# Real-time Planning: Reducing Complexity for Scaling Up

**Composition du jury:**

Sylvain Lagrue  
Professeur  
Université de Technologie de Compiègne  
Président

Peter Jonsson  
Professeur  
Université de Linköping, Suède  
Rapporteur

Aurélie Beynier  
Maître de Conférences, HDR  
Université de la Sorbonne  
Rapporteur

Tristan Cazenave  
Professeur  
Université Paris-Dauphine-PSL  
Directeur

Éric Jacopin  
Senior AI Programmer, HDR  
(ex-)Centre de Recherche de Saint-Cyr  
Co-directeur

Stéphane Cardon  
Maître de Conférences  
Centre de Recherche de Saint-Cyr  
Co-encadrant

Christophe Guettier  
Manager, Expert Emeritus  
Safran Electronics & Defense  
Invité

Robert Gabriel  
Lead R&D Developer  
Ubisoft La Forge, Québec, Canada  
Invité

Simon Girard  
AI Programmer  
Ubisoft Québec, Québec, Canada  
Invité
# Contents

1 Introduction 5
   1.1 AI Planning in Real-Time 5
   1.2 Motivation 6
   1.3 Approach 7
   1.4 Outline of the Thesis: my Plan. 7

2 Goal-Oriented Action Planning 11
   2.1 The Theory of GOAP 14
      2.1.1 The Action Representation 14
      2.1.2 The Search Algorithm 16
      2.1.3 The Case of FEAR 18
      2.1.4 The Context Preconditions 18
   2.2 Results 19
      2.2.1 The Context Precondition Impact on the Branching Factor 19
      2.2.2 Action Plan Construction 20
      2.2.3 Construction Time 23
   2.3 Discussion 24
      2.3.1 The Context Precondition Impact on Planning 24
      2.3.2 Planning Failures 25
   2.4 Conclusions 26

3 Simplified Action Structure 29
   3.1 SAS Planning Framework 30
      3.1.1 World State Variables 30
      3.1.2 Operators and Actions 32
      3.1.3 SAS Planning Structures 34
      3.1.4 SAS Planning Problem Instances 36
3.2 A Tractable Class of Problems: SAS-PUS ........................................ 36
  3.2.1 Syntactical Restrictions ....................................................... 36
  3.2.2 SAS-PU and SAS-PUS ....................................................... 38
  3.2.3 Discussion ................................................................. 43

4 Topological Planning ............................................................... 45
  4.1 Prerequisites ........................................................................ 46
  4.2 Operator Graphs ................................................................. 50
    4.2.1 Operator Graph ............................................................ 50
    4.2.2 Domain Operator Graph ............................................... 51
    4.2.3 Action Graph ............................................................... 56
    4.2.4 Discussion ................................................................. 56
  4.3 Truth Criterion ..................................................................... 57
    4.3.1 General Form of an Action Chain ................................. 57
    4.3.2 $\Upsilon_1$ Truth Criterion ............................................. 59
  4.4 A New Structural Restriction: $C_k$ ................................. 64
    4.4.1 The Class of Problems SAS-PUC$_0$ .............................. 66
    4.4.2 The Class of Problems SAS-PUC$_2$ .............................. 66
    4.4.3 The Class of Problems SAS-PUC$_S$ .............................. 67
    4.4.4 The Class of Problems SAS-PUC$_2^S$ .......................... 69
    4.4.5 The Class of Problems SAS-PUC$_k$, with $k \geq 3$ .......... 72
  4.5 TopoPlan $P$ ...................................................................... 73
    4.5.1 Setting up the Problem .................................................. 74
    4.5.2 Algorithm .................................................................... 76
    4.5.3 Discussion ................................................................. 81

5 Benchmarks and Results ............................................................. 85
  5.1 Scalable Benchmarks .............................................................. 86
    5.1.1 Description of the Scalable Benchmarks .................... 86
    5.1.2 Empirical Comparison of Time Complexities ($P$ vs $P$) .... 89
    5.1.3 Another Scalable Example: BlockWorld ...................... 92
  5.2 Realistic Benchmarks ............................................................ 96
    5.2.1 Read Dead Redemption 2 .............................................. 97
    5.2.2 Assassin’s Creed: Origins .............................................. 99
    5.2.3 Horizon Zero Dawn ...................................................... 104
    5.2.4 Runtime Performances ................................................ 111
    5.2.5 Discussion ................................................................. 119

6 Conclusion ............................................................................. 123
  6.1 How to Reduce Time Complexity to Scale up Real-time Planning? 123
  6.2 Future Work ....................................................................... 124

Appendices ............................................................................. 127
# Table of Contents

A Modified Planning Algorithm for SAS\(^+\)-PUS Problems ........................................ 129

B Domain Operator Graphs with (PU) Operators ......................................................... 135

C Restrictions C\(_2\), C\(_2^S\) and C\(_2^*\)  ......................................................... 139

\hspace{1em} C.1 Proof of SAS-PUC\(_2^S\) ................................................................. 144

\hspace{1em} C.2 Proof of SAS-PUC\(_2^*\) ................................................................. 145

D Proof of Theorems in Chapter 4 ................................................................................. 149

\hspace{1em} D.1 Proof of BuildChain .............................................................................. 149

\hspace{1em} D.2 Proof of TopoPlan \(\mathbb{P}\) ........................................................................... 152

E Step by Step Examples of \(\mathbb{P}\) ............................................................................ 157

F Red Dead Redemption 2: Operator Sets ................................................................. 165

G Résumé en Français ................................................................................................... 171

\hspace{1em} G.1 Introduction ............................................................................................. 172

\hspace{1em} G.2 Goal-Oriented Action Planning ............................................................. 174

\hspace{1em} G.3 Simplified Action Structure ..................................................................... 176

\hspace{1em} G.4 La Planification Topologique ................................................................. 178

\hspace{1em} G.5 Benchmarks et Résultats ......................................................................... 183

\hspace{1em} G.6 Conclusion ............................................................................................... 186
This thesis is about Artificial Intelligence (AI) Action Planning. In particular, my researches focus on action planning in real-time and I use video games to test, experiment and analyze my findings. AI is an area of knowledge and research aiming to design intelligent computational agents, i.e. agents that act intelligently in their environment and according to their goals. In my opinion, the best definition of AI, or Computational Intelligence, is the one of D. Poole, A. Mackworth and R. Goebel [1, p.1]:

> Computational intelligence is the study of the design of intelligent agents. An agent is something that acts in an environment — it does something. Agents include worms, dogs, thermostats, airplanes, humans, organizations, and society. An intelligent agent is a system that acts intelligently: What it does is appropriate for its circumstances and its goal, it is flexible to changing environments and changing goals, it learns from experience, and it makes appropriate choices given perceptual limitations and finite computation. The central scientific goal of computational intelligence is to understand the principles that make intelligent behavior possible, in natural or artificial systems. The main hypothesis is that reasoning is computation. The central engineering goal is to specify methods for the design of useful, intelligent artifacts.

This introduction is organized as follows: I first give a definition of AI planning in real-time, I then give my motivation followed by my approach to answer the main question: how to reduce time complexity to scale up real-time planning? I end the introduction with the outline of my thesis.

### 1.1 AI Planning in Real-Time

An AI planning problem, or more precisely an AI action planning problem (to distinguish with pathfinding and other sub-fields of planning), is an AI problem where an intelligent agent can plan in its environment to reach its goals. To do so, the environment and relevant
parameters of the agent are symbolized to form states that the agent uses to evaluate a real situation at a given time in an abstract way. Most of the time, an agent states its current situation or its goals. The agent is then equipped with actions or operators that the planner uses to create action plans in order to look ahead in time and perform actions in the world. Furthermore, actions and action plans both act as a transition function between two states. They can have a cost in order to characterize the quality of a transition, i.e. the relevance of an action or an action plan for a given situation.

Planning in real-time concerns planners that plan in a very short amount of time, namely in a millisecond or two at most. A good example of intelligent agents using such planners are the Non-Playable Characters (NPCs) populating some video games [2–5]. For those games, real-time means 10% of the time between two frames\(^1\), i.e. 3.33\(\text{ms}\) for a video game with a framerate of 30 Frame per Second (FPS), 1.67\(\text{ms}\) for a 60 FPS games, 0.83\(\text{ms}\) for a 120 FPS game, and so on. As I am interested in NPCs controlled by planners, I will present the Goal-Oriented Action Planning system as it is the technology used in today’s video games to plan in real-time (cf. Chapter 2). According to a study I conducted on GOAP, I found patterns that can be used to restrict planning problems. These patterns lead me to study the Simplified Action Structure formalism [6] which I present in Chapter 3. I then use this formalism in Chapter 4 and Chapter 5 as a basis for my work.

\section*{1.2 Motivation}

I strongly believe planning is a field of AI that is the future for NPCs behaviour. Video games are more and more realistic and beautiful due to the graphics. Maps move from corridors (Call Of Duty, The Witcher 1 & 2, Pacman...) to huge open-worlds (The Elder Scrolls: Skyrim, Gran Theft Auto, Assassin’s Creed, Red Dead Redemption,...), and even infinite open-worlds thanks to procedural generation (Minecraft, No Man’s Sky...). Some stories are so immersive that you feel as if you are an actor in a movie (The Witcher 3, Detroit: Become Human, Fable: The Lost Chapter,...). Virtual Reality headsets are close to be found in every household, especially with the announcement of the Metaverse. That is to say, there are many new technologies to create more and more immersive video games. This is therefore important to populate these high quality environments with believable NPCs, i.e. NPCs that do not only patrol or interact purposelessly with the player.

Believable NPCs are NPCs that live in the environment, that seem to have a real life no matter what the player does. If the NPCs have to interact with the player, their actions have to be meaningful according to their goals. That is where planning is interesting, depending on the goal of one NPC, the planner will try to create an action plan that links the current state of the NPC to the goal state. An action plan is more meaningful than just one action in a sense that the first action of the plan is useful for one or more actions that will come next. As an example, an NPC that carries a box may do that to strengthen a position, and the same NPC will then use his strengthened position to take cover so as to prepare an ambush and so on. Goal-Oriented Action Planning (GOAP) [7]

\footnote{Only 10\% of the processing budget is granted for AI in video games.}
CHAPTER 1. INTRODUCTION

is the most used planner in video games. F.E.A.R. [2], Shadow of Mordors [3], Assassin’s Creed [5], Rise of the Tomb Raider [4], all these games have used GOAP to make their NPCs more believable. NPCs of F.E.A.R are still perceived as some of the best NPCs so far in the video game industries [8]. Pedestrians, faunas, enemies of recent Assassin’s Creed (Odyssey, Valhalla...) are driven by GOAP: it makes these games more dynamic and immersive for the players [9].

Real-time means the planner has to provide as many action plans as possible within the time budget, e.g. between two frames in a video game. The more agents to control, the less time to generate one plan. Idem for the number of actions inside a plan, the more actions, the more time to generate one plan. The worst-time complexity is therefore a significant criteria to evaluate the planning algorithm. Unfortunately, the current planning system used in GOAP is search-based and has an exponential time complexity [7]. This leads to development restrictions to respect the time budget while trying to meet the growing expectations of the players. My main question throughout this thesis is therefore: how to reduce time complexity to scale up real-time planning?

1.3 Approach

My approach to answer my main question was to first study a GOAP system inside a commercial video game. F.E.A.R., which is still a great video game reference in term of believable NPCs, is also one of the few commercial video games using a GOAP system and whose source code is freely available online [10]. Based on the results of my AI planning analytics of F.E.A.R., I then analyzed several dozen GOAP GitHub projects to see whether the planning system is always implemented the same way. I finally reached some Lead AI and teams that implement GOAP in their video games to ask them some questions about their development habits. Eventually, I drew conclusions that I then turned into assumptions.

These assumptions then led me to study the Simplified Action Structure (SAS) formalism. Indeed, two of my assumptions are in fact existing input restrictions of this formalism. C. Bäckström, the founder of SAS, restricted SAS planning problems to find tractable subclasses of problems. He also developed a polynomial time algorithm that solves instances of his tractable problems. I re-used C. Bäckström’s approach to create new classes of problems adapted to NPCs in video games and I then developed a linear-time algorithm to solve these new problems. Eventually, I carried out several experiments using several different benchmarks to evaluate the effectiveness of my algorithm and to check that it indeed respects the real-time constraint.

1.4 Outline of the Thesis: my Plan.

Chapter 2: Goal-Oriented Action Planning  This Chapter presents the GOAP system, my planning analytics on F.E.A.R. and my findings. The conclusions allowed me to
define Proposition 2.4.1 which presents my assumptions for the rest of the manuscript.

**Chapter 3: Simplified Action Structure** I studied the work of C. Bäckström [6] to answer my main question and to create a planner that can plan in real-time for more than a few dozen of NPCs and with longer plans. I re-use C. Bäckström’s approach to reduce the time complexity. He restricted the input of planning problems to create tractable ones that can be solved by a polynomial time algorithm. Before going any further in his direction, I recall the theory behind SAS and adapt it to my notations and my needs. I present the classes of problems SAS-PU and SAS⁺-PUS. The latter is a class of tractable problems whose instances can be solved by an existing polynomial time algorithm. I implemented it in C++ and tested it on realistic benchmarks.

**Chapter 4: Topological Planning** This chapter is the core of this thesis where I present new classes of problems and my linear-time algorithm, named TopoPlan and denoted \( \mathcal{P} \), that solves instances of these new problems. In this chapter, I first present prerequisites that take advantage of the post-uniqueness and unariness restrictions. In particular, I present how to identify operators and actions in constant time while planning and how the identifiers can be used to create new sets of predecessors. Then, I introduce operator graphs and action graphs which are graphical representations of the input and the output of a SAS-PU planning problem. I then use these graphs to prove the \( \Upsilon_1 \)-Truth Criterion, a truth criterion that take into account my assumptions, and to introduce a new structural restriction, denoted \( C_k \), which allows me to create new classes of problems respecting my assumptions. I finally present my linear-time algorithm that takes advantage of the prerequisites and which I prove correct and complete with the previously the \( \Upsilon_1 \)-Truth Criterion.

**Chapter 5: Benchmarks and Results** This chapter is divided in two parts followed by a discussion. The first section presents scalable benchmarks which are benchmarks designed to grow with respect to the number of state variables or with respect to the maximum number of values in a state variable domain (cf. \( \mathcal{M} \) and \( \mathcal{D}_v \) in Definition 3.1.1). Among these benchmarks are the BlockWorld and the Tunnel examples [6]. I use these benchmarks to strengthen empirically my theoretical results. The purpose of the second section is to show that realistic benchmarks can be represented with my new classes of tractable problems. Among those realistic benchmarks are RDR-2, ACO and HZD which are respectively based on the commercial video games Red Dead Redemption [11], Assassin’s Creed: Origins [12] and Horizon Zero Dawn [13]. I then use these benchmarks to compare my C++ implementation of both \( \mathcal{P} \) and \( \mathcal{P} \). The results are astonishing: \( \mathcal{P} \) is about > 30 thousands times faster than \( \mathcal{P} \) to solve instances of these realstic problems. I end the chapter with a discussion that gives explanations on why \( \mathcal{P} \) is much faster than \( \mathcal{P} \) as well as some other interesting differences between the two.
Chapter 6: Conclusion  I eventually conclude my thesis by summarizing each chapter and, then, by answering my main question: how to reduce time complexity to scale up real-time planning? I then present some possible future works based on my thesis results.
Planning in real-time is one of the alternatives [14] for modeling and editing the behavior of Non-Player Characters (NPC) in commercial video-games. Goal-Oriented Action Planning (GOAP) was first introduced in F.E.A.R. [2], a First-Person Shooter (FPS) released in 2005. When the goal of an NPC is updated according to the current game state, GOAP searches for a plan in a backward manner to achieve this new goal: GOAP starts with searching an action which can achieve this new goal in the set of actions of the NPC; the preconditions of such an action become new goals to achieve, and so on until the current game state can satisfy the preconditions of an action. During the search, context preconditions are used to query of the current game state and compute various numerical values (e.g. distances to dodge).

In F.E.A.R., actions are implemented as C++ classes, and search reuses A* [15] from the path planning module of the game engine; the heuristic function used for GOAP evaluates each search node summing the costs of actions, and then computes the number of goals yet to be satisfied. Action costs in F.E.A.R. are not dynamic (on the contrary of the implementation of GOAP in the Tomb Raider series released by Crystal Dynamics since 2013 [4], for instance): once designed and implemented in the game, action costs are constant values that will never be modified during the search.

In this chapter, I address the use of action costs in GOAP, and more generally the representation of actions in practice in commercial games using GOAP. Indeed, this system handles unexpected game situations better than other behavior engineering alternatives; but game designers feel a lack of control [14] over an AI architecture able to produce unexpected NPC behaviors [7]. Consequently, action costs appear as a mean of tweaking what happens on-screen, thus entailing the recurring question of a new to GOAP development team: how to design the action costs?

F.E.A.R.’s AI has been widely recognized as the best in class when released [16], and still is recognized as such [8]. As controversial as such reviews can be, F.E.A.R.’s GOAP nevertheless has had a great impact among game AI developers and definitely sparked the use of planning in commercial video-games in general and in FPSes in particular.

Thanks to the SDK released by Monolith in 2006 [10], the game AI development dif-
CHAPTER 2. GOAL-ORIENTED ACTION PLANNING

Abstract
Attack(6)
AttackGrenade
GrenadeFromCover(2)
FromVehicle(1) Melee(3) Ready(7)
AttackFromNode
Crouch(5) FromView(4.5)
FromAmbush(4)
FromCover(4)
BlindFire(2)

Figure 2.1: C++ action hierarchy for attack actions of the action set of the basic soldier in F.E.A.R. where each node represent a class; the root is the Abstract node colored in yellow. The cost of each action appears in brackets after the name of the action. Actions whose names appear in italics are abstract C++ classes that cannot be instantiated. All actions except Ready follow Jeff’s proposal of assigning lower costs to the leaves of the class hierarchy.

Difficulty is no longer a coding problem: the C++ code of F.E.A.R.’s GOAP can be studied and its main parts can be duplicated. Although today many games use GOAP for NPC behavior engineering, F.E.A.R. remains to this day the only game whose SDK allows access to the C++ code of the planner. There are also more than 300 GOAP repositories on Github, implementing planning functionality for potential commercial video-games in python, ruby, and javascript. C++ and C# share 54 implementations, not all of which will end up in a commercial game, of course; but if that’s the case, then these planners will face what is now the crucial step of the use of GOAP in a video-game: the design of the action costs for the game for which they have to generate plans. Indeed, using GOAP for a new game means new actions, and consequently means new action costs must be designed.

The original author of GOAP, Jeff Orkin, proposes to “apply a cost to actions to force A* to consider more specific actions before more general ones.” [7, page 108]: that is, the higher the action in the C++ inheritance hierarchy, the larger the action cost; and that is indeed what I observe when studying F.E.A.R.’s SDK. Figure 2.1 shows the distribution of the action costs in F.E.A.R.

GOAP uses A* to search for plans: when a solution is found, its cost merely is the sum of the cost of its actions, as all goals of a solution-plan are satisfied; therefore, this cost is the minimal cost that can be found. At each step during search, when faced with a choice among several actions, GOAP will choose the one with the lowest action cost to achieve the minimal cost for a solution. We can observe that this is not what happens while playing F.E.A.R., however; not only any kind of expensive action can appear more often than a lower cost action, but this is also the case if I restrict my study to actions achieving the same goal.

Figure 2.2 shows in-game data representing 58 hours of playing F.E.A.R. over 617
In F.E.A.R., low-cost actions are not necessarily used more than higher-cost actions

Figure 2.2: x-axis represents the action usage and y-axis is the action cost; the 72 (red) dots of this graphic represent the 72 actions logged in the F.E.A.R. sessions. We can observe that there is no clear influence of the action cost over the action usage: some, more expensive, actions are used much more than other, less expensive, actions.

Sessions from 10 players where 337600 plans were generated and recorded so that action occurrences could be computed; each red dot represents one of the 72 actions logged in the in-game data, whose abscissa indicates its usage and the ordinate indicates its cost. We can observe that the right-most red dot, which represents the most used action in the game (i.e. this action occurs about 22.5% of the time), has a cost of 3 which is much higher than the cost of much less used actions that appear in the bottom left corner of the figure.

Indeed, we can observe in Figure 2.4 the following: for any pair of actions \((a_1, a_2)\) such that action \(a_2\) occurs more than action \(a_1\) (that is, action \(a_2\) appears more on the right in Figure 2.2 than action \(a_1\)) then the average value of \(\frac{\text{cost}(a_2)}{\text{cost}(a_1)}\) is 1.67. Consequently, (1) despite the use of A* and the search for minimal cost plans, expensive actions occur more often than cheap actions, and (2) more abstract actions occur more often than more specific actions, despite Jeff Orkin’s proposal.

If I restrict my study to attack actions achieving the same KillEnemy goal, for instance, Figure 2.3 shows the same result: some expensive attack actions are used more than attack actions with a lower cost, which definitely brings to the forefront the question of how to design the costs of actions.

If game designers cannot rely on action costs to impact action usage, then why use action costs at all? Are action costs useful for real-time purposes or for anything else in GOAP? If the use of action costs in GOAP is unclear, can we safely remove action costs? In the next sections I conduct a close study of the action costs in GOAP in order to provide answers to these questions; I begin with explaining GOAP.
In F.E.A.R., low-cost attack actions are not necessarily used more than higher-cost attack actions.

Figure 2.3: The 10 (red) dots of this graphic represent the 10 attack actions logged in the F.E.A.R. sessions achieved the same KillEnemy goal for the action set of the basic soldier and are the 10 actions from Figure 2.1; x-axis represents the action usage and y-axis is the action cost; We can observe that there is no clear influence of the action cost over the action usage: some, more expensive, attack actions are used much more than other, less expensive, attack actions.

2.1 The Theory of GOAP

GOAP is a decision-making system coupled with a planning one. The world representation of an agent is based on STRIPS: it is an action-centric representation but with multi-value state variables, and each action has a cost. Then, the agent has to be equipped with sensors, a set of goals and a set of actions which will allow the NPC to sense, plan and act in its environment. Each time a perturbation occurred, the agent will sense it and then perform a goal relevance calculation thanks to its decision-making system to decide what to do. It will then decide how to achieve its goal thanks to its planning system which will search for an action plan by using the A* search algorithm. Finally, all of this must be achieved in a short period of time in order to respect the constraint of agent control in real-time. In this section, I illustrate my explanations with examples from the video game F.E.A.R.. I also explain how GOAP is implemented inside this game in subsection 2.1.3.

2.1.1 The Action Representation

As GOAP is STRIPS-based, the actions are represented with the classical planning model, that is with pre- and post-conditions that are conditions on state variables. This action-centric representation coupled with a search algorithm will allow the planner to build sequences of actions in order to build solutions plans and, therefore, to link two different states. The following will present how these parameters are used.
In F.E.A.R., actions used more often than others also cost 1.67 times more on average.

The median value of the action cost ratio, represented by the white line, is 1: 50% of the action cost ratios are ≤ 1 while 50% of the action cost ratios are ≥ 1; the mean value of the action cost ratio, represented by the black line is 1.67: any action $a_2$ used more than $a_1$ is on average 1.67 more expensive than $a_1$. If we restrict the study to attack actions from Figure 2.1 then the average value is 2.1.

The postconditions

They are used to satisfy goals or unsatisfied preconditions of other actions (these preconditions can be seen as sub-goals). Then, as A* goes backward to search for actions (see 2.1.2 and Algorithm 1 for more details), the first pruning is made by selecting actions with the proper postconditions. Only these actions can become candidates in the search space of A*. For example in F.E.A.R., for the soldier, if \texttt{KillEnemy} is called, the goal is to set the state symbol \texttt{TargetIsDead} to the value \texttt{True}. Thus, the first step for the planner is to only keep actions having the postcondition \texttt{TargetIsDead} = \texttt{True}. For the soldiers, which are the most basic NPCs in F.E.A.R., 10 actions possess this postcondition. Given the fact that A* goes backward to search, this means the brute branching factor at this node is 10. So, the more actions with the same and only postcondition, the higher the brute branching factor is to satisfy a goal or an unsatisfied precondition. The average branching factor is never equal to the brute one due to context preconditions, however. (See 2.1.4 for more details). It is also worth noticing that actions are unary in F.E.A.R., i.e. each action only have one post-condition. This can be seen in the source code provided by the SDK.

The preconditions

When an action is chosen during the plan building process, its preconditions are then consulted to figure out the next step. All its unsatisfied preconditions become goals to achieve. That is, this action can not be activated by the NPC at its current state, therefore other actions have to be considered. The process continue until all goals is achieved. One
should notice that the more actions whose preconditions are unsatisfied, the greater the depth of search, i.e. the greater the number of states to be crossed during the search.

### 2.1.2 The Search Algorithm

Originally, i.e. in the video game F.E.A.R., the search algorithm used to search for valid plans is A* [7]. It was already implemented for the pathfinding: its reuse avoided the implementation of a new search algorithm, and reduced both development time and bugs. It is the same for other commercial games such as Middle-earth: Shadow of Mordor [3] or Rise of the Tomb Raider [4]: they use A* in their GOAP planning system. On GitHub, I have analyzed the source code of all the C++ and C# projects, which represent 54 projects. Only half of them uses A*. A third of them do not use any search algorithm: they build the entire graph before selecting the cheapest path. The remaining ones use a best-first search algorithm. Due to the preponderance of A* in the various GOAP systems, and because it is the original one chosen by Jeff Orkin, I decided to only considered A* in my study.

A* is used to search for valid plans. Game developers had to configure the actions and the planner properly so as to use together [7]. A* works with heuristics and cost metrics. At the frontier of the search (See SF on Algorithm 1), the node with the lowest score will be selected. The score, returned by the fitness function $f$, is the sum of the heuristic function $h$ and the cost function $g$: $h$ estimates the remaining distance from a node at the search frontier to the goal node whereas $g$ evaluates the current cost to reach this node from the start. To use A* in GOAP, a node is characterized by its goals (a subset of a goal state, noted $G$ in Algorithm 1) and the action plan built to reach itself (noted $P$); a move from a node to another is fulfilled thanks to an action. The selection criterion for the next node to explore is still made by the fitness function $f$: the one with the lowest score is selected (noted $\text{min}(\text{SF})$). To evaluate $f$, the cost and the heuristic functions had to be adapted: the first one evaluates the sum of the action costs of a plan ($g(P)$), while the second one counts the number of goals ($h(P)$). The output of the algorithm is an action plan, and it will be the cheapest one if A* is admissible [17].

In F.E.A.R., most of the actions have only one postcondition, therefore the heuristic does not only evaluate the number of goals, it also evaluates the number of actions required to achieve the goals of a node: if a node possesses two goals, then at least two actions are required. Then, all the action costs\footnote{Except for the cost of EscapeDanger which is equal to 0.5.} are equal to or greater than 1.0, therefore the heuristic is admissible and it is optimal in F.E.A.R.: the planner always returns the best solution [17], i.e. the cheapest one. One should notice that the cheapest plan does not mean the shortest one. Indeed, let imagine two plans $P_1$ and $P_2$ that solve a goal from the NPC’s current state, if $P_1$ is composed of three actions with a cost of 1 (so $g(P_1) = 3$) while the second one is composed of one action with a cost of 10 (so $g(P_2) = 10$), then the cheapest plan returned by A* is $P_1$ and it is not the shortest one (3 actions against 1). Thus, in F.E.A.R., as the heuristic function $h$ also estimates the number of actions to use,
Algorithm 1 A* for GOAP

{The search for a plan goes backward from the goals to the NPC’s current state.}

**Inputs:** $G \neq \emptyset$, a list of goals;
$A \neq \emptyset$, the NPC’s action set;
$S_C$, the NPC’s Current state;

**Output:** $P$, an action Plan.

$P \leftarrow \emptyset$
$g(P) \leftarrow 0$ {the cost function of $P$}
$h(P) \leftarrow \text{Length}(G)$ {the heuristic function of $P$}
$f(P) \leftarrow g(P) + h(P)$ {the fitness function of $P$}

$SF \leftarrow \{(G, P)\}$
{min($SF$) is the couple $(G, P) \in SF$ (Search Frontier) such that $\forall (G_i, P_i) \in SF, f(P) \leq f(P_i)$}

while $SF \neq \emptyset$ do

$(G_i, P_i) \leftarrow \text{min}(SF)$

if $G_i \in S_C$ then

Return $P_i$ {Exit with success}

else

for each action $a \in A$ whose at least one postcondition belongs to $G_i$ do

$P_j \leftarrow \{a\} + P_i$

$G_j \leftarrow G_i \setminus \text{Post}(a)$

for each precondition pre of $a$ do

if $\text{pre} \notin S_C$ and $\text{pre} \notin G_j$ then

$G_j \leftarrow G_j + \{\text{pre}\}$

end if

end for

$g(P_j) \leftarrow g(P_i) + \text{Cost}(a)$

$h(P_j) \leftarrow \text{Length}(G_j)$

$f(P_j) \leftarrow g(P_j) + h(P_j)$

$SF \leftarrow SF + \{(G_j, P_j)\}$

end for

end if

$SF \leftarrow SF \setminus \{(G_i, P_i)\}$

end while

Return $\emptyset$ {Exit with failure}
it fosters short plans. Indeed, the more action required, the greater the heuristics. And if 
\( h \) increases, \( f \) also increases (\( f = g + h \), \( g \) being independent of \( h \)), which results in not favoring nodes that lead to long action plans (according to the selection criterion of \( A^* \)).

In my previous example, if we focus on the first step of the search, for each plan, \( P_1^1 \) and \( P_1^2 \) (where \( P_2^1 = P_2 \)), we have \( 2 \geq h(P_1^1) \geq 1 \) whereas \( h(P_2^2) = h(P_2) = 0 \). Therefore, the heuristic fosters \( P_2 \), the shortest plan and not \( P_1 \) the cheapest one.

### 2.1.3 The Case of FEAR

In the video game F.E.A.R., sensors (e.g. line of sight, detecting light, hearing footsteps, etc.) allow the Non-Playable Characters (NPCs) to have an individual understanding of their surroundings. The acquired pieces of information are stored in their working memory and if it is a shareable knowledge, it is then managed in a blackboard [7]; this technique allowed developers to minimize the number of the state variables for the NPCs [7]. There are 21 different state variables in F.E.A.R. like `TargetIsAimingAtMe`, `TargetIsDead`, `AtNodeType` or `WeaponLoaded`. These two components (the individual understanding and the state variables) are both required by the decision-making system to assign goals to an NPC. Then, the planning system is requested to search for a valid plan to achieve the goals, as explained in subsections 2.1.1 and 2.1.2.

Although the previous implementation description is the basic one for GOAP [18], _context preconditions_ have been added to actions so as to lighten the planning process [7].

### 2.1.4 The Context Preconditions

According to what I explained in subsections 2.1.1 and 2.1.2, actions are first selected with their postconditions. As several actions can have the same postconditions (in F.E.A.R., 84 actions are available for 21 state variables, which entails that some actions possess the same postcondition), with no other selection criterion than the postconditions, the branching factor of each node of the search space is equal to the number of actions having the same postconditions: it is the brute branching factor. In order to maintain “Lightweight Planning” [7] in F.E.A.R., context preconditions have been added to action. Their purpose is to minimize the search space and optimize precondition validation so as to maintain a fast enough planning system for real-time. Indeed, let suppose we have \( b \) actions every step and a depth of search of \( d \), the complexity of the \( A^* \) search algorithm is \( O(b^d) \).

Therefore, as a developer, you definitely want to keep \( b \) and \( d \) as low as possible so as to avoid a combinatorial explosion and, consequently, a frame rate drop while planning in real time. The context preconditions, which are queries on the world, will impact \( b \) by pruning candidate actions. Each action will have their own ones and it will allow the planner to decide if the current action is viable or not. If not, then the action is merely withdrawn from the search, and so \( b \) decreases. For instance in F.E.A.R., if an NPC detects a disturbance, its goal will be to set the state symbol `DisturbanceExists` to the value `False`. Four actions possess this postcondition but only two of them, `ReactToDanger` and `EscapeDanger`, will be considered if the disturbance is dangerous while the other two, `InspectDisturbance` and
CHAPTER 2. GOAL-ORIENTED ACTION PLANNING

LookAtDisturbance, will be considered for safer disturbances. This technique, the pruning with context preconditions, has also been used by other studios than Monolith Productions. Crystal Dynamics (“Rise of the Tomb Raider”) or High Moon Studio (“Transformers 2”) have also implemented it in their commercial games using GOAP [19]. Furthermore, among the GitHub projects observed, a quarter of them use this technique.

In the next sections, I study the impact of these context preconditions on planning and I am going to show that they are behind the no clear influence of the action cost over the action usage.

2.2 Results

A* is the search algorithm used in GOAP to search for plans, actions must therefore be designed with a cost: it will be used by the planner to distinguish actions that will compete for a same goal. Therefore, the proposal made by Jeff Orkin was: “apply a cost to actions to force A* to consider more specific actions before more general ones” [7], i.e. the lower the cost, the more likely an action is to be chosen. As explained in the introduction of this chapter, in the video game F.E.A.R., for attack actions having the same postcondition (KillEnemy, i.e. TargetIsDead = true) some expensive attack actions are used more than attack actions with a lower cost. This contradicts Jeff Orkin’s proposal. In fact, it seems that context preconditions, by pruning candidate actions, play a more important role in planning than expected.

2.2.1 The Context Precondition Impact on the Branching Factor

As I explained in subsection 2.1.1, many actions can have the same postcondition, which means for a same goal we can have several candidates: this is the brute branching factor. But, in subsection 2.1.4, I have highlighted that the context preconditions impact the branching factor (BF) by withdrawing actions that are not viable depending on the context: indeed, BF decreases each time an action is removed. I made an experiment in F.E.A.R. in order to visualize the average branching factor while planning. The table 2.2.1 gives the influence of the context preconditions on the branching factor with respect to four of the most diverse goals, i.e. the number of actions for these goals are greater than the others. (“WSE” stands for state Event.)

<table>
<thead>
<tr>
<th>Soldier</th>
<th>Name of the postcondition</th>
<th>Target Is Dead (true)</th>
<th>Disturbance Exists (false)</th>
<th>Target Is Suppressed (true)</th>
<th>ReactTo WSE (damage)</th>
<th>BF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brute BF</td>
<td></td>
<td>10</td>
<td>5</td>
<td>2</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Average BF</td>
<td></td>
<td>1.52</td>
<td>2.18</td>
<td>0.03</td>
<td>0.33</td>
<td></td>
</tr>
<tr>
<td>Median BF</td>
<td></td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Quartile 1 BF</td>
<td></td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Quartile 3 BF</td>
<td></td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Min BF</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Max BF</td>
<td></td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
According to the results, the context preconditions have been a strong pruning: they have greatly reduced the branching factor of the search. In F.E.A.R., a soldier can choose between 10 attack actions with TargetIsDead = True as their only postcondition. It implies a brute branching factor of 10 but, in practice, the branching factor has an average been equal to 1.52. In other words, the search algorithm for the goal TargetIsDead = True has on average had half a chance to only have one candidate action and half a chance to have two candidate actions. Therefore, the context preconditions can have a great impact on the branching factor: what is the place left for the action costs which such an impact?

2.2.2 Action Plan Construction

In order to see the impact of the action costs and the impact of the context preconditions on the planning activity while playing, I have carried out two experiments on the video game F.E.A.R. (1) I have asked some players to play the game entirely and several times with two different cost configurations: the first one is the original game with the original costs (called OC+ configuration), while the second one set all the costs to 1, i.e., unitary costs (called UC configuration). (2) I have played the game entirely with the original costs but with the context preconditions removed from the planning activity (called OC− configuration). During all these gaming sessions, I have gathered data: 647,548 action plans were generated during (1): 337,600 plans come from the OC+ sessions (they represent 58 hours 21 minutes of playing time) while 309,948 plans come from the UC sessions (58 hours 35 minutes); 754,205 action plans were generated during (2) and it only represents 4 hours 5 minutes of playing time. Among all these plans, there are only 87 distinct ones in the framework of OC+, 73 in UC and 56 in OC−. In other words, the GOAP planner in F.E.A.R. has only created 87 plans in OC+, 73 in UC and 56 in OC− and have repeated them many times during the game sessions. It corroborates results in [20] explaining that AI planning pattern can be found with GOAP. In addition, UC and OC+ share 69 common plans (68 if we remove the empty plan: {}), i.e. if we only keep the successful plans) which represent 95.61% of the successful OC+ plans and 99.93% of the successful UC plans. OC+ and OC− share 48 common plans (47 if {} is removed) which represent 95.43% of the successful OC+ plans and 87.08% of the successful OC− plans.

Let POC+, PUC and POC− be respectively the finite set of distinct plans generated with OC+, the finite set of distinct plans generated with UC and the finite set of distinct plans generated with OC−; we have #POC+ = 87, #PUC = 73 and POC− = 56. POC+ − PUC, PUC − POCS, POC+ − POC− and POC− − POC+ represent the respective complements between POCS and PUC and between POCS and POC−. The table in subsection 2.1 gives information on these finite set of distinct plans with respect to the plan length (where the length is the number of actions). If we remove the planning failures from the results, which represent a large part for the UC and OC+ configurations, the occurrence probability of POCS − PUC is 4.503% (instead of 1.133%) whereas the occurrence probability of PUC − POCS is 0.065% (instead of 0.016%). Consequently, I consider that PUC ⊂ POCS, i.e., all the plans generated in UC were generated in OC+, i.e. the setting of the costs in F.E.A.R. has brought a diversity represented by the complement POCS− − PUC, which is 4.503% if we
only consider the successful plans.

Furthermore, the majority of the successful plans have one or two actions. Indeed, planning failures set aside, they represent about 98.8% of the generated plans in OC$^+$ and UC, which corroborates the results in [21] showing that GOAP returns short plans.

Figure 2.5 shows that the probability of generating a plan as a function of the number of actions constituting it has been broadly the same for OC$^+$ and UC. Planning failure represents respectively 74.7% and 75.2% for both of these configurations, where a failure means the planner was not able to build a valid plan to achieve a given goal, so it returns an empty plan (See subsection 2.3.2 for more details about planning failure). Thus, to modify the costs did not affect the length of the plans. On the contrary, with OC$^-$, only single-action plans (almost) were generated by the planner. Therefore, in OC$^+$, context preconditions allowed the construction of longer plans. Furthermore, almost no planning failure can be seen with OC$^-$, which means the planner never failed to find a plan. But, planning failure is not necessarily a bad thing because action plans returned with OC$^-$ were not often the most realistic ones: the NPCs behaved oddly while playing. This last analyze can be highlighted with Figure 2.6 which compares the occurrence probability of the common attack action plans between the three configurations. The most likely plan for OC$^-$ is the single action plan \{\textit{AttackFromVehicle}\} with an occurrence probability of 32.79%. This plan makes sense in F.E.A.R. when an NPC is close to a vehicle, but this is not often the case. On the other hand, the most likely attack action plan for UC

Table 2.1: F.E.A.R. Data from my experiments

<table>
<thead>
<tr>
<th>Plan Length</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_{OC^+}$</td>
<td>1</td>
<td>62</td>
<td>15</td>
<td>6</td>
<td>3</td>
<td>87</td>
</tr>
<tr>
<td>$P_{UC}$</td>
<td>1</td>
<td>50</td>
<td>15</td>
<td>5</td>
<td>2</td>
<td>73</td>
</tr>
<tr>
<td>$P_{OC^-}$</td>
<td>1</td>
<td>43</td>
<td>8</td>
<td>3</td>
<td>1</td>
<td>56</td>
</tr>
<tr>
<td>Number of occurrences $P_{OC^+}$</td>
<td>252,275</td>
<td>56,418</td>
<td>27,902</td>
<td>785</td>
<td>220</td>
<td>337,600</td>
</tr>
<tr>
<td>Number of occurrences $P_{UC}$</td>
<td>233,158</td>
<td>50,960</td>
<td>24,954</td>
<td>657</td>
<td>219</td>
<td>309,948</td>
</tr>
<tr>
<td>Number of occurrences $P_{OC^-}$</td>
<td>754</td>
<td>738,048</td>
<td>11,878</td>
<td>3,491</td>
<td>34</td>
<td>754,205</td>
</tr>
<tr>
<td>Occurrence probability $P_{OC^+}$ (%)</td>
<td>74.73</td>
<td>16.71</td>
<td>8.26</td>
<td>0.23</td>
<td>0.07</td>
<td>100</td>
</tr>
<tr>
<td>Occurrence probability $P_{UC}$ (%)</td>
<td>75.23</td>
<td>16.44</td>
<td>8.05</td>
<td>0.21</td>
<td>0.07</td>
<td>100</td>
</tr>
<tr>
<td>Occurrence probability $P_{OC^-}$ (%)</td>
<td>0.01</td>
<td>97.86</td>
<td>1.57</td>
<td>0.46</td>
<td>0.00</td>
<td>100</td>
</tr>
</tbody>
</table>

\[
\begin{array}{c|cccccc}
\text{Number of occurrences } & 0 & 3,825 & 14 & 2 & 1 & 3,842 \\
\text{Number of occurrences } & 0 & 3,978 & 420 & 542 & 220 & 3,883 \\
\text{Number of occurrences } & 0 & 97,284 & 15 & 16 & 34 & 97,349 \\
\text{Occurrence probability } & 0 & 1.133 & 0.005 & 0.001 & 0 & 1.138 \\
\text{Occurrence probability } & 0 & 0 & 0.015 & 0.001 & 0 & 0.016 \\
\text{Occurrence probability } & 0 & 12.899 & 0.002 & 0.002 & 0.005 & 12.907 \\
\end{array}
\]
CHAPTER 2. GOAL-ORIENTED ACTION PLANNING

Figure 2.5: Probability of plans occurrence as a function of their length; 0 means GOAP has failed to build a plan.

is the single action plan \{Attack\} while, for OC\(^+\), it is \{AttackFromCover\}. There is also a slightly higher difference in favor of OC\(^+\) for the other attack plans. To summarize, OC\(^+\) has been the configuration with the most diversity and consistency for the NPC’s behavior, but the differences with UC are very small unlike the differences with OC\(^-\). It highlights that the context preconditions have played a significant role in the action plan construction.

Concerning the plan \{AttackGrenadeFromCover\} in OC\(^-\), its occurrence probability exists and is high because the action AttackGrenadeFromCover has as a second postcondition: TargetIsFlushedOut = True (It is one of the very few actions having more than one post-condition). Which means the planner were able to solve the goal TargetIsFlushedOut = True with this action. Otherwise, for the goal TargetIsDead = True the only possible attack action plan for the OC\(^-\) configuration is \{AttackFromVehicle\}. Indeed, the attack action costs range from 1 (AttackFromVehicle) to 7 (Attack-Ready) with the original configuration and, as attack action plans are most of the time single action plans, the heuristic functions during the search have always been equal to 0 (meaning the action can be activated in the NPC’s current state), the only possible action plan to be returned is: \{AttackFromVehicle\}, the cheapest one.

Concerning the other possible action plans, they were no significant differences between...
CHAPTER 2. GOAL-ORIENTED ACTION PLANNING

Figure 2.6: Comparison of the occurrence probabilities of the attack action plans between OC\(^+\), UC and OC\(^-\).

the OC\(^+\) and the UC configurations (the greatest differences lie in the attack action plans). On the other hand, the differences between the original game and the game without the context preconditions have been more notable and were clearly visible in game. NPCs have had inconsistent behaviors and were contextually more irrelevant. Finally, context preconditions have a significant impact compared to the costs. The context preconditions shape the search space while the use of action cost only brings a slight diversity.

2.2.3 Construction Time

In 2005, a video game had a frame rate of 30 FPS, which entails the time budget for one frame is 0.033s. As the computing resources have to be shared with many different modules, such as animation, rendering, and physics systems, the allocated time budget for the AI module is no more than 10% of the total time budget, i.e. 3.3ms for a 30 FPS video game. Because GOAP has the constraint to plan in real-time, i.e. between two frames, it seems obvious that any kind of optimization can be useful so that its implementation does not hinder the fluidity of the game [7]: could the action costs help to achieve real-time planning?

As we can see on Figures 2.7 and 2.8, which represent the planning runtime of the three configurations (Figure 2.8 differentiates the length of the plans), the modification of the costs, from OC\(^+\) to UC, have barely influenced the runtime unlike the context preconditions. Therefore, adding context preconditions extend the planning runtime by a few tens of microseconds. It was good for the video game F.E.A.R., the context preconditions avoided unnecessary or unfeasible plans. In fact, returned plans with OC\(^-\) were often not feasible, so NPCs had to ask the planner again and again for new plans, which led to frame rate drops and odd behaviors from NPCs in game.
CHAPTER 2. GOAL-ORIENTED ACTION PLANNING

2.3 Discussion

Through this section, I discuss previous works and bring further explanations thanks to my results (section 2.2) on the impact of the context preconditions on GOAP’s plan construction.

2.3.1 The Context Precondition Impact on Planning

As explained in section 2.1.4, context preconditions have been designed to maintain light-weight planning [7]. They have also been used as a way to contextually keep or remove a candidate action without burdening the symbolic representation of STRIPS. Therefore, they have also been part of the planning process. There is no study yet that have examined, in practice, their impact on planning, however. I have highlighted that they have not just been part of the planning system: if they are overused, they can basically shape the whole planning activity by overtaking the use of the action costs. Figure 2.6 corroborates this observation: although the action costs have had a sensible impact on the attack action plans, the overall behavior of the search algorithm has not changed with the cost modification. In 2007, Edmund Long in its report [22] criticized the fact the action costs are static and that the behavior of the planner could be improved further with dynamic

Lastly, the allocated time budget (3.3 ms – 10% of a frame) has been far from being overused (0.040 ms on average), which means there is plenty of time left to create longer plans. In fact, the faced computational limits were related to memory usage [22], and not the time as explained by my results. Which means the GOAP system with these settings manages the real-time constraints.
costs\(^2\). I do believe on this point but with the use of context preconditions, and according to my results, dynamic cost may not modify the planner behavior that much if context preconditions are overused. Finally, my advice to restore the usefulness of action costs would have been to reduce the use of the context preconditions.

2.3.2 Planning Failures

As far as I know, planning failures in GOAP have never been mentioned. I thought it was important to mention them: they have appeared more than 74\% for both OC\(^+\) and UC configurations in my experiment (1). A planning failure occurs when the planner failed to link the goal state and the NPCs’ current state, so it returns an empty plan. There are two possible reasons for it: (i) when all actions are pruned by context preconditions or (ii) when (at least) one precondition cannot be satisfied. The first reason is easy to understand and, Figure 2.5 highlights that it is almost all the time due to context preconditions. The second reason can be highlighted in F.E.A.R. with the action AttackFromView. This attack has \textit{AtNodeType} = View as a precondition, which can only be satisfied when the NPC is on a node of the map labeled as View. So, either the NPC is already on such a node or a moving action such as GotoNodeOfType has to be performed. But the surroundings of an NPC can be without a View node due to map design, therefore in this situation, the precondition cannot be satisfied and the planner will fail to build a valid plan.

\(^2\)Crystal Dynamics in Rise of the Tomb Raider have used dynamic costs [4] [19].
2.4 Conclusions

I analyzed in-game planning data from playing F.E.A.R. to evaluate the use of action costs in GOAP in order to provide an answer to the important game AI development problem of designing these action costs when starting a new game. I found that the behaviors of NPCs are the result of context preconditions: action costs have a negligible impact on NPC behaviors. Therefore if your implementation of GOAP closely follows that of F.E.A.R., action costs can be removed from this implementation in favor of context preconditions; this comes at the cost of some CPU cycles, however, and procedural precondition run time should not be overused but closely monitored to satisfy the time budget for GOAP.

I also analyzed how the heuristic for the A* search algorithm has been designed and used in GOAP: I explained that it actually fosters short action plans instead of cheap action plans. Video games are facing the major drawbacks of planning, pointed by D. Chapman [23], it is intractable in general. Game industries try to contain the intractability by restricting the number of calls per NPCs and per frame [3,5,24] and by restricting the size of the solution plans [21]. Despite these restrictions, planners in GOAP remain search algorithms like A* [15] or Djikstra [25] whose complexity are exponential with respect to the size of the solution plan, which, in return, does not help to improve GOAP or real-time planning system.

Eventually, even though the context preconditions in F.E.A.R. decrease the runtime performance of the GOAP planner, I strongly believe that this is the way to go. In fact, among the answers I get from asking Lead AI: “Do you use context preconditions?”, are Troy Humphrey’s answer (Highmoon Studio) who told us that these kind of preconditions are a bad because developers will put too many instructions inside, including heavy function like pathfinding (This is the case in some actions of F.E.A.R.), even though they are told not to. The result is that these instructions can be run several times while planning because there is no dedicated variable for them. This is the second part of Troy Humphrey’s answer, the current form of the context preconditions is not optimal at all, they must be stored someway because sensing and planning are two different abstractions. Eventually, by considering that it exists a more efficient way to handle context preconditions and the fact that they overshadow the action costs in practice, I am convinced that action costs can be fully removed in favor of the context preconditions. The direct consequence will be that actions will become contextually post-unique.

I therefore made the following assumptions which I use in this thesis to produce tractable classes of problems and provide a correct and complete linear time algorithm to solve any instance of these classes.

**Proposition 2.4.1 (Working Assumptions).** The following assumptions are compatible with the creation of realistic behavior for NPCs:

1. Actions are unary
2. Actions are contextually post-unique
3. Solution plans are totally ordered
4. **There is not twice the same action inside a solution plan**

The last assumption can be surprising at first glance. In fact, the size of the action plans are rather small, e.g. 1 or 2 actions at most in F.E.A.R. And it is slightly higher in some other video games using GOAP. And in these small plans, in particular in F.E.A.R., there is not twice the same action.

In the next Chapter, I introduce the *Simplified Action Structure*, a formalism created by [6] whose approach to counter the general intractability of planning was to apply restrictions on the action representation in order to find tractable subclasses of problems, and then to provide polynomial time algorithms to solve any instance of these problems. Then, because the assumptions of Proposition 2.4.1 are restrictions that are already implemented by game developers, the idea of the thesis is to find a tractable class of problems out of these assumptions and to provide an algorithm that can solve this new class of problems.
The *Simplified Action Structure* (SAS) was introduced by C. Bäckström [6] in the early 90’s and it is a restricted version of the *Action Structure* formalism [26]. SAS is similar to the very well-known propositional formalism: Stanford Research Institute Problem Solver (STRIPS) [27], except that: (i) state variables are multi-valued and (ii) operators have two types of preconditions. (i) The multi-valuedness, which was said to be appealing for automatic control researchers, is also a feature of the planning system of GOAP (cf. Chapter 2). It is indeed more convenient than the add and delete lists of the STRIPS formalism, especially when using a programming language like C++ that allows the assignment of values. (ii) The two types of preconditions, which are latter described, are easily adaptable to the current representation of GOAP operators. The way these preconditions are described is algorithmically convenient because their description allows a simple definition of a *Truth Criterion* [23] which is a criterion useful to prove the correctness and the completeness of an algorithm. Eventually, GOAP operators are closer to those of SAS than those of STRIPS.

D. Chapman proved that planning is intractable in general [23] and, unfortunately, the GOAP planning system is not an exception (cf. Chapter 2). C. Bäckström’s approach to tackle the general intractability is to apply restrictions on SAS operators so as to create and study sub-classes of problems in the hope of finding usable and tractable ones [6,28]. Among these restrictions are the *(P)*ost-uniqueness and the *(U)*nariness [29] which directly concern 2 of my 4 assumptions (cf. Proposition 2.4.1). Without the addition of other restrictions, the class of problems SAS-PU is NP-Hard [30], however. The addition of the *(S)*ingle-valuedness and the *(B)*inariness restrictions to (PU) lead to SAS-PUBS and SAS+PUS which are both tractable classes and which can both be solved by the polynomial time algorithms described in C. Bäckström’s thesis [6].

In this Chapter, I recall how a planning problem is described with the SAS formalism and what is a minimal solution plan for a SAS problem instance. Then, I recall existing restrictions before focusing on SAS-PU and SAS-PUS, two consistent classes of problems for GOAP according to my assumptions.
CHAPTER 3. SIMPLIFIED ACTION STRUCTURE

3.1 SAS Planning Framework

In SAS, a planning problem is composed of a structure (Φ). Φ = (M, S, O) is a triplet made up of a set of multi-valued state variables (M), a set of states (S) and a set of SAS operators (O). The purpose of a SAS planner is to solve a planning problem instance (Π) by building an action plan (Δ). Π = (Φ, s₀, s⋆) is a triplet made up of a structure (Φ), an initial state (s₀) and a goal state (s⋆).

In this section, I first describe how to build a SAS structure, starting with the world state variables (M) and their associated domain. I then described how to design operators and what is the difference between an action and an operator. I then detail three different and existing SAS structures. Then, I explain what is an action plan and give further definitions useful to understand how a SAS problem instance can be solved.

3.1.1 World State Variables

In SAS, states of the world are described with state variables that can take several different values. A state is no more than a set of defined world state variables and it is used to describe an agent’s situation at any time, or some goals.

Definition 3.1.1 (State description and world modeling).

1. M = {v₁,...,vₘ} is the set of state variables, where vᵢ is the iᵗʰ state variable.

2. ℰvᵢ is the domain associated to vᵢ and ℰvᵢ⁺ = ℰvᵢ ∪ {u} is the extended domain, where ‘u' is the undefined value. S = {(x₁, x₂,...,xₙ) | ∀x ∈ ℰvᵢ} is the total-state space while S⁺ = {(x₁, x₂,...,xₙ) | ∀x ∈ ℰvᵢ⁺} is the partial-state space.

3. Let s ∈ S⁺ and vᵢ ∈ M, s[vᵢ] is the value of vᵢ in s.

4. s₀, s⋆ are resp. the start and the goal states of SAS problem instances¹:
   - s₀, s⋆ ∈ S, with the SAS-structure [6, p.61].
   - If s₀, s⋆ ∈ S⁺, with the SAS⁺-structure [6, p.51-52].
   - If s₀ ∈ S, s⋆ ∈ S⁺, with the SAS⁺-structure [28, p.42-43].

In order to illustrate how a video game world can be modeled with the SAS formalism, let introduce the NPCs of Figures 3.1 and 3.2. The first one represents the Horse Breeder’s world: he can interact with haystacks, a bucket, a horse feeder, a source of water and a horse trough. I thus added the following state variables: v₀ for a haystack, v₁ for the bucket, v₂ to state the transfer of a certain amount of water ; associated with the following domain:

¹The difference between structures SAS, SAS⁺ and SAS⁺ are recalled in subsection 3.1.3
Figure 3.1: Read Dead Redemption based NPCs: the Horse Breeder.

- \(v_0: \text{HayStack}, \mathcal{D}_{v_0} = \{0: \text{none}, 1: \text{inHands}, 2: \text{inFeeder}\}\).
- \(v_1: \text{Bucket}, \mathcal{D}_{v_1} = \{0: \text{none}, 1: \text{inHands}\}\).
- \(v_2: \text{Water}, \mathcal{D}_{v_2} = \{0: \text{inSource}, 1: \text{inBucket}, 2: \text{inTrough}\}\).

The second figure represents the Native hunter’s world, where he has a hunting routine. I created the following state variables:

- \(v_3: \text{HuntingRoutine}, \mathcal{D}_{v_3} = \{0: \text{notStarted}, 1: \text{onGoing}, 2: \text{done}\}\)
- \(v_4: \text{HuntingReadiness}, \mathcal{D}_{v_4} = \{0: \text{notReady}, 1: \text{ready}\}\)
- \(v_5: \text{Energy}, \mathcal{D}_{v_5} = \{0: \text{low}, 1: \text{rested}\}\)
- \(v_6: \text{Thirst}, \mathcal{D}_{v_6} = \{0: \text{thirsty}, 1: \text{hydrated}\}\)

The sets \(\mathcal{M}_{\text{HB}} = \{v_0, v_1, v_2\}\) and \(\mathcal{M}_{\text{NH}} = \{v_3, v_4, v_5, v_6\}\) are respectively the set of state variables of the Horse Breeder and of the Native Hunter. Each of these two sets models the world for the corresponding type of NPC, focusing on what one wants or allows them to do. Of course, state variables can be combined differently: \(\{v_0, v_1, v_2, v_5, v_6\}\) is a Horse Breeder that feels the thirst and tiredness; \(\{v_0, v_1, v_2, v_3, v_4, v_5, v_6\}\) is a Native Horse Breeder and Hunter.

A state is written \(s = \{v_i = x \mid \forall v_i \in \mathcal{M}, x \in \mathcal{D}_{v_i}\}\) and it describes a particular situation. In video games, it is safe to say that, at any time, the state of an NPC is fully known, which implies we will only use the SAS- or SAS*-structures [6,28]. Let gives some state examples with the Horse Breeder:

- \(s = \langle 0, 0, 0 \rangle\), the trough, the feeder and the bucket are empty, and the Horse Breeder carries nothing.
- \(s = \langle 2, 0, 2 \rangle\), the trough and the feeder are filled and the bucket is dropped.
• \( s = (1, 1, 1) \), the Horse Breeder is carrying a haystack and a bucket full of water.

• \( s = (2, u, u) \), the feeder is filled and we do not know the state of neither the bucket nor the water travel.

I mainly use this state representation to define the current state of NPCs \( (s_0) \) and their goals \( (s_\star) \). The NPC’s current state \( s_0 \) is updated by its sensors (ears, eyes, memory, collision boxes etc.) and by the successes of its action executions. Sensors update the current state in real-time while actions update it when their execution is over and succeeded. In case of action failure, the concerned state variables are obviously not modified by the action. The goals represent what NPCs want to accomplish according to their will, their belief, the context, etc. The goal determination, which is the Goal-Oriented system role in GOAP, is not studied in this thesis. I will define some of them in Chapter 5, however, where experiments and results are given.

### 3.1.2 Operators and Actions

Once the world is modeled, NPCs need operators to plan and actions to act in its environment. In fact, operators are instantiated by a SAS planner to add actions in the solution plan. Hence, an action is an instance of an operator. In other words, a SAS planner has operators as input and it return action plans as output. An operator can be instantiated as many times as needed by the planner, it results that a solution plan can have the exact same action several times.

**Definition 3.1.2.** An action is an instance of an operator, I use the following notations throughout the entire manuscript:

- The operator set is denoted: \( \mathcal{O} \). An operator of \( \mathcal{O} \) is denoted: \( o \).
CHAPTER 3. SIMPLIFIED ACTION STRUCTURE

Table 3.1: Horse Breeder operator set. His goal is to feed the horses: $s^* = \langle 2, 0, 2 \rangle$.

<table>
<thead>
<tr>
<th>$O$</th>
<th>Pre</th>
<th>Post</th>
<th>$\langle v_0, v_1, v_2 \rangle$</th>
<th>Name of the operators/actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$o_0$</td>
<td>$\langle 1, u, u \rangle$</td>
<td>$\langle 0, u, u \rangle$</td>
<td>$\langle u, u, u \rangle$</td>
<td>DropHaystack</td>
</tr>
<tr>
<td>$o_1$</td>
<td>$\langle 0, u, u \rangle$</td>
<td>$\langle 1, u, u \rangle$</td>
<td>$\langle u, 0, u \rangle$</td>
<td>TakeHaystack</td>
</tr>
<tr>
<td>$o_2$</td>
<td>$\langle 1, u, u \rangle$</td>
<td>$\langle 2, u, u \rangle$</td>
<td>$\langle u, u, u \rangle$</td>
<td>FillHorseFeeder</td>
</tr>
<tr>
<td>$o_3$</td>
<td>$\langle u, 1, u \rangle$</td>
<td>$\langle u, 0, u \rangle$</td>
<td>$\langle u, u, u \rangle$</td>
<td>DropBucket</td>
</tr>
<tr>
<td>$o_4$</td>
<td>$\langle u, 0, u \rangle$</td>
<td>$\langle u, 1, u \rangle$</td>
<td>$\langle 0, u, u \rangle$</td>
<td>PickUpBucket</td>
</tr>
<tr>
<td>$o_5$</td>
<td>$\langle u, u, 0 \rangle$</td>
<td>$\langle u, u, 1 \rangle$</td>
<td>$\langle u, 1, u \rangle$</td>
<td>FillBucketWithWater</td>
</tr>
<tr>
<td>$o_6$</td>
<td>$\langle u, u, 1 \rangle$</td>
<td>$\langle u, u, 2 \rangle$</td>
<td>$\langle u, 1, u \rangle$</td>
<td>FillHorseTrough</td>
</tr>
</tbody>
</table>

$v_0 : HayStack, \mathcal{D}_{v_0} = \{ 0 : \text{none}, 1 : \text{inHands}, 2 : \text{inFeeder} \}$.

$v_1 : Bucket, \mathcal{D}_{v_1} = \{ 0 : \text{none}, 1 : \text{inHands} \}$.

$v_2 : Water, \mathcal{D}_{v_2} = \{ 0 : \text{inSource}, 1 : \text{inBucket}, 2 : \text{inTrough} \}$.

- The action set is denoted: $A$. An action of $A$ is denoted: $a$.

- $\text{ope}() : A \rightarrow O$ is a function s.t. $a \in A, o \in O$, $\text{ope}(a) = o$ means $a$ is an instance of $o$.

In C. Bäckström’s thesis, an operator is an action type [6, p.48]. I have decided to use the more usual term, operator, which is also what C. Bäckström did with P. Jonsson in [28]. I also believe the term operator to be less confusing than the term action type.

Traditionally in planning, operators are described with preconditions and postconditions. With the SAS formalism, however, operators are represented with three different conditions: the pre-conditions, the post-conditions and the prevail-conditions. Actions being operator instances, they inherit this representation. The pre- and prevail-conditions are both preconditions. They both describe what must be satisfied before an action execution. The difference is that prevail-conditions hold values that will not be modified, whereas pre-conditions describe what will be modified. The pre-conditions are in fact linked with the post-conditions as the post-conditions describe the new values of the state variables affected by the action. Each of the three conditions define a partial-state or a total-state.

**Definition 3.1.3 (SAS Operators).** Let $o \in O$, $\text{post}(o)$, $\text{pre}(o)$ and $\text{prv}(o)$ are functions that respectively return the post-, pre- and prevail-conditions of the operator $o$. These functions are defined as follows:

$$\text{pre}(), \text{post}(), \text{prv}() : O \rightarrow S^+$$

**Remark:** Let $v_i \in M$, $\text{pre}(o)[v_i]$, $\text{post}(o)[v_i]$, $\text{prv}(o)[v_i]$ are the value of $v_i$ in the pre-, post- and prevail-conditions respectively (cf. 3. Def 3.1.1).
CHAPTER 3. SIMPLIFIED ACTION STRUCTURE

Let illustrates the three types of conditions in SAS with the Horse Breeder whose operators are described in Table 3.1. Let consider in particular the FillBucketWithWater operator, its execution will modify the Water value from 0 to 1, i.e. a certain amount of water from the source (fountain) (0) is transferred in the bucket (1). Therefore, \((v_0 = 0)\) is a pre-condition of FillBucketWithWater and \((v_0 = 1)\) is a post-condition. To perform this action the NPC must have the bucket in his hands, and it must remain in it during the whole action execution. Hence, \((v_1 = 1)\) is a prevail-condition of FillBucketWithWater.

3.1.3 SAS Planning Structures

Any SAS problem is composed of a structure, denoted \(\Phi\). A structure \(\Phi = (\mathcal{M}, \mathcal{S}, \mathcal{O})\) is a triplet composed of a set of state variables \(\mathcal{M}\), a set of states \(\mathcal{S}\) and a set of operators \(\mathcal{O}\). There exists three different structures, respectively denoted: SAS, SAS* and SAS+. These three structures all have the additional restrictions S1, S3, S4 and S5, but they differ with the restrictions S2, S6, S7 and S7\((s_0)\). With \(\dim(s) = \{s[v_i] \neq u \mid v_i \in \mathcal{M}\}\) being the set of defined state variables in \(s \in \mathcal{S}^+\), the restriction S1 to S7\((s_0)\) are described as followed:

- S1: For all \(o \in \mathcal{O}\), \(\text{pre}(o), \text{post}(o), \text{prv}(o)\) are consistent.
- S2: For all \(o \in \mathcal{O}\), \(\dim(\text{pre}(o)) \subseteq \dim(\text{post}(o))\).
- S3: For all \(o \in \mathcal{O}\), \(\dim(\text{post}(o))\) and \(\dim(\text{prv}(o))\) are disjoint.
- S4: For all \(o \in \mathcal{O}\) and for all \(v_i \in \mathcal{M}\), if \(\text{pre}(o)[v_i] \neq u\), then \(\text{pre}(o)[v_i] \neq \text{post}(o)[v_i]\).
- S5: For all \(o, o' \in \mathcal{O}\), if \(\text{post}(o) = \text{post}(o')\) and \((\text{pre}(o) \sqcup \text{prv}(o)) \subseteq (\text{pre}(o') \sqcup \text{prv}(o'))\), then \(o = o'\).
- S6: For all \(o \in \mathcal{O}\), \(\dim(\text{pre}(o)) = \dim(\text{post}(o))\).
- S7: \(s_0, s_* \in \mathcal{S}\).
- S7\((s_0)\): \(s_0 \in \mathcal{S}\).

Restrictions S1 to S5, and S6, S7, were introduced in [6, p.52, p.61]. I denote S7\((s_0)\) the restriction 1. of Definition 7.1 [28, p.42,43] as it is restriction S7 but only for \(s_0\). S7 and S7\((s_0)\) could be rewritten differently but I wanted to keep the notation S7 for the sake of overall consistency, so that the readability between researches is consistent.

The restrictions S1, S3, S4 and S5 are applied to the three structures (SAS, SAS* and SAS+) and they can be seen as “intuitive” restrictions. S1 is meaningless in this manuscript because it is about contradictory values \(k\) which we do not use in video games\(^3\). Therefore, S1 will always be respected in the following. S3 concerns the definitions of the post-conditions and the prevail-conditions. The post-conditions describe what will change after

\(^2\)The operators of the Native Hunter are given in Table F.7, Appendix F.

\(^3\)The reader is invited to read C.Bäckström thesis, Chapter 3, for more details on contradictory values.
the action execution while the prevail-conditions describe what must not change. So any
defined state variable in the post-condition cannot be defined in the prevail-condition, and
vice versa. A state variable cannot change its value while being asked to remain constant.
S4 means that for each \( v_i \in \mathcal{M} \), if an operator \( o \in \mathcal{O} \) has its pre-condition defined on \( v_i \), its
post-condition on \( v_i \) cannot have the same defined value. An action cannot change a value
from a state variable to the same value, if the value of a precondition must not change
during the execution of an action, it is merely a prevail-condition. S5 is about operators
having the same post-conditions. If two operators have the same post-conditions, then the
pre- and prevail-conditions of one of the two must not be subsumed to the other one. If
S5 is respected, then the structure is said to be parsimonious. As I’m interested in post-
unique operators, S5 will always be respected in my research because post-unique means
that there are no two operators with the same post-conditions. Finally, and additionally to
S1, S3, S4 and S5, applying the restrictions S2, S6, S7 or S7\((s_0)\) highlight which structure
is being used:

- SAS uses the additional restrictions S6 and S7.
- SAS\(^*\) uses the additional restrictions S6 and S7\((s_0)\).
- SAS\(^+\) uses the additional restriction S2.

What differs between the different structures is the definition of the start and the goal
states and their interaction with them. The SAS\(^+\)-structure is the more general one. It
allows the start and the goal to be partial-states. Hence, the restriction S2 in SAS\(^+\) allows
some operators to define a previously undefined state variable. That is for a given state
variable, a pre-condition of an operator can be undefined and its post-condition defined.
The SAS\(^*\)-structure is a bit more restrictive as the start state must be totally defined. The
SAS-structure, finally, is even more restrictive as the goal state must be totally defined as
well. As a consequence for SAS and SAS\(^*\), the restriction S2 is no longer relevant and it
is replaced with the restriction S6. The latter says that, if the post-conditions are defined
for some state variables, then these state variables must be defined in the pre-conditions
as well. Eventually, restrictions S2 and S6 both respect: \( \dim(\text{pre}(o)) \subseteq \dim(\text{post}(o)) \),
which means that, in each structure, a post-condition cannot be undefined for a given
state variable if it is defined in the pre-conditions. That is, an operator cannot “undefine”
a defined state variable.

In order to evaluate partial-states in both SAS\(^+\) and SAS\(^*\), the partial order \( \sqsubseteq \) was
defined as followed [6]:

**Definition 3.1.4** (Partial order \( \sqsubseteq \)).

- \( \sqsubseteq \) is a reflexive partial order on \( \mathcal{D}^+_v \) defined as:
  \[ \forall p, q \in \mathcal{D}^+_v, p \sqsubseteq q \iff p = q \text{ or } p = u. \]

- \( \sqsubseteq \) is a reflexive partial order over \( \mathcal{S}^+ \) defined as:
  \[ \forall s, s' \in \mathcal{S}^+, (s \sqsubseteq s' \iff (\forall v_i \in \mathcal{M}, s[v_i] \sqsubseteq s'[v_i])). \]
Let $v_i \in \mathcal{M}$ and $s, s' \in \mathcal{S}^+$, the partial order $\sqsubseteq$ allows to compare a defined state variable $s[v_i] \neq u$ with a possibly undefined one $s'[v_i]$. ($s[v_i] \sqsubseteq s'[v_i]$) is equivalent to $(s[v_i] = s'[v_i])$, and the answer remains true if $s'[v_i]$ is undefined. For instance, let $v_i \in \mathcal{M}$, $p \in D_{v_i}$ and $s, s', s'' \in \mathcal{S}^+$ three possible states of an intelligent agent, if $s[v_i] = s'[v_i] = p$ and $s''[v_i] = u$, then $(s[v_i] \sqsubseteq s'[v_i])$ and $(s[v_i] \sqsubseteq s''[v_i])$ are both true.

### 3.1.4 SAS Planning Problem Instances

An instance of a SAS planning problem $\Pi = (\Phi, s_0, s_*)$ is composed of a structure $\Phi$, an initial state $s_0 \in \mathcal{S}$ and a goal state $s_* \in \mathcal{S}$. $\Pi$ is then solved by an action plan ($\Delta$).

There is no cost function in SAS, so there is no “optimal” solution plan. There is minimal solution plan, however, and they are defined as follows:

**Definition 3.1.5 (Minimal Solution Plan).** Let $\prec$ be a strict partial-order on a set of actions. $\Delta = (A, \prec_A)$ is a minimal solution plan solving $\Pi$ if there is no other plan $\Delta' = (B, \prec_B)$ solving $\Pi$ such that $|B| \leq |A|$.

In other words, a minimal solution plan solving $\Pi$ is a solution plan with the minimum number of actions.

### 3.2 A Tractable Class of Problems: SAS-PUS

The SAS formalism is as expressive as Ground TWEAK, a propositional version of TWEAK, according to Corollary 28 [31], which implies unfortunately that, in general, SAS planning is intractable as well [23]. The approach of C. Bäckström to bypass this general intractability was to apply restrictions on SAS problems in order to create tractable classes of problems. Among those tractable classes of problems is SAS-PUS. It has three restrictions (P), (U) and (S), whose definitions is reminded below. Two of them, (P) and (U), which respectively stand for Post-unique and Unary, fit two of my assumptions from the conclusion of my study of GOAP (cf. Chapter 2). The following is organized as follows, I first remind the syntactical restrictions (P), (U), (B) and (S) and give useful existing definitions that I use in this manuscript. I then explain why I cannot use the class of problems SAS-PU for my research and give some results with the class SAS-PUS after reminding the theory behind.

### 3.2.1 Syntactical Restrictions

The restrictions Post-unicness (P), Unariness (U), Binary (B), Single-Valuedness (S) are syntactical [6]. They give rules on how to design operators (P, U, S), or on how many values a single state variable can have (B). They are defined as follows:

**Definition 3.2.1 (P,U,B ans S).** Let $\Phi = (\mathcal{M}, \mathcal{S}, \mathcal{O})$ a SAS structure, $\Phi$ is:
CHAPTER 3. SIMPLIFIED ACTION STRUCTURE

![Figure 3.3: CubeWorld with 2 cubes.](image)

(a) The initial state $s_0$ (b) The goal state $s_*$

$\begin{align*}
\text{(P): Post-Unique } & \iff \text{no two operators have the same post-condition. For all } o, o' \in O, \text{ if } \text{post}(o)[v_i] = \text{post}(o')[v_i] \neq u \text{ for some } v_i \in M \text{ then } o = o'. \\
\text{(U): Unary } & \iff \text{all operators of } O \text{ is unary, i.e. they all have a single post-condition: } o \in O, \text{ if } \text{post}(o)[v_i] \neq u \text{ then } \text{post}(o)[v_j] = u \text{ for all } v_j \in M \setminus \{v_i\}. \\
\text{(B): Binary } & \iff \text{all state variables have only two values: } \forall v_i \in M, |D_{v_i}| = 2. \\
\text{(S): Single-Valued } & \iff \text{for some } v_i \in M, \text{ if it exists } o, o' \in O \text{ s.t. } \text{prv}(o)[v_i] \neq u \land \text{prv}(o')[v_i] \neq u, \text{ then } \text{prv}(o)[v_i] = \text{prv}(o')[v_i].
\end{align*}$

The single-valuedness means that it does not exist two operators with a defined prevail-condition on the same state variable that is not equal. Let consider the following SAS framework for the BlockWorld with 2 cubes: $M = \{AonB, BonA\}$, with $D_{AonB} = D_{BonA} = \{F, T\}$, i.e. each state variable only has two values: True or False. If $AonB = F$ (resp. $BonA = F$), then it means $A$ (resp. $B$) is on the table. We have the following operator set:

<table>
<thead>
<tr>
<th>$O$</th>
<th>pre</th>
<th>post</th>
<th>prv</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$o_0$</td>
<td>$\langle T, u \rangle$</td>
<td>$\langle F, u \rangle$</td>
<td>$\langle u, F \rangle$</td>
<td>A from B to Table</td>
</tr>
<tr>
<td>$o_1$</td>
<td>$\langle F, u \rangle$</td>
<td>$\langle T, u \rangle$</td>
<td>$\langle u, F \rangle$</td>
<td>A from Table to B</td>
</tr>
<tr>
<td>$o_2$</td>
<td>$\langle u, T \rangle$</td>
<td>$\langle u, F \rangle$</td>
<td>$\langle F, u \rangle$</td>
<td>B from A to Table</td>
</tr>
<tr>
<td>$o_3$</td>
<td>$\langle u, F \rangle$</td>
<td>$\langle u, T \rangle$</td>
<td>$\langle F, u \rangle$</td>
<td>B from Table to A</td>
</tr>
</tbody>
</table>

Table 3.2: BlockWorld operator set.

The BlockWorld problem is (P), (U), (B) and (S). It is (P) because no two operators affects the same post-condition, (U) because each operator only has one defined post-condition. It is (B) because $|D_{AonB}| = |D_{BonA}| = 2$, i.e. the domains of $AonB$ and $BonA$ are binary. Finally, it is (S) because, for each $v_i \in M$, for all $o, o' \in O$ st. $\text{prv}(o)[v_i] \neq u$, $\text{prv}(o)[v_i] \sqsubseteq \text{prv}(o')[v_i]$. The problem would not have been (S) if, for example, $\text{prv}(o_0)[v_1] = F \land \text{prv}(o_1)[v_1] = T$. That is, a problem is not (S) if there exists at least 2 operators with a defined previal-condition on the same state variable that is different.

C. Bäckström and P. Jonsson in [28] defined a useful set of values, called the requestable values, which is a set that stores, for each $v_i \in M$ and for unary operators, the values of $D_{v_i}$ that are used as a prevail-condition on $v_i$ for some operators. Its definition is recalled below:
CHAPTER 3. SIMPLIFIED ACTION STRUCTURE

Definition 3.2.2. For each \( v_i \in M \), the set \( R_{vi}^O \) of requestable values for unary operators is defined as \( R_{vi}^O = \{ \text{prv}(o)[v_i] \mid o \in O \} \setminus \{ u \} \).

To illustrate this set, let consider the Horse Breeder (Table 3.1) and the BlockWorld (Table 3.2) problems:

- **Horse Breeder**: \( R_{v_0}^O = \{ 0 \} \), \( R_{v_1}^O = \{ 0, 1 \} \), \( R_{v_0}^O = \emptyset \).
- **BlockWorld**: \( R_{v_0}^O = R_{v_1}^O = \{ F \} \).

The requestable values are also useful to define the restriction (S). Indeed, a SAS problem is (S) iff \( |R_{vi}^O| \leq 1 \) for each \( v_i \in M \). I use this set of values later in the manuscript.

### 3.2.2 SAS-PU and SAS-PUS

My approach to reduce the complexity of actual planning systems in video games is to restrict the planning domain, i.e. the input instead of the output. With respect to the assumptions made at the end of Chapter 2, the class of problems SAS-PU is the best match. Indeed, I made the assumptions that operators are (P) and (U) in commercial video games, and because the initial state of an NPC is always known, i.e. totally defined at each frame, the SAS- and SAS*-structures are more relevant than SAS+. SAS* authorizes uncertainty in the goal state, I therefore decided to focus on SAS first because SAS* can inherit the result of SAS while the reciprocal is false. Hence, the interest towards SAS-PU. But the class has been proven intractable (Theorem 6.14 [6]), unfortunately, there is therefore a necessity to add at least another restriction in order to have a tractable class of problems. This is feasible because, by adding the restriction (S), C. Bäckström proved that the resulting class of problems SAS-PUS is tractable. In addition, he has provided several versions of a correct and complete polynomial-time algorithm solving SAS-PUS problem instances. His algorithm has been proven correct and complete thanks to a truth criterion called IMP for Intended Minimal Plans. This truth criterion is based on the notion of chain of actions that is recalled below.

Definition 3.2.3 (Chain of actions “chain\(v_i\)”). Given a SAS-structure \( \Phi = (M, S, O) \), two states \( s, t \in S \) and \( v_i \in M \), \( \sigma = \text{chain}_{v_i}(s[v_i], t[v_i]) = (a_1, ..., a_n) \) is a chain of actions over \( \Phi \) from \( s[v_i] \) to \( t[v_i] \) iff either:

1. a) \( a_k \) affects \( v_i \) for \( k \in [1; n] \)
   b) \( \text{pre}(a_1)[v_i] = s[v_i] \)
   c) \( \text{post}(a_n)[v_i] = t[v_i] \)
   d) \( \text{post}(a_k)[v_i] = \text{pre}(a_{k+1})[v_i], \text{ for } k \in [1; n[ \)
   or,
   2. \( \sigma \) is the empty chain \( \text{chain}_{v_i} = \langle \rangle \) and \( s[v_i] = t[v_i] \).
Definition 3.2.4 (Minimal chain of actions “\(\delta_{v_i}\))”. For each \(v_i \in M, x, y \in \mathcal{D}_{v_i}, x \neq y\), I denote \(\delta_{v_i}(x, y)\) the minimal chain from \(x\) to \(y\). \(\delta_{v_i}(x, y)\) is a minimal chain of actions from \(x\) to \(y\) iff there is no other chain’\(_{v_i}(x, y)\) such that: \(|\text{chain’}_{v_i}(x, y)| < |\delta_{v_i}(x, y)|\).

For SAS-PU problems, SAS-PUS ones included, a chain of actions is therefore a linear sequence of actions that focus on the relation between the post-condition and the pre-condition for a given state variable. Then, minimal chains of actions have two interesting properties: there is not twice the same action in a minimal chain and it is unique.

Lemma 3.2.1. Let \(x, y \in \mathcal{D}_{v_i}, \delta_{v_i}(x, y)\) does not have twice the same actions.

Proof. Let \(v_i \in M, \mathcal{O}(v_i)\) a set of (PU) operators affecting \(v_i\) and \(\mathcal{A}(v_i)\) a set of instances of the operators of \(\mathcal{O}(v_i)\). \(\mathcal{A}(v_i)\) is therefore a set of actions affecting \(v_i\).

**Base case:** Let \(\delta^1_{v_i} \in \mathcal{A}(v_i)\) such that \(\text{pre}(\delta^1_{v_i}) = 0\) and \(\delta^1_{v_i}\) is an instance of \(\delta^1_{v_i}\). \(\delta_{v_i}(0, 1) = \langle a^1_{v_i} \rangle\) is the minimal chain and only has one action.

**Induction hypothesis:** Let \(k \in \mathcal{D}_{v_i}, \{a^1_{v_i}, ..., a^k_{v_i}\} \subseteq \mathcal{A}(v_i)\) such that \(\delta_{v_i}(0, k) = \langle a^1_{v_i}, ..., a^k_{v_i} \rangle\) is the minimal chain from \(0\) to \(k\) and there is not twice the same action in it, i.e. for each action \(a\) in \(\delta_{v_i}(0, k)\), \(a\) is the only instance of \(\text{ope}(a) \in \mathcal{O}(v_i)\). \(\delta_{v_i}(0, l)\) is the minimal chain from \(0\) to \(l\) and \(\exists c \in \delta_{v_i}(0, l)\) such that \(\text{ope}(c) = \text{ope}(b)\). This concludes the proof.

Lemma 3.2.2 (Minimal chain uniqueness). Let \(v_i \in M, x, y \in \mathcal{D}_{v_i}\) such that \(x \neq y\). \(\delta_{v_i}(x, y)\) is the unique minimal chain between \(x\) and \(y\).

Proof. Let \(v_i \in M, x, y \in \mathcal{D}_{v_i}\) and \(\delta_{v_i}(x, y)\) and \(\delta'_{v_i}(x, y)\) two minimal chains between \(x\) and \(y\). We have \(|\delta_{v_i}(x, y)| = |\delta'_{v_i}(x, y)| = n\). **Lemma 3.2.1** and post-uniqueness imply that: \(\forall k \in [1, n], \text{ope}(\delta_{v_i}(x, y)[k]) = \text{ope}(\delta'_{v_i}(x, y)[k])\). Therefore, \(\delta_{v_i}(x, y) \equiv \delta'_{v_i}(x, y)\) and it concludes the proof.

C. Bäckström relied on chains of actions to define the IMP. Given a SAS planning problem \(\Pi = (\Phi, b_0, b_*)\), the IMP says that: a minimal solution plan \(\Delta = (A, \prec)\) is composed of chains such that, for each \(v_i \in M\), chain\(_{v_i}(s_0[v_i], s_*[v_i])\) link \(s_0[v_i]\) to \(s_*[v_i]\) ; Then, for each \(v_i \in M\), for each \(a \in A\), if \(\text{pre}(a)[v_i] = q \in \mathcal{R}_v^{\mathcal{O}}\), then there exists \(\delta_{v_i}\) from \(s_0[v_i]\) to \(q\) and from \(q\) to \(s_*[v_i]\) such that \(\delta_{v_i}(s_0[v_i], q) \prec a\) and \(a \prec \delta_{v_i}(q, s_0[v_i]\)) ; Finally, \(A\) is minimal wrt. the above and \(\prec\) is a minimal strict partial order on \(A\) satisfying the above.

The notion of truth criterion was developed by D. Chapman [23, p.340] with his modal truth criterion (MTC). He proves that plans from TWEAK, his algorithm, are correct if and only if the MTC is respected. The MTC describes, for a given planning problem instance, how actions must at least be ordered to ensure a correct non-linear plan. To this
end, Chapman orders two types of action, namely the establisher $T$ and the clobberer $C$, in relation to a situation $s$ where a proposition $p$ has to be asserted. The establisher is the action that assert $p$ to $s$, while the clobberer is an action that threatens (or clobbers) the assertion of $p$. In other words, a clobberer is an action that remove $p$ from $s$. While the establisher must obviously be ordered before the situation $s$, Chapman suggested three possibilities for deblobbering: (1) Promotion: it consists in ordering the clobberer after $s$; (2) Demotion with a White Knight: it consists in finding another establishing situation after the clobberer, called the white knight, and where $p$ has to be asserted; (3) Separation: it consists in avoiding the codesignation between the clobberer and the establisher. Based on this work, some other researchers [32,33] described that the situation $s$ is in fact related to the precondition of an action, so the definition of a truth criterion can be more action-focused by considering an action to assert instead of a situation. Then, É. Jacopin proved the “uselessness” of white knights [34]: a clobberer must in fact not be placed in between the establisher and the action to assert, it results that a clobberer has to be ordered before the establisher (Demotion) or after the action to assert (Promotion). Figure 3.4 presents a graphic parsing of PWEAK truth criterion, with PWEAK being É. Jacopin’s planning algorithm: $T$ is the establisher, $C$ is the clobberer and $A$ is the action to assert.

Figure 3.4: A graphic parsing of PWEAK truth criterion.

IMP is not without similarities to what is described above. First of all, like D. Chapman’s MTC, the IMP was used by C. Bäckström as a specification for plan generation algorithm. Then, in SAS, the action to assert, the establisher and the clobberer can be defined as follows:

**Definition 3.2.5.** Let $\Phi = (\mathcal{M}, S, \mathcal{O})$ a SAS-PU structure and $\mathcal{A}$ the set of actions that are instances of the operators of $\mathcal{O}$. I denote $a \in \mathcal{A}$ the action to assert with a proposition $p$, $T \in \mathcal{A}$ the establisher and $C \in \mathcal{A}$ the clobberer. They are defined as follows:

1. The assertion of $p$ either concerns the pre-condition of $a$: $\text{pre}(a)[v_i] = p$, or its prevail-condition on $v_i$: $\text{prv}(a)[v_i] = p$.

2. The action $T$ establishes $p$ iff $\text{post}(T)[v_i] = p$. 
CHAPTER 3. SIMPLIFIED ACTION STRUCTURE

3. The action \( C \) clobbers \( p \) iff \( \text{pre}(C)[v_i] = p \).

Even though there are two types of preconditions in SAS: the pre- and prevail-conditions, it is just a case distinction in the truth criterion, i.e. the assertion of \( p \) concerns one or the other. The distinction between the pre- and prevail-condition in SAS is besides very convenient for a truth criterion on (PU) operators, especially thanks to the definition of the pre-condition which allows to define clobberers very easily. Indeed, by definition, the pre-condition of a (PU) operator in a SAS-structure is precisely the value of a state variable that will be changed by the instance of it, i.e. the action. In other words, each action clobbers what is defined in their pre-condition. Therefore, a clobberer in a SAS-PU problem is an action whose pre-condition is equal to the pre- or one of the prevail-conditions of another action. If a clobberer \( C \) clobbers the pre-condition of \( a \), this is a clobbering situation that occurs in a chain of actions. But, thanks to post-uniqueness and unariness, there is one possible ordering in a chain of actions, i.e. there is no need for declorning because actions are chained based on their post-condition and not based on their pre-condition. On the contrary, each defined prevail-condition of an action \( a \in \mathcal{A} \) may have an establishing situation and a clobbering situation to handle. This was taken into account by the IMP with the two minimal chains \( \delta(v)(s_0[v_i], q) \) and \( \delta(v)(q, s_∗[v_i]) \): for each \( v_i \in \mathcal{M} \), \( q \in \mathcal{D}_{v_i} \) such that \( \text{prv}(a)[v_i] = q \), \( q \) is the proposition to assert, hence the establisher of \( a \) is the minimal chain \( \delta(v)(s_0[v_i], q) \) and the clobberer of \( a \) is the minimal chain \( \delta(v)(q, s_∗[v_i]) \). More precisely with Definition 3.2.5, the establisher is the last action of \( \delta(v)(s_0[v_i], q) \) and the clobberer is the first action of \( \delta(v)(q, s_∗[v_i]) \). In the IMP, the declorning technique only consists in promoting \( a \) before the clobberer: \( a \prec C = \text{First}(\delta(v)(q, s_∗[v_i])) \). The reason the IMP only requires the promotion technique is due to \( \mathcal{S} \). For each \( v_i \in \mathcal{M} \), we have \( |\mathcal{R}_{v_i}^\mathcal{O}| \leq 1 \) due to \( \mathcal{S} \), let \( q \in \mathcal{R}_{v_i}^\mathcal{O} \) be the single-value of \( \mathcal{D}_{v_i} \), Theorem 4.4 [6, p.76] indicates that a minimal solution plan \( \Delta \) for a SAS\(^+\)-PUS planning problem can have at most two instances of the same operator, but it can be highlighted that the establisher of \( q \) in \( \Delta \) is never concerned by the double instantiation, i.e. there is a unique establisher of \( q \) in every minimal solution plan of a SAS-PUS problem. Therefore, it only exists one situation where the proposition \( q = \text{prv}(a)[v_i] \) is asserted. In a minimal solution plan of an SAS\(^+\)-PUS problem, the clobberer must therefore be ordered after the action \( a \) to assert because once \( q \) has been clobbered, it does not exist another situation where \( q \) is asserted. It explains why, due to \( \mathcal{S} \), the promotion technique is sufficient to declorney in C. Bäckström’s IMP.

Eventually, like PWEAK truth criterion, the IMP does not use the separation technique. É. Jacopin [33] removed the separation technique of D. Chapman’s MTC for PWEAK by considering that an action cannot be both an establisher and a clobberer. Due to the SAS structure restriction (S4), which says that an operator cannot have its post-condition equal to its pre-condition, an operator cannot both establish and clobber a proposition, so the separation technique is also not needed in the IMP.

The idea of the IMP is that: for each \( v_i \in \mathcal{M} \), the single-value \( q \in \mathcal{R}_{v_i}^\mathcal{O} \) is a unique subgoal on \( v_i \) and, therefore, it has to be reached by the chain of actions on \( v_i \) between the start \( (s_0[v_i]) \) and the goal \( (s_∗[v_i]) \). Let denote \( T \) the establisher of \( q \) \( (\text{post}(T)[v_i] = q) \) and \( C \) the clobber \( (\text{pre}(C)[v_i] = q) \), the chain of actions chain\(_{v_i}(s_0[v_i], s_∗[v_i]) \) has two possible
forms:
If \( T \in \delta_v(s_0[v_i], s_*[v_i]) \), then:

\[
\text{chain}_{v_i}(s_0[v_i], s_*[v_i]) = \delta_{v_i}(s_0[v_i], s_*[v_i])
\]  

(3.1)

If \( T \notin \delta_v(s_0[v_i], s_*[v_i]) \), then \( T \in \delta_v(s_0[v_i], s_0[v_i]) \neq \langle \rangle \) such that:

\[
\text{chain}_{v_i}(s_0[v_i], s_*[v_i]) = \delta_{v_i}(s_0[v_i], q) \oplus \delta_{v_i}(q, s_*[v_i])
\]

\[
= \delta_{v_i}(s_0[v_i], q) \oplus \delta_{v_i}(q, s_0[v_i]) \oplus \delta_{v_i}(s_0[v_i], s_*[v_i])
\]

(3.2)

In both equations, there is the minimal chain of actions \( \delta_{v_i}(s_0[v_i], s_*[v_i]) \) (possibly empty) and then, depending on whether \( q \) is established by the minimal chain between \( s_0[v_i] \) and \( s_*[v_i] \), case 3.1 or case 3.2 should be considered. This is the planning strategy of C. Bäckström’s algorithm so as to find the minimal solution plan \( \Delta = (\mathcal{A}, \mathcal{R}) \), with \( \mathcal{R} \) the set of orders, while respecting the IMP:

Phase 1: For each \( v_i \in \mathcal{M} \), build the minimal chain \( \delta_{v_i}(s_0[v_i], s_*[v_i]) \). We have \( \text{chain}_{v_i}(s_0[v_i], s_*[v_i]) = \delta_{v_i}(s_0[v_i], s_*[v_i]) \). Actions of these chains are added to \( \mathcal{A} \) and their relations between their pre- and post-conditions are added to \( \mathcal{R} \).

Phase 2: For each \( a \in \mathcal{A} \), for each \( v_i \in \mathcal{M} \), (i) if \( \text{prv}(a)[v_i] = q \neq u \) and \( T, C \in \text{chain}_{v_i}(s_0[v_i], s_*[v_i]) \), then \( a \) is ordered after \( T \) and before \( C \). If \( q = s_0[v_i] \), \( a \) is simply ordered before \( C \). If \( q = \text{goal}[v_i] \), \( a \) is simply ordered after \( T \). The orders are added to \( \mathcal{R} \). (ii) If \( T, C \notin \text{chain}_{v_i}(s_0[v_i], s_*[v_i]) \), then the minimal chains \( \delta_{v_i}(s_0[v_i], q) \) and \( \delta_{v_i}(q, s_0[v_i]) \) are built, the actions added to \( \mathcal{A} \) and their relation added to \( \mathcal{R} \). The chain between \( s_0[v_i] \) and \( s_*[v_i] \) is updated: \( \text{chain}_{v_i}(s_0[v_i], s_*[v_i]) = \delta_{v_i}(s_0[v_i], q) \oplus \delta_{v_i}(q, s_0[v_i]) \oplus \delta_{v_i}(s_0[v_i], s_*[v_i]) \).

Now, \( T, C \in \text{chain}_{v_i}(s_0[v_i], s_*[v_i]) \) and step 2.(i) can be executed.

Phase 3: Check whether the set of orders \( \mathcal{R} \) is not irreflexive. If it is not irreflexive, then fail. The plan \( \Delta = (\mathcal{A}, \mathcal{R}) \) is returned otherwise.

There are three versions of the algorithm, with different worst-case time complexities:

1. \( O(n^3 \cdot m^3) \) [35], this version performs a transitive closure to check the irreflexivity. The transitive closure is based on the Floyd-Warshall algorithm [36] whose worst-case time complexity is \( O(|V|^3) \), with \( V \) the vertex set of a graph. \( O(|V|^3) \equiv O(|\mathcal{A}|^3) = O(n^3 \cdot m^3) \).

2. \( O(n^2 \cdot m^2) \) [6, p.77-79], in this version, the irreflexivity test is done by a topological sort as we only need to see whether \( \mathcal{R} \) has no cycle. The time complexity is dominated by Phase 2.

3. \( O(n \cdot m^2) \) [6, p.85-89], this version is equivalent as 2 except that there is a preprocessing to store the (PU) operators in a hashing table. I denote \( \mathcal{P} \) this algorithm and I provide it in Appendix A. Let \( v_i \in \mathcal{M} \), \( p \in \mathcal{D}_{v_i} \), due to post-uniqueness and unariness, operators are identifiable by the pair \(^4(v_i, p) \). It takes \( O(|\mathcal{O}|) =

\[^4\text{I formally define the operator identifiers in Definition 4.1.1.}\)
$O(n \cdot m)$ steps to store all the operators in an hashing table. Actions can therefore be instantiated in constant time while planning by searching the corresponding operator in the hashing table.

### 3.2.3 Discussion

Given the fact $P$ has the best worst-case time complexity, I chose to work with this one. I have implemented it in C++14 with default settings for Microsoft Visual Studio 2019. The code was optimized with the use of references and pointers as much as possible to avoid unnecessary copy constructors and other underlying procedures that can burden the planning runtime performance, especially with the manipulation of actions that are instantiated from operators. I then tested $P$ on scalable and realistic benchmarks to observe its runtime performance. The experiments and results are fully explained in Chapter 5.

It can be noticed, however, from Figure 5.18, that my implementation of $P$ was able to provide a plan with a length of at most 10 to a hundred NPCs in realtime (<1.67 ms) for each of my realistic benchmarks. It represents on average one plan every 15 μs. In F.E.A.R., the planner returns one plan in 43 μs on average (cf. Figure 2.7), and in 33 μs on average without the context preconditions. But plans in F.E.A.R. were of length 1 or 2, and at most 4. In Shadow of Mordors [3], developers bound the number of calls to 50 NPCs per frame and action plans are shorts, i.e. the same length as those of F.E.A.R. In other words, if developers make the effort to create problems of the class SAS+-PUS for their NPCs, $P$ is capable of controlling a hundred of them in realtime and with longer plans than current planning systems.

The single-valuedness (S) is very restrictive [35], however. It is tricky to represent realistic problems with it and can sometimes be a difficult conundrum, if the representation is possible without concessions though. (S) does not allow problems where an action needs a room to be on and another the room to be off for instance. This restriction will only allow the room to be on (or off) as a prevail-condition. Removing (S) from SAS-PUS is equivalent to considering the intractable class of problems SAS-PU, however. The source of intractability is linked to exponentially sized minimal solution plans. It exists problems where the minimal solution plan requires an exponential number of operator instances, while with (S), minimal solution plans have at most two times the same instance it. One of my assumption from Chapter 2 is that action plans in video games have at most one occurrence of the same action, i.e. one instance per operator. Is it possible to only consider SAS-PU problems whose minimal solution plans have at most one instance per operator? Does it exist a restriction to SAS-structures whose resulting class of problems with (P) and (U) only have minimal solution plans with one occurrence of the same action?
In Chapter 2, I defined four assumptions (Proposition 2.4.1) that should help reaching real-time planning. Among them are the restrictions (P) and (U), which, once combined with a SAS-structure, create the class of problems SAS-PU, proven NP-Hard [30]. In the previous Chapter, I recalled the restriction (S) which creates the tractable class of problems SAS-PUS with (P) and (U). (S) is very restrictive, however, plus it exists problems in this class where minimal solution plans have two occurrences of the same action which does not fit my fourth assumption.

The intractability of the class SAS-PU is due to exponentially sized minimal solution plans. I can quote the following observation from C. Bäckström [6, p. 175] at the end of his thesis:

One source of intractability for planning is that instances are allowed that require exponentially sized minimal solutions. [...] This is non trivial (to remove such instances), however. There is no single set of restrictions having this effect. [...] Although it is not sufficient to bound the plans for achieving tractability, it is an inevitable step to take.

What is missing from this observation is the reason why some minimal solutions are exponentially sized action plans. An action is an instance of an operator and, therefore, an exponentially sized minimal solution is an action plan where some operators have been instantiated an exponential number of times. **Proof 6.14** [6, p. 138] is a good example. The proof is based on the Gray Code [37] represented as a SAS-PUB problem where each operator modifies a single bit. Let $\Phi = (\mathcal{M}, \mathcal{S}, \mathcal{O})$ be a structure of such a Gray Code problem, the instance $\Pi = (\Phi, s_0 = \langle 0, 0, ..., 0 \rangle$, $s_\ast = \langle 1, 0, ..., 0 \rangle)$ leads to an exponentially sized minimal plan mainly because the operator that switches the least significant bit from 0 to 1 is at least instantiated $2^{(|\mathcal{M}| - 2)}$ times. Table 4.1 shows a problem of a 3-bits Gray Code where $o_{v_0}^+ \circ$ is instantiated twice. With a 4-bits Gray Code and a similar problem instance, $o_{v_0}^+ \circ$ would have been instantiated 4 times. With 5-bits, 8 times and so on. The operators of the Gray Code problem can be instantiated many times because operators
CHAPTER 4. TOPOLOGICAL PLANNING

Table 4.1: The SAS+–PUB 3-bits Gray Code. (a) lists the operators with their pre-, post- and prevail-conditions; (c) shows the minimal solution plan for the 3-bits Gray Code problem instance \( \Pi = (\Phi, s_0 = \langle 0, 0, 0 \rangle, s_* = \langle 1, 0, 0 \rangle) \). \( s_0 \) and \( s_* \) are respectively the start and goal states; (b) shows the succession of states between \( s_0 \) and \( s_* \) when \( \Delta \) is applied; \( a_{v_i}^+ \) (resp. \( a_{v_i}^- \)) is an instance of \( o_{v_i}^+ \) (resp. \( o_{v_i}^- \)) and it switches the bit \( v_i \) from 0 to 1 (resp. from 1 to 0).

(a)  

<table>
<thead>
<tr>
<th>Operators</th>
<th>pre-</th>
<th>post-</th>
<th>prevail-</th>
</tr>
</thead>
<tbody>
<tr>
<td>( o_{v_0}^+ )</td>
<td>( 0, u, u )</td>
<td>( 1, u, u )</td>
<td>( u, u, u )</td>
</tr>
<tr>
<td>( o_{v_0}^- )</td>
<td>( 1, u, u )</td>
<td>( 0, u, u )</td>
<td>( u, u, u )</td>
</tr>
<tr>
<td>( o_{v_1}^+ )</td>
<td>( u, 0, u )</td>
<td>( u, 1, u )</td>
<td>( 1, u, u )</td>
</tr>
<tr>
<td>( o_{v_2}^+ )</td>
<td>( u, 1, u )</td>
<td>( u, 0, u )</td>
<td>( 1, u, u )</td>
</tr>
<tr>
<td>( o_{v_2}^- )</td>
<td>( u, u, 0 )</td>
<td>( u, u, 1 )</td>
<td>( 0, 1, u )</td>
</tr>
</tbody>
</table>

(b)  

\[ \begin{array}{c|c|c|c} v_2 & v_1 & v_0 \\ \hline s_0 & 0 & 0 & 0 \\ & 0 & 0 & 1 \\ & 0 & 1 & 1 \\ & 0 & 1 & 0 \\ & 1 & 1 & 0 \\ & 1 & 1 & 1 \\ & 1 & 0 & 1 \\ & 1 & 0 & 0 \\ s_* & 0 & 0 & 0 \end{array} \]

(c) \( \Delta = \langle a_{v_0}^+, a_{v_1}^+, a_{v_0}^-, a_{v_2}^+, a_{v_0}^+, a_{v_1}^-, a_{v_0}^- \rangle \).

This chapter is organized as follows. In section 4.1, I provide prerequisites that will be used throughout the entire chapter. Operator and action graphs are then defined in section 4.2 and are useful to present the \( \Upsilon_1 \)-Truth Criterion (section 4.3) and the new structural restriction \( C_k \) with which I created new classes of problems (section 4.4). Eventually, I present TopoPlan, a correct and complete linear time algorithm that solves SAS-PUC\(_0\), SAS-PUC\(_2^S\) and SAS-PUC\(_2^*\) problem instances.

4.1 Prerequisites

In this section, I present several definitions that take advantage of the restrictions (P) and (U), and that are useful for what follows.

Remark: As actions are instances of operators, all the next definitions concerning operators are applicable to actions.

I denote (PU) operators the operators restricted with (P) and (U). The restrictions (P) and (U) allow to identify a (PU) operator \( o \) with the pair \( (v_i, p) \), where \( p \) is the postcondition of \( o \) on \( v_i \).

**Definition 4.1.1 ((PU) Operator Identifier).** I denote \( o_{v_i}^p \) the unique (PU) operator such that: \( v_i \in M, p \in D_{v_i}, \text{post}(o_{v_i}^p)[v_i] = p \).
**Definition 3.1.3** gives the original functions to represent pre-, post- and prevail-conditions of SAS operators, i.e. these functions return a partial-state. Because I'm using unary operators, however, it is relevant to represent the pre- and post-condition of any unary operator \( o^i_v \) as a function that return the value of the affected state variable \( v_i \) instead. Table 4.2 (a) shows the Horse Breeder operator set with the new operator representation using the following definition.

**Definition 4.1.2.** Let \( v_i \in M \) and \( O(v_i) \subseteq O \) the set of unary operators affecting the state variable \( v_i \). For each \( o^i_v \in O(v_i) \), I redefine the \( \text{pre()} \) and \( \text{post()} \) functions by taking into account the restrictions (P) and (U), and with respect to SAS structures (cf. subsection 3.1.3):

- SAS or SAS*: \( \text{pre()} : O(v_i) \rightarrow D_{v_i} \),
- SAS\(^+\): \( \text{pre()} : O(v_i) \rightarrow D^+_{v_i}, \text{post()} : O(v_i) \rightarrow D_{v_i} \).

The function \( \text{prv()} \) does not change: \( \text{prv()} : O \rightarrow S^+ \)

<table>
<thead>
<tr>
<th>( O )</th>
<th>Pre</th>
<th>Post</th>
<th>Prevail</th>
<th>( N_{pre} )</th>
<th>( N_{prv} )</th>
<th>( N_{clob} )</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( o^0_{v_1} )</td>
<td>( v_0 = 1 )</td>
<td>( v_0 = 0 )</td>
<td>( {0,0,0} )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>DropHaystack</td>
<td></td>
</tr>
<tr>
<td>( o^1_{v_1} )</td>
<td>( v_0 = 0 )</td>
<td>( v_0 = 1 )</td>
<td>( {0,0,1} )</td>
<td>( {0,0,1} )</td>
<td>( \emptyset )</td>
<td>( {0,0,1} )</td>
<td>FillHorseFeeder</td>
</tr>
<tr>
<td>( o^2_{v_1} )</td>
<td>( v_0 = 1 )</td>
<td>( v_0 = 2 )</td>
<td>( {0,0,2} )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>DropBucket</td>
<td></td>
</tr>
<tr>
<td>( o^1_{v_2} )</td>
<td>( v_1 = 0 )</td>
<td>( v_1 = 1 )</td>
<td>( {0,1,0} )</td>
<td>( {0,1,0} )</td>
<td>( \emptyset )</td>
<td>PickUpBucket</td>
<td></td>
</tr>
<tr>
<td>( o^2_{v_2} )</td>
<td>( v_2 = 0 )</td>
<td>( v_2 = 1 )</td>
<td>( {0,2,1} )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>FillBucketWithWater</td>
<td></td>
</tr>
<tr>
<td>( o^2_{v_2} )</td>
<td>( v_2 = 1 )</td>
<td>( v_2 = 2 )</td>
<td>( {0,2,2} )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>FillHorseTrough</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: The different representation of the Horse Breeder operators: (a) This resemble the original representation except I take advantage of the restrictions (PU) to represent the pre- and post-conditions (Definition 4.1.2). (b) The sets of Neighbors (Definition 4.1.3). \( N_{pre} \) is a singleton. As \( n = |D_{v_0}| = |D_{v_2}| = 3 \), operators can be indexed with the formula \( i \times 3 + p \). For instance, PickUpBucket index is 4. \( o^0_{v_2} \), the predecessor of \( o^1_{v_2} \) via the precondition, is a ghost operator (Definition 4.2.3). (c) The set of clobberers (Definition 3.2.5).

The notation \( o^i_{vi} \) carries the assigned state variable, namely \( v_i \in M \), thus: \( \text{post}(o^i_{v_i})[v_i] \equiv \text{post}(o^i_{v_i}) \) and \( \text{pre}(o^i_{v_i})[v_i] \equiv \text{pre}(o^i_{v_i}) \). Hence Definition 4.1.2. In practice, it is possible to index each operator \( o^i_{v_i} \) with the formula \( i \times n + p \) for instance, with \( n \) being the size of the biggest state variable domain. Such identification allows to create a hashing table, like in the algorithm \( P \), to instantiate actions in constant time (\( O(1) \)) while planning. Definition 4.1.1 combined with Definition 3.2.5 allow to define two new types of operator sets. Indeed, the defined pre- or prevail-conditions of an operator determine its predecessors.
Definition 4.1.3 (Predecessors sets). For each $o_{v_i}^p \in \mathcal{O}$, the predecessors of $o_{v_i}^p$ are defined as follows:

- $\mathcal{N}_{\text{pre}}(o_{v_i}^p) = \{ o_{v_j}^q \mid v_j = v_i \land q = \text{pre}(o_{v_i}^p) \}$, the set of operators that establish the pre-condition of $o_{v_i}^p$. $\mathcal{N}_{\text{pre}}(o_{v_i}^p)$ is a singleton due to (P).

- $\mathcal{N}_{\text{prev}}(o_{v_i}^p) = \{ o_{v_j}^q \mid v_j \neq v_i \land q = \text{prv}(o_{v_i}^p)[v_j] \}$, the set of operators that establish the prevail-conditions of $o_{v_i}^p$.

In an action plan, an action instance of these predecessors is an establisher of an action instance of $o_{v_i}^p$. Table 4.2 (b) gives examples of predecessor sets. For instance with the operator FillHorseTrough ($o_{v_2}^2$), it requires water in the bucket ($v_2 = 1$), hence its predecessor via the pre-condition is FillBucketWithWater ($\mathcal{N}_{\text{pre}}(o_{v_2}^2) = o_{v_2}^1$). To fill the horse trough, the Horse Breeder also requires the bucket in his hands ($v_1 = 1$), therefore the predecessor of $o_{v_2}^2$ via the prevail-condition is PickUpBucket ($\mathcal{N}_{\text{prev}}(o_{v_2}^2) = o_{v_1}^v$).

Let $o_{v_i}^q, o_{v_j}^q \in \mathcal{O}$, if $\mathcal{N}_{\text{pre}}(o_{v_i}^q) = \{ o_{v_j}^q \}$, i.e. $v_i = v_j \land \text{post}(o_{v_j}^q) = \text{pre}(o_{v_i}^q)$, then there is a post-pre dependency between $o_{v_j}^q$ and $o_{v_i}^q$: the post-condition of $o_{v_i}^q$ solves the pre-condition of $o_{v_j}^q$. If $\mathcal{N}_{\text{prev}}(o_{v_j}^q) = \{ o_{v_i}^q \}$, i.e. $v_j \neq v_i \land \text{post}(o_{v_i}^q) = \text{prv}(o_{v_i}^q)[v_j]$, then there is a post-prevail dependency between $o_{v_j}^q$ and $o_{v_i}^q$: the post-condition of $o_{v_j}^q$ solves the prevail-condition of $o_{v_i}^q$ on $v_j$. Let consider the notations $\prec_{\text{pre}}$ and $\prec_{\text{prev}}$ to represent the two different dependencies. They are two strict partial-orders between two operators $o_{v_i}^q, o_{v_j}^q \in \mathcal{O}$ such that $(o_{v_j}^q \prec_{\text{pre}} o_{v_i}^q) \equiv o_{v_j}^q \in \mathcal{N}_{\text{pre}}(o_{v_i}^p)$ and $(o_{v_j}^q \prec_{\text{prev}} o_{v_i}^q) \equiv o_{v_j}^q \in \mathcal{N}_{\text{prev}}(o_{v_i}^p)$.

$\mathcal{N}_{\text{pre}}(o_{v_i}^p)$ and $\mathcal{N}_{\text{prev}}(o_{v_i}^p)$ can both be initialized during a pre-processing by looking after the defined pre- or prevail-conditions. The process takes no more than $|\mathcal{M}|$ steps per operator to complete as there are $|\mathcal{M}|$ state variables to browse and it takes a constant time to identify each predecessor via a hashing table using Definition 4.1.1.

Similar to the requestable values (cf. Definition 3.2.2) which, for unary operators, are values that are assigned to the prevail-condition of some other operators, I define the requestable (PU) operators. Due to restrictions (P) and (U), and with the identifiers (cf. Definition 4.1.1), a requestable value is in fact the post-condition of a unique (PU) operator.

Definition 4.1.4 (Requestable Operator $\mathcal{RO}$). I define $\mathcal{RO}$ the set of requestable operators of $\mathcal{O}$. These operators solve a prevail-condition of at least one other operator of $\mathcal{O}$.

$$\mathcal{RO} = \{ o \mid v_i \in \mathcal{M}, o \in \mathcal{O}(v_i), \text{post}(o)[v_i] \in \mathcal{R}_{v_i}^O \}$$

Similar to $\mathcal{O}(v_i)$, $\mathcal{A}(v_i)$ or Cycle($v_i$), I denote $\mathcal{RO}(v_i)$ the set of requestable operators affecting the state variable $v_i$. It can be highlighted that $\mathcal{RO} = \bigcup_{o \in \mathcal{O}} (\mathcal{N}_{\text{prev}}(o))$. Let give two examples of requestable operator sets: the one of the Horse Breeder (cf. Table 4.2) is $\mathcal{RO} = \{ \text{DropHaystack, DropBucket, PickUpBucket} \}$; that of the 3-bits Gray Code (cf. Figure 4.1) is $\mathcal{RO} = \{ o_{v_0}^v, o_{v_0}^v, o_{v_1}^v \}$. 

48
CHAPTER 4. TOPOLOGICAL PLANNING

It results from Definition 3.1.2 and Definition 3.2.5 that it exists problem instances where an action instance of a requestable operator is an establisher of the prevail-condition of one or more other actions in the solution plan.

Because I am interested in the number of action instances of an operator in a solution plan, I introduce the following output restriction denoted \( \Upsilon_k \).

**Definition 4.1.5 (Output restriction \( \Upsilon_k \)).** Let \( k \in \mathbb{N} \), an action plan \( \Delta \) is \( \Upsilon_k \) iff:

- the number of distinct action instances of the same operator occurring in \( \Delta \) is at most \( k \).
- it exists at least one operator that has \( k \) distinct action instances in \( \Delta \).

I will use this definition to characterize any sequence of actions such as action chains or minimal action plans. It results from this definition that the minimal solution plan of the instance of the 3-bits Gray Code presented Figure 4.1 is \( \Upsilon_2 \), because the operators \( o_{v_0}^+ \) and \( o_{v_0}^- \) are instantiated twice. Then, Lemma 3.2.1 can be rewritten: for each \( v_i \in M \), \( \delta_{v_i} \) is \( \Upsilon_1 \). In addition, if a plan is said to be at least \( \Upsilon_k \) then it exists \( n \in \mathbb{N} \) such that \( n \geq k \) and the plan is \( \Upsilon_n \).

I extend the definition of \( \Upsilon_k \) to problems. A SAS problem is at most \( \Upsilon_k \) iff it exists problem instances whose minimal solution plan is \( \Upsilon_k \) and iff it does not exist problem instances whose minimal solution plan is \( \Upsilon_n \), \( \forall n > k \). Then, a SAS problem is at least \( \Upsilon_k \) iff it exists problem instances whose minimal solution plan is \( \Upsilon_k \) and it may exist problem instances whose minimal solution plan is \( \Upsilon_n \), \( \forall n > k \). I add this definition or this characterization of SAS problems because I am interested in problems that are at most \( \Upsilon_1 \). In other words, I aim to discard any SAS problem that is at least \( \Upsilon_2 \).

Moreover, according to the introduction of this Chapter and Theorem 6.14 [6, p.138], the Gray Code problem with \( n \)-bits is at least \( \Upsilon_{2(n-2)} \), as it exists problem instances where the least significant bit is switched (from 0 to 1) \( 2^{(n-2)} \) times. Hence, there are \( 2^{(n-2)} \) distinct action instances of the operator doing this switch. Concerning the class of problems SAS\(^+\)PUS, Theorem 4.4 [6, p.76] implies that these problems are at most \( \Upsilon_2 \).

**Lemma 4.1.1.** Let \( \Pi = (\Phi, s_0, s_*) \) a SAS problem instance such that \( s_0 \neq s_* \). If \( \Pi \) is solvable, then there exists a non-empty minimal solution plan solving \( \Pi \) and it is at least \( \Upsilon_1 \).

**Proof.** The idea of the proof is to show that a \( \Upsilon_0 \) plan is equivalent to an empty plan, and that an empty plan cannot solve a problem instance where the start is different from the goal state. Then, I justify the statement “and it is at least \( \Upsilon_1 \)” by showing that it exists problem instances whose minimal solution plan is exactly \( \Upsilon_1 \).

Let denote \( \Delta \) the minimal solution plan of \( \Pi \). If \( \Pi \) is solvable and \( \Delta \) is \( \Upsilon_0 \), then \( \Delta = \langle \rangle \).

Indeed, \( \Upsilon_0 \) means the number of distinct action instances of the same operator in \( \Delta \) is at
most 0. And if $\Delta = \langle \rangle$, then $\text{result}(s_0, \Delta) = s_0 \neq s_*$. Therefore, $\Delta$ does not solve $\Pi$ which is a contradiction. Therefore, $\Delta$ cannot be $\Upsilon_0$, i.e. empty.

Let $o$ an operator of a SAS problem and $a$ its action instance, i.e. $\text{ope}(a) = o$. If $\Pi$ is made up of $s_0 = \text{pre}(o) \oplus \text{prv}(o)$ and $s_* = \text{post}(o) \oplus \text{prv}(o)$, then $\Delta = \langle a \rangle$ is the minimal solution plan solving $\Pi$ and it is $\Upsilon_1$. It is possible to create such an instance in every SAS problem.

It results the statement of Lemma 4.1.1.

Corollary 4.1.1. Every SAS problem is at least $\Upsilon_1$.

Proof. Every SAS problem has at least one operator and therefore, every SAS problem has at least the problem instance described in Proof 4.1.1 and whose minimal solution plan is $\Upsilon_1$.

4.2 Operator Graphs

Similar to the state transition graph and the domain transition graph [28], or causal graph [38, 39], which consist in representing a planning problem as a graph, I introduce three new graphs that focus on operators and actions and their relations between them: two of them are operator graphs and represent the planning problem, the third one is an action graph and is a graphical representation of the solution plan. These operator- and action-focused graphs shed light on operators and actions instead of states, state values or state variables. Furthermore, the output of a SAS planner being action plans it is convenient to represent them as a graph to visualize the relations between the actions.

As mentioned in the introduction of this Chapter, it exists (PU) operators that loop together and, as a result, for some problem instances, they can be instantiated an exponential number of times, thus creating exponentially sized plans. My approach to break this intractability is therefore to study these cycles so as to suggest a structural restriction that insures these operators are instantiated at most once in each minimal solution plan of each instance of a problem. A structural restriction is a concept that was introduced by C. Bäckström and P. Jonsson in [28]. They have used graphs to study the interaction between values of state variable domains and then suggested restrictions to avoid some unwilling planner behavior.

4.2.1 Operator Graph

The first graph I introduce is the Operator Graph and it is described as follows:

Definition 4.2.1 (Operator Graph). I denote $G_O = (\mathcal{O}, E_O)$ the operator graph, with $\mathcal{O}$ the vertex set and $E_O$ the directed edge set. Let $v_i, v_j \in \mathcal{M}$, $E_O = E_{\text{pre}}(\mathcal{O}) \cup E_{\text{prv}}(\mathcal{O})$, and:

\footnotetext{result is a function defined by C.Bäckström in Definition 3.4 [6, p.58], it merely returns the state resulting from executing an action sequence.}
CHAPTER 4. TOPOLOGICAL PLANNING

- $E_{\text{pre}}(\mathcal{O}) = \{(o_{vi}^a, o_{vi}^p) \mid o_{vi}^a, o_{vi}^p \in \mathcal{O}, o_{vi}^a \in \mathcal{N}_{\text{pre}}(o_{vi}^p)\}$ is the set of directed edges between two (PU) operators of $\mathcal{O}$ having a post-pre dependency. Each post-pre dependency such that $o_{vi}^a \prec_{\text{pre}} o_{vi}^p$ is represented as a directed edge from $o_{vi}^a$ to $o_{vi}^p$.

- $E_{\text{prv}}(\mathcal{O}) = \{(o_{v}^a, o_{v}^p) \mid o_{v}^a, o_{v}^p \in \mathcal{O}, o_{v}^a \in \mathcal{N}_{\text{prv}}(o_{v}^p)\}$ is the set of directed edges between two (PU) operators having a post-prevail dependency. Each post-prevail dependency such that $o_{v}^a \prec_{\text{prv}} o_{v}^p$ is represented as a directed edge from $o_{v}^a$ to $o_{v}^p$.

An operator graph shows where the planner will navigate to search for a plan and gives a hint on how actions will be connected to each other for a given problem instance. In the introduction of this Chapter, I mentioned that only operators affecting the same state variable and looping between each other can be instantiated several times. In the Horse Breeder example, the operators of the pairs PickUpBucket/DropBucket and DropHaystack/TakeHaystack loop between each other and can be instantiated several times by the planner. For instance, if the horse breeder wants to work out with his haystacks, he can repeatedly take and drop them. $\sigma = \langle a_{v0}^1, a_{v0}^0, a_{v0}^1, a_{v0}^0, a_{v0}^1, a_{v0}^0 \rangle$ is therefore a possible sequence of actions where both take and drop operators have been instantiated three times: $\text{ope}(a_{v0}^1) = \text{ope}(a_{v0}^1) = \text{ope}(a_{v0}^1) = a_{v0}^1$ and $\text{ope}(a_{v0}^0) = \text{ope}(a_{v0}^0) = \text{ope}(a_{v0}^0) = a_{v0}^0$. It is exactly the same behaviour with the operators of the 3-bits Gray Code (cf. Figure 4.1), the operators of the pairs $a_{v0}^+/_v a_{v0}^-, a_{v1}^+/_v a_{v1}^-$ or $a_{v2}^+/_v a_{v2}^-$ loop between each other and the induced cycles enable $a_{v0}^+$ and $a_{v0}^-$ to be instantiated twice by the planner in order to solve $\Pi = (\Phi, s_0 = \langle 0, 0, 0 \rangle, s_* = \langle 1, 0, 0 \rangle)$. Figure 4.1 shows the operator graph of the Horse Breeder and that of the 3-bits Gray Code. The blue directed edges represent the post-pre dependencies ($E_{\text{pre}}(\mathcal{O})$), whereas the green directed edges represent the post-prv dependencies ($E_{\text{prv}}(\mathcal{O})$). It may exist cycles through the green edges (post-prv dependencies) or through the green and the blue edges. By definition of the pre- and post-conditions, it is through the post-pre dependencies that values of state variables are modified. I therefore focused on the cycles created by the blue arcs in order to spot what cause the minimal solution plans to grow exponentially in some problems. The following therefore introduces the Domain Operator Graph which is an operator graph that focuses on post-pre dependencies only. This is through this type of graph that chains of actions are built.

### 4.2.2 Domain Operator Graph

The following definition concerns the Domain Operator Graph, a useful graph to study chains of actions (cf. Definition 3.2.3).

**Definition 4.2.2** (Domain Operator Graph). For each $v_i \in \mathcal{M}$, I define the Domain Operator Graph of $v_i$ the directed graph $\mathcal{G}_{\mathcal{O}}(v_i) = (\mathcal{O}(v_i), E_{\text{pre}}(\mathcal{O}(v_i)))$, with $\mathcal{O}(v_i)$ the vertex set and $E_{\text{pre}}(\mathcal{O}(v_i)) = \{(o', o) \mid o, o' \in \mathcal{O}(v_i), o' \in \mathcal{N}_{\text{pre}}(o)\}$ the directed edge set. $\mathcal{G}_{\mathcal{O}}(v_i)$ has the following properties:

1. $\mathcal{G}_{\mathcal{O}}(v_i)$ is weakly connected.
2. \( \mathcal{G}_\mathcal{O}(v_i) \) has \( |\mathcal{D}_{v_i}| \) operators.

The property (1.) of Definition 4.2.2 is kind of a structural restriction. Although it may exist problems with disconnected domain operator graphs, each connected parts are weakly connected. Plus, to solve a problem instance, a SAS planner will only navigate through one of the connected part to find a solution. Let consider the following SAS planning framework, \( \mathcal{M} = \{v_0\}, \mathcal{D}_{v_0} = \{0, 1, 2, 3\} \) and \( \mathcal{O} = \{o_{v_0}^1, o_{v_0}^3\} \) such that \( \text{pre}(o_{v_0}^1) = 0 \land \text{post}(o_{v_0}^1) = 1 \) and \( \text{pre}(o_{v_0}^3) = 2 \land \text{post}(o_{v_0}^3) = 3 \). If the problem instance is made up of \( s_0[v_0] = 0 \) and \( s_\ast[v_0] = 3 \), then it is not solvable because \( o_{v_0}^1 \) and \( o_{v_0}^3 \) are disjoint and, therefore, it does not exist a chain of actions between \( s_0[v_0] = 0 \) and \( s_\ast[v_0] = 3 \). No matter the problem, the instances of \( o_{v_0}^1 \) and \( o_{v_0}^3 \) will never end up in the same chain of actions. The idea behind the property (1.) is therefore to say that, in such cases, the disjointed parts can be considered as different domain operator graphs, even though the operators affect the same state variable. It does not matter as their action instances will never end up in the same chain of actions, nor in the same action plan.

The property (2.) is directly linked to Definition 4.1.1. We can identify exactly \( |\mathcal{D}_{v_i}| \) operators, hence \( \mathcal{G}_\mathcal{O}(v_i) \) has \( |\mathcal{D}_{v_i}| \) operators. In fact, the post-unicity implies that there are at most \( |\mathcal{D}_{v_i}| \) operators, i.e. a unique operator for each value of the domain \( v_i \). The weakly connected property implies that there are at least \( |\mathcal{D}_{v_i}| - 1 \) operators. Indeed, consider the domain transition graph \( \mathcal{G}_{v_i} = (\mathcal{D}_{v_i}, \mathcal{T}_{v_i}) \) [28, p.10], with \( \mathcal{T}_{v_i} = \{(\text{pre}(o), o, \text{post}(o)) \mid o \in \mathcal{O}(v_i), \text{pre}(o), \text{post}(o) \in \mathcal{D}_{v_i}\} \) the set of transition between two values of the domain of \( v_i \), in order to connect each value of \( \mathcal{D}_{v_i} \) we need at least \( |\mathcal{D}_{v_i}| - 1 \) arcs in \( \mathcal{T}_{v_i} \). As an arc of \( \mathcal{T}_{v_i} \) is merely an operator, it results that there are at least \( |\mathcal{D}_{v_i}| - 1 \) existing operators in \( \mathcal{G}_\mathcal{O}(v_i) \). Therefore, there may exist a unique \( x \in \mathcal{D}_{v_i} \) such that \( o_{v_i}^2 \notin \mathcal{O}(v_i) \), i.e. it exists an unique operator that does not exist in the operator set. I call this missing operator the ghost operator. Proof 4.2.2 gives more details on why there are at least \( |\mathcal{D}_{v_i}| - 1 \) operators with a defined pre-condition in \( \mathcal{G}_\mathcal{O}(v_i) \).

**Definition 4.2.3** (Ghost operator). Let \( v_i \in \mathcal{M}, x \in \mathcal{D}_{v_i} \). The ghost operator\(^2\) is an

---

\(^2\)I could have created a notation to distinguish the operator set with ghost operators (like \( \mathcal{O}^+ \)) with...
CHAPTER 4. TOPOLOGICAL PLANNING

(a) Gray Code

Ghost V2

Fill Bucket with Water

Fill Horse Trough

Drop Haystack

Take Haystack

Fill Horse Feeder

Pick up Bucket

Drop Bucket

(b) Horse Breeder

Figure 4.2: All the domain operator graphs of the Horse Breeder and the Gray Code.

The Horse Breeder has a ghost operator in \( G(v_2) \), named Ghost V2, whose identifier is \( o_{v_2}^0 \) (cf. Definition 4.2.3).

operator that can be identified with the pair \((v_i, x)\) according to Definition 4.1.1 but that does not exist in \( \mathcal{O} \), i.e. \( o_{v_i}^x \notin \mathcal{O} \). The pre-condition of \( o_{v_i}^x \) is unknown, therefore:

\[
\text{pre}(o_{v_i}^x) = \emptyset.
\]

In the Horse Breeder problem, there is a ghost operator, named Ghost V2. It is represented in Figure 4.2 with a dashed node and a dashed directed edge to FillBucketWithWater. In fact, \( (v_2 = 0) \) states the water is in a source and, in my Horse Breeder model, no action can fill the source as I consider the filling to be an external event. Therefore, identifying \( (v_2 = 0) \) as an operator results in a ghost operator, i.e. an operator that can be identified but that does not exist.

Lemma 4.2.1 (Ghost Operator Uniqueness). For each \( v_i \in \mathcal{M} \), if there is a ghost operator in \( G(v_i) \), then it is unique.

Proof. Direct from the fact a ghost operator, if it exists, is a missing operator in \( G(v_i) \). By definition of \( G(v_i) \), there is at least \( |D_{v_i}| - 1 \) defined operator. The last one, if not defined, is the ghost operator and is unique in \( G(v_i) \).

The domain operator graphs of (PU) operators have two specific characteristics:

Lemma 4.2.2. For each \( v_i \in \mathcal{M} \):

1. \( G(v_i) \) is a directed tree if there is a ghost operator.

2. \( G(v_i) \) has a unique cycle otherwise.

...
Proof. The two points are proven by recursion. Let $\Phi = (\mathcal{M}, \mathcal{S}, \mathcal{O})$ a SAS-structure, $v_i \in \mathcal{M}$ a state variable with $\mathcal{D}_{v_i}$ its domain, and $G_{\mathcal{O}(v_i)}|_{\mathcal{D}_{v_i}} = (\mathcal{O}(v_i), E_{\text{pre}}(\mathcal{O}(v_i)))$ the associated domain operator graph.

Base case: Let $\mathcal{D}_{v_i} = \{x_1, x_2\}$, we can identify exactly 2 operators with Definition 4.1.1, namely $o_1$ and $o_2$, such that $\text{post}(o_1) = x_1$ and $\text{post}(o_2) = x_2$. We have $\mathcal{O}(v_i) = \{o_1, o_2\}$

Case 1: There is a ghost operator in $G_{\mathcal{O}(v_i)}$. Let consider $o_2$ to be the ghost operator. Therefore, $o_1$ is not a ghost operator, and due to restriction (S4) and (S6) from the SAS-structure (cf. subsection 3.1.3), $\text{pre}(o_1) \neq \text{post}(o_1) \neq u$ and $\text{pre}(o_1) \in \mathcal{D}_{v_i} \setminus \{x_1\}$, that is: $\text{pre}(o_1) = x_2$. It implies that, $o_2 \prec_{\text{pre}} o_1$, that is $G_{\mathcal{O}(v_i)}$ has 2 nodes and 1 directed edge. Therefore, $G_{\mathcal{O}(v_i)}$ is a directed tree and the root node $o_2$ is the ghost operator. If $o_1$ and $o_2$ are swapped in the proof, the reasoning is equivalent.

Case 2: There is no ghost operator in $G_{\mathcal{O}(v_i)}$. Therefore, due to restriction (S4) and (S6), $\text{pre}(o_1) \neq \text{post}(o_1) \neq u$ and $\text{pre}(o_1) \in \mathcal{D}_{v_i} \setminus \{x_1\}$ and $\text{pre}(o_2) \neq \text{post}(o_2) \neq u$ and $\text{pre}(o_2) \in \mathcal{D}_{v_i} \setminus \{x_2\}$. That is $\text{pre}(o_1) = x_2$ and $\text{pre}(o_2) = x_1$, which implies $o_2 \prec_{\text{pre}} o_1$ and $o_1 \prec_{\text{pre}} o_2$. In other words, $o_1$ and $o_2$ are looping together, i.e. they create a cycle in $G_{\mathcal{O}(v_i)}$ and the cycle is unique.

Induction hypothesis: $(\mathcal{D}_{v_i})_n = \{x_1, \ldots, x_n\}$ and $(\mathcal{O}(v_i))_n = \{o_1, \ldots, o_n\}$ such that $\forall k \in [1, n], \text{post}(o_k) = x_k$.

Case 1: $(G_{\mathcal{O}(v_i)})_n$ is a directed tree and the root node is the ghost operator. The ghost operator is denoted $o_j$ and $\text{post}(o_j) = x_j$.

Case 2: There is no ghost operator and $(G_{\mathcal{O}(v_i)})_n$ has a unique cycle.

Induction step: Let $(\mathcal{D}_{v_i})_{n+1} = \{x_1, \ldots, x_{n+1}\}$ and $(\mathcal{O}(v_i))_{n+1} = \{o_1, \ldots, o_{n+1}\}$ such that $\forall k \in [1, n + 1], \text{post}(o_k) = x_k$.

Case 1, there are two possibilities:

- $o_{n+1}$ is a ghost operator, it means $o_{n+1}$ is disconnected from $(G_{\mathcal{O}(v_i)})_n$. Indeed, $\text{post}(o_{n+1}) = x_{n+1} \notin (\mathcal{D}_{v_i})_n$ and its pre-condition is not set. Hence, $o_{n+1}$ does not precedes nor succeeds any operator of $(G_{\mathcal{O}(v_i)})_n$. So, $o_{n+1}$ cannot be a ghost operator, otherwise $(G_{\mathcal{O}(v_i)})_{n+1}$ is not weakly connected which contradicts the definition of the domain operator graph.

- $o_{n+1}$ is not a ghost operator, and due to restriction (S4) and (S6), $\text{post}(o_{n+1}) \neq \text{pre}(o_{n+1}) \neq u$, therefore $\text{pre}(o_{n+1}) \in (\mathcal{D}_{v_i})_n \setminus \{x_{n+1}\}$, that is $\text{pre}(o_{n+1}) \in (\mathcal{D}_{v_i})_n$. It implies $o_{n+1}$ is preceded by one operator in $(G_{\mathcal{O}(v_i)})_n$. Therefore, this adds a new node $o_{n+1}$ and a new directed edge from $(G_{\mathcal{O}(v_i)})_n$ to $o_{n+1}$, it implies $(G_{\mathcal{O}(v_i)})_{n+1}$ has $n + 1$ nodes and $n$ directed edges. $(G_{\mathcal{O}(v_i)})_{n+1}$ is therefore a directed tree and the root node $o_j$ is the only ghost operator. It concludes point 1. of the Lemma: If there is a ghost operator, $G_{\mathcal{O}(v_i)}$ is a directed tree.

Case 2, there are two possibilities:

- If $o_{n+1}$ is a ghost operator, then $(G_{\mathcal{O}(v_i)})_{n+1}$ is not weakly connected because $o_{n+1}$ does not connect $(G_{\mathcal{O}(v_i)})_n$. Indeed, $\text{post}(o_{n+1}) = x_{n+1} \notin (\mathcal{D}_{v_i})_n$ and all the operators
of \((\mathcal{G}_O(v_i))_n\) have their pre-condition defined in \((\mathcal{D}_v)_n\). It contradicts the definition of a domain operator graph, so \(o_{n+1}\) cannot be a ghost operator.

- If \(o_{n+1}\) is not a ghost operator, then due to restriction (S4) and (S6), \(\text{pre}(o_{n+1}) \in (\mathcal{D}_v)_{n+1} \setminus \{x_{n+1}\}\), that is \(\text{pre}(o_{n+1}) \in (\mathcal{D}_v)_n\), which implies \(o_{n+1}\) is necessary preceded by an operator of \((\mathcal{G}_O(v_i))_n\). This therefore adds a new node \(o_{n+1}\) and a new directed edge from \((\mathcal{G}_O(v_i))_n\) to \(o_{n+1}\). The new directed edge does not create another cycle as \(x_{n+1} \notin (\mathcal{D}_v)_n\). Therefore, \((\mathcal{G}_O(v_i))_{n+1}\) has a unique cycle and no ghost operator. It concludes the point 2. of the lemma: If there is no ghost operator, \(\mathcal{G}_O(v_i)\) has a unique cycle.

\[ \square \]

**Remark:** The restriction (S4) (cf. subsection 3.1.3) says that the post-condition and the pre-condition of an operator cannot be equal, it implies domain operator graphs have no loops\(^3\).

Figure 4.2b presents the domain operator graphs of the Horse Breeder (cf. Table 4.2). \(\mathcal{G}_O(v_0)\) and \(\mathcal{G}_O(v_1)\) have a unique cycle and no ghost operator while \(\mathcal{G}_O(v_2)\) is a directed tree with a ghost operator and a single branch. In Appendix B, I provide Mathematica functions to produce all possible shapes of a domain operator graph with (PU) operators with respect to the number of values in the domain. Let \(v_i \in \mathcal{M}\), I also provide the result of these function for \(|\mathcal{D}_v| \in [2, 5]\). I was not able to go higher due to memory and time limitations. Then, it can be noticed that the shapes of \(\mathcal{G}_O(v_i)\) as a function of \(|\mathcal{D}_v|\) is equivalent to the graphs presented under the OEIS reference “A002861”, whose integer sequence name is “Number of connected functions (or mapping patterns) on \(n\) unlabeled points, or number of rings and branches with \(n\) edges”.

I previously stated that what cause the intractability of the class of problems SAS-PU are the operators that can be instantiated more than once, that is, operators that are inside a cycle. It is therefore relevant to study the cycle highlighted by (2.) **Lemma 4.2.2** because this is exactly the starting point of the intractability of the class of problems SAS-PU. To this end, I introduce a new set of operators that, for each \(v_i \in \mathcal{M}\), stores operators looping together in the unique cycle of \(\mathcal{G}_O(v_i)\).

**Definition 4.2.4.** Let \(v_i \in \mathcal{M}\), I denote \(\text{Cycle}(v_i)\) the set of operators that are inside the unique cycle of \(\mathcal{G}_O(v_i)\). If \(\mathcal{G}_O(v_i)\) is a directed tree, then \(\text{Cycle}(v_i) = \emptyset\). Otherwise, \(\text{Cycle}(v_i) \neq \emptyset\).

Let consider the Horse Breeder example to illustrate this new notation: \(\text{Cycle}(v_0) = \{\text{DropHaystack}, \text{TakeHaystack}\}\), \(\text{Cycle}(v_1) = \{\text{DropBucket}, \text{PickUpBucket}\}\), \(\text{Cycle}(v_2) = \emptyset\). Similarly, in the 3-bits Gray Code example, there is for each \(v_i \in \{v_0, v_1, v_2\}\): \(\text{Cycle}(v_i) = \{o_{v_i}^{-}, o_{v_i}^{+}\}\).

\(^3\)A loop in a graph is when there is an edge from a vertex to itself.
4.2.3 Action Graph

The action graph is the graphical representation of an action plan. In particular, the graphical representation of the minimal action plan $\Delta$ is described as follows:

**Definition 4.2.5 (Minimal Action Graph).** Let $\Delta = \langle A, \prec \rangle$ a minimal action plan, I denote $G_\Delta = (A, E_A)$ the minimal action graph, with $A$ the set of vertices and $E_A$ the set of directed edges, such that:

- $G_\Delta \equiv \Delta$, $G_\Delta$ is the graphical representation of $\Delta$.
- $E_A = \{(a, a') \mid a, a' \in A \land a \prec a'\}$.

The topological sort of $G_\Delta$ is equivalent to a totally-ordered minimal action plan. Figure 4.3 gives an example of an action graph: (4.3a) represents the non-linear minimal action plan solving the 3-bits Gray Code problem described in the introduction of this Chapter, and (4.3b) is the topological sort of it. The blue edges represent the post-pre dependencies and the green edges represent the post-prv dependencies. The red ones are orders to handle clobberers (cf. Definition 3.2.5).

4.2.4 Discussion

It should be noticed that the operator graph and the domain operator graph for (PU) operators are respectively the state transition graph and the P-Graph of [28] except that my graphs focus on operators instead of state variable values. That is, the arcs of their
graphs are the nodes of mine. Definitions 7.3 and 7.4 [28, p.44] say that the P-Graph is composed of a “center” which is a set of values either empty or it forms a simple cycle where vertex has incoming and outgoing degree of 1. This set of values, once their associated operator identified (Definition 4.1.1), is equivalent to the set of operators: \( v_i \in M, \text{Cycle}(v_i) \). They did not prove, however, why the cycle is unique. This is now proved with Lemma 4.2.2. Definition 7.4 gives another interesting description of the P-Graph and which can be transposed to the domain operator graph. Each element of the cycle is a root node of a (possibly empty) directed tree. Let takes \( G_Q(v_0) \) from the Horse Breeder (cf. Figure 4.2b), the center is: \( \text{Cycle}(v_0) = \{ \text{DropHaystack}, \text{TakeHaystack} \} \). DropHaystack is a root node of an empty tree while TakeHaystack is a root node of a directed tree composed of a single branch that connects TakeHaystack to FillHorseFeeder.

4.3 Truth Criterion

A minimal solution plan for a SAS-PU problem instance is composed of several chains of actions between the start and the goal state, and more precisely it is composed of a chain of action (possibly empty) for each state variable. This assertion from C. Bäckström’s IMP holds when (S) is relaxed. His truth criterion, however, is adapted for SAS+PUS problems that are at most \( \Upsilon_2 \). In this section, I aim to provide a truth criterion for SAS-PU problem instances whose minimal solution plan is \( \Upsilon_1 \). The section is organised as follows: I present the general form of a chain of actions, i.e. the possible forms of a chain of actions that is \( \Upsilon_k \). Based on that, I present the possible forms when \( k = 1 \). Out of these forms, I present several clobbering situations that must be handle in order to define a truth criterion for SAS-PU problem instances whose minimal solution plan is \( \Upsilon_1 \).

4.3.1 General Form of an Action Chain

We can observe from the 3-bits Gray Code instance \( \Pi = (M, O, s_0 = \langle 0, 0, 0 \rangle, s_* = \langle 1, 0, 0 \rangle) \) (cf. Figure 4.1) that the chain of actions on \( v_0 \) of the solution plan can be written:

\[
\text{chain}_{v_0}(s_0[v_0], s_*[v_0]) = \text{chain}_{v_0}(0, 0) = \langle a^+_v, a^-_v, a^+_v, a^-_v \rangle = \delta_{v_0}(s_0[v_0], s_0[v_0]) \oplus \delta_{v_0}(s_0[v_0], s_0[v_0]) = \langle a^+_v, a^-_v \rangle = \langle a^+_v, a^-_v \rangle
\]

If we extend the instance of Figure 4.1 to the n-bits Gray Code (\( \Pi = (M, O, s_0 = \langle 0, 0, ..., 0 \rangle, s_* = \langle 1, 0, ..., 0 \rangle) \)), then the chain of actions of the least significant bit in the solution plan looks like:

\[
\text{chain}_{v_0}(s_0[v_0], s_*[v_0]) = \delta_{v_0}(s_0[v_0], s_0[v_0]) \oplus ... \oplus \delta_{v_0}(s_0[v_0], s_0[v_0]) \quad 2^{n-2} \text{occurrences of } \delta_{v_0}
\]
Figure 4.4: A generic Domain Operator Graph. Let $v_i \in M$, $o_{v_i}^x, o_{v_i}^y, o_1, o_2 \in \text{Cycle}(v_i)$, $o_1$ and $o_2^y$ are the root nodes of two different directed trees. The dashed directed edges represent possible hidden nodes, i.e. it is a choice of representation to illustrate that paths between nodes can be longer. The dotted directed trees whose root nodes are respectively $o_y^{v_i}$ and $o_1$ are here as an example to illustrate a generic shape of a domain operator graph.

Based on section 4.2, a domain operator graph is composed of a unique cycle (cf. Lemma 4.2.2) and each node of this cycle is the root node of a, possibly empty, directed tree. Figure 4.4 gives a generic representation of a domain operator graph. Since action chains are composed of actions instantiated from operators affecting the same state variable, these domain operator graphs help to understand how the size of a chain grows. Let $k \in \mathbb{N}$, $v_i \in M$, $x, y, z \in \mathcal{D}_{v_i}$, based on Definition 4.1.5 and the study of the domain operator graphs (cf. section 4.2), a chain of actions is $\Upsilon_k$ iff:

$$
\text{chain}_{v_i}(x, z) = \underbrace{\delta_{v_i}(x, x) \oplus \ldots \oplus \delta_{v_i}(x, x)}_{k-1 \text{ times}} \oplus \delta_{v_i}(x, y) \oplus \delta_{v_i}(y, z)
$$

$$\land \forall a \in \delta_{v_i}(x, x), ope(a) \in \text{Cycle}(v_i)
\land \forall b \in \delta_{v_i}(x, y), ope(b) \in \text{Cycle}(v_i)
\land \forall c \in \delta_{v_i}(y, z), ope(c) \notin \text{Cycle}(v_i)
$$

(4.1)

Let consider Figure 4.4 to understand why $\text{chain}_{v_i}(x, z)$ is $\Upsilon_k$. Let $a_{v_i}^x, a_{v_i}^y, a_{v_i}^z, a_1, a_2$ and $a_3$ be the action instances of the operators $o_{v_i}^x, o_{v_i}^y, o_{v_i}^z, o_1, o_2$ and $o_3$ respectively, we have:

$$
\delta_{v_i}(x, x) = \langle a_1, ..., a_{v_i}^y, a_2, ..., a_{v_i}^x \rangle
\land \delta_{v_i}(x, y) = \langle a_1, ..., a_{v_i}^y \rangle
\land \delta_{v_i}(y, z) = \langle a_3, ..., a_{v_i}^z \rangle
$$
chain_{v_i}(x, z) is therefore Υ_k because the actions a_1, ..., a_{v_i}^y are instantiated k times: \((k - 1)\) times in \(\delta_{v_i}(x, x)\) and once in \(\delta_{v_i}(x, y)\). The chain \(\delta_{v_i}(x, x)\) can be seen as the unfolding of Cycle(\(v_i\)) from x to x, while \(\delta_{v_i}(x, y)\) is an unfolding from x to y. The chain \(\delta_{v_i}(y, z)\) represents an external path to the cycle, from the post condition of a root node \(y = \text{post}(o_{v_i}^y)\) to the goal \(s_\ast[v_i] = \text{post}(o_{v_i}^x) = z\). This external chain is obviously Υ_1.

In practice, what causes the multiple passage in the graph cycle are the requests of the requestable values that are satisfied by operators belonging to the cycle. If there is an alternation of appearance in the solution plan for instance: one value is requested, then another, then the first again, etc. That will cause the planner to create chains with several occurrences of the same actions. A typical example of this phenomena is the Gray Code: the logical state of each binary digit switch from 0 to 1 and from 1 to 0 constantly. Another good example is given in Appendix C.4c, the “odd” operators \((o_1, o_3, o_5)\) have a positive prevail-condition on \(v_i\) while the “even” operators \((o_2, o_4, o_6)\) have a negative one. The solution plan of the appendix problem instance is composed of the sequence of actions \(\langle a_1, a_2, ..., a_6 \rangle\), so Cycle(\(v_i\)) = \{o_{v_i}^-, o_{v_i}^+\} has to be unfolded three times in order to produce a chain of actions that satisfies the prevail-condition on \(v_i\) of each action of the sequence by alternation. Another possibility is that the requestable values are required by the solution plan in an inverse order to that of the operators that satisfy them. That will also cause the planner to create chains of actions with many occurrences of the same actions. It can also explain why chains of actions are at most Υ_2 when the restriction (S) is applied. There is at most one requestable value per state variable, so the alternation of appearance or the appearance in an inverse order have no meaning since there is only one requestable value per state variable domain. This value only needs to be reached once, at most, and there is no other requestable value that will require the planner to create a chain to reach it, which, in return, also prevents the planner from creating another chain of actions to reach the already reached once requestable value. Lastly, the Gray Code is a good example to illustrate that two requestable operators looping together are sufficient to create intractable problems.

4.3.2 Υ_1 Truth Criterion

Based on the notion of truth criterion described in subsection 3.2.2 and the adaptation of the terms “establisher”, “clobberer” and “action to assert” for SAS-PU problems given by Definition 3.2.5, I neatly described in this subsection all the situations to handle for SAS-PU problems whose minimal solution plan is Υ_1. I especially describe situations that were not possible with the restriction (S). Firstly, based on the previous subsection, chains of actions are Υ_1 iff they respect the following Lemma:

**Lemma 4.3.1.** Let \(v_i \in M\), \(x, y, z \in D_{v_i}\), \(o_{v_i}^x, o_{v_i}^y \in \text{Cycle}(v_i)\) such that \(o_{v_i}^y\) is a root node of a directed tree containing \(o_{v_i}^x \notin \text{Cycle}(v_i)\). \(\text{chain}_{v_i}(x, z)\), a chain of actions between x
and $z$, is $\Upsilon_1$ iff:

\[ \text{chain}_{v_i}(x, z) = \delta_{v_i}(x, y) \oplus \delta_{v_i}(y, z) \]

\[ \land \forall a \in \delta_{v_i}(x, y), \ ope(a) \in \text{Cycle}(v_i) \]

\[ \land \forall b \in \delta_{v_i}(y, z), \ ope(b) \notin \text{Cycle}(v_i) \]

(4.2)

**Proof.** Direct from the equation (4.1), by taking into account that $k = 1$. \hfill \square

Depending on the relation between $x$, $y$ and $z$, a $\Upsilon_1$ chain of actions will have the possible forms described in Table 4.3.

| $x \neq y$ | $y \neq z$ | $\delta_{v_i}(x, y) \oplus \delta_{v_i}(y, z)$ | $\forall a \in \delta_{v_i}(x, y), \ ope(a) \in \text{Cycle}(v_i)$ | (a) |
| $y = z$ | $\delta_{v_i}(x, y) \oplus \langle \rangle$ | $\forall a \in \delta_{v_i}(x, y), \ ope(a) \in \text{Cycle}(v_i)$ | (b) |
| $x = y$ | $y \neq z$ | $\delta_{v_i}(y, y) \oplus \delta_{v_i}(y, z)$ | $\forall a \in \delta_{v_i}(y, y), \ ope(a) \in \text{Cycle}(v_i)$ | (c) |
| $\langle \rangle \oplus \delta_{v_i}(y, z)$ | $\forall b \in \delta_{v_i}(y, z), \ ope(b) \notin \text{Cycle}(v_i)$ | (d) |
| $y = z$ | $\delta_{v_i}(y, y) \oplus \langle \rangle$ | $\forall a \in \delta_{v_i}(y, y), \ ope(a) \in \text{Cycle}(v_i)$ | (e) |
| $\langle \rangle$ | | | (f) |

Table 4.3: Let $v_i \in \mathcal{M}$, $x, y, z \in \mathcal{D}_{v_i}$. The possible forms of a $\Upsilon_1$ action chain depending on the relations between $x$, $y$ and $z$. $x$ is the start value, $z$ the goal one and $y$ the root one. $\sigma^x_{v_i}, \sigma^y_{v_i} \in \text{Cycle}(v_i)$ and $\sigma^z_{v_i} \notin \text{Cycle}(v_i)$.

Let consider $\sigma^x_{v_i}, \sigma^y_{v_i}, \sigma^z_{v_i}, a_1, a_2$ and $a_3$ the action instances of the operators $\sigma^x_{v_i}, \sigma^y_{v_i}, \sigma^z_{v_i}, o_1, o_2$ and $o_3$, respectively, from Figure 4.4, to illustrate Table 4.3. We have:

(a) $\text{chain}_{v_i}(x, z) = \delta_{v_i}(x, y) \oplus \delta_{v_i}(y, z) = \langle a_1, ..., a^y_{v_i} \rangle \oplus \langle a_3, ..., a^z_{v_i} \rangle = \delta_{v_i}(x, z)$

(b) $\text{chain}_{v_i}(x, y) = \delta_{v_i}(x, y) = \langle a_1, ..., a^y_{v_i} \rangle$

(c) $\text{chain}_{v_i}(y, z) = \delta_{v_i}(y, y) \oplus \delta_{v_i}(y, z) = \langle a_2, ..., a^x_{v_i}, a_1, ..., a^y_{v_i} \rangle \oplus \langle a_3, ..., a^z_{v_i} \rangle$

(d) $\text{chain}_{v_i}(y, z) = \delta_{v_i}(y, z) = \langle a_3, ..., a^z_{v_i} \rangle$

(e) $\text{chain}_{v_i}(y, y) = \delta_{v_i}(y, y) = \langle a_2, ..., a^x_{v_i}, a_1, ..., a^y_{v_i} \rangle$

(f) $\text{chain}_{v_i}(y, y) = \langle \rangle$

(c) and (e) are chains that both fully unfold the cycle, i.e. they unfold the cycle from $y$ to $y$. (a) and (b), on the contrary, partially unfold it, from $x$ to $y$. Finally, (d) is an external path. With $s_0[v_i]$ and $s_*[v_i]$ as parameters for the chain of actions, and based on the equations (a) to (e) (the equation (f) is trivial), there are three cases to handle in order to define a proper truth criterion for $\Upsilon_1$ solution plan in SAS.
1. \( \sigma_1 = \text{chain}_{v_i}(s_0[v_i], s_*[v_i]) = \delta_{v_i}(s_0[v_i], s_*[v_i]) \) (which represents the chains (a), (b) and (d))

2. \( \sigma_2 = \text{chain}_{v_i}(s_0[v_i], s_0[v_i]) = \delta_{v_i}(s_0[v_i], s_0[v_i]) \) (which represents (e))

3. \( \sigma_3 = \text{chain}_{v_i}(s_0[v_i], s_*[v_i]) = \delta_{v_i}(s_0[v_i], s_0[v_i]) \oplus \delta_{v_i}(s_0[v_i], s_*[v_i]) \) (which represents (c)).

We have \( \sigma_3 = \sigma_2 + \sigma_1 \).

Let \( v_i \in M \), I denote \( q \in \mathcal{R}_v^O \) the proposition to assert and \( a \in A \) the action to assert such that \( \text{prv}(a)[v_i] = q \). Similar to C. Bäckström’s IMP, and based on Definition 3.2.5, the estisher of \( q \), that I henceforth denote \( T \), and its clobberer, that I henceforth denote \( C \), both affect the state variable \( v_i \) such that: \( \text{post}(T)[v_i] = q \) and \( \text{pre}(C)[v_i] = q \). So it means that \( T \) and \( C \) are in the same chain of actions between \( s_0[v_i] \) and \( s_*[v_i] \): \( T, C \in \text{chain}_{v_i}(s_0[v_i], s_*[v_i]) \). Therefore, to define any promotion or demotion technique, it is relevant to study the chains of actions of an action plan.

Case (1): \( \sigma_1 = \text{chain}_{v_i}(s_0[v_i], s_*[v_i]) = \delta_{v_i}(s_0[v_i], s_*[v_i]) \). For each \( T \in \delta_{v_i}(s_0[v_i], s_*[v_i]) \), \( T \) is post-unique due to \( T_1 \) and due to the fact operators are post-unique. Each value \( q \) between \( s_0[v_i] \) and \( s_*[v_i] \) is therefore established once, either by \( s_0[v_i] \), or by \( T \in \delta_{v_i}(s_0[v_i], s_*[v_i]) \). By definition of the clobberer, \( q \) is clobbered once except when \( q = s_*[v_i] \), where there is no clobberer. There are three possibilities:

- If \( q \neq s_0[v_i] \land q \neq s_*[v_i] \), \( a \) is established by \( T \) and promoted before \( C \):
  \[
  \exists T, C \in \text{chain}_{v_i}(s_0[v_i], s_*[v_i]) \text{ s.t. } T \prec C, \ (T \prec a \land a \prec C) \tag{4.3}
  \]

- If \( q = s_0[v_i] \), \( a \) is established by the start and promoted before \( C \) which is the first action of the minimal chain:
  \[
  C = \text{First}(\delta_{v_i}(s_0[v_i], s_*[v_i])), \ a \prec C \tag{4.4}
  \]

- If \( q = s_*[v_i] \), \( a \) is established by \( T \) which is the last action of the minimal chain:
  \[
  T = \text{Last}(\delta_{v_i}(s_0[v_i], s_*[v_i])), \ T \prec a \tag{4.5}
  \]

Case (2): \( \sigma_2 = \text{chain}_{v_i}(s_0[v_i], s_0[v_i]) = \delta_{v_i}(s_0[v_i], s_0[v_i]) \). If \( q = s_0[v_i] \) then \( q \) is established twice: once with the start and once with \( T \) s.t. \( \text{post}(T)[v_i] = s_*[v_i] = s_0[v_i] \). That is, \( q \) is established by the start or by \( \text{Last}(\delta_{v_i}(s_0[v_i], s_*[v_i])) \). Then, the proposition \( q \) is clobbered once with \( C = \text{First}(\delta_{v_i}(s_0[v_i], s_*[v_i])) \). So, if \( q = s_0[v_i] \), there is two possibilities to assert the action \( a \):

- The action \( a \) is established by the start and promoted before \( C \) which is the first action of the minimal chain:
  \[
  C = \text{First}(\delta_{v_i}(s_0[v_i], s_*[v_i])), \ a \prec C \tag{4.6}
  \]

- The action \( a \) is established by \( T \), which is the last action of the chain, and denoted after \( C \):
  \[
  \begin{cases}
  T = \text{Last}(\delta_{v_i}(s_0[v_i], s_*[v_i])), \ T \prec a \\
  C = \text{First}(\delta_{v_i}(s_0[v_i], s_*[v_i])), \ C \prec T
  \end{cases} \tag{4.7}
  \]
It is worth noticing that: \( \delta_{v_i}(s_0[v_i], s_0[v_i]) = \langle C, ..., T \rangle \). Therefore, \( C \prec T \) is redundant because \( C \) is already ordered before \( T \) in the minimal chain of actions. Ordering \( a \) after \( T \) is therefore sufficient in these case as the demotion technique is spontaneously achieved by the building of \( \delta_{v_i}(s_0[v_i], s_0[v_i]) \).

If \( q \neq s_0[v_i] \), then the assertion of the action \( a \) follows the rule given by the equation (4.3).

**Case (3):** \( \sigma_3 = chain_{vi}(s_0[v_i], s_0[v_i]) = \delta_{v_i}(s_0[v_i], s_0[v_i]) \oplus \delta_{v_i}(s_0[v_i], s_0[v_i]) \). Similar to case (2), if \( q = s_0[v_i] \), then the proposition has two establishers: \( s_0[v_i] \) and \( T = Last(\delta_{v_i}(s_0[v_i], s_0[v_i])) \). Unlike (2), the proposition \( q \) has two clobberers: \( C_1 \) and \( C_2 \) such that \( C_1 = First(\delta_{v_i}(s_0[v_i], s_0[v_i])) \) and \( C_2 = First(\delta_{v_i}(s_0[v_i], s_0[v_i])) \). The chain between \( s_0[v_i] \) and \( s_0[v_i] \) can be written:

\[
\text{chain}_{vi}(s_0[v_i], s_0[v_i]) = \delta_{v_i}(s_0[v_i], s_0[v_i]) \oplus \delta_{v_i}(s_0[v_i], s_0[v_i]) = \langle C_1, ..., T \rangle \oplus \langle C_2, ..., a_{v_i}^s[v_i] \rangle
\]

So, if \( q = s_0[v_i] \), then there are two possibilities to assert \( a \):

- The action \( a \) is established by the start and promoted before \( C_1 \) and before \( C_2 \). Because \( C_1 \) is already ordered before \( C_2 \) in the chain of actions, the ordering of \( a \) with \( C_1 \) only is sufficient:

\[
C_1 = First(\delta_{v_i}(s_0[v_i], s_0[v_i])), \ a \prec C_1 \tag{4.8}
\]

- The action \( a \) is established by \( T \), demoted after \( C_1 \) and promoted before \( C_2 \). Similar to the equation (4.7), the demotion with \( C_1 \) is not necessary because \( C_1 \) is already ordered before \( T \) via \( \delta_{v_i}(s_0[v_i], s_0[v_i]) \). Eventually:

\[
\land \left\{ \begin{array}{l}
T = Last(\delta_{v_i}(s_0[v_i], s_0[v_i])), \ T \prec a \\
C_2 = First(\delta_{v_i}(s_0[v_i], s_0[v_i])), \ a \prec C_2
\end{array} \right. \tag{4.9}
\]

If \( q \neq s_0[v_i] \) and \( q \neq s_0[v_i] \) then the assertion of the action \( a \) follows the rule given by the equation (4.3). If \( q = s_0[v_i] \), it follows that of the equation (4.5).

It ensues from these different cases the following definition for the \( \Upsilon_1 \)-Truth Criterion.

**Theorem 4.3.1** (\( \Upsilon_1 \)-Truth Criterion). Let \( \Delta = (A, \prec) \) be a \( \Upsilon_1 \) non-linear solution plan for the SAS-PU problem instance \( \Pi = (\Phi, s_0, s_1) \) and \( \sigma_{vi} = (A(v_i), \prec) \) a chain of actions affecting \( v_i \in M \). \( \Delta \) is a \( \Upsilon_1 \) correct non-linear plan iff,

For each \( v_i \in M \),

1. \( \sigma_{vi} = chain_{vi}(s_0[v_i], s_0[v_i]) \subset \Delta \Rightarrow \sigma_{vi} \) is \( \Upsilon_1 \) or empty.

2. \( \forall a \in A \setminus A(v_i), \ p = \text{prv}(a)[v_i], \ (\exists T \in \sigma_{vi}, \ p = \text{post}(T)[v_i] \land T \prec a) \land (\exists C \in \sigma_{vi}, \ p = \text{pre}(C)[v_i] \land (T \prec C \land p = s_0[v_i]) \Rightarrow (C \text{ is unique } \land a \prec C) \)
Proof. A SAS-PU solution plan is a collection of chain of actions with additional relations between the chains via the prevail-conditions. A solution plan is therefore $\Upsilon_1$ if each non-empty chain of actions composing it is $\Upsilon_1$. The second part is an adaption of Theorem 1 [33, p.4] to the class SAS-PU with Definition 3.2.5 and the equations (4.3) to (4.9). The establishment process ($T \prec a$) is proven by the equations (4.3), (4.5), (4.7) and (4.9). The promotion process ($a \prec C$) is proven by the equations (4.3) (4.4), (4.6), (4.8) and (4.9). Concerning the uniqueness of $C$, there is only one case (cf. case (3)) where there are 2 clobberers, otherwise, cases (1) and (2) prove that $C$, if it exists, is simply unique. Concerning case (3) and its two clobberers $C_1$ and $C_2$, because a chain of actions is linear, only $C_1$ clobbers the start ($p = s_0[v_i] = \text{pre}(C_1)$) while only $C_2$ clobbers the post-condition of the establisher $T$ ($p = \text{post}(T) = \text{pre}(C_2)$, i.e. $T \prec C_2$). Although $C_1$ and $C_2$ clobber the same proposition $p$ in case (3), ordering $a$ in function of $C_1$ or in function of $C_2$ is sufficient because the ordering in function of the second will be set automatically by transitivity via $\sigma_{v_i}$.

- Equation (4.8): $a \prec C_1 \Rightarrow a \prec C_2$, because $C_1$ is ordered before $C_2$ in $\sigma_{v_i}$.

- Equation (4.9): $T \prec a \wedge a \prec C_2 \Rightarrow C_1 \prec T$, because $C_1$ is already ordered before $T$ in $\sigma_{v_i}$.

It results there is a unique clobberer that needs an additional order with the action to assert, and this unique additional order is set by promotion.

This truth criterion seems to be adaptable to $\Upsilon_k$, with for each $\sigma_{v_i} \subset \Delta$, $\sigma_{v_i}$ is either empty or at most $\Upsilon_k$, and point 2 remains true. Point 2 may be proven by recursion with:

$$\sigma_{v_i} = \text{chain}_{v_i}(s_0[v_i], s_s[v_i]) = \langle b, ..., T_1 \rangle \oplus \langle C_1, ..., T_2 \rangle \oplus ... \oplus \langle C_{k-1}, ..., T_k \rangle \oplus \langle C_{k}, ..., a^*_v[v_i] \rangle$$

with $b \in A(v_i), \text{pre}(b)[v_i] = s_0[v_i]$

That is, the action to assert, denoted $a$, has $k$ clobberers, denoted $C_k$. Let $l \in \mathbb{N}$ such that $1 < l \leq k$, if $T_l$ establishes a prevail-condition of $a$, then $\forall n \in [l, l+1]$, $a$ is demoted after $C_n$, i.e. $C_n \prec T_l$, and $\forall m \in [l, k]$, $a$ is promoted before $C_m$, i.e. $a \prec C_m$. But similar to the argument used in the proof of the $\Upsilon_1$-Truth Criterion, if $T_l$ establishes $a$ and $T_l \prec C_l$ in $\sigma_{v_i}$, then it is sufficient to promote $a$ before $C_l$. $C_l$ is also the unique clobberer that succeeds $T_l$ in $\sigma_{v_i}$. The other clobberers are already properly ordered with respect to $T_l$ and $C_l$ in $\sigma_{v_i}$.

It should be notice that the situations described in cases (2) and (3) when $q = s_0[v_i]$ does not exist with the restriction (S). Indeed, in such situations, there is no need for the minimal chain $\delta_{v_i}(s_0[v_i], s_0[v_i])$ to be built, because (S) insures there is a unique subgoal in $v_i$, which, here, is the start. So, $\delta_{v_i}(s_0[v_i], s_0[v_i]) \neq \langle \rangle$ increases the length of the chain of actions for nothing because it passes through no subgoal. It results that $A$ will not be minimal if actions of $\delta_{v_i}(s_0[v_i], s_0[v_i])$ are added, thus contradicting C. Bäckström’s IMP. The situation of cases (2) and (3) when $q = s_0[v_i]$ therefore occur iff $|R^O_{v_i}| \geq 2$ and
CHAPTER 4. TOPOLOGICAL PLANNING

\( q \in \mathcal{R}_{v_i}^O, q = s_0[v_i] \) and if \( \delta_{v_i}(s_0[v_i], s_0[v_i]) \) passes through other requested values in \( \mathcal{R}_{v_i}^O \) \( \setminus \{q\} \) to solve the problem.

**Corollary 4.3.1.** If \( q = s_0[v_i] \) is a proposition to assert in a \( \Upsilon_1 \) solution plan \( \Delta \) and it exists \( a^q_{v_i} \in \Delta \) s.t. \( \text{ope}(a^q_{v_i}) \in \text{Cycle}(v_i) \), then \( q \) is either established by the start or by the unique establisher in \( \Delta \), namely \( a^q_{v_i} \).

Otherwise, \( q \) has a unique establisher:

\[
\begin{cases} 
\text{the start} & \text{if } q = s_0[v_i] \\
\ a^q_{v_i} & \text{if } q \neq s_0[v_i] 
\end{cases}
\]

**Proof.** Direct from the explanation of the equations (4.3) to (4.9).

### 4.4 A New Structural Restriction: \( C_k \)

\( \Upsilon_1 \) is an output restriction, however, and all SAS problems is at least \( \Upsilon_1 \) (Corollary 4.1.1). In term of development for planning projects, it results that \( \Upsilon_1 \) does not give rules or instructions to create classes of problems without exponentially sized-minimal plans. The only way to ensure a problem is at most \( \Upsilon_1 \) is to run a planner that respects the \( \Upsilon_1 \)-Truth Criterion on every instance of the problem. Another solution is to find a new restriction that, once combined with the syntactical restrictions (P) and (U), produces a class where each problem is at most \( \Upsilon_1 \). In this section, I thereby introduce a new structural restriction that produces such problems.

\( C_k \) is a structural restriction that focuses on the unique cycle of each domain transition graph. It is defined as follows:

**Definition 4.4.1 (Structural Restriction \( C_k \)).** Let \( \Phi = (M, S, O) \) a structure of the class SAS-PU, \( v_i \in M \) and \( k \in \mathbb{N} \). I denote \( C_k \) the structural restriction that limits to at most \( k \) the number of distinct operators inside each \( \text{Cycle}(v_i) \) having at least one requestable operator. Let \( o^p_{v_i} \in O \), if \( o^p_{v_i} \in \mathcal{RO} \wedge o^p_{v_i} \in \text{Cycle}(v_i) \), then \( |\text{Cycle}(v_i)| \leq k \).

\( C_k \) is a restriction on structures of the class SAS-PU, so it is necessary combined with (P) and (U). The restriction \( C_k \) is obviously compatible with the restriction (S) as well. For example, the Horse Breeder and the Gray Code problems are of the class SAS-PUC2:

- \( \mathcal{RO}_{\text{HB}} = \{\text{TakeHaystack}, \text{DropBucket}, \text{TakeBucket}\} \) with \( \text{TakeHaystack} \in \text{Cycle}(v_0) \) and \( \text{DropBucket}, \text{TakeBucket} \in \text{Cycle}(v_1) \). Therefore, only \( \text{Cycle}(v_0) \) and \( \text{Cycle}(v_1) \) are affected by the restriction \( C_2 \): \( |\text{Cycle}(v_0)| \leq 2, |\text{Cycle}(v_1)| \leq 2 \). \( \text{Cycle}(v_2) \) is not affected by the restriction. Even though it is empty, its size could have been greater than 2.

- \( \mathcal{RO}_{\text{GC}} = \{o^+_{v_0}, o^+_{v_0}, o^+_{v_1}\} \) with \( o^-_{v_0}, o^+_{v_0} \in \text{Cycle}(v_0) \) and \( o^+_{v_1} \in \text{Cycle}(v_1) \). Therefore, only these two cycles are affected by the restriction \( C_2 \): \( |\text{Cycle}(v_0)| \leq 2, |\text{Cycle}(v_1)| \leq 2 \).

It should be noticed that SAS-PUC1 problems do not exist due to the restriction (S4) of the SAS-structure. Let \( v_i \in M \), if an operator \( o \) is requestable and inside \( \text{Cycle}(v_i) \), \( C_1 \) imposes
that: Cycle($v_i$) ≤ 1, i.e. Cycle($v_i$) = {o} which means o is looping with itself, which is not possible due to (S4): pre(o)[v_i] ≠ u ⇒ pre(o)[v_i] ≠ post(o)[v_i]. In the following, I therefore study SAS-PUC_k problems with k = 0, k = 2 and k ≥ 3. I first introduce some useful theorems and lemmata for what follows.

**Theorem 4.4.1.** Let $k \in \mathbb{N}$,

1. SAS-PUC$_{(k-1)}$ ⊂ SAS-PUC$_k$
2. SAS-PUSC$_k$ ⊂ SAS-PUC$_k$

**Proof.** Let $v_i \in \mathcal{M}$ and $k \in \mathbb{N}$,

1. Let Φ = ($\mathcal{M}, \mathcal{S}, \mathcal{O}$) a structure of the class SAS-PUC$_{k-1}$. Let $\mathcal{C}$ the set of cycles having at least one requestable operator, then $\max_{c \in \mathcal{C}} |c| \leq k - 1$. Φ is therefore also of the class SAS-PUC$_k$, because $\max_{c \in \mathcal{C}} |c| < k$. Hence, SAS-PUC$_{(k-1)}$ ⊂ SAS-PUC$_k$.

2. The restriction $\mathcal{C}_k$ restricts the number of (PU) operators inside non-empty Cycle($v_i$) having at least one requestable operator. So, with (S), $\mathcal{C}_k$ merely restricts the number of operators inside Cycle($v_i$) having exactly one requestable operator. Hence, SAS-PUSC$_k$ ⊂ SAS-PUC$_k$.

In the following, if a problem is exactly of the class SAS-PUC$_k$, then the problem is in SAS-PUC$_k \setminus$ SAS-PUC$_{k-1}$. In other words, the structure of such a problem possesses at least one cycle with $k$ operators in it and one of them at least is requestable. **Theorem 4.4.1** implies that a problem that is SAS-PUC$_2$ is also a SAS-PUC$_3$ problem, i.e. it is a SAS-PUC$_k$ problem with $k \geq 2$. This precision is important for what follows because, depending on $k \in \mathbb{N}$, problems that are exactly of the class SAS-PUC$_k$ may have characteristics that problems of the class SAS-PUC$_{k+1}$ do not have.

**Theorem 4.4.2.** The limit of the class SAS-PUC$_k$ as $k$ approaches infinity is the class SAS-PU:

$$\lim_{k \to \infty} (\text{SAS-PUC}_k) = \text{SAS-PU}$$

**Proof.** According to **Lemma 4.2.2**, the domain operator graphs of (PU) operators are either a directed tree or a directed graph with a single cycle. The class SAS-PU accepts all these domain operator graphs whereas the restriction $\mathcal{C}_k$ discards those whose unique cycle has more than $k$ nodes. Thus, the higher $k$, the fewer cyclic domain operator graphs discarded. Therefore, when $k$ approaches infinity, SAS-PUC$_k$ accepts all the domain operator graphs which implies: $\lim_{k \to \infty} (\text{SAS-PUC}_k) = \text{SAS-PU}$. 

**Corollary 4.4.1.** Let $k \in \mathbb{N}$, SAS-PUC$_k$ ⊆ SAS-PU.
Proof. Direct from Theorem 4.4.1 (1.) and Theorem 4.4.2.

In the following subsections, I first present the class of problems SAS-PUC\(_0\) and prove that all these problems are \(\Upsilon_1\). I then present the class of problems SAS-PUC\(_2\) and prove that this class is generally intractable. With additional restrictions, I present SAS-PUC\(_S^2\) and SAS-PUC\(_*^2\) problems which are subclasses of SAS-PUC\(_2\) and whose problems are \(\Upsilon_1\). Finally, I prove that problems that are exactly of the class SAS-PUC\(_k\) with \(k \geq 3\) are at least \(\Upsilon_2\). That is, every SAS-PUC\(_k\) problem with \(k \geq 3\) has at least one problem instance whose minimal solution plan is \(\Upsilon_2\).

### 4.4.1 The Class of Problems SAS-PUC\(_0\)

**Theorem 4.4.3.** SAS-PUC\(_0\) problems are \(\Upsilon_1\).

*Proof. Let \(v_i \in \mathcal{M}, p \in \mathcal{R}^O_v\) a requestable value. \(C_0\) implies that there is no requestable operator in a cycle: \(\forall o^p_{v_i} \in \mathcal{RO}, \forall v_i \in \mathcal{M}, o^p_{v_i} \notin \text{Cycle}(v_i)\). Therefore, every chain of actions in a minimal solution plan solving an instance \(\Pi = (\Phi, s_0, s_*)\) is of the form:

\[
\sigma_{v_i} = \text{chain}_{v_i}(s_0[v_i], s_*[v_i]) = \delta_{v_i}(s_0[v_i], p) \oplus \delta_{v_i}(p, s_*[v_i]) = \delta_{v_i}(s_0[v_i], s_*[v_i])
\]

If there is a requestable value \(p\), then the establisher of \(p\) is either the start or an action of \(\sigma_{v_i}\). Otherwise, it means the establisher must be instantiated from a requestable operator that are in the cycle of the domain operator graph, which cannot occur due to \(C_0\). Therefore, for each \(v_i \in \mathcal{M}\), if \(\sigma_{v_i}\) is a non-empty chain of actions of the plan, then it is the minimal chain from the start to the goal: \(\delta_{v_i}(s_0[v_i], s_*[v_i])\). Therefore, every non-empty chain of a plan solving an instance \(\Pi\) is \(\Upsilon_1\). It implies that the minimal solution plan of every solvable instance of any SAS-PUC\(_0\) problem is \(\Upsilon_1\). \qed

This class of problems does not prevent the use of the unique cycle of the domain operator graph, but prevents the definition of requestable operators inside that cycle. It results that actions establishing prevail-conditions, which are instances of requestable operators, will always be part of the minimal action chain between the start and the goal state, which is unique (cf. **Lemma 3.2.2**) and \(\Upsilon_1\) (cf. **Lemma 3.2.1**).

It also results that every problem structure whose domain operator graphs are directed trees (cf. **Lemma 4.2.2**), i.e. \(\forall v_i \in \mathcal{M}, \text{Cycle}(v_i) = \emptyset\), is of the class SAS-PUC\(_0\).

### 4.4.2 The Class of Problems SAS-PUC\(_2\)

**Theorem 4.4.4.** The class of problems SAS-PUB is included in the class of problems SAS-PUC\(_2\).
CHAPTER 4. TOPOLOGICAL PLANNING

Proof. The restriction $C_2$ restricts the number of operators to at most 2 in the unique cycle of domain operator graphs that possess requestable operators. The restriction (B) restricts every domain operator graph to only have two operators, i.e. two nodes. In other words, (B) implies that, for each $v_i \in M$, $|O(v_i)| \leq 2$ while the restriction $C_2$ implies $|\text{Cycle}(v_i)| \leq 2$. As $\text{Cycle}(v_i) \subseteq O(v_i)$, it results that SAS-PUB $\subseteq$ SAS-PUC. \qed

Corollary 4.4.2. The class of problems SAS-PUC is intractable.

Proof. The class of problems SAS-PUB is intractable because there exists problems such that the minimal solution plan is of length $\Theta(2^{|M|})$ (Theorem 6.14 [6, p.138]). As SAS-PUB is included in SAS-PUC (Theorem 4.4.4), the latter is intractable for the exact same reason. \qed

Corollary 4.4.3. The class of problems SAS-PUC$_k$, with $k \geq 2$ is intractable.

Proof. Let $k \in \mathbb{N}$, Theorem 4.4.1 implies SAS-PUC$_2$ $\subset$ SAS-PUC$_k$ and SAS-PUC$_k$ is therefore intractable according to Corollary 4.4.2. \qed

Table 4.4 presents the operator set of a SAS-PUC$_2$ problem:

<table>
<thead>
<tr>
<th>$O$</th>
<th>pre()</th>
<th>post()</th>
<th>prv()</th>
</tr>
</thead>
<tbody>
<tr>
<td>$o^1_{v_0}$</td>
<td>$v_0 = 0$</td>
<td>$v_0 = 1$</td>
<td>$(u, 1, u)$</td>
</tr>
<tr>
<td>$o^0_{v_1}$</td>
<td>$v_1 = 1$</td>
<td>$v_1 = 0$</td>
<td>$(u, u, u)$</td>
</tr>
<tr>
<td>$o^1_{v_1}$</td>
<td>$v_1 = 0$</td>
<td>$v_1 = 1$</td>
<td>$(u, u, u)$</td>
</tr>
<tr>
<td>$o^1_{v_2}$</td>
<td>$v_2 = 0$</td>
<td>$v_2 = 1$</td>
<td>$(u, 0, u)$</td>
</tr>
</tbody>
</table>

Table 4.4: Operators of a SAS-PUC$_2$ structure $\Phi$. $\Pi = (\Phi, s_0 = (0, 0, 0), s_* = (1, 1, 1))$ is a possible problem instance.

There are exactly 64 different problem instances with this structure ($2^3$ different starts times $2^3$ different goals), 40 of which are solvable. It can be proved by hand or with Algorithm 4 that each of the solvable problem instances has a minimal solution plan that is $\Upsilon_1$. Therefore, it exists at least one problem of the class SAS-PUC$_2$ that is $\Upsilon_1$. In other words, it may exist subclasses of SAS-PUC$_2$ whose problems are $\Upsilon_1$. The following two subsections present such subclasses.

4.4.3 The Class of Problems SAS-PUC$_2^S$

Definition 4.4.2. A SAS-PU structure is restricted by $C^S_2$ iff:

- it is restricted by $C_2$;
- there is at most one requestable operator per domain operator graph cycle.
Let \( v_i \in \mathcal{M} \) such that \( \text{Cycle}(v_i) \neq \emptyset \) is a cycle affected by the structural restriction \( C_2 \), the size of \( \text{Cycle}(v_i) \) is therefore: \(|\text{Cycle}(v_i)| = 2\). Let denote \( o_{v_i}^− \) and \( o_{v_i}^+ \) the two operators of \( \text{Cycle}(v_i) \) such that \( \text{Cycle}(v_i) = \{o_{v_i}^−, o_{v_i}^+\} \). At least one of the two, let consider \( o_{v_i}^+ \), is a requestable operator (Definition 4.4.1). If \( \text{Cycle}(v_i) \) respects the structural restriction \( C_2^* \), then \( o_{v_i}^+ \) is the single requestable operator of \( \text{Cycle}(v_i) \). The notion \( C_2^* \) is directly inspired from the restriction (S). The main difference with the syntactical restriction (S) is that the set of requestable value for a given state variable can be greater than 1. Let \( v_i \in \mathcal{M} \), we can have:

\[
\mathcal{R}^\mathcal{O}_{v_i} = \{+, p_1, \ldots, p_{\bar{n}}\}, \text{ with } \bar{n} \leq |D_{v_i}| - 2
\]

such that:

\[
o_{v_i}^+ \in \mathcal{RO}(v_i) \cup \text{Cycle}(v_i) \quad \forall k \in [1, \bar{n}], o_{v_i}^{p_k} \in \mathcal{RO}(v_i) \setminus \text{Cycle}(v_i)
\]

That is, each requestable operator \( o_{v_i}^{p_k} \) is not in the cycle of the domain operator graph, which implies these operators can only have at most one action instance each, no matter the problem instance. It remains to show that \( o_{v_i}^+ \), the single requestable operator of \( \text{Cycle}(v_i) \), can also be instantiated at most once, no matter the \( \text{SAS-PUC}_2^* \) problem instance.

In \( \text{SAS}^+\text{-PUS} \) problems, \(|\mathcal{R}^\mathcal{O}_{v_i}| \leq 1 \) for each \( v_i \in \mathcal{M} \). Let denote \( p \) the single value of the domain \( D_{v_i} \), such that \( \mathcal{R}^\mathcal{O}_{v_i} = \{p\} \). When defining the IMP, C. Bäckström described \( p \) as a subgoal that each planner respecting the IMP may have to consider in order to have a correct minimal solution plan for a given SAS-PUO problem instance. With D. Chapman’s terms, \( p \) is a proposition to assert, i.e. \( p \) must be established and declobbered if it is a prevail-condition of an action \( a \) in the solution plan. If I apply these terms to each \( p \in \{+, p_1, \ldots, p_{\bar{n}}\} \), then there are all possible propositions to assert before reaching the goal state.

**Definition 4.4.3.** \( v_i \in \mathcal{M} \), \( x, y \in \mathcal{R}^\mathcal{O}_{v_i} \) s.t. \( x \neq y \). \( \delta_{v_i}(x,y) \) is a meaningful minimal chain if \( x \) is a clobbering situation and \( y \) is an establishing one.

Let \( \mathcal{R}^\mathcal{O}_{v_i} = \{+, p_1, \ldots, p_{\bar{n}}\} \) such that \( \mathcal{RO}(v_i) \cup \text{Cycle}(v_i) = \{o_{v_i}^+\} \) and \( \mathcal{RO}(v_i) \setminus \text{Cycle}(v_i) = \{p_1, \ldots, p_{\bar{n}}\} \). For \( 1 \leq k < \bar{n} \), let consider \( o_{v_i}^{p_k} \prec o_{v_i}^{p_{k+1}} \) in \( \mathcal{G}_\mathcal{O}(v_i) \). It implies that \( o_{v_i}^+ \prec o_{v_i}^{p_i} \) in \( \mathcal{G}_\mathcal{O}(v_i) \). It results from **Definition 4.4.3** that:

- \( \delta_{v_i}(s_0[v_i], s_*[v_i]) \) is meaningful.
- \( \forall p \in \mathcal{RO}(v_i), \delta_{v_i}(s_0[v_i], p) \) and \( \delta_{v_i}(p, s_*[v_i]) \) are both meaningful.
- \( \forall p_i, p_j \in \{p_1, \ldots, p_{\bar{n}}\}, \delta_{v_i}(p_i, p_j) \) is meaningful iff \( 1 \leq i < j \leq \bar{n} \). And \( \delta_{v_i}(p_j, p_i) = () \).
- \( \forall p \in \{p_1, \ldots, p_{\bar{n}}\}, \delta_{v_i}(+, p) \) is meaningful and \( \delta_{v_i}(p, +) = () \).

I used this notion of meaningfulness to prove that \( \text{SAS-PUC}_2^* \) problems are \( \Upsilon_1 \).

**Theorem 4.4.5.** \( \text{SAS-PUC}_2^* \) problems are \( \Upsilon_1 \).
The proof is provided in Appendix C.1.

In comparison with the restriction (S), the structural restriction $C_2^S$, although less restrictive for operators outside the cycle of their domain operator graph, does not allow On/Off situations neither. Problems of the class SAS-PUC$_2^S$ do not allow two requestable operators affecting the same state variable to be in the unique cycle of their domain operator graph. So in a problem instance, if an action required the switch to be on and another action the switch to be off, then there is one operator to instantiate “turn on” and one operator to instantiate “turn off”. The two operators create a cycle and are both requestable, therefore the problem is not $C_2^S$. The main purpose of this class of problems, however, is to prove the following class.

\subsection{The Class of Problems SAS-PUC$_2^*$}

A consequence of the $\Upsilon_1$-Truth Criterion (Theorem 4.3.1), is that there is at most one clobberer per prevail-condition of an action that needs an additional order, by promotion, to ensure a $\Upsilon_1$ correct solution plan. Similar to the predecessor sets $N_{\text{pre}}(o_{vi}^p)$ and $N_{\text{pre}}(o_{vi}^q)$ of an operator $o_{vi}^p$, we can thus represent a set of predecessors for $o_{vi}^p$ that represents the set of operators whose instance in a $\Upsilon_1$ solution plan may need to be ordered after an instance of another operator $o_{vj}^q$. That is, an instance of $o_{vj}^q$ may be promoted before an instance of $o_{vi}^p$.

\begin{definition}(N_{\text{clob}}(o_{vi}^p)). Let $o_{vi}^p \in O$, $N_{\text{clob}}(o_{vi}^p)$ is the set of predecessors whose instance may be clobbered by an instance of $o_{vi}^p$: $N_{\text{clob}}(o_{vi}^p) = \{ o_{vj}^q \mid v_i \neq v_j \land o_{vj}^q \notin N_{\text{pre}}(o_{vi}^p) \land \text{pre}(o_{vi}^p) = \text{pre}(o_{vj}^q)[v_i] \}$. \end{definition}

Table 4.2(c) illustrate $N_{\text{clob}}(o_{vi}^p)$ with the Horse Breeder. Operator graphs can then be enhanced with a set of directed edges that represents the promotion orders.

\begin{definition}(Declobbered Operator Graph).\end{definition}

Let $o_{vi}^p \in O$, I denote $G_2^c = (O, E_2^c)$ the declobbered operator graph, with $O$ the vertex set and $E_2^c$ the directed edge set. The declobbered operator graph is an operator graph where orders from the promotion process are displayed. $E_2^c = E_{\text{pre}}(O) \cup E_{\text{pre}}(O) \cup E_{\text{clob}}(O)$ with $E_{\text{pre}}(O)$ and $E_{\text{pre}}(O)$ the edge sets of Definition 4.2.1 and $E_{\text{clob}}(O) = \{(o_{vi}^p, o_{vj}^q) \mid o_{vi}^p \in N_{\text{clob}}(o_{vi}^p)\}$, the set of directed edges representing the order by promotion between the operator to assert $o_{vi}^p$ and its clobberer $o_{vj}^q$.

The set of directed edges $E_{\text{clob}}(O)$ will be colored in red in Figures representing declobbered operator graphs. Figure 4.5 respectively shows the declobbered operator graph of the Horse Breeder and that of the Gray Code. I use these graphs to increment the restriction $C_2$ as (P), (U) and $C_2$ are not sufficient to ensure a tractable class of problems (cf. Corollary 4.4.2). I denote $C_2^*$ the incremented version of $C_2$.

\begin{definition}(Restriction $C_2^*$). Let $v_i \in M, x, y \in D_{v_i}, x \neq y$. \end{definition}

- The class of problems is $C_2^*$
Figure 4.5: The yellow nodes are requestable operators inside the unique cycle of the same domain operator graph. The blue and green nodes are operators preceded by a different yellow operator via a post-prv dependency. The blue edges represent the post-pre dependencies, the green edges the post-prv ones and the red edges represent orders by promotion (cf. Theorem 4.3.1).

- if it exists \( o_{v_i}^x, o_{v_i}^y \in \mathcal{RO}(v_i) \) s.t. \( o_{v_i}^x, o_{v_i}^y \in \text{Cycle}(v_i) \) and for all \( o_1, o_2 \in \mathcal{O} \setminus \mathcal{O}(v_i) \) s.t. \( \text{prv}(o_1)[v_i] = x \land \text{prv}(o_2)[v_i] = y \) then:
  
  \( o_1 \) is not related to \( o_2 \) in \( \mathcal{G}_{\mathcal{O} \setminus \text{Cycle}(v_i)} \)

Let illustrate \( C_2^* \) with the Horse Breeder (HB) and the Gray Code (GC), the nodes of the two graphs in Figure 4.5 have three colors: yellow, green and blue. The yellow color is for cycle with at least two requestable operators, i.e. it colors \( o_{v_i}^x \) and \( o_{v_i}^y \) from the definition:

- Yellow\textsubscript{HB} = \{PickupBucket, DropBucket\}
- Yellow\textsubscript{GC} = \{o_{v_0}^+, o_{v_0}^-\}

Due to \( C_2 \), the yellow cycle has exactly two operators in it and it is two requestable operators. They therefore precede other operators in \( \mathcal{O} \setminus \mathcal{O}(v_i) \) by post-prv dependency, and these operators are colored either in blue or in green depending on the requestable operator that precedes them. We have:

- Blue\textsubscript{HB} = \{TakeHaystack\}, it is preceded by DropBucket.
CHAPTER 4. TOPOLOGICAL PLANNING

Figure 4.6: $G_{O \setminus \text{Cycle}(v_i)}$ are declobbered operator graphs with the domain operator graph on $v_i$ removed. It is useful to check whether green and blue operators are related in it. Here are two examples: the Horse Breeder and the Gray Code.

- Green$_{HB} = \{\text{FillBucketWithWater, FillHorseTrough}\}$, they are preceded by $\text{TakeBucket}$.
- Blue$_{GC} = \{o_{v_2}^-, o_{v_2}^+\}$, they are preceded by $o_{v_0}^-$. 
- Green$_{GC} = \{o_{v_1}^-, o_{v_1}^+\}$, they are preceded by $o_{v_0}^+$.

These blue and green operators represent $o_1$ and $o_2$ respectively in the definition of $C_2^*$.

In order to see if the Horse Breeder and the Gray Code structures respect the restriction $C_2^*$, the domain operator graph that contains the yellow operators is removed from the declobbered operator graph and we then check the relation between the green operators and the blue ones. Figure 4.6 shows the declobbered operator graphs with the domain operator graphs containing the yellow nodes removed of both the Horse Breeder and the Gray Code. The resulting graphs are respectively denoted $G_{HB}$ and $G_{GC}$.

- The Horse Breeder structure respect $C_2^*$: each blue operator is not related to any of the green operator in $G_{O \setminus \text{Cycle}(v_i)}^{+}_{HB} \setminus \text{Cycle}(v_1)$.
- The Gray Code structure does not respect $C_2^*$: each blue operator is preceded by the green operators and each green operator is preceded by the blue operators in $G_{O \setminus \text{Cycle}(v_0)}^{+}_{GC} \setminus \text{Cycle}(v_0)$. For instance, $o_{v_2}^+ \prec o_{v_1}^-$ and it exists the sequence of operators $\langle o_{v_1}^-, o_{v_1}^+, o_{v_2}^+ \rangle$ in $G_{O \setminus \text{Cycle}(v_0)}^{+}_{GC} \setminus \text{Cycle}(v_0)$, which implies $o_{v_1}^- \prec o_{v_1}^+$ by transitivity. Hence $C_2^*$ is not respected.

The idea of this incremented restriction $C_2^*$ is to keep the relaxation of (S) and to prevent the creation of problems whose instances may require the planner to loop through the unique cycle of some domain operator graphs. If two operators $o_1$ and $o_2$ with a different defined prevail-condition on a state variable (so the problem is not (S)) are related to each other in some ways, and if the requestable operators that solve these prevail-conditions are in the unique cycle of their domain operator graph, then it exists problem instances where one of the requestable operators needs to be instantiated at least twice in order to respect the existing order between $o_1$ and $o_2$ in the declobbered operator graph. In appendix C, I provide four operator graphs that could represent four possible SAS-PU structures to
illustrate the issue when \( o_1 \) and \( o_2 \) are related. Figure C.4 efficiently shows how a planner can unfold many times the yellow cycle of a simple SAS-PUC\(_2\) structure (C.4a) in order to provide a minimal and correct non-linear solution plan (C.4c). Figure C.3 also shows a SAS-PUC\(_2\) problem where instances, except one (C.1c), have a minimal solution plan that is \( \Upsilon_1 \). The exception problem instance has a minimal solution plan that is \( \Upsilon_2 \). Eventually, Figures C.1 and C.2 show two SAS-PUC\(_2^*\) problems and several minimal and \( \Upsilon_1 \) solution plans that solve different problem instances. These two examples illustrate prominently the effect of the restriction. The non-relationship between plans that solve different problem instances. These two examples illustrate prominently the effect of the restriction. The non-relationship between \( o_1 \) and \( o_2 \), or between the “green” and the “blue” operators, (cf. Definition 4.4.6 and the figures), implies that the instance of the requestable operators can frame once all the instance of the green and blue operators.

In fact, we saw in subsection 4.3.2 that a \( \Upsilon_1 \) chain of actions implies that, for each \( v_1 \in M \), a situation \( p \in D_{v_1} \) is most of the time established once (cases 1, 2 and 3) and is established twice iff \( p = s_0[v_1] \) (cases 2 and 3). As the start \( s_0[v_1] \) is not an action, in every case there is at most one action in a \( \Upsilon_1 \) chain that can establish \( p \). As a consequence, every action instance of operators that has \( p \) as a prevail-condition must be ordered after the establisher of \( p \) (For examples, see the blue actions in the action graphs of Figures C.2c, C.2d and the left graph of C.2e, they are all ordered after \( a^+ \)), or can also be ordered right after the start if \( p = s_0[v_1] \) (For examples, see the blue actions of Figures C.2f and the right graph of C.2e).

**Theorem 4.4.6.** SAS-PUC\(_2^*\) problems are \( \Upsilon_1 \).

The proof is provided in Appendix C.2.

### 4.4.5 The Class of Problems SAS-PUC\(_k\), with \( k \geq 3 \)

**Lemma 4.4.1.** SAS-PUC\(_k\) problems with \( k \geq 3 \) are at least \( \Upsilon_2 \).

**Proof.** The idea of the proof is to show that each problem that are exactly of the class SAS-PUC\(_k\), with \( k \geq 3 \), has a problem instance whose minimal solution plan is \( \Upsilon_2 \). To do so, I define a scalable problem instance in function of \( k \) whose minimal solution plan is \( \Upsilon_2 \), thus proving that SAS-PUC\(_k\) problems, with \( k \geq 3 \), are at least \( \Upsilon_2 \).

The scalable problem instance in function of \( k \), with \( k \geq 3 \), is defined as follows. Each problem that are exactly of the class SAS-PUC\(_k\) has at least one domain operator graph with a cycle of the size \( k \), and in this cycle, at least one operator is requestable. Let consider Cycle\((v_i)\) as the one respecting: \( |\text{Cycle}(v_i)| = k \). Cycle\((v_i) = \{o_{v_i}^0, \ldots, o_{v_i}^{k-1}\} = \{o_{v_i}^l | \text{pre}(o_{v_i}^l) = l-1 \text{ if } l > 0, \text{ pre}(o_{v_i}^0) = k-1, \text{ otherwise}\} \). As \( k \geq 3 \), then \( \{o_{v_i}^0, o_{v_i}^1, o_{v_i}^2\} \subseteq \text{Cycle}(v_i) \) and at least one of them is requestable. If one of them is requestable, let say \( o_{v_i}^0 \), then it exists an operator \( o' \in O \setminus O(v_i) \) such that \( \text{pre}(o')[v_i] = \text{post}(o_{v_i}^0) \). Let consider \( v_j \in M \) being the state variable affected by \( o' \). The scalable problem instance is:

\[
\Pi_k = (\Phi, s_0, s_v) \text{ with: } \begin{cases}
    s_0[v_i] = 1, & s_v[v_i] = 2 \\
    s_0[v_j] = \text{pre}(o'), & s_v[v_j] = \text{post}(o') \\
    s_0[v_i] = s_v[v_i], & \forall v_i \in M \setminus \{v_i, v_j\}
\end{cases}
\]
This problem instance is solved by $\Delta$ such that, $a'$ is an action instance of $o'$:

$$\Delta = \delta_{v_i} (s_0[v_i], prv(o')[v_i]) \oplus \langle a' \rangle \oplus \delta_{v_i} (prv(o')[v_i], s_*[v_i])$$

$$= \delta_{v_i} (1, 0) \oplus \langle a' \rangle \oplus \delta_{v_i} (0, 2)$$

$$= \delta_{v_i} (1, 2) \oplus \delta_{v_i} (2, 0) \oplus \langle a' \rangle \oplus \delta_{v_i} (0, 1) \oplus \delta_{v_i} (1, 2)$$

And $\Delta$ is $\Upsilon_2$ because $\delta_{v_i} (1, 2)$ appears twice. Let $a^0_{v_i}$ the action instance of $o^0_{v_i}$, this problem instance is scalable in function of $k$ because $|\delta_{v_i} (2, 0)| = k - 2$ such that:

- $\delta_{v_i} (0, 1) = \langle a^1_{v_i} \rangle$
- $\delta_{v_i} (1, 2) = \langle a^2_{v_i} \rangle$
- $\delta_{v_i} (2, 0) = \begin{cases} \langle a^0_{v_i} \rangle, & \text{if } k = 3 \\ \langle a^3_{v_i}, ..., a^{k-1}_{v_i}, a^0_{v_i} \rangle, & \text{if } k > 3 \end{cases}$

This problem instance can be created in every problem of the class SAS-PUC$_k$, with $k \geq 3$. It concludes the proof.

Because I am interested in problem instance whose minimal solution plan is $\Upsilon_1$, it results that I cannot consider SAS-PUC$_k$ problems with $k \geq 3$.

4.5 TopoPlan $\mathbb{P}$

This section presents the algorithm $TopoPlan$, denoted $\mathbb{P}$, which solves SAS-PUC$_k$ problems that are at most $\Upsilon_1$. Each problem instance are denoted $\Pi = (\Phi, s_0, s_*)$. $\mathbb{P}$ is based on $\mathcal{P}$: it plans backwards to take advantage of the post-unicity and does it in three phases.
Phase 1 builds the minimal chains of actions between the start \( s_0 \) and the goal \( s_\star \) for each state variable \( v_i \in \mathcal{M} \). All the actions of these chains are coloured yellow; Phase 2 checks the prevail-conditions of all the yellow actions. It creates orders between the yellow actions of the different chains so as to establish and/or promote each defined prevail-condition of each the yellow actions correctly. Let \( v_i \in \mathcal{M} \) and \( p \in \mathcal{R}_v^\circ \) a prevail-requestable value, two minimal chains of actions, one between \( s_0[v_i] \) and \( p \), and one between \( p \) and \( s_0[v_i] \) may be built to ensure the establishment of \( p \) while respecting the \( \Upsilon_1 \)-Truth Criterion (cf. Cases (2) and (3) in subsection 4.3.2). Eventually, a minimal and \( \Upsilon_1 \) non-linear plan is returned; Phase 3 topologically sorts the non-linear plan. \( \mathbb{P} \) differs from \( \mathcal{P} \) in several ways, however: I added extra conditions to take into account the relaxation of the restriction (S); I went deeper in the pre-processing and created additional sets of operators (cf. Definition 4.1.3) to reduce the complexity of \( \mathbb{P} \); The pre-processing of \( \mathbb{P} \) is not done while planning compared to \( \mathcal{P} \), but while compiling; Finally, \( \mathbb{P} \) does not instantiate actions, it colours the operators and returns plans of identifiers.

The section is organised as follows: I first describe the pre-processing, then I describe the algorithm and give complexity theorems. It follows a step by step explanation of the resolution of two problem instances of the Horse Breeder by \( \mathbb{P} \).

### 4.5.1 Setting up the Problem

First of all, I described operators in an external file, namely a .txt file, which is also a technique used by developers. In F.E.A.R. for instance, new assets, operators included, are described in a dedicated application called “GDBEdit” (which stands for Game DataBase Editor) that creates .record files used by the compiler [10]. Figure 4.8 shows an example of my .txt files, it is the Horse Breeder from the RDR-2 benchmark (described in subsection 5.2.1). The first column, coloured in gray, numbered the lines. The first line stores the number of state variables (l.1, \( m = 3 \)), then there is the number of operators (l.2, \( |\mathcal{O}| = 7 \)), followed by the maximum number of values in a state variable domain (l.3, \( n = 3 \)). Each line, from l.4 to l.10, is the representation of a (PU) operator: the first column (without considering the gray one) is the index of the affected state variable; the second column stores the value of the pre-condition; the third column is the post-condition; the 4th column to the \((4+m)^{th}\) column are the prevail-conditions. In the Horse Breeder
example, there are 3 state variables so there are 3 prevail-conditions per operator. u is the undefined value; The last column stores the name of each operator. Is called Setup the pre-processing phase of my algorithm, which is done prior to any planning sessions. That is, the setup is performed while compiling the code. The .txt file is translated to create a SAS structure $\Phi = (M, S, O)$. Each operator gets the state variable it affects, the value of its pre- and post-condition, and finally gets its prevail-conditions. They also get an identifier (cf. Definition 4.1.1), and their predecessor sets $N_{pre}$ and $N_{prv}$ are built (cf. Definition 4.1.3). Operators are stored in a hashing table, in the position given by their identifier. All these steps can be done line per line while the setup is browsing the .txt files.

Once each operator has been configured, then the structure is verified to see whether the restriction $C_0$, $C^S_2$ or $C^*_2$ is respected. It is done in 4 steps:

1. Mark every operator $o_{v_i}^p \in O$ such that $o_{v_i}^p \in \text{Cycle}(v_i)$ and $o_{v_i}^p \in \mathcal{RO}$.
   (a) If $o_{v_i}^p$ is marked and $|\text{Cycle}(v_i)| > 2$, then the problem is $C_k$ with $k > 2$. The structure is not valid. (cf. Lemma 4.4.1).

2. If no operator has been marked, then the structure is $C_0$ and therefore valid (cf. Theorem 4.4.3).

3. Else if exactly one operator per cycle has been marked, then the structure is $C^S_2$, i.e. it is valid (cf. Theorem 4.4.5).

4. Else
   Let $\text{Cycle}(v_i) = \{o^-_{v_i}, o^+_{v_i}\}$ and $o^-_{v_i}, o^+_{v_i} \in \mathcal{RO}(v_i)$,
   (a) If for each pair $o, o' \in O$ such that $\text{prv}(o)[v_i] = +$ and $\text{prv}(o')[v_i] = -$, $o$ is related to $o'$ in $\mathcal{G}_{O\setminus \text{Cycle}(v_i)}$, then the problem does not respect $C^*_2$.
   (b) Else, the problem is $C^*_2$ and therefore valid (cf. Theorem 4.4.6).

At last, I equipped actions with a getter called $\text{Next}()$ whose role is to return in constant time their successor of in the chain. First of all, because of $\Upsilon_1$, each operator is instantiated at most once in the solution plan, it results that actions, which are instances of an operator, can be identified with the same identifier as their operator (cf. Definition 4.1.1). It results that establishers, which are characterized by the post-condition (cf. Definition 3.2.5), are easily identified and can therefore be addressed in constant time while planning. Clobberers, on the contrary, are characterized by their pre-condition and post-uniqueness does not imply pre-uniqueness. This can be highlight with the Horse Breeder example (Table 3.1), DropHaystack and FillFeeder are post-unique but they share the same pre-condition. It results that clobberers cannot be identified the same way as establishers. Let $q \in \mathcal{RO}_{v_i}$ be a requestable value on $v_i \in M$ such that $T$ and $C$ respectively are the establisher and the clobberer of $q$, it can be highlighted that: $\text{post}(T)[v_i] = \text{pre}(C)[v_i] = q$. Due to $\Upsilon_1$, if $q \neq s_0[v_i]$, $\sigma_{v_i} = \text{chain}_{\nu}(s_0[v_i], s_*[v_i])$ passes through $q$ exactly once. As explained in subsection 4.3.2, $q$ is established by $T$ and promoted before $C$, the equality
post(T)[v_i] = pre(C)[v_i] therefore implies that C succeeds T in \( \sigma_{v_i} \). As T can be identified in \( O(1) \), I therefore introduce \( \text{Next}() \), a getter that is dynamically set, to return in constant time the successor of T in \( \sigma_{v_i} \). \( \text{Next}() \) must be dynamically set because an operator may have several successor via the post-pre dependencies but an action only has one in the chain. So, for each action in \( A \), \( \text{Next}() \) is set while building the chain of actions (cf. procedure 2 \textit{BuildChain}).

### 4.5.2 Algorithm

In this subsection, I present my algorithm \textit{TopoPlan}, denoted \( \mathbb{P} \). Due to the output restriction \( \Upsilon_1 \), the size of the action set \( A \) in a minimal solution plan is at most equal to the size of the operator set \( O \): \( |A| \leq |O| \). In addition, because each operator of \( O \) has at most one action instance in \( A \), the action instance and its operator can both share the same identifier while planning (cf. \textbf{Definition 4.1.1}). Let \( v_i \in \mathcal{M} \), \( p \in \mathcal{D}_{v_i} \), I therefore use the identifier \( a^p_{v_i} \) to refer to the operator \( a^p_{v_i} \in O \) and its action instance \( a^p_{v_i} \in A \) (so, \( \text{ope}(a^p_{v_i}) = a^p_{v_i} \)). Similarly, hoping that the reader will not be confused, I will only use the term action and the notation \( a^p_{v_i} \) in the following. For example: \( a^p_{v_i} \in O \) and \( a^p_{v_i} \in A \).

To avoid having to browse \( A \) to know if an action is inside, I implemented a labeling technique with colors such that:

- (White) the action is not instantiated, i.e. it is not in \( A \). Every action of \( O \) is white at the beginning of the resolution of any problem instance;
- (Yellow) the action is instantiated, i.e. it is in \( A \);
- (Blue) the action is being topologically sorted;
- (Green) the action is topologically sorted, i.e. added in a linear solution plan \( \Delta \).

The identifier \( a^p_{v_i} \) refers to an action of the hashing table that was built in the setup. The use of colors with the hashing table therefore allows to know in constant time the state of an action.

\( \mathbb{P} \) plans backwards using the identification technique \( a^p_{v_i} \in O \) and the predecessor sets \( N_{\text{pre}}(a^p_{v_i}) \) and \( N_{\text{prev}}(a^p_{v_i}) \) to find the minimal solution plan \( \Delta \) that solves an instance \( \Pi = (\Phi, s_0, s_*) \) of the class SAS-PUC\(_0\), SAS-PUC\(_S\) or SAS-PUC\(_S^*\) (cf. \textbf{Definitions 4.1.1} and 4.1.3). The resolution of \( \Pi \) by \( \mathbb{P} \) consists in building \( A \) and a sufficient set of orders between the actions of \( A \), denoted \( E_A \), while respecting the \( \Upsilon_1 \)-Truth Criterion. \( \mathbb{P} \) relies on the procedure 2 (\textit{BuildChain}) to build \( A \).

**Definition 4.5.1.** The specification of \textit{BuildChain} is:

- **Input:** \( O \), \( A \), \( E_A \), \( v_i \in \mathcal{M} \), \( s, g \in \mathcal{D}_{v_i} \) s.t. \( s \neq g \) and \( a^g_{v_i} \in O \) is white.
- **Output:** \( \delta_{v_i}(s,g) \). Each action \( a \) of \( \delta_{v_i}(s,g) \) is colored yellow, added to \( A \) and each order \((b,a)\), s.t. \( b \in N_{\text{pre}}(a) \), is added to \( E_A \).
CHAPTER 4. TOPOLOGICAL PLANNING

Theorem 4.5.1. BuildChain satisfies its specification, so it is correct.

Proof. The proof is given Appendix D. 

BuildChain is used by \( \mathbb{P} \) to build chains of actions that respect the \( \Upsilon_1 \)-Truth Criterion. I showed in subsection 4.3.2 that there are three cases to take into account with \( \Upsilon_1 \) chains of actions between the start \( s_0[v_i] \) and the goal \( s_\ast[v_i] \):

1. \( \sigma_1 = \text{chain}_{v_i}(s_0[v_i], s_\ast[v_i]) = \delta_{v_i}(s_0[v_i], s_\ast[v_i]) \)
2. \( \sigma_2 = \text{chain}_{v_i}(s_0[v_i], s_0[v_i]) = \delta_{v_i}(s_0[v_i], s_0[v_i]) \)
3. \( \sigma_3 = \text{chain}_{v_i}(s_0[v_i], s_\ast[v_i]) = \delta_{v_i}(s_0[v_i], s_0[v_i]) \oplus \delta_{v_i}(s_0[v_i], s_\ast[v_i]) \)

We can highlight that: \( \sigma_3 = \sigma_2 \oplus \sigma_1 \). The building of \( \sigma_1 = \delta_{v_i}(s_0[v_i], s_\ast[v_i]) \) for each \( v_i \in M \) is easy to understand in order to solve a problem instance. Indeed, a planner is designed to solve planning problem instances, therefore, for SAS-PU problem instances, finding all the chains of actions that solve \( s_0[v_i] \neq s_\ast[v_i] \) is a relevant starting point. This is the output specification of the phase 1 of TopoPlan: find all the \( \sigma_1 \), i.e. the minimal chains of actions that link the start to the goal states.

The action prevail-conditions then give instructions to the planner to correctly order the actions of the different chains together. This is done during the phase 2 of TopoPlan while respecting the \( \Upsilon_1 \)-Truth Criterion. An action \( T \) that establishes one prevail-condition (Post-uniqueness) of another action \( A \) is an establisher (cf. Definition 3.2.5). If \( T \) is already ordered in a chain, then it is yellow and the \( \Upsilon_1 \)-Truth Criterion tells us that: \( T \) must be ordered before \( A \), and if \( \text{Next}(T) \) exists, then it is a clobberer and \( A \) must be promoted before \( \text{Next}(T) \). If \( T \) is required but not yet ordered in a chain, then the problem instance is either not solvable, or the establisher is in a minimal chain of the form of \( \sigma_2 = \delta_{v_i}(s_0[v_i], s_0[v_i]) \). In other words, if the problem instance is solvable, \( \text{ope}(T) \) is a requestable operator inside the cycle of its domain operator graph. Let \( v_i \in M \), if \( \sigma_1 = \delta_{v_i}(s_0[v_i], s_\ast[v_i]) \) is built during Phase 1 and if \( \sigma_2 = \delta_{v_i}(s_0[v_i], s_0[v_i]) \) is built during Phase 2, then the chain of actions on \( v_i \) is of the form of \( \sigma_3 \). The concatenation of \( \sigma_2 \) and \( \sigma_1 \) is made through the additional order created by BuildChain (cf. Appendix D). At the end of its procedure, BuildChain adds the order \( (b, \text{First}(\sigma_1)) \) in the set \( E_A \), s.t. \( b \in \mathcal{N}_{\text{pre}}(\text{First}(\sigma_1)) \). \( b \in \mathcal{N}_{\text{pre}}(\text{First}(\sigma_1)) \) implies that: \( \text{pre}(\sigma_1) = \text{post}(b) \), and therefore \( \text{post}(b) = s_0[v_i] \). Then, \( \text{post}(\sigma_2) = s_0[v_i] \), therefore by post-unicity and \( \Upsilon_1 \), \( \text{Last}(\sigma_2) = b \). Eventually, \( (b, \text{First}(\sigma_1)) = (\text{Last}(\sigma_2), \text{First}(\sigma_1)) \), which corresponds to the concatenation: \( \sigma_2 \oplus \sigma_1 \). Eventually, if the input of \( \mathbb{P} \) is a solvable problem instance \( \prod \) of the SAS-PUC, SAS-PUC\(^S\) or SAS-PUC\(^\ast\), then Phase 2 returns a \( \Upsilon_1 \), non-linear, minimal solution plan that solves \( \prod \) while respecting the \( \Upsilon_1 \)-Truth Criterion. If the input is not a solvable problem instance, Phase 2 can yield a failure through the BuildChain procedure, otherwise it returns a sequence of actions that is not irreflexive.

The last phase is a topological sort performed by the procedure DFSTopo. This procedure is a depth first search topological sort, proven correct [40]. Its input specification
is a non-linear sequence of actions and its output specification is the linearisation of that sequence of actions, if it exists. In the algorithm TopoPlan, if the problem instance is solvable, the input of DFSTopo is a $\Upsilon_1$, non-linear and minimal solution plan that respect the $\Upsilon_1$-Truth Criterion. Therefore, DFSTopo returns a $\Upsilon_1$, linear and minimal solution plan and it still respects the $\Upsilon_1$-Truth Criterion. If the problem instance is not solvable, Phase 2 may return a non-linear sequence of action that is not irreflexive. DFSTopo therefore yields a failure thanks to the blue color that spots looping issue.

**Definition 4.5.2.** The specification of TopoPlan is:

- **Input:** $\Pi = (M, O, s_0, s_\ast)$, a problem instance of the class SAS-PUC$_0$, SAS-PUC$_2$ or SAS-PUC$_{\ast}$.

- **Output:** If $\Pi$ is solvable, then $\mathcal{P}$ returns $\Delta$ a linear, $\Upsilon_1$ and minimal solution plan that solves $\Pi$ while respecting the $\Upsilon_1$-Truth Criterion. If $\Pi$ is not solvable, $\mathcal{P}$ yields a failure.

**Theorem 4.5.2.** The algorithm TopoPlan satisfies its specification, so it is correct and complete.

**Proof.** The proof is given in Appendix D.

**Theorem 4.5.3** (Time Complexity). $\mathcal{P}$ worst-case time complexity is $O(|A| + |E_A|)$, with $A$ the set of actions and $E_A$ the set of orders between the actions of $A$.

**Proof.** Phase 1 has at most $O(|O|) = O(mn)$ steps: $m$ steps via the for loop (3.2) times $n$ steps via the $BuildChain$ procedure of (3.4).

Phase 2: Due to the restriction $\Upsilon_1$, $|A| \leq |O|$, so the for loop (3.9) requires at most $|A| \leq |O| = O(mn)$ steps. The two for loops (3.9) and (3.10) require $O(|A| + |E_{A,prv}|)$ steps with $E_{A,prv} = \{(b, a) \mid a \in A \land b \in N_{prv}(a)\}$. Indeed, during this step, the two for loops check the prevail-conditions of the whole set of actions of $A$: there are thus $|A|$ steps to go through all the actions of $A$, plus $|E_{A,prv}|$ steps to visit all the predecessors of all the actions of $A$. The procedures $BuildChain$ (3.15) and (3.21) can be triggered if the action being evaluated has a prevail-condition that is neither satisfied by the initial state nor by an action of $A$. In this case, both procedures look for actions that are in a set $D \subseteq O \setminus A$. The procedures $BuildChain$ thus add $|D|$ steps for their calls. The prevail-conditions of these newly added actions in $D$ must be checked as well, it thus adds $|D|$ more steps via the for loop (3.9) as well as $|E_{D,prv}|$ more steps for the for loop (3.10). Finally, phase 2 has a complexity of $O(|A| + |E_{A,prv}| + 2 \cdot |D| + |E_{D,prv}|) \equiv O(|A| + |E_{A,prv}| + 2 \cdot |O \setminus A| + |E_{O \setminus A,prv}|) \equiv O(|O| + |E_{O,prv}|)$. All the instructions from (3.27) to (3.30) are done in constant time ($O(1)$).

Phase 3: At the end of Phase 2 is returned $\Delta = (A, E_A)$, a non-linear sequence of actions. Phase 3 topologically sorts $\Delta$ and its worst-case time complexity is $O(|A| + |E_A|)$; in the worst case, this is equivalent to $O(|O| + |E_O|)$, which ultimately dominates the whole and proves the linear time complexity of $\mathcal{P}$. □
CHAPTER 4. TOPOLOGICAL PLANNING

Procedure 2 BuildChain($v_i, s, g, A, E_A, O$)

Input/Output: cf. Definition 4.5.2.
Parameters: $x, y$, two values of $D_{v_i}$.

1: $x \leftarrow \emptyset$; $y \leftarrow \emptyset$
2: if $a^q_{v_i}$ is a ghost action then fail
3: end if
4: Color($a^q_{v_i}$) $\leftarrow$ yellow; $A \leftarrow A \cup \{a^q_{v_i}\}$; $y \leftarrow pre(a^q_{v_i})$; $E_A \leftarrow E_A \cup \{(a^q_{v_i}, a^q_{v_i})\}$
5: if Next($a^q_{v_i}$) $\neq \emptyset$ \{a$^q_{v_i}$ can be a ghost action.\} then
6: Next($a^q_{v_i}$) $\leftarrow a^q_{v_i}$
7: end if
8: while $y \neq s$ do
9: if $a^y_{v_i}$ is a ghost action then fail
10: end if
11: if Color($a^y_{v_i}$) $=$ yellow then fail
12: end if
13: Color($a^y_{v_i}$) $\leftarrow$ yellow; $A \leftarrow A \cup \{a^y_{v_i}\}$; $x \leftarrow y$; $y \leftarrow pre(a^y_{v_i})$; $E_A \leftarrow E_A \cup \{(a^y_{v_i}, a^x_{v_i})\}$
14: if Next($a^y_{v_i}$) $\neq \emptyset$ \{a$^y_{v_i}$ can be a ghost action.\} then
15: Next($a^y_{v_i}$) $\leftarrow a^x_{v_i}$
16: end if
17: end while

Procedure 3 DFSTopo($a^p_{v_i}, A, E_A, s_0, \Delta$)

Input: $a^p_{v_i}$, a yellow action affecting $v_i$ with the value $p \in D_{v_i}$; $s_0 \in S$; $\Delta$, the totally ordered solution plan.
Output: $a^p_{v_i}$ is colored in green once all its neighbors have been topologically ordered; It is then enqueued to $\Delta$.

1: Color($a^p_{v_i}$) $\leftarrow$ blue
2: for $a^q_{v_j} \in E_A(a^p_{v_i})$ do
3: if $a^q_{v_j} \notin N_{pre}(a^p_{v_i}) \vee a^p_{v_i}$ is not the first action to modify $s_0[v_i]$ then
4: if Color($a^q_{v_j}$) $=$ blue then fail \{Cycle spotted.\}
5: end if
6: if Color($a^q_{v_j}$) $=$ yellow then
7: DFSTopo($a^q_{v_j}, A, E_A, s_0, \Delta$)
8: end if
9: end if
10: end for
11: Color($a^p_{v_i}$) $\leftarrow$ green; $\Delta \leftarrow \Delta + \{a^p_{v_i}\}$;
Procedure 4 $P(\mathcal{M}, \mathcal{O}, s_0, s_*)$

Input/Output: cf. Definition 4.5.2.

Parameters: $\mathcal{A}$, the set of yellow actions; $E_A$, the set of orders between yellow actions.

1: $\Delta \leftarrow \emptyset$; $\mathcal{A} \leftarrow \emptyset$; $E_A \leftarrow \emptyset$
2: for $v_i \in \mathcal{M}$ do {Phase 1}
3: 
4: if $s_0[v_i] \neq s_*[v_i]$ then
5: 
6: end if
7: end for
8: if $\mathcal{A} = \emptyset$ then return $\emptyset$ {s_0 and s_* are equal.}
9: end if
10: for $a_{v_i}^p \in \mathcal{A}$ do {Phase 2}
11: 
12: for $a_{v_j}^q \in N_{prv}(a_{v_i}^p)$ do
13: 
14: if $q \neq s_0[v_j]$ then
15: 
16: end if
17: if $a_{v_j}^q \notin \mathcal{O}$ then fail
18: end if
19: if $s_0[v_j] = s_*[v_j]$ then
20: 
21: BuildChain($v_j, s_0[v_j], q, \mathcal{A}, E_A, \mathcal{O}$)
22: end if
23: if $prv(Next(a_{v_j}^q))[v_i] \neq p$ then
24: 
25: end if
26: end if
27: if $q = s_0[v_j] \land a_{v_j}^q \in \text{Cycle}(v_j) \land \text{Cycle}(v_j)$ is concerned by $C_2^*$ then
28: 
29: end if
30: end for
31: for $a \in \mathcal{A}$ do {Phase 3}
32: 
33: if $Color(a) = \text{yellow}$ then
34: 
35: end if
36: end for
37: return $\Delta$
Theorem 4.5.4 (Space Complexity). \( \mathbb{P} \) requires at most \( O(n^2 \cdot m^2) \) space.

Proof. A \((PU)\) operator takes \( O(m) \) space: the pre- and post-condition can be reduced to one variable each, plus a variable for the index of the affected state variable (cf. Definition 4.1.2), and the set \( \mathcal{N}_{\text{pre}} \) is a singleton due to restrictions \((P)\) and \((U)\); the prevail-conditions, on the contrary, are lists of \( m \) elements, and the set \( \mathcal{N}_{\text{prv}} \) has at most \( m \) elements (cf. Definition 4.1.3). Then, the hashing table that stores identifiers (cf. Definition 4.1.1) takes \( O(n \cdot m) \) space. Finally, \( \mathbb{P} \) creates an action graph \( \mathcal{G}_{\Delta} = (\mathcal{A}, E_{\mathcal{A}}) \) which takes at most \( O(n^3 \cdot m^2) \) space: The set of actions takes \( O(|\mathcal{A}|) \equiv O(|\mathcal{O}|) \equiv O(n \cdot m) \) space due to \( \Upsilon_1 \), and each action can have at most \( O(|\mathcal{A}|) \equiv O(n \cdot m) \) predecessors. Hence, \( O(|\mathcal{A}|^2) \equiv O(n^2 \cdot m^2) \), which dominates the whole.

It should be noticed that, if an action \( C \in \mathcal{A} \) has \( |\mathcal{A}| - 1 \) predecessors, it is the following scenario: \( C \) is the clobberer of all the actions of the problem, except for the establisher, and these actions have been promoted. It results that \( C \) is preceded by the establisher and by all the promoted actions, hence \( O(|\mathcal{A}|) \). Another consequence is that the other actions only have one predecessor: the establisher. In most problems, the number of predecessors of an action \( a \) is closer to \( O(|\mathcal{N}_{\text{prv}}(a)|) \equiv O(m) \) as it corresponds to the number of establishers to assert all the prevail-conditions of \( a \). And the clobberers, which are the successors of the establishers, are ordered after \( a \) (most of the time, by promotion, cf. subsection 4.3.2), i.e. each clobberer only gets one additional predecessor: \( a \). Overall, there are at most \( O(n \cdot m^2) \) orders between actions, but a single action can have up to \( O(|\mathcal{A}|) \) predecessors.

Two examples on how \( \mathbb{P} \) resolves a problem instance are given in Appendix E.

4.5.3 Discussion

• TopoPlan \((\mathbb{P})\) is based on C. Bäckström’s algorithm that solve SAS\(^+\)-PUS problems. I have not considered the SAS\(^+\)-structure, however. The next step would be to consider the SAS\(^*\)-structure, i.e. to allow undefined state variables in the goal state. I believe \( \mathbb{P} \) can be used as a good basis to consider solving SAS\(^*\)-PUC\(_0\), SAS\(^*\)-PUC\(_S^2\) or SAS\(^*\)-PUC\(_S^*\) problem instances. It may be useful to add a new color than yellow to mark actions inside cycles, though, so that yellow can be used by BuildChain as a stop condition to concatenate chains of actions instead of yielding a failure. And the other color, on the contrary, can be used as a stop condition to yield a failure as we do not want BuildChain to loop inside the unique cycle of each domain operator graph. Indeed, an undefined state variable in the goal state does not mean the value given by the start cannot be changed. But not all values are meaningful, the meaningful one are the requestable values. So, to consider the SAS\(^*\) structure, it is necessary to know the last requestable value required by the solution plan solving a given problem instance. The action chain from the start to the last useful requestable value can thus be built by BuildChain incrementally, hence the use of two colors. If the incremental construction takes place inside the cycle of the domain operator graph, then it probably means the planner is looping, so a stop condition is required. If the incremental construction takes place outside the cycle, then the planner must ensure it is not building several different branches of the domain operator graph, but only one.
operators with an undefined precondition and affecting the same state variable. That is, \( O = \{ o_{v_i}^k \mid v_i \in \mathcal{M}, k \in \mathcal{D}_{v_i}, \text{pre}(o_{v_i}^k) = u \} \). This kind of operator sets are useful to instantiate movement actions for instance: \( o_{v_i}^k = \text{MoveTo}(k) \), such that \( \text{pre}(\text{MoveTo}(k)) = u \). But movement operators/actions are related to pathfinding, which is known to be hard.

• TopoPlan can be used to solve SAS-PUS problem instances, but only those whose minimal solution plan is \( \Upsilon_1 \). I haven’t proved theoretically this assertion, however. But any solution plan returned by \( \mathcal{P} \) can be verified with \( \mathcal{P} \).

• Concerning (3.27) to (3.29) in \( \mathcal{P} \), it only concerns operators of cycles strictly restricted by \( C_2^s \) because operators of \( C_2^S \) or \( C_0 \) are never concerned by the double establishment situation described in Case 1 and Case 2 (subsection 4.3.2) when the proposition to assert is equal to the start. \( v_i \in \mathcal{M} \), action chains in minimal solution plan solving SAS-PUC_0 problem instance are of the form of \( \sigma_1 = \delta_{v_i}(s_0[v_i], s_0[v_i]) \). Therefore, if the proposition to assert \( q \) is equal to the start (\( q = s_0[v_i] \)), then \( q \) only needs to be promoted before \( \text{First}(\sigma_1) \). This is handled by \( \mathcal{P} \) between (3.19) and (3.26). For SAS-PUC_2^S problem instances, minimal solution plans resemble those of SAS-PUS problem instances that are \( \Upsilon_1 \). The action chain can be of the form of \( \sigma_1, \sigma_2 = \delta_{v_i}(s_0[v_i], s_0[v_i]) \) or \( \sigma_3 = \sigma_2 \oplus \sigma_1 \). But if \( \sigma_2 \) is built in SAS-PUC_2^S or SAS-PUS problem instances, this is to reach \( q \) inside the cycle, so that:

\[
\sigma_2 = \delta_{v_i}(s_0[v_i], q) \oplus \delta_{v_i}(q, s_0[v_i])
\]

Otherwise, if \( q = s_0[v_i] \) or outside the cycle, then \( \sigma_2 \) is never built because it is meaningless and therefore breaks the minimality of the solution plan if it is built. It results that if \( q = s_0[v_i] \), then \( q \) is always promoted before \( \sigma_1 \).

The instructions between (3.27) to (3.29) work for SAS-PUC_2^S problem instances because of the structural properties induced by the restriction \( C_2^s \). It may exists SAS-PU problem instances whose minimal solution plan is \( \Upsilon_1 \) and where Case 2 and Case 3 with \( q = s_0[v_j] \) may be handled differently. That is, it may exist problem instances where the order (4.10) must be applied over (4.11), and vice versa. But this is not the case with \( C_2^s \).

\( \mathcal{P} \) works with SAS-PUC_0, SAS-PUC_2^S and SAS-PUC_2^S problem instances, which means that it does not exist domain operator graph cycles with more than 3 operators and at
least one of them requestable. It results the BuildChain procedures (3.15) and (3.21) will always return a minimal action chain of size 1. These procedures aim at building:

\[ \sigma_2 = \delta_{v_i}(s_0[v_i], s_0[v_i]) = \delta_{v_i}(s_0[v_i], q) \oplus \delta_{v_i}(q, s_0[v_i]) \]

Such that \( q \) is a requestable value established by \( a_{v_i}^q \). Therefore, \(|\text{Cycle}(v_i)| = |\sigma_2| = 2 \) and \(|\delta_{v_i}(s_0[v_i], q)| = |\delta_{v_i}(q, s_0[v_i])| = 1 \), hence the assertion that these BuildChain procedures will always return a chain with one action.

Eventually, TopoPlan has a linear-time complexity thanks to the predecessor sets (\( N_{\text{pre}} \) and \( N_{\text{prv}} \)) that were created during the setup. I will present experiments in the next chapter and will show that, in addition to reducing complexity, these sets of predecessors prevent \( P \) from doing the unnecessary work that \( P \) does by checking all the prevail-conditions. In other words, with the predecessor sets \( P \) only checks the defined prevail-conditions while \( P \) also checks the undefined ones.
CHAPTER 4. TOPOLOGICAL PLANNING
In this chapter, I present the runtime performance of both $\mathcal{P}$ and $\mathbb{P}$ implemented in C++14 with default settings for Microsoft Visual Studio 2019. All experiments were performed with an AMD Ryzen™ 7 2700X (cf. Table 5.11 for the characteristics), 32Gb of RAM and Windows 10 (64 bits). Unlike C. Bäckström who implemented $\mathcal{P}$ with SUN Common Lisp with “no attempts at optimizing it”, I personally optimized each of my code: the C++ implementation of $\mathcal{P}$ works with vectors and functions’ parameters are integers, references and/or pointers in order to avoid copy constructors. Actions are only instantiated once by the $\text{FindPost}$ function (cf. Appendix A), and the other functions work with references to address the instantiated actions. Concerning the C++ implementation of $\mathbb{P}$, I remove the vectors in favor of fixed sized arrays. There were some weird results linked to the vectors’ dynamic allocation that occurred automatically while the code is being executed. There are therefore a memory management with $\mathbb{P}$, which allowed me to get smooth results. Finally, $\mathbb{P}$ also works with pointers and references to keep planning as light as possible.

I present several different benchmarks in this chapter. Let $m$ be the number of state variables and $n$ the size of each state variable domain, the first round of benchmarks are scalable, i.e. they can be scaled in function of $m$, or in function of $n$. Their purpose is to show empirically the time complexity of both algorithm $\mathcal{P}$ and $\mathbb{P}$. As a reminder, the complexity of $\mathcal{P}$ is $O(n \cdot m^2)$ and that of $\mathbb{P}$ is $O(|\mathcal{A}| + |E_{\mathcal{A}}|) \approx O(n \cdot m^2)$, so both algorithms are affected by the variables $m$ and $n$. The second round of benchmarks are more realistic, they are based on existing commercial video games. I design them in order to see how many Non-Playable Characters of these benchmarks can get a plan from the C++ implementation of both $\mathcal{P}$ and $\mathbb{P}$ in real-time.

The chapter is organized as follows: I first introduce the scalable benchmarks and I use them to show empirically that $\mathcal{P}$ is indeed quadratic in $m$ and linear in $n$, and that $\mathbb{P}$ is indeed linear in the number of actions plus the number of orders. In a second section, I present three realistic benchmarks based on the existing commercial video games Red Dead Redemption, Assassin’s Creed: Origins and Horizon Zero Dawn. I created these benchmarks for two reasons: to test both $\mathbb{P}$ and $\mathcal{P}$ on more realistic problems and to highlight that it is possible to represent these problems with the new class SAS-PUC$^*_2$. I
end this chapter with a discussion and conclusions on the results of these experiments.

5.1 Scalable Benchmarks

In this section, I empirically show the time complexity of both $\mathcal{P}$ and $\mathcal{P}$, firstly with respect to $m$, the number of state variables, and secondly with respect to $n$, the size of the state variable domains. To do so, I use two scalable instances of 2 new benchmarks designed to have a non-linear minimal solution plan whose size increases in function of the number of state variables ($m$). The difference between the two problem instances lies in the number of orders between the actions: in the first problem instance, the increase of the number of orders is quadratic with respect to $m$, while in the second the increase is linear with respect to $m$.

5.1.1 Description of the Scalable Benchmarks

MultiPrv_n and MultiPrv_n_Cycle: Let $n, p \in \mathbb{N}$ such that $n$ is the maximum number of values in a state domain, i.e. $n = \max_{v_i \in \mathcal{M}}(\mathcal{D}_{v_i})$ and $0 < p \leq n - 1$. For each $v_i \in \mathcal{M}$, the operators of $\mathcal{O}(v_i)$ of MultiPrv_n are described as follows:

- $\text{pre}(o^p_{v_i}) = p - 1$
- $\text{post}(o^p_{v_i}) = p$
- $\text{prv}(o^p_{v_i})[v_j] = u$, for $i \leq j < m$
- $\text{prv}(o^p_{v_i})[v_j] = \lfloor n/2 \rfloor$, for $0 \leq j < i$.

MultiPrv_n is of the class SAS-PUSC$^0$ and, therefore, it can be solved by both $\mathcal{P}$ and $\mathcal{P}$. It is a very constrained problem as there are $O(n \times m^2)$ orders between the operators. For each $v_i \in \mathcal{M}$, if all operators of $\mathcal{O}(v_i)$ are instantiated to create a chain of actions, we have:

$$\text{chain}_{v_i}(0, n - 1) = (\mathcal{A}(v_i), \prec_{\text{pre}}) = \langle a^1_{v_i}, a^2_{v_i}, \ldots, a^{n-1}_{v_i} \rangle$$

The structure of MultiPrv_n_Cycle is equivalent to that of MultiPrv_n, except that, for each $v_i \in \mathcal{M}$, the operator $o^0_{v_i}$ such that $\text{post}(o^0_{v_i}) = 0$, $\text{pre}(o^0_{v_i}) = n - 1$ and $\text{prv}(o^0_{v_i}) = \text{prv}(o^p_{v_i})$ is added to $\mathcal{O}(v_i)$. It adds the order: $a^{n-1}_{v_i} \prec_{\text{pre}} a^0_{v_i}$ which implies that:

$$\text{chain}_{v_i}(0, n - 1) \oplus \langle a^0_{v_i} \rangle = \text{chain}_{v_i}(0, 0)$$
That is the additional operator and order create a cycle such that \(\mathcal{O}(v_i) = \text{Cycle}(v_i)\). \text{MultiPrv}_2\_\text{Cycle} is nothing but the Tunnel example of [6, p.17] and \text{MultiPrv}_n\_\text{Cycle} is the generalisation of it. \text{MultiPrv}_n\_\text{Cycle} is of the class SAS-PUSC\(_n\). For all \(n \in \mathbb{N}\), it implies only \(\mathcal{P}\) is complete on \text{MultiPrv}_n\_\text{Cycle}, i.e. \text{MultiPrv}_2\_\text{Cycle}, only. More precisely, if \(n > 2\), for every problem \((s_0, s_*)\) where it exists \(v_i \in \mathcal{M}\) such that the single prevail-condition on \(v_i\) is not in between \(s_0[v_i]\) and \(s_*[v_i]\), then the minimal solution plan is \(\Upsilon_2\). \mathcal{P}\) will thus return the minimal solution plan whereas \(\mathcal{P}\) will yield a failure. Otherwise, if the prevail-condition is in between \(s_0[v_i]\) and \(s_*[v_i]\) or equal to one of the two, for each \(v_i \in \mathcal{M}\), the minimal solution plan is \(\Upsilon_1\) and both algorithms return it (cf. Chapter 4).

I now describe a possible scalable instance for each of the two problems: \text{MultiPrv}_n\_\text{Cycle} and \text{MultiPrv}_n\_\text{Cycle}. These scalable instances are the one I used to empirically strengthen my theoretical results on \(\mathcal{P}\). For \text{MultiPrv}_n\_\text{Cycle}, I define the scalable problem instance \(\Pi_{\text{multi}} = (\Phi_{\text{multi}}, s_0, s_*)\) with:

\[
\begin{align*}
s_0 &= \langle v_i = 0 \mid v_i \in \mathcal{M} \rangle; \quad s_* = \langle v_i = n - 1 \mid v_i \in \mathcal{M} \rangle
\end{align*}
\]

And for \text{MultiPrv}_n\_\text{Cycle}, \(\Pi_{\text{mcy}} = (\Phi_{\text{mcy}}, s_0, s_*)\) with:

\[
\begin{align*}
s_0 &= \langle v_i = 0 \mid v_i \in \mathcal{M} \rangle; \quad s_* = \langle v_{m-1} = n - 1 \rangle \oplus \langle v_i = 0 \mid v_i \in \mathcal{M} \setminus \{v_{m-1}\} \rangle
\end{align*}
\]

Let consider the scalable problem \(\Pi_{\text{mcy}} = \Pi_{\text{Tun3}} = (\Phi_{\text{Tun3}}, s_0 = \langle 0, 0, 0 \rangle, s_* = \langle 0, 0, 1 \rangle)\) of \text{MultiPrv}_2\_\text{Cycle} with \(m = 3\), which is the Tunnel planning problem with 3 sections (\(\Phi_{\text{Tun3}}\), cf. Figure 5.1). \(\Pi_{\text{Tun3}}\) states that every section is off initially and the goal is to turn on the third section only. The operator set is described in Table 5.1. The minimal solution plan of \(\Pi_{\text{Tun3}}\) is: \(\Delta = \langle a^0_{v_0}, a^1_{v_1}, a^1_{v_2}, a^0_{v_3}, a^0_{v_3} \rangle\), i.e. to turn on the 3rd section, the sections 1 to 3 are turned on successively, then the sections 2 and 1 are turned off successively. Then, \(|\Delta| = |\mathcal{O}| - 1\), i.e. the planner has to instantiate all the operator once (except one) to solve the problem instance. Both planners, \(\mathcal{P}\) and \(\mathcal{P}\), create 10 orders to solve the problem instance.

It can be proved that both \(\mathcal{P}\) and \(\mathcal{P}\) have to create at least \((2 \cdot m^2 - 3 \cdot m + 1)\) orders to solve \(\Pi_{\text{mcy}}\) of Tunnel with \(m\) sections, and they both return a minimal action plan \(\Delta\) of \((2 \cdot m - 1)\) actions and that is \(\Upsilon_1\).

<table>
<thead>
<tr>
<th>(\mathcal{O}_{\text{Tun3}})</th>
<th>pre()</th>
<th>post()</th>
<th>prv()</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>(o_{v_0}^{-})</td>
<td>(v_0 = 1)</td>
<td>(v_0 = 0)</td>
<td>(u, u, u)</td>
<td>Switch off Sec 1</td>
</tr>
<tr>
<td>(o_{v_0}^{+})</td>
<td>(v_0 = 0)</td>
<td>(v_0 = 1)</td>
<td>(u, u, u)</td>
<td>Switch on Sec 1</td>
</tr>
<tr>
<td>(o_{v_1}^{-})</td>
<td>(v_1 = 1)</td>
<td>(v_1 = 0)</td>
<td>(1, u, u)</td>
<td>Switch off Sec 2</td>
</tr>
<tr>
<td>(o_{v_1}^{+})</td>
<td>(v_1 = 0)</td>
<td>(v_1 = 1)</td>
<td>(1, u, u)</td>
<td>Switch on Sec 2</td>
</tr>
<tr>
<td>(o_{v_2}^{-})</td>
<td>(v_2 = 1)</td>
<td>(v_2 = 0)</td>
<td>(1, 1, u)</td>
<td>Switch off Sec 3</td>
</tr>
<tr>
<td>(o_{v_2}^{+})</td>
<td>(v_2 = 0)</td>
<td>(v_2 = 1)</td>
<td>(1, 1, u)</td>
<td>Switch on Sec 3</td>
</tr>
</tbody>
</table>

Table 5.1: Operator set of \(\Phi_{\text{Tun3}}\).
OnePrv_n and OnePrv_n_Cycle  The structures of OnePrv_n and OnePrv_n_Cycle are very close to that of MultiPrv_n and MultiPrv_n_Cycle. They are less constrained, however, as each operator only has one defined prevail-condition. There are $O(n \times m)$ orders between the operators. The operators of OnePrv_n follows the following rules, $\forall v_i \in \mathcal{M}$ and $0 < p \leq n - 1$:

- $\text{pre}(a_{vi}^p) = p - 1$
- $\text{post}(a_{vi}^p) = p$
- $\text{prv}(a_{vi}^p)[v_{i+1}] = \lfloor n/2 \rfloor$, if $v_i \neq v_{m-1}$
- $\text{prv}(a_{vi}^p)[v_j] = u$, $\forall v_j \in \mathcal{M} \setminus \{v_{i+1}\}$.

For each $n \in \mathbb{N}$, OnePrv_n is of the class SAS-PUSC_0. It can be solved by both $\mathbb{P}$ and $\mathcal{P}$. For each $v_i \in \mathcal{M}$, the longest chain of actions is:

$$\text{chain}_{v_i}(0, n - 1) = (\mathcal{A}(v_i), \prec_{\text{pre}}) = \langle a_{vi}^1, a_{vi}^2, ..., a_{vi}^{n-1} \rangle$$

OnePrv_n_Cycle consists in OnePrv_n but, for each $v_i \in \mathcal{M}$, there is the additional operator $a_{vi}^0$: $\text{post}(a_{vi}^0) = p = 0$, $\text{pre}(a_{vi}^0) = n - 1$. The prevail-conditions of $a_{vi}^0$ follow the same rules as the other operators of OnePrv_n. For each $v_i \in \mathcal{M}$, $a_{vi}^0$ creates a cycle in $\mathcal{G}_\mathcal{O}(v_i)$, and it allows chain of actions from 0 to 0 for instance:

$$\text{chain}_{v_i}(0, n - 1) \oplus \langle a_{vi}^0 \rangle = \text{chain}_{v_i}(0, 0)$$

OnePrv_n_Cycle is of the class SAS-PUSC_n, that is, according to Lemma 4.4.1 and Theorem 4.4 [6, p.76], if $n \geq 3$, the minimal solution plans are at most $\Upsilon_2$, which implies only $\mathcal{P}$ is complete. If $n < 2$, SAS-PUSC_2 is equivalent to the class SAS-PUC_2, i.e. $\Upsilon_1$, therefore all problems of this class can be solved by both $\mathbb{P}$ and $\mathcal{P}$. I now describe two scalable instances for these two problems. For OnePrv_n, I define $\Pi_{\text{one}} = (\Phi_{\text{one}}, s_0, s_*)$ with:

$$s_0 = \langle v_i = 0 \mid v_i \in \mathcal{M} \rangle, \ s_* = \langle v_i = n - 1 \mid v_i \in \mathcal{M} \rangle$$

For OnePrv_n_Cycle, I define $\Pi_{\text{ocy}} = (\Phi_{\text{ocy}}, s_0, s_*)$ with:

$$s_0 = \langle v_i = 0 \mid v_i \in \mathcal{M} \rangle, \ s_* = \langle v_0 = n - 1 \rangle \oplus \langle v_i = 0 \mid v_i \in \mathcal{M} \setminus \{v_0\} \rangle$$

Let consider the scalable problem instance $\Pi_{\text{ocy}} = (\Phi_{\text{ocy}}, s_0 = \langle 0, 0, 0 \rangle, s_* = \langle 1, 0, 0 \rangle)$ of OnePrv_2_Cycle with $m = 3$. The operator set is described in Table 5.2.

In this problem, the minimal solution plan is: $\Delta = \langle a_{v_2}^1, a_{v_1}^1, a_{v_0}^1, a_{v_1}^0, a_{v_2}^0 \rangle$. Both planners, $\mathbb{P}$ and $\mathcal{P}$, create 8 orders to solve the problem and return $\Delta$, which is a minimal solution plan of 5 actions.

It can be proved that both $\mathbb{P}$ and $\mathcal{P}$ have to create at least $(5 \cdot m - 7)$ orders to solve $\Pi_{\text{ocy}}$ of OnePrv_2_Cycle with respect to $m$, and they both return a minimal action plan $\Delta$ of $(2 \cdot m - 1)$ actions and that is $\Upsilon_1$. 

88
### Table 5.2: Operator set of $\Phi_{oCy3}$.

<table>
<thead>
<tr>
<th>$O_{Tun3}$</th>
<th>pre()</th>
<th>post()</th>
<th>prv()</th>
</tr>
</thead>
<tbody>
<tr>
<td>$o_{v_0}^0$</td>
<td>$v_0 = 1$</td>
<td>$v_0 = 0$</td>
<td>$(u,1,u)$</td>
</tr>
<tr>
<td>$o_{v_0}^1$</td>
<td>$v_0 = 0$</td>
<td>$v_0 = 1$</td>
<td>$(u,1,u)$</td>
</tr>
<tr>
<td>$o_{v_1}^0$</td>
<td>$v_1 = 1$</td>
<td>$v_1 = 0$</td>
<td>$(u,u,1)$</td>
</tr>
<tr>
<td>$o_{v_1}^1$</td>
<td>$v_1 = 0$</td>
<td>$v_1 = 1$</td>
<td>$(u,u,1)$</td>
</tr>
<tr>
<td>$o_{v_2}^0$</td>
<td>$v_2 = 1$</td>
<td>$v_2 = 0$</td>
<td>$(u,u,u)$</td>
</tr>
<tr>
<td>$o_{v_2}^1$</td>
<td>$v_2 = 0$</td>
<td>$v_2 = 1$</td>
<td>$(u,u,u)$</td>
</tr>
</tbody>
</table>

5.1.2 Empirical Comparison of Time Complexities ($\mathbb{P}$ vs $\mathcal{P}$)

In this subsection, I empirically compare the time complexity of both $\mathbb{P}$ and $\mathcal{P}$. To do so, I used the two scalable problem instances $\Pi_{mCy} (\text{MultiPrv}_n\_\text{Cycle})$ and $\Pi_{oCy} (\text{OnePrv}_n\_\text{Cycle})$ described in subsection 5.1.1. As a reminder, the worst-case time complexity of $\mathcal{P}$ is $O(n \cdot m^2)$ (cf. Theorem 4.2 [6, p.90]), whereas the worst-case time complexity of $\mathbb{P}$ is $O(|A| + |E_A|) \equiv O(n \cdot m^2)$ (cf. Theorem 4.5.3). With $n$ being the number of values inside a variable domain, $m$ the number of state variables in the problem, $A$ the actions of the plans, $|E_A|$ the number of orders between the action of $A$. $\mathcal{P}$ is linear in $n$ and quadratic in $m$ while $\mathbb{P}$ is linear in the number of actions plus the number of orders between them, which, in very constrained problems, is equivalent to the complexity of $\mathcal{P}$.

In the following, I present two experiments. In the first one (1), $\mathbb{P}$ and $\mathcal{P}$ have to solve $\text{MultiPrv}_2\_\text{Cycle}$ and $\text{OnePrv}_2\_\text{Cycle}$ in function of $m$. That is, $n = 2$ is fixed, while $m$ ranges from 0 to 4000. In the second one (2), $\mathbb{P}$ and $\mathcal{P}$ have to solve $\text{MultiPrv}_n\_\text{Cycle}$ and $\text{OnePrv}_n\_\text{Cycle}$ in function of $n$ and with three different fixed $m$. (1) and (2) consist in observing and comparing the runtime performance of both algorithms in function of $m$ or $n$ respectively.

**Runtime performance comparison in function of $m$:** As explained previously, $\text{MultiPrv}_2\_\text{Cycle}$ and $\text{OnePrv}_2\_\text{Cycle}$ are both problems of the class SAS-PUSC$_2$ which means all their problem instances can be solved by both $\mathbb{P}$ and $\mathcal{P}$. Hence, both algorithm can be compared with the problem instances $\Pi_{mCy} (\text{MultiPrv}_n\_\text{Cycle})$ and $\Pi_{oCy} (\text{OnePrv}_n\_\text{Cycle})$ presented in subsection 5.1.1.

In the following, I fixed the number of values to 2: $n = 2$. Figure 5.2 shows the runtime performance of both $\mathbb{P}$ (Green data set) and $\mathcal{P}$ (Red data set) solving the problem instances of $\text{MultiPrv}_2\_\text{Cycle}$ and $\text{OnePrv}_2\_\text{Cycle}$ respectively, and in function of an increasing number of state variables $m$. I will use the name of the problems, $\text{MultiPrv}_2\_\text{Cycle}$ and $\text{OnePrv}_2\_\text{Cycle}$, to refer to the problem instances $\Pi_{mCy} (\text{MultiPrv}_n\_\text{Cycle})$ and $\Pi_{oCy} (\text{OnePrv}_n\_\text{Cycle})$ presented in subsection 5.1.1.

In the following, I fixed the number of values to 2: $n = 2$. Figure 5.2 shows the runtime performance of both $\mathbb{P}$ (Green data set) and $\mathcal{P}$ (Red data set) solving the problem instances of $\text{MultiPrv}_2\_\text{Cycle}$ and $\text{OnePrv}_2\_\text{Cycle}$ respectively, and in function of an increasing number of state variables $m$. I will use the name of the problems, $\text{MultiPrv}_2\_\text{Cycle}$ and $\text{OnePrv}_2\_\text{Cycle}$, to refer to the problem instances $\Pi_{mCy} (\text{MultiPrv}_n\_\text{Cycle})$ and $\Pi_{oCy} (\text{OnePrv}_n\_\text{Cycle})$ presented in subsection 5.1.1.

For example, for $m = 2002$, the average runtime of $\mathbb{P}$ to solve $\text{MultiPrv}_2\_\text{Cycle}$ 15 times is 0.08ms. I previously explained that $\text{MultiPrv}_2\_\text{Cycle}$ is designed to be solved with
CHAPTER 5. BENCHMARKS AND RESULTS

Figure 5.2: \( P \) can run in \( O(m) \) when \( |A| + |E_A| = O(m) \) whereas \( P \) always runs in \( O(m^2) \).

Figure 5.3: Runtime ratio between the two algorithms \( \frac{\mathcal{P}}{\bar{P}} \), with respect to the scalable problems of \textit{MultiPrv}_2\_Cycle and \textit{OnePrv}_2\_Cycle.
plans having a quadratic growth of the number of orders between actions in function of \( m \): \( |E_A| = O(m^2) \), while OnePrv_2_Cycle is solved by plans having a linear growth of orders in function of \( m \): \( |E_A| = O(m) \). As the number of orders depends on the number of state variables \( m \), Figure 5.2 therefore empirically strengthens Theorem 4.5.3 and Theorem 4.2 [5, p.90]. Indeed, \( \mathcal{P} \) worst-case time complexity is quadratic in \( m \) and the quadratic behaviors on Figure 5.2 with the red data sets. \( \mathcal{P} \), however, has a worst-case time complexity of \( |A| + |E_A| \), and thus, it can run in \( O(m) \) when \( |A| + |E_A| = O(m) \) or it can run in \( O(m^2) \) when \( |A| + |E_A| = O(m^2) \). This is why \( \mathcal{P} \) solves OnePrv_2_Cycle in a linear manner: this problem is designed to have a non-linear minimal solution plan with a linearly increasing number of orders as a function of \( m \). \( \mathcal{P} \) then solves MultiPrv_2_Cycle in a quadratic manner: this problem is designed to have a non-linear minimal solution plan with a quadratically increasing number of orders as a function of \( m \) (Green data sets). Figure 5.3 shows that \( \mathcal{P} \) is at least 2 times faster than \( \mathcal{P} \) to solve MultiPrv_2_Cycle, especially when \( m \) is big. When \( m \) is small (< 200), then \( \mathcal{P} \) is even faster, up to a factor 10 when \( m = 102 \). Then, we can also see in Figure 5.3 that \( \mathcal{P} \) is at least 65 times faster than \( \mathcal{P} \) (\( m = 150 \)) to solve OnePrv_2_Cycle. And the higher \( m \), the higher the ratio. When \( m = 3900 \), \( \mathcal{P} \) is 178 times faster than \( \mathcal{P} \). MultiPrv_2_Cycle has a very constrained structure and shows that the setup and the use of identifiers for \( \mathcal{P} \) helps achieving better results. OnePrv_2_Cycle, on the contrary, is a less constrained and shows that the use of predecessor sets are relevant, especially for bigger problem instances, as it greatly enhances the performances. The solving of OnePrv_2_Cycle also shows that \( \mathcal{P} \) can do a lot of useless work. It evaluates every prevail-condition of each action, which, in the case of OnePrv_2_Cycle, means \( \mathcal{P} \) evaluates \( m - 2 \) undefined prevail-conditions per action. It should finally be noticed that real-world problems are very unlikely to be as constrained as MultiPrv_2_Cycle. In fact, they are more likely to be low or medium constrained, which further strengthens my strategy of using a setup before planning.

**Runtime performance comparison in function of \( n \):** C. Bäckström explained in his thesis that he didn’t succeed in empirically show the linearity of his algorithm in function of the size of the state variable domains \( n \). He instead got a superlinear behavior and justified the result with some other experiments. He explained that the possible reason for the superlinearity was the use of Lisp and that “many details, like memory management, are out of control of the programmer” [6, p.92]. Contrary to Lisp, the C++ programming language allows memory management and the use of other tools, such as pointers, to counteract or avoid disruption from external events during data collection. With the help of my benchmarks OnePrv_n_Cycle and MultiPrv_n_Cycle, I was able to carry out experiments to prove the linearity in function of \( n \) of both \( \mathcal{P} \) and \( \mathcal{P} \). The results are given in Figure 5.4. The red data refers to the performance of \( \mathcal{P} \) while the green data refers to the performance of \( \mathcal{P} \). Figure 5.4a shows the experiment with MultiPrv_n_Cycle. Each linear curve shows the solving of MultiPrv_n_Cycle with a fixed \( m \) (\( m \in \{2, 4, 8\} \)) in function of \( n \), by \( \mathcal{P} \) (red curves) and by \( \mathcal{P} \) (green curves). Figure 5.4b presents similar results but for the problem OnePrv_n_Cycle. These results corroborate that both \( \mathcal{P} \) and
\(P\) are linear in \(n\).

Figure 5.5 presents the runtime performance ratio of \(P\) over \(P\) when solving \texttt{MultiPrv\_n\_Cycle} (Figure 5.5a) or \texttt{OnePrv\_n\_Cycle} (Figure 5.5b), with fixed \(m\) and in function of \(n\). Overall, my algorithm (\(P\)) was faster than C. Bäckström’s one (\(P\)) in these experiments. In both subfigures of Figure 5.5, \(P\) is several hundred times faster than \(P\) when \(n\) is small. Then, the higher \(n\) is, the more the ratio tends towards 10.

It should be reminded that both \texttt{OnePrv\_n\_Cycle} and \texttt{MultiPrv\_n\_Cycle} are of the class \texttt{SAS-PUSC\_n}, which implies only \(P\) is complete to solve them. Because it exists instances that are \(\Upsilon_2\) (Theorem 4.4 [6, p.76] and Lemma 4.4.1). However, the scalable instances \(\Pi_{oCy}\) and \(\Pi_{mCy}\) used in the experiments and described in subsection 5.1.1 are \(\Upsilon_1\), i.e. the minimal solution plan of these instances is always \(\Upsilon_1\). Therefore, these instances are solvable by \(P\). In addition to that, the solution plans of \(P\) were correct as they were equal to those of \(P\).

### 5.1.3 Another Scalable Example: BlockWorld

The \texttt{BlockWorld} is a very well-known planning problem [41]. C. Bäckström has created a SAS-PUBS version of it in his thesis [6, p.93] with 3 blocks. It can be generalized as follows: \(\mathcal{M} = \{X_1 on X_2, ..., X_1 on X_k, ..., X_k on X_{k-1}\}\), where \(k \in \mathbb{N}\) is the number of blocks and \(X_k\) the \(k^{th}\) blocks. There are \(m = |\mathcal{M}| = k \cdot (k - 1)\) state variables. Let consider the two blocks \(X\) and \(Y\) such as \(X \neq Y\), there are two operators named \texttt{MoveXfromYtoTable} and \texttt{MoveXfromTabletoY} and define as:

- \texttt{MoveXfromYtoTable}
  
  - \(post(\texttt{MoveXfromYtoTable})[X on Y] = 0\)
  
  - \(pre(\texttt{MoveXfromYtoTable})[X on Y] = 1\)
  
  - \(prv(\texttt{MoveXfromYtoTable}) = \langle u \mid \forall v \in \mathcal{M}\rangle\), then:
    
    - \(\forall C \in Blocks \setminus \{X, Y\}, prv(\texttt{MoveXfromYtoTable})[X on C] = 0\)
    
    - \(\forall C \in Blocks \setminus \{X\}, prv(\texttt{MoveXfromYtoTable})[Con X] = 0\)
    
    - \(\forall C \in Blocks \setminus \{X, Y\}, prv(\texttt{MoveXfromYtoTable})[Con Y] = 0\)

- \texttt{MoveXfromTabletoY}
  
  - \(post(\texttt{MoveXfromTabletoY})[X on Y] = 1\)
  
  - \(pre(\texttt{MoveXfromTabletoY})[X on Y] = 0\)
  
  - \(prv(\texttt{MoveXfromTabletoY}) = prv(\texttt{MoveXfromYtoTable})\)

The following \(s_0\) and \(s_*\) form the reverse tower scalable example. That is, \(s_0\) represents a tower of \(k \in \mathbb{N}\) blocks s.t. \(\forall 1 < i \leq k\), the block \(X_i\) is on top of \(X_{i-1}\) (With 3 blocks, cf. (a) in Fig 5.6), and \(s_*\) represents the reverse tower: \(\forall 1 \leq i < k\), the block \(X_{i-1}\) is on top of \(X_i\) (With 3 blocks, cf. (b) in Fig 5.6).
Figure 5.4: Comparison of the runtime performances of $\mathbb{P}$ and $\mathcal{P}$ when solving MultiPrv_n_Cycle or OnePrv_n_Cycle in function of $n$ and with three fixed $m \in \{2, 4, 8\}$. 

(a) MultiPrv_n_Cycle.

(b) OnePrv_n_Cycle.
Figure 5.5: Runtime performance ratio of $P$ over $P$ when solving MultiPrv_n_Cycle or OnePrv_n_Cycle in function of $n$ and with a fixed $m \in \{2, 4, 8\}$. 

(a) MultiPrv_n_Cycle.

(b) OnePrv_n_Cycle.
CHAPTER 5. BENCHMARKS AND RESULTS

![Diagram of the BlockWorld problem]

(a) The initial state $s_0$  
(b) The goal state $s_*$

Figure 5.6: BlockWorld. $s_0 = \langle 0, 0, 1, 0, 0, 1 \rangle$ and $s_* = \langle 1, 0, 0, 1, 0, 0 \rangle$.

$\mathcal{M} = \{A \cap B, A \cap C, B \cap A, B \cap C, C \cap A, C \cap B\}$.

- $\forall 1 < i \leq k$, $s_0[X_i \cap X_{i-1}] = 1$. All the other state variables are set to 0.
- $\forall 1 \leq i < k$, $s_*[X_{i-1} \cap X_i] = 1$. All the other state variables are set to 0.

**BlockWorld with 3 blocks:** In this paragraph, I illustrate the SAS-PUBS representation of the BlockWorld with 3 blocks. Let $\Phi_{3\text{blocks}} = (\mathcal{M}, \mathcal{S}, \mathcal{O})$ a SAS-structure of the BlockWorld with 3 blocks, and $\Pi_{ev} = (\Phi_{3\text{blocks}}, s_0 = \langle 0, 0, 1, 0, 0, 1 \rangle, s_* = \langle 1, 0, 0, 1, 0, 0 \rangle)$ the reverse tower problem presented on Fig. 5.6. The operator set $\mathcal{O}$ of $\Phi_{3\text{blocks}}$ is presented in Tab. 5.3.

<table>
<thead>
<tr>
<th>$\mathcal{O}$</th>
<th>pre()</th>
<th>post()</th>
<th>prv()</th>
<th>Operator Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O_{\text{v}_3}$</td>
<td>$v_0 = 1$</td>
<td>$v_0 = 0$</td>
<td>$\langle u, 0, 0, u, 0, 0 \rangle$</td>
<td>AfromBtoT</td>
</tr>
<tr>
<td>$O_{\text{v}_3}$</td>
<td>$v_0 = 0$</td>
<td>$v_1 = 0$</td>
<td>$\langle u, 0, 0, u, 0, 0 \rangle$</td>
<td>AfromTtoB</td>
</tr>
<tr>
<td>$O_{\text{v}_3}$</td>
<td>$v_1 = 1$</td>
<td>$v_1 = 0$</td>
<td>$\langle 0, u, 0, 0, 0, u \rangle$</td>
<td>AfromCtoT</td>
</tr>
<tr>
<td>$O_{\text{v}_3}$</td>
<td>$v_1 = 0$</td>
<td>$v_1 = 1$</td>
<td>$\langle 0, u, 0, 0, 0, u \rangle$</td>
<td>AfromTtoC</td>
</tr>
<tr>
<td>$O_{\text{v}_3}$</td>
<td>$v_2 = 1$</td>
<td>$v_2 = 0$</td>
<td>$\langle 0, u, u, 0, 0, 0 \rangle$</td>
<td>BfromAtoT</td>
</tr>
<tr>
<td>$O_{\text{v}_3}$</td>
<td>$v_2 = 0$</td>
<td>$v_2 = 1$</td>
<td>$\langle 0, u, u, 0, 0, 0 \rangle$</td>
<td>BfromTtoA</td>
</tr>
<tr>
<td>$O_{\text{v}_3}$</td>
<td>$v_3 = 1$</td>
<td>$v_3 = 0$</td>
<td>$\langle 0, 0, 0, u, 0, u \rangle$</td>
<td>BfromCtoT</td>
</tr>
<tr>
<td>$O_{\text{v}_3}$</td>
<td>$v_3 = 0$</td>
<td>$v_3 = 0$</td>
<td>$\langle 0, 0, 0, u, 0, u \rangle$</td>
<td>BfromTtoC</td>
</tr>
<tr>
<td>$O_{\text{v}_3}$</td>
<td>$v_4 = 1$</td>
<td>$v_4 = 0$</td>
<td>$\langle u, 0, 0, 0, u, 0 \rangle$</td>
<td>CfromAtoT</td>
</tr>
<tr>
<td>$O_{\text{v}_3}$</td>
<td>$v_4 = 0$</td>
<td>$v_4 = 1$</td>
<td>$\langle u, 0, 0, 0, u, 0 \rangle$</td>
<td>CfromTtoA</td>
</tr>
<tr>
<td>$O_{\text{v}_3}$</td>
<td>$v_5 = 1$</td>
<td>$v_5 = 0$</td>
<td>$\langle 0, 0, u, 0, 0, u \rangle$</td>
<td>CfromBtoT</td>
</tr>
<tr>
<td>$O_{\text{v}_3}$</td>
<td>$v_5 = 0$</td>
<td>$v_5 = 1$</td>
<td>$\langle 0, 0, u, 0, 0, u \rangle$</td>
<td>CfromTtoB</td>
</tr>
</tbody>
</table>

Table 5.3: The operator set to solve problems of the BlockWorld with 3 blocks. $m = 6$, $n = 2$, $|\mathcal{O}_\text{set}| = 12$.

**Runtime performance of $P$ on the BlockWorld:** The BlockWorld problem is interesting to illustrate some drawbacks of the SAS-PU representations. This BlockWorld problem is of the class SAS-PUC$^3$ and therefore solvable by $\mathbb{P}$. This representation, however, does not allow the displacement of a block from a block to another, each block displacement is
from one block to the table and from the table to a block. A valid plan for the problem instance $\Pi_{rev}$ is therefore to move every block on the table and then build the goal tower.

| #(Blocks) | $|\mathcal{M}|$ | Runtime $\mathcal{P}(\mu s)$ | Runtime $\mathcal{P}(\mu s)$ | Ratio $\mathcal{P}/\mathcal{P}$ |
|-----------|---------------|-----------------|-----------------|-----------------|
| 2         | 4             | 0.1             | 275             | 2750            |
| 3         | 6             | 0.1             | 636             | 6360            |
| 10        | 90            | 3.40            | 678             | 199.4           |
| 25        | 600           | 21.0            | $1.15 \cdot 10^3$ | 54.7            |
| 50        | 2450          | 88.6            | $8.99 \cdot 10^3$ | 101.4           |

Table 5.4: Runtime performance of both $\mathcal{P}$ and $\mathcal{P}$ on the scalable BlockWorld.

Table 5.4 shows that $\mathcal{P}$ is extremely fast to solve $\Pi_{rev}$, it solved $\Pi_{rev}$ with 50 blocks in 88.6$\mu$s. The number of blocks, however, ranges from 0 to 50 and it is difficult to add more. The reason is that the number of state variables grows in a quadratic manner in function of the number of blocks ($m = |\mathcal{M}| = k \cdot (k - 1)$) with this representation. It is therefore not possible to consider ten thousand of blocks [42]. With that amount of block, one SAS-PUC$_2$ operator would at least take $O(m^2) = 10^3$ To. Eventually, although $\mathcal{P}$ is extremely fast, the representation takes a lot of space.

5.2 Realistic Benchmarks

As I aim to provide a real-time planning algorithm it is crucial to test the C++ implementation of my planner $\mathcal{P}$ with realistic benchmarks. First to show that it is possible to represent practical problems with my class of problems, and second to prove that my algorithm satisfies the real-time constraint.

For this purpose, I introduce 3 practical benchmarks that I created inspired by the following commercial video games: Red Dead Redemption 2 (RDR-2) [11], Assassin’s Creed: Origins (ACO) [12] and Horizon Zero Dawn (HZD) [13]. I name the benchmarks after those of the games. I present the world representation, some operators and some goals for each game. I also introduce operator graphs for some NPCs of each game so as to show my structural restrictions $C_2$ and $C_\ast_2$ in practice. Finally, each benchmark has its specificity: the RDR-2 model has several different types of NPCs, each with its own representation of the world; the ACO model has a base-NPC and a specialized one that inherits the former. The operators in ACO have parameters$^1$ that allow contextualization; the HZD model has a single type of NPC with a single goal. This model shows how to represent an NPC having a single goal with my class of problem.

$^1$It is possible to add parameters to the operators of RDR-2 and HZD as well, but they are useless while planning. That is why I did not spend time adding parameters in these benchmarks. The benchmark ACO is enough to understand how to contextualize actions convincingly without affecting the behavior of the planning system.
Table 5.5: Operator set of the Horse Breeder. His goal is to \( s_*(\text{Feed the horses}) = \langle 2, 0, 2 \rangle \). The Horse Breeder is a SAS-PUC\(^*_2\) problem, solvable by \( P \) only.

The section is organized as followed: I introduce RDR-2, ACO and HZD; Then, I present the experiments I carried out to study the performances of both TP and MPUS. I finally explain the results, suggest a discussion and give a conclusion.

5.2.1 Read Dead Redemption 2

Red Dead Redemption 2 (RDR-2) was released by Rockstar Studios in October 2018 [11]. It is a action-adventure and open-world video game that takes place in Southern United States in the late Wild West era, in 1899. The player plays as Arthur Morgan, an outlaw member of the Van Der Linde gang who must deal with the decline of the Wild West whilst attempting to survive several different adversaries like the government forces, the rival gangs, etc. This very successful video game [43] has NPCs of great quality which improves credibility and dynamism of the virtual world of RDR-2. Christopher Livingston, authors of articles for PC Gamer, wrote the following about the NPCs of RDR-2 [44]:

The most compelling open worlds are the ones that feel like they would exist even if you never visited them, and the key to that illusion is to fill the world with convincing people. I’m not talking about the main characters, the quest-givers, or the speech-makers. I’m talking about everyone else — the background extras that fill the gaps in the world, the ones you tend to barely notice unless you slow down and start paying attention to them.

Videos of NPCs’ daily life can be found on-line and show the details of their behaviors. At the time stamp 5:36 of the video [45], we follow a Horse Breeder that is off to work. He goes to feed his horses by filling the water trough first and then the feeder.

Table 5.5 presents a possible SAS-PUC\(^*_2\) operator set for the Horse Breeder. Similar to the video, my version of the Horse Breeder can interact with a bucket and the haystacks in order to fill the water trough and the feeder for his horses. Then, his work goal is to
feed the horses:

\[ s_*(\text{Feed the horses}) = (2, 0, 2) \]

The main difficulty in this representation is to ensure the NPCs do not carry a haystack and
the bucket at the same time as it is physically demanding. Hence the prevail-conditions:
\[ \text{prv(Haystack)}[v_1] = 0 \quad \text{and} \quad \text{prv(Bucket)}[v_0] = 0. \]

With the goal \( s_*(\text{Feed the horses}) \), it is not possible for the \textbf{Horse Breeder} to fill the feeder first, however. The planner will
necessary return a plan whose first subgoal is to fill the water trough then the feeder.

Because the action \textbf{Fill Horse Feeder} sets \((v_0 = 2)\) permanently, which means the prevail-
condition of \textbf{Take Bucket} can never be satisfied after \textbf{Fill Horse Feeder} while planning. If
we want to fill the feeder first anyway, then we need two subgoals:

- \( s_*(\text{Fill feeder}) = (2, 0, s_0[v_2]) \)
- \( s_*(\text{Fill trough}) = (s_0[v_0], 1, 2) \)

Once the \textbf{Horse Breeder} has finished the goal \( s_*(\text{Fill feeder}) \), then he no longer carries
an haystack. Therefore, the achievement of \( s_*(\text{Fill feeder}) \) set \( v_0 \) to the value 0. Same for
\( s_*(\text{Fill trough}) \), once achieved \( v_2 \) can be reset to 0 for the \textbf{Horse Breeder}. It works the
same way with \( s_*(\text{Feed the horses}) \). In a way, the state variables \textit{HayStack} and \textit{Water}
can be seen as targetable objects. Once in the feeder, the haystack is no longer targetable.

That is, the \textbf{Horse Breeder} has to target another one if he wants to repeat the process.
This is the same thing for the water, once in the water trough, it is no longer usable
and the \textbf{Horse Breeder} needs to find a water source to refill the bucket. Hence, if the
\textbf{Horse Breeder} calls the planner again to \( s_*(\text{Feed the horses}) \) with new targeted items,
then \( s_0[v_0] = 0 \) and \( s_0[v_2] = 0 \) necessarily.

Figures 5.7 and 5.8 respectively shows the domain operator graphs and the full operator
graph of the \textbf{Horse breeder}. What is interesting with this \textbf{Horse Breeder} representation
is that it highlights post-uniqueness does not imply pre-uniqueness. The operators
\textbf{Drop Haystack} and \textbf{Fill Horse Feeder} are post-unique but they share the same precondi-
tion: \( \text{pre(Drop Haystack)}[v_0] = \text{pre(FillHorseFeeder)}[v_0] = 1 \), i.e. they both require the
\textit{Haystack} to be \textit{inHands}(1). It results a domain operator graph with an unusual shape (cf.
Figure 5.7), that is a graph with an unique cycle that does not contain all the nodes. The
shape is unusual because, most of the time, when representing a problem, the resulting do-
main operator graphs are either a directed tree (cf. appendices B: id=2 B.2; ids={2,4} B.3;
ids={2,6,7,9} B.4; etc.) or a full cycle (cf. appendices B: id=1 B.2; id=3 B.3; id=8 B.4;
etc.)

This \textbf{Horse Breeder} problem is obviously (P) and (U). It is not (S), however, because
\textbf{Fill Bucket with Water} and \textbf{Fill Horse Trough} have a different prevail-condition on \( v_1 \) than the
operator \textbf{Take Haystack}. It is \( C^*_2 \), however, as it can be seen with Figure 5.9. (i) The prevail
requestable operators, \textbf{Pick up Bucket} and \textbf{Drop Bucket}, colored in yellow in Figure 5.9a,
are inside \text{Cycle}(v_1) \text{ and } |\text{Cycle}(v_1)| = 2 \text{ (cf. Figure 5.7). (ii) Then, when these operators}
are set apart, the resulting graph is Figure 5.9b, and we observe that \textbf{Take Haystack} is not
linked to either \textbf{Fill Bucket with Water} or \textbf{Fill Horse Trough}. (i) and (ii) ensure the restriction
\( C^*_2 \) is respected.
CHAPTER 5. BENCHMARKS AND RESULTS

Figure 5.7: The domain operator graphs of the Horse Breeder. These graphs represent the post-pre dependencies only. **Ghost V2** is a ghost operator, i.e. an operator whose identifier exists but the operator itself does not exist (Definition 4.2.3).

Figure 5.8: The operator graph of the Horse Breeder. The blue directed edges are the post-pre dependencies whereas the green directed edges are the post-prevail dependencies.

Similar to the Horse Breeder, I have designed 6 more NPC types: the **Farrier**, whose goals are $s_*(	ext{Sooing the horse})$ and $s_*(	ext{Forge horseshoes})$; the **Native hunter**, whose goal is to $s_*(	ext{Hunt})$; the **Bandit**, whose goal is to $s_*(	ext{Rob a bank})$; the **Innkeeper**, whose goals are to $s_*(	ext{Serve a drink})$ and $s_*(	ext{Assing a room})$; the **Dancer**, whose goal is to $s_*(	ext{Dance})$; and, finally, the **Drunk Citizen**, whose goal is to $s_*(	ext{Forget})$. The SAS representation of these NPCs can be found in appendix F.

5.2.2 Assassin’s Creed: Origins

Assassin’s Creed Origins (ACO) is action role playing video game developed by Ubisoft Montreal and released in October 2017 [12]. The tenth installment of the Assassin’s Creed serie sets in Egypt, near the end of the Ptolemaic period (49–43 BC), where the player plays as a Medjay named Bayek of Siwa. In order to make their magnificent open-world feel more alive and believable, the developers went in great detail with their NPCs, giving them autonomy and freedom through a Goal-Oriented AI system [46]. Soldiers, citizens, and even the fauna, have goals and needs that the player can witness or feel while playing [9]. At night, for example, the guards are less present because they are eating or resting. This allows the player to draw tactics based on the habits of the NPCs.

As there are a lot of different types of citizens in ACO, I have decided to create a base-NPC named **Citizen**. Then I introduce the **Armorer**, a citizen whose work is to create swords
(a) The declobbered operator graph of the Horse Breeder with red directed edges representing orders by promotion (Definition 3.2.5). If one red edge is added between two actions by Pwhile planning, it means the queue action clobbers a prevail-condition of the head action. If the Horse Breeder needs to Drop Haystack to Pick up Bucket, then the Take Haystack action, if required, threatens the prevail-condition of the action Pick up Bucket. Therefore, to ensure a proper topological sort, the add of the directed red edge from Pick up Bucket to Take Haystack by Pwhile planning is necessary.

(b) The declobbered operator graph of the Horse Breeder when the operators Pick up Bucket and Drop Bucket are removed (yellow operators in Figure 5.9a).

Figure 5.9: This two subfigures highlight why the Horse Breeder problem respects the restriction C₂. “Yellow”= \{Pick up Bucket, Drop Bucket\}, “Green”= \{Fill Bucket with Water, Fill Horse Trough\} and “Blue”= \{Take Haystack\} are affected by the restriction C₂. It is C₂ because the “Yellow” nodes form a cycle of size 2 in their domain operator graph (cf. Figure 5.7). It is C₂ because none of the “Green” nodes are linked to the “Darker Green” one when the “Yellow” nodes are removed (cf. Figure 5.9b).
Table 5.6: Possible operator representation for the citizens of Assassin’s Creed: Origins (based on the game). $\mathcal{M}_{\text{Citizen}} = \{v_i \mid i \in [0, 6]\}$ is the citizen state variable set.

| Citizen | Pre | Post | Prevail $\langle v_i | i \in [0, 6] \rangle$ | Explanation |
|---------|-----|------|--------------------------------|-------------|
| $O_0$  | $v_0 = 0$ | $v_0 = 1$ | $\langle u, u, u, u, u, u, 0 \rangle$ | Buy Food(?pos) |
| $O_{v_0}$ | $v_0 = 1$ | $v_0 = 2$ | $\langle u, u, u, u, u, u, u \rangle$ | Eat |
| $O_{v_1}$ | $v_1 = 0$ | $v_1 = 1$ | $\langle u, u, u, u, u, u, 0 \rangle$ | Buy Drink(?pos) |
| $O_{v_1}$ | $v_1 = 1$ | $v_1 = 2$ | $\langle u, u, u, u, u, u, u \rangle$ | Drink |
| $O_{v_2}$ | $v_2 = 0$ | $v_2 = 1$ | $\langle u, u, u, 0, u, u, 0 \rangle$ | Store (?item,?pos) |
| $O_{v_2}$ | $v_2 = 1$ | $v_2 = 2$ | $\langle u, u, u, 0, u, u, u \rangle$ | Take (?item,?pos) |
| $O_{v_3}$ | $v_3 = 0$ | $v_3 = 1$ | $\langle u, u, u, u, u, u \rangle$ | Stop Wandering |
| $O_{v_3}$ | $v_3 = 1$ | $v_3 = 1$ | $\langle u, 2, u, u, u, u \rangle$ | Have a Walk |
| $O_{v_4}$ | $v_4 = 1$ | $v_4 = 0$ | $\langle u, u, u, u, u, u \rangle$ | Stop Debate |
| $O_{v_4}$ | $v_4 = 0$ | $v_4 = 1$ | $\langle u, u, u, u, u, u \rangle$ | Debate(?with, ?pos) |
| $O_{v_5}$ | $v_5 = 0$ | $v_5 = 1$ | $\langle u, u, 0, 0, u, 0 \rangle$ | Rest (?pos) |
| $O_{v_5}$ | $v_5 = 1$ | $v_5 = 1$ | $\langle u, u, u, u, u, u \rangle$ | End Work |
| $O_{v_6}$ | $v_6 = 0$ | $v_6 = 1$ | $\langle 2, 2, 1, 0, u, u, u \rangle$ | Work (?item, ?pos) |

$V_0 : \text{Hunger}$, $D_{v_0} = \{0 : \text{noFood}, 1 : \text{hasFood}, 2 : \text{fed}\}$.

$V_1 : \text{Thirst}$, $D_{v_1} = \{0 : \text{noDrink}, 1 : \text{hasDrink}, 2 : \text{hydrated}\}$.

$V_2 : \text{hasItem}$, $D_{v_2} = \{0 : \text{false}, 1 : \text{true}\}$.

$V_3 : \text{Wandering}$, $D_{v_3} = \{0 : \text{false}, 1 : \text{true}\}$.

$V_4 : \text{Debating}$, $D_{v_4} = \{0 : \text{false}, 1 : \text{true}\}$.

$V_5 : \text{isRested}$, $D_{v_5} = \{0 : \text{false}, 1 : \text{true}\}$.

$V_6 : \text{Working}$, $D_{v_6} = \{0 : \text{false}, 1 : \text{true}\}$.

Table 5.7: Example of a detailed NPC for Assassin’s Creed: Origins ($\text{Armorer}$). $\mathcal{M}_{\text{Armorer}} = \mathcal{M}_{\text{Citizen}} \cap \{v_j \mid j \in [7, 10]\}$.
and sabers. We have given our citizen the possibility to eat, drink, buy some food, wander, debate and work (Table 5.6). Although our operator representation is quite generic, I have added some parameters to some of them (Buy Food, Buy Drink, Store, Take, Debate, Rest and Work) so as to allow some contextualization. For instance, if the NPC is a musician who like to play in the street, then we can contextualize: Work(?Luth, ?MarketPlace). We can also represent a group of musicians: it is merely a set of NPCs playing different instruments at the same place:

- Work$_{NPC_1}$(?Luth,?MarketPlace)
- Work$_{NPC_2}$(?Drum,?MarketPlace)
- Work$_{NPC_3}$(?Triangle,?MarketPlace)
- Work$_{NPC_4}$(?Voice,?MarketPlace)

All these musicians must take their instrument as well, so:

- Take$_{NPC_1}$(?Luth,npc1HomeLuthStand)
- Take$_{NPC_2}$(?Drum,npc2HomeChest)
- Take$_{NPC_3}$(?Triangle,npc3HomeChest)
- NPC$_4$ always has is voice, so his hasItem($v_2$) state variable is always true

I have added these parameters in order to illustrate that it is possible to contextualize an operator without modifying the overall representation. Furthermore, I wanted to point out that MoveTo operators must not be represented in any SAS-PU representation. MoveTo operators are related to pathfinding, representing them in any SAS-PU representation is almost impossible as it is very likely to not respect the post-uniqueness. A position to reach can merely be added as a parameter. Indeed, almost every action is preceded by a MoveTo, so it can implicitly be added inside the animation of the action.

Table 5.7 is a more specific Citizen: the Armorer. The Armorer is a Citizen, so it inherits the operators presented in Table 5.6. Plus, he has more detailed operators to accomplished its duty, namely: Put Metal in Foundry, Collect Smelted Metal, Forge a Sword, Forge a Saber. As these operators have to be performed while working, they all have the prevail-condition ($prv$): ($v_6 = 1$). Then, it can be noticed that the Armorer can debate while working, but not have a walk. Debate has no specific prevail-conditions, but Have a Walk requires the work in progress to be stopped ($prv$(Have a Walk)$[v_6] = 0$).

Let now describe some possible goal states for the citizen$^2$:

1. $s_*$ (Feed) = $\langle v_0 = 2, v_1 = 2, v_6 = 0 \rangle \oplus \langle v_i = s_0[v_i] \mid v_i \in \mathcal{M} \setminus \{v_0, v_1, v_6\} \rangle$
2. $s_*$ (Rest) = $\langle v_3 = 0, 0, 1, 0 \rangle \oplus \langle v_i = s_0[v_i] \mid v_i \in \mathcal{M} \setminus \{v_3, v_4, v_5, v_6\} \rangle$

$^2$We used $\oplus$ as a concatenation symbol here.
3. \( s_*(\text{Debate}) = \langle v_4 = 1 \rangle \oplus \langle v_i = s_0[v_i] \mid v_i \in \mathcal{M} \setminus \{v_4\} \rangle \)

4. \( s_*(\text{Wander}) = \langle v_1 = 2, v_3 = 1, v_6 = 0 \rangle \oplus \langle v_i = s_0[v_i] \mid v_i \in \mathcal{M} \setminus \{v_1, v_3, v_6\} \rangle \)

5. \( s_*(\text{Work}) = \langle v_0 = 2, 2, 1, 0, v_6 = 1 \rangle \oplus \langle v_i = s_0[v_i] \mid \mathcal{M} \setminus \{v_0, v_1, v_2, v_3, v_6\} \rangle \)

6. \( s_*(\text{ForgeSword}) = s_*(\text{Work}) \oplus \langle v_7 = 1, v_8 = 1, v_9 = 1 \rangle \)

7. \( s_*(\text{ForgeSaber}) = s_*(\text{Work}) \oplus \langle v_7 = 1, v_8 = 1, v_{10} = 1 \rangle \)

8. \( s_*(\text{WanderAndRest}) = \langle v_1 = 2, v_3 = 1, v_5 = 1 \rangle \oplus \langle v_i = s_0[v_i] \mid v_i \in \mathcal{M} \setminus \{v_1, v_3, v_5\} \rangle \)

9. \( s_*(\text{Eat,Debate,Rest}) = \langle v_0 = 2, v_4 = 1, v_5 = 1 \rangle \oplus \langle v_i = s_0[v_i] \mid v_i \in \mathcal{M} \setminus \{v_0, v_4, v_5\} \rangle \)

10. \( s_*(\text{BuySupplies}) = \langle v_0 = 1, v_2 = 1 \rangle \oplus \langle v_i = s_0[v_i] \mid v_i \in \mathcal{M} \setminus \{v_0, v_1\} \rangle \)

The description of the goals as well as some prevail-conditions are very precise because of SAS in which restriction (S7) imposes that \( s_0 \) and \( s_* \) must be totally defined. It may exists way to automate this, however. For example, the goal \( s_*(\text{Work}) \) could first be instantiated with the start state, then each required values on specific state variables could be found and then defined with the main operator, namely \( \text{Work} (a^6_{v_1}) \), by looking at its prevail-conditions and its post-condition:

\[
\begin{align*}
\forall v_i \in \mathcal{M} & \text{ s.t. } \text{prv}(\text{Work})[v_i] \neq u, s_*(\text{Work})[v_i] = \text{prv}(\text{Work})[v_i], \\
\text{s_*(Work)} & = s_0,
\end{align*}
\]

Let imagine an idle armorer that is hungry, thirsty and tired, we have:

\[
s_0^{\text{Armorer}} = \langle 0, 0, 0, 0, 0, 0, 0, 0, 0 \rangle
\]

This is the beginning of the day and he has to work, however. And we also want him to have rested and tidied up his belongings at the end of the day, we can give him the following goal:

\[
s_*^{\text{Armorer}} = \langle 2, 2, s_0[v_2], s_0[v_3], s_0[v_4], s_0[v_5], s_0[v_6], 1, 1, 1, 1 \rangle
\]

\( s_*^{\text{Armorer}} \) expresses that he has fed \( (v_0 = 2) \) and hydrated \( (v_1 = 2) \) himself during the day, that he has forged sabers and swords \( (v_9 = 1, v_{10} = 1) \) using his foundry \( (v_8 = 1, v_9 = 1) \), and that he is now doing what he was doing at the beginning of the day, namely idling \( (v_i = s_0[v_i], \forall i \in [2,6]) \). If we also want our armorer to rest, we can add to the goal: \( s_*[v_3] = 1 \). Which will give the opportunity for our NPC to rest.

With our topological planning algorithm, the minimal solution plan is:

\[
\Delta = \langle \text{Buy Food}(?\text{pos}), \text{Eat}, \text{Buy Drink}(?\text{pos}), \text{Drink}, \text{Rest}(?\text{pos}), \text{Take}(?\text{item}, ?\text{pos}), \text{Work}(?\text{item}, ?\text{pos}), \text{Put Metal in Foundry}, \text{Collect Smelted Metal}, \text{Forge a Sword}, \text{Forge a Saber}, \text{End work}, \text{Store}(?\text{item}, ?\text{pos}) \rangle
\]

Concerning the parameters \textit{item} and \textit{pos}, our armorer works at a blacksmith and he needs a hammer that could be stored in a chest. We can thus contextualize the
CHAPTER 5. BENCHMARKS AND RESULTS

Figure 5.10: All the domain operator graphs of my representation of Assassin’s Creed: Origins. These graphs present the post-pre dependencies between operators. Ghost operators are operators that can be identified with the pair \((v_i, p)\) (Definition 4.1.1) but do not exist in the problem (Definition 4.2.3).

following operators: Take (?hammer, ?armorerChest); Store (?hammer, ?armorerChest); Work(?hammer, ?blacksmith). He can buy food and drinks at the shop, thus: Buy Food (?shop), Buy Drink(?shop). And finally, he can rest at many places, such as his bedroom, stairs, benches, etc. Thus, if there is a staircase while going to work, we can have: Rest(?stairs). It should be noticed that the contextualization of an action must not be done while planning. It must either be update before planning thanks to dedicated variables, or it can be evaluated during the execution of the action. To contextualize an action while planning is burdening the planner and threatens the real-time constraint.

5.2.3 Horizon Zero Dawn

Horizon Zero Dawn (HZD) is an action role play game developed by Guerrilla Games and released in early 2017 [13]. The player plays a huntress named Aloy in a world full of high-tech colossal machines that have appeared at the fall of human civilization. These machines are categorized in classes: the acquisition class, whose role is to harvest resources; the combat class, whose role is to guard vulnerable machines from hunters; the recon class, whose initial role was to search for suitable terraforming land, they finally serve as guards and lookouts in the game; the transport class, whose role is to help acquisition machines to move large amount of resources. There are other types of machines whose description can be found in [47]. In the following, I present a possible SAS-PUSC\(_0\) representation for acquisition machines.

The acquisition machines, which I henceforth denote A-M, harvest resources of the HZD world. Their role is crucial as they assist the terraforming program and allow for the further construction of machines. This description, however, is more folkloric than anything else as this is not what really happens in-game. Although there exist stunning combat scenes in HZD, which are mainly due to great animations, the overall behavior of the machines
Figure 5.11: The operator graph of an armorer citizen of Assassin’s Creed: Origins. The post-pre dependencies are in blue whereas the post-prevail dependencies are in green.
Figure 5.12: The declobbered operator graph of an armorer citizen of Assassin’s Creed: Origins, with red directed edges representing the possible threats between actions (Definitions 4.2.1 and 3.2.5). For instance with Rest and Debate (on the right), if these operators are instantiated while planning, then the action Debate threatens the prevail-condition of the action Rest. The red edge between them may thus be added by $P$ to ensure correct sorting. The operators colored in yellow, in green and in darker green are involved in the restriction $C^*_2$. We need to remove the yellow operators to see whether the restriction is respected (cf. Figure 5.13).
Figure 5.13: The graph of operators of Assassin’s Creed: Origins, except that the operators Work and End Work are removed. This graph is here to illustrate the structural restriction $C_2$. The operators Work and End Work are in Cycle($v_6$) and both satisfy the $v_6$ prevail-condition of some other operators: Work satisfies the $v_6$ prevail-condition of “Green” = {Put Metal in Foundry, Collect Smelted Metal, Forge a Sword, Forge a Saber}, whereas End Work satisfies the $v_6$ prevail-condition of “Darker Green” = {Buy Food, Buy Drink, Take, Store, Have a Walk, Rest}. No green operator is connected to any of the darker green operators, therefore the structural restriction of $C_2^*$ is respected (Definition 4.4.6).
can be disappointing to those who seek consistency with the life goal of these machines. Through planning, however, it is possible to create operator sets and represent life goals to fit the description of these machines.

Let consider the following SAS-PUSC$_0$ problem that models an A-M. It has a unique goal: Harvest. An A-M repeats this goal each time it is completed, or reevaluates it each time the on-going plan is invalidated by one or more disruptions. For each A-M, we can model a harvest routine and some disruptions with the following state variable descriptions:

- $v_0: oreDepositFound$, $D_{v_0} = \{0 : false, 1 : true\}$.
- $v_1: oreDepositPosShared$, $D_{v_1} = \{0 : false, 1 : true\}$.
- $v_2: harvestRoutine$, $D_{v_2} = \{0 : none, 1 : oreHarvested, 2 : oreRefined, 3 : refinedOreStored\}$.
- $v_3: isCharged$, $D_{v_3} = \{0 : false, 1 : true\}$.
- $v_4: system$, $D_{v_4} = \{0 : damaged, 1 : operational\}$.
- $v_5: disturbanceExist$, $D_{v_5} = \{0 : false, 1 : true\}$.
- $v_6: isThreatened$, $D_{v_6} = \{0 : false, 1 : true\}$.

$v_0$ checks whether the A-M is surrounded by ore deposits. If not, the A-M can Scan for Ore Deposit, which allows it to find points of interest. I wanted the position of ore deposits to be a shared information between A-M. Thus, once the action Scan for Ore Deposit is over and succeeded, or if at least one A-M knows where there are some deposits, it can Share Ore Position to other machines. This operator affects the state variable $v_1$. As we cannot share a position without knowing it, $(v_0 = 1)$ is a prevail-condition for Scan for Ore Deposit. Then, $v_2$ takes care of the harvest routine. At the beginning of a scene, or when starting the game, $v_2$ is initially set to 0. It means two things: first the harvest routine has not yet started, and second the machine does not have any resource. $v_2$ will be reset to 0 each time the subgoal $(v_2 = 3)$ is accomplished, that is, each time an A-M succeeds to store a refined ore. Otherwise, $v_2$ keeps its value. It means that if the harvest routine is disturbed, the machine does not restart the on-going routine, but resume it instead once all the disruptions have been taken care of. Let imagine an A-M heading towards a storage to Store Refined Ore, then $v_2 = 2$. If the player attacks the A-M at this point, then the A-M has to reevaluate the situation with $s_0[v_2] = 2$ (and $s_0[v_6] = 1$ because it is under attack). Let now consider the A-M has accomplished its goal $s_4[v_2] = 3$, then the A-M has to reevaluate the situation with $s_0[v_2] = 0$ because it has to restart the whole harvest routine. I also give A-M some sensing state variables: $v_3$, $v_4$, $v_5$ and $v_6$. $v_3$ and $v_4$ are internal sensors that respectively check whether the system is charged or damaged. $v_5$ and $v_6$ sense the environment, updating their value if there is any disturbance or threat. To react to these disruptions, the A-M can: Rest, Repair, Scan towards Disturbance and Attack. These actions respectively affect $v_3$, $v_4$, $v_5$ and $v_6$. Tab. 5.8 gives the full representation of these operators.
CHAPTER 5. BENCHMARKS AND RESULTS

<table>
<thead>
<tr>
<th>$\mathcal{O}$</th>
<th>Pre</th>
<th>Post</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$o_{v_9}^1$</td>
<td>$v_0 = 0$</td>
<td>$v_0 = 1$</td>
<td>Scan for Ore Deposit(?fromPos)</td>
</tr>
<tr>
<td>$o_{v_1}^1$</td>
<td>$v_1 = 0$</td>
<td>$v_1 = 1$</td>
<td>Share Ore Position(?listOrePos)</td>
</tr>
<tr>
<td>$o_{v_2}^1$</td>
<td>$v_2 = 0$</td>
<td>$v_2 = 1$</td>
<td>Harvest Ore(?oreType,?orePos)</td>
</tr>
<tr>
<td>$o_{v_2}^2$</td>
<td>$v_2 = 1$</td>
<td>$v_2 = 2$</td>
<td>Refine Ore</td>
</tr>
<tr>
<td>$o_{v_2}^3$</td>
<td>$v_2 = 2$</td>
<td>$v_2 = 3$</td>
<td>Store Refined Ore(?storePos)</td>
</tr>
<tr>
<td>$o_{v_3}^1$</td>
<td>$v_3 = 0$</td>
<td>$v_3 = 1$</td>
<td>Repair</td>
</tr>
<tr>
<td>$o_{v_4}^1$</td>
<td>$v_4 = 0$</td>
<td>$v_4 = 1$</td>
<td>Scan towards Disturbance</td>
</tr>
<tr>
<td>$o_{v_4}^2$</td>
<td>$v_4 = 0$</td>
<td>$v_4 = 1$</td>
<td>Attack</td>
</tr>
<tr>
<td>$o_{v_5}^1$</td>
<td>$v_5 = 1$</td>
<td>$v_5 = 0$</td>
<td></td>
</tr>
<tr>
<td>$o_{v_6}^1$</td>
<td>$v_6 = 1$</td>
<td>$v_6 = 0$</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.8: (Based on HZD) Operator representation for the Acquisition Machines ($A$-Ms). $\mathcal{M}_{AM} = \{v_i \mid i \in [0,6]\}$ is the state variable set of the $A$-Ms.

There are two main points in this representation. First, an $A$-M must take care of all existing disruptions before doing anything else. That is, they have to be charged, operational, not threatened and not disturbed to perform its routine actions. Hence the prevail-conditions of Harvest Ore, Refine Ore, Store Refined Ore and Scan for Ore Deposits in Tab. 5.8. These prevail-conditions ensure that an $A$-M will not harvest resources while under attack, for instance. That, instead, it will Attack first because $(v_6 = 0)$ is required by these actions. Second, there is an order to the treatment of the disruptions. $(v_6 = 1)$ is more serious than $(v_5 = 1)$, which is more serious than $(v_3 = 0)$, which, finally, is more serious than $(v_4 = 0)$. This results in the prevail-conditions of the operators Attack, Scan towards Disturbance, Rest, Repair presented in Tab. 5.8. As $P$ returns a totally ordered plan thanks to a topological sort, this representation ensures that $P$ will only return relevant topological sorts as plans. For instance, the operator Attack solves the worst disruption, $(v_6 = 1)$ into $(v_6 = 0)$. As all other operators have $(v_6 = 0)$ as a prevail-condition, $P$ will have no choice but to return an action plan whose first action is Attack. On the contrary, Repair solves the least worst disruption, hence, if the other disruptions are on, the prevail-conditions of Repair ensure that Attack, Scan towards Disturbance and Rest will be first sorted by $P$. This way of writing operators is a knowledge representation, if we want to ensure that our planner returns a plan whose first action is to Attack when threatened, then $(v_6 = 0)$ must necessary be a prevail condition for all the other operators that may be instantiated while planning.

Remark: Concerning the eyes coloring in HZD, it is possible to represent the change of eye color with operators. I personally consider it as an animation that can be triggered when some state variables are being updated to specific values. As an example, if $v_6$ is updated to 1, the $A$-M is threatened so its eyes can turn red. If the system is disturbed $(v_5 = 1)$ and not threatened $(v_6 = 0)$, then the $A$-M can turn its eyes in yellow. In any other cases, its eyes are blue.
This way of describing dependencies between actions creates a very constrained problem. This can be highlighted with Figure 5.14 which shows the complete operator graph of my SAS-PUSC\(_0\) version of HZD. The blue directed edges represent the post-pre dependencies while the green directed edges represent the post-prevail dependencies. As there is only one goal, however, it is necessary to constraint the problem so as to ensure a pertinent totally ordered plan by the planner.

Eventually, this HZD version is a SAS-PUSC\(_0\) problem. It is obviously (P), (U), and (S). The new structural restriction, \(C_k\) with \(k = 0\), means no requestable operator (Definitions 4.4.1 and 4.1.4) is in the cycle of its domain operator graph. This is the case in this HZD version because there is no cycle in any of the domain operator graph of the problem, as it can be seen in Figure 5.15.

Now that the world and some operators for the A-Ms have been modeled, the goal Harvest and some possible start states can be defined.

\[ s_*(\text{Harvest}) = (1, 1, 3, 1, 1, 0, 0) \]

This goal means that the A-M is charged, operational and safe, it knows where there are ore deposits and the positions are shared with its counterparts. Plus, it has completed its harvest routine: \(v_2 = 3\). In this representation, the A-Ms will always try to reach this goal. So let’s create some start variations so that the planner can return different plans:
Figure 5.15: The domain operator graphs of the Acquisition Machine (A-M) model. Each graph represents the post-pre dependencies between some operators. Ghost operators are operators with an existing identifier (cf. Definition 4.1.1) but that do not exist in the problem.

- $s_0(1) = \langle 0, 0, 0, 0, 1, 1 \rangle$, this is the worst scenario, the A-M is threatened, disturbed, damaged, not charged, has not started its harvest routine and does not have any clue where ore deposits are.

- $s_0(2) = \langle 1, 1, 0, 0, 0, 1 \rangle$, Same situation but this time the A-M knows where to start harvesting.

- $s_0(3) = \langle 1, 1, 0, 1, 1, 0 \rangle$, the A-M can safely start to Harvest Ore as all its prevail-conditions are satisfied.

- $s_0(4) = \langle 1, 1, 2, 0, 1, 0 \rangle$, the A-M is low on energy, it needs to rest before resuming its harvest routine.

- $s_0(5) = \langle 0, 0, 0, 1, 1, 0 \rangle$, the A-M is safe and operational, it can harvest ores but it first needs to Scan for Ore Deposits. It also has to Share Ore Position at some point.

- $s_0(6) = \langle 1, 1, 1, 1, 1, 1 \rangle$, the A-M can start to refine the ore it has already harvested, but it is actually threatened and disturbed, so it has to manage the disruptions first.

5.2.4 Runtime Performances

Now that the realistic benchmarks have been introduced, I will use them to carry out experiments, presented in the first paragraph of this subsection, that aim to show and compare the performances of the algorithms $P$ and $P$. In a second paragraph, I show the results of the experiments and then discuss the performances of both algorithms. In the third paragraph, I go deeper in the analysis of the performance of $P$ by redoing an experiment with the SAS-PUC$^2$ version of ACO and by using the software AMD $\mu$Prof to perform a CPU analysis. I finally discuss and conclude the results.
CHAPTER 5. BENCHMARKS AND RESULTS

Figure 5.16: The performance of $P$ when solving HZD (blue square) and the SAS-PUC$_2^*$ versions of RDR-2 and ACO (red and yellow crosses resp.)

Description of the experiments  Even though my algorithm $P$ is theoretically linear in the number of operators plus the number of orders between actions, it does not imply the real-time constraint is respected in practice. As I aim to provide a planner capable of giving as many plans as possible in a very short amount of time, my C++ implementation of $P$ must be tested accordingly. The question is: “How many NPCs in RDR-2, ACO or in HZD can get a plan in real-time from my C++ implementation of $P$?”

First of all, planning in real-time, in video games, means planning between two frames. Taking into account that only 10% of the processing budget is granted for AI system, planning in real-time for a 60 FPS$^3$ video game means the planner has $\frac{1}{60} \times 0.1 = 1.67 ms$ of planning time. As a consequence, the runtime threshold I have set for the experiment is $1.5 ms$. This is below $1.67 ms$ and leaves a margin of error of 10%, or $0.17 ms$, in practice. For the experiment, I did not implement $P$ nor the benchmarks described in subsections 5.2.1, 5.2.2 and 5.2.3 in their respective and original game as I do not have their source code nor the time to do it. Therefore, I have implemented the benchmarks RDR-2, ACO and HZD in a C++14 project with default settings for Microsoft Visual Studio 2019, in order to solve them with my C++ implementations of $P$. Some logs were displayed with the Command Prompt in Windows 10 whereas runtime data (and some other data) were collected on .txt files and then analyzed with Mathematica. Each C++ simulation has a specific configuration detailed below:

$^3$FPS: Frames per Second
CHAPTER 5. BENCHMARKS AND RESULTS

- **RDR-2**: As described in the subsection 5.2.1 and in appendix F, there are 7 types of NPCs: the Horse Breeder, the Farrier, the Bandit, the Native Hunter, the Innkeeper, the Dancer and the Drunk Citizen. Overall, there are 30 state variables, 56 operators and the biggest state variable domain is $D_{v_5}$ from the Farrier with 5 values. For the experiment, all NPCs start with $s_0 = \langle v_i = 0 \mid v_i \in \mathcal{M}_{RDR-2} \rangle$ and try to reach one of their goals (Table F.8). $P$ had to return a plan for each of the 9 goals successively and an increasing number of times, thus simulating the growth of the number of NPCs.

- **ACO**: There are two types of NPCs: the Citizen and the Armorer. Overall, there are 11 state variables, 18 operators and the biggest domains are $D_{v_0}$ and $D_{v_1}$ with 3 values. Each NPC starts with $s_0 = \langle v_i = 0 \mid v_i \in \mathcal{M}_{ACO} \rangle$ and try to reach one of the 9 goals described in subsection 5.2.2. $P$ had to return a plan for each of the 9 goals successively and an increasing number of times.

- **HZD**: There is only the acquisition machines (A-Ms). Each A-M has 7 state variables, 9 operators and its biggest domain is $D_{v_2}$ with 3 values. Compared to the NPCs of RDR-2 and ACO, the A-Ms only have one goal but 6 different starts described in subsection 5.2.3. $P$ had to return a plan for each of the 6 starts successively and an increasing number of times.

Finally, with these configuration, I measure and collect the runtime it takes $P$ to provide a plan to a growing number of NPCs in these C++ simulations of RDR-2, ACO and HZD. The result of these experiments are given in Figure 5.16.

I have carried out the same experiment with $P$ to test it and to compare its performances with those of $P$. However, $P$ solves problems of the class SAS$^+-$PUS. There are some common problems with those of SAS-PUC$^*$ that can be solved by both algorithm, however. These problems are grouped into classes of problems like SAS-PUSC$_0$ and SAS-PUSC$_2$. Hence, HZD does not need to be reworked as it is SAS-PUSC$_0$, RDR-2 and ACO, on the contrary, require some changes. RDR-2 is SAS-PUC$^*$ because of the Horse Breeder and the Farrier problems. If we remove these two types of NPCs, then RDR-2 becomes SAS-PUSC$_2$. Indeed, all the remaining NPC problems are either SAS-PUSC$_0$ (Dancer, Innkeeper) or SAS-PUSC$_2$ (Bandit, Drunk Citizen, Native hunter). This SAS-PUSC$_2$ version of RDR-2 has 20 state variables, 33 actions and the highest state variable domain is $D_{v_{27}}$ from the Dancer with 4 values$^4$. Concerning ACO, the Citizen problem is SAS-PUSC$_2$ but the Armorer problem is SAS-PUSC$_2$. I have reworked the Armorer to produce a SAS-PUSC$_2$ version of it, presented in Table 5.9. Concerning the goals of the Armorer, most of them remain unchanged (cf. subsection 5.2.2), but he no longer has $s^*$ (Work) and:

- $s^*$ (ForgeSword) = $\langle 2, 2, 1, 0, 0, 1, s_0[v_6], 1, 1, 1, s_0[v_{10}] \rangle$
- $s^*$ (ForgeSaber) = $\langle 2, 2, 1, 0, 0, 1, s_0[v_6], 1, 1, s_0[v_9], 1 \rangle$

$^4$I have kept the state variable indices for the SAS-PUSC$_2$ version of RDR-2. That is why the Dancer still has a state variable domain on $v_{27}$ even if only 20 state variables remain.
CHAPTER 5. BENCHMARKS AND RESULTS

Table 5.9: The SAS-PUSC\textsubscript{2} version of the Armorer of ACO. This version of the Armorer inherits the operators from the Citizen, except the operators Work and End Word. The state variable \( v_6 \): Working is still part of \( M_{\text{Armorer}} \) but its value is set to 0 permanently to satisfy the \( v_6 \) prevail-condition of the inherited operators. In this version we consider the Armorer works as soon as the planner uses the operators presented in this table. The Citizen problem is SAS-PUSC\textsubscript{2} in its original form.

This new version of ACO has 11 state variables, 18 operators and the biggest domains remain \( D_{v_0} \) and \( D_{v_1} \) with 3 values. Eventually, HZD and the new versions of RDR-2 and ACO are solvable by both \( \mathbb{P} \) and \( \mathcal{P} \). The runtime performances of these two algorithms solving these benchmarks are shown in Figure 5.16 for \( \mathbb{P} \) and in Figure 5.17 for \( \mathcal{P} \).

Finally, the runtime performances of \( \mathbb{P} \) and \( \mathcal{P} \) presented in Figures 5.16, 5.17 and 5.18 take into account the initialization runtime, the planning runtime and the runtime of the variables clearing for each problem instance. For example with the Horse Breeder, \( s_0 = \langle 0,0,0 \rangle \) and \( s_* = \langle 2,0,2 \rangle \) form an instance of the Horse Breeder problem. For the experiment, I have measure the time it takes to initialize this instance, how many times it takes \( \mathbb{P} \) to solve it and how many times it takes to clear all the local variables. This measurement procedure is done for every single NPC to produce the data presented in this subsection.

Explanation of the performances of both \( \mathbb{P} \) and \( \mathcal{P} \) First of all, concerning the data symbols used in Figure 5.16 and in Figure 5.17, the square is for SAS-PUSC\textsubscript{k} problems (\( k = 0 \) or \( k = 2 \)) whereas the cross is for SAS-PUC\textsubscript{2} problems. HZD is a SAS-PUSC\textsubscript{0} problem in both experiments, the data shown in Figures 5.16 and 5.17 are the same. In Figure 5.16, the red and yellow crosses refer to the SAS-PUC\textsubscript{2} version of RDR-2 and ACO respectively, whereas the red and yellow squares in Figure 5.17 refer to the SAS-PUSC\textsubscript{2} versions of RDR-2 and ACO. Finally, in Figure 5.18, I have used the black color for the performances of \( \mathcal{P} \) to distinguish them from those of \( \mathbb{P} \): the triangle and dot symbols are for the SAS-PUSC\textsubscript{2} versions of RDR-2 and ACO respectively, the diamond is for HZD.

The linear growth of these curves is explained by the fact the worst-case time complexities of both \( \mathbb{P} \) and \( \mathcal{P} \) are not a function of the number of NPCs. Therefore, the slope of these curves is merely the ratio of the runtime growth to the number of NPCs growth. It
results that the slope highlights the difficulty of both algorithms to solve a problem. The steeper the slope, the more difficult the problem. From these explanations, we can observe that $P$ and $P$ do not have the same criteria of difficulty. For instance, HZD is the hardest problem for $P$ whereas this is RDR-2 for $P$. Table 5.10 gives the causality of this difference in difficulty. Table 5.10 shows the linear regressions of the sets of data, where the argument $x$ refers to the number of NPCs, shows the number of state variables ($|M|$) and the average number of orders between actions ($\text{avg}(|E_A|)$) for each problem. The performances of $P$ are sorted in descending order of $\text{avg}(|E_A|)$ in Table 5.10. It highlights the slope are correlated with the average number of orders$^5$. It is consistent with the worst-case time complexity of $P$ which is linear in the number of orders plus the number of actions: $O(|A| + |E_A|)$. In other words, the more orders, the more difficult for $P$. It explains why HZD is the most difficult problem for $P$ despite being the one with the less state variables and with the less operators. It is in fact more constrained than the other two, and besides, it is directly linked to the number of the defined prevail-conditions shown in Tab 5.8. Concerning $P$, its performances are sorted in descending order of $|M|$ in Table 5.10. It highlights that the difficulty to solve a problem is correlated with the number of state variable ($|M|$). This is also consistent with its worst-case time complexity, which is quadratic in the number of state variables. Hence, the more state variables, the more difficult for $P$.

$^5$The slope of HZD is lower than the one of the SAS-PUSC$_2$ version of RDR-2 but the curve is still above the one of RDR-2 in Figure 5.16. This inaccuracy is linked to the noise of the data.
Finally, the key result of these data is that my C++ implementation of $\mathcal{P}$ was able to provide a plan to 3.4 millions of NPCs in less than 1.5 ms for each of the three realistic benchmarks, which is huge and definitely respect the real-time constraint. On the contrary, $\mathcal{P}$ was only able to provide a plan to a hundred NPCs for each benchmark in this same amount of time. It implies my C++ implementation of $\mathcal{P}$ is about 34,000 times faster than my C++ implementation of $\mathcal{P}$. There are many reasons explaining the huge performance differences between $\mathcal{P}$ and $\mathcal{P}$. The first one is the pre-processing, or setup, for $\mathcal{P}$. I have setup a very rigorous memory management (malloc, memset, hashing table), and I have allocated and filled data structures as much in advance so as to not lose time while planning. This setup does not explain everything, however. I have implemented a setup version of $\mathcal{P}$ and the setup version has only speeded up the result by a factor of 3. Another reason is that $\mathcal{P}$ does less work than $\mathcal{P}$ while planning. During the setup of $\mathcal{P}$, The neighborhood sets $\mathcal{N}_{\text{pre}}, \mathcal{N}_{\text{prev}}$ are created and some space are allocated for $\mathcal{N}_{\text{clol}}$, which is the neighborhood set of threatening actions (Def. 3.2.5). By doing so, $\mathcal{P}$ only browses the defined pre- and prevail-conditions while planning, whereas $\mathcal{P}$ browses all the prevail-conditions, the undefined one included. It makes $\mathcal{P}$ do a lot of useless work, especially with realistic benchmarks where the prevail-conditions are more often undefined than defined. Even with the most constrained SAS$^+$-PUS problem, which I have named Multi_Cycle_X, only half of all the prevail-conditions are defined, which means only half of the prevail checking is useful in $\mathcal{P}$.
CHAPTER 5. BENCHMARKS AND RESULTS

| Benchmark (class) | Figure       | Linear combination of $x$ | $|\mathcal{M}|$ | $\text{avg}(|E_A|)$ |
|-------------------|--------------|---------------------------|----------------|-------------------|
| HZD (SAS-PUSC$_0$) | Fig. 5.16 and 5.17 | $0.0676405 + (3.7637 \cdot 10^{-1})x$ | 7              | 12.6667          |
| RDR-2 (SAS-PUC$_2^0$) | Fig. 5.16 | $0.00337846 + (3.83027 \cdot 10^{-7})x$ | 30             | 11.7778          |
| ACO (SAS-PUC$_2^0$) | Fig. 5.16 | $-0.00938495 + (2.89321 \cdot 10^{-7})x$ | 11             | 10.0000          |
| RDR-2 (SAS-PUSC$_2$) | Fig. 5.17 | $0.0313575 + (2.46048 \cdot 10^{-7})x$ | 20             | 8.16667          |
| ACO (SAS-PUSC$_2$) | Fig. 5.17 | $0.0157308 + (1.54132 \cdot 10^{-7})x$ | 11             | 6.66667          |

Performances of $\mathcal{P}$

| Benchmark (class) | Figure       | Linear combination of $x$ | $|\mathcal{M}|$ | $\text{avg}(|E_A|)$ |
|-------------------|--------------|---------------------------|----------------|-------------------|
| RDR-2 (SAS-PUSC$_2$) | Fig. 5.18 | $0.00432027 + 0.015978x$ | 20             | 8.16667          |
| ACO (SAS-PUSC$_2$) | Fig. 5.18 | $0.00516015 + 0.00941788x$ | 11             | 6.66667          |
| HZD (SAS-PUSC$_0$) | Fig. 5.18 | $0.00467229 + 0.00731254x$ | 7              | 12.6667          |

Table 5.10: This table shows the performances of $\mathcal{P}$ and $\mathcal{P}$ in the light of the linear combinations of the number of NPCs ($x$) constructed from the data presented in the Figures 5.16, 5.17 and 5.18. The performances of $\mathcal{P}$ are sorted in descending order of the average number of orders between actions ($\text{avg}(|E_A|)$). The performances of $\mathcal{P}$ are sorted in descending order of the number of state variables ($|\mathcal{M}|$).

<table>
<thead>
<tr>
<th>Brand</th>
<th>Product</th>
<th>Default TDP</th>
<th>105W</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD</td>
<td>AMD Ryzen™ 7 2700X</td>
<td>Processor tech for CPU cores</td>
<td>12 nm FinFET</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Frequency</td>
<td>3.7GHz, up to 4.3GHz</td>
</tr>
<tr>
<td></td>
<td></td>
<td># CPU cores</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td># of Threads</td>
<td>16</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Total L1 cache</td>
<td>768Ko</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Total L2 cache</td>
<td>4MB</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Total L3 cache</td>
<td>16MB</td>
</tr>
</tbody>
</table>

Table 5.11: Characteristics of the AMD Ryzen™ 7 2700X used for the experiments.

CPU Analysis by Profiling $\mathcal{P}$ Given the impressive results of $\mathcal{P}$, it is crucial to have an internal view of the processor while performing $\mathcal{P}$. The experiments were performed with the following configuration: AMD Ryzen™ 7 2700X (8-Core) CPU (3.7GHz), 32Gb of RAM and Windows 10 (64 bits) (cf. Table 5.11 for more details); $\mathcal{P}$ was written in C++14 with default settings for Microsoft Visual Studio 2019. To observe the internal performance of the processor, we have used the software AMD µProf [48]. AMD µProf is a software used to perform CPU, GPU, and power analysis of applications running on Window®, Linux®, FreeBSD® operating systems on AMD processors. The AMD Ryzen™ 7 2700X has a base clock of 3.7GHz with a boost of up to 4.3GHz, which corresponds to $\approx 4 \cdot 10^9$ clock cycles per second. There is thus approximately a clock cycle every quarter of a nanosecond. Yet, our linear time algorithm $\mathcal{P}$ is able to provide a plan of size 4 (on average) to 3.4 millions citizens in the SAS-PUC$_2^*$ version of Assassin’s Creed: Origins (cf. Figure 5.16). It approximately gives $\approx \frac{1}{3} \cdot 10^{-9}$ second to build a plan, on average. That is, a third of a nanosecond to build one action plan. Eventually, it means $\mathcal{P}$ builds on average one plan in 1.2 clock cycles with the AMD Ryzen™ 7 2700X.

---

$^6$ $\mathcal{P}$ was written with the same configuration.
Figure 5.19: Inner analysis of the AMD Ryzen™ 7 2700X, on how the C++ implementation of \( \mathbb{P} \) is processed during the experiment. This graph shows the number of Retired Instructions (RI) and the Instruction Per Cycle (IPC) wrt. the number of NPCs. The left x-axis is for the number of RI, the right x-axis is for the IPC, and the y-axis is the number of NPCs. The solid curves show the RI whereas the dashed curves show the IPC, both wrt. \textit{TopoPlan}, the name of our algorithm; and wrt. \textit{SearchExtension}, \textit{BuildChain} and \textit{DFSTopo}, functions (methods) used by \textit{TopoPlan} and \textit{Init}, the method used to initialize the start and the goal states for each problem instance.

So, to observe how \( \mathbb{P} \) is processed while solving ACO by the AMD Ryzen™ 7 2700X, I have carried out the following experiments with the AMD µProf software: I asked \( \mathbb{P} \) to provide a plan to respectively 500,000, 1,000,000, 2,000,000, 3,000,000, 4,000,000 and 5,000,000 armorers and citizens in ACO. All at once respectively, i.e. I did not set any repetition to smooth the results. For each of the 6 numbers, I have profiled \( \mathbb{P} \) with the AMD µProf software, using the \textit{Assess Performance} profile type, in order to collect internal performance data from the AMD Ryzen™ 7 2700X. Figure 5.19 shows the internal performance, focusing on the \textit{number of retired instructions} (RI, solid curves with solid dots) and the \textit{instructions per cycle} (IPC, dashed curves with empty square symbols). The number of RI is the number of instructions retired from execution and the IPC is the average number of RI per CPU cycle.

Concerning the Figure 5.19, \textit{TopoPlan} is the C++ name I have given to \( \mathbb{P} \). \textit{BuildChain}, \textit{SearchExtension} and \textit{DFSTopo} are C++ methods used by \textit{TopoPlan} to solve a given problem instance. \textit{Init} is a method to initialize the start \( (s_0) \) and the goal \( (s_\ast) \) of a problem instance. The x-axis is the number of NPCs, for each curve there is respectively a point at 500,000, 1,000,000, 2,000,000, 3,000,000, 4,000,000 and 5,000,000 NPCs. The left y-axis is for the RI whereas the right y-axis is for the IPC. The AMD µProf software does not count
CHAPTER 5. BENCHMARKS AND RESULTS

precisely how many instructions it takes the processor to execute a program but counts samples instead. Thus, the left y-axis of Figure 5.19 does not give the precise number of instructions but a sample count of it, hence the term “retired” in retired instructions.

The higher the IPC, the better. It means the processor does not loose time due to performance bottlenecks like the cache misses or the branch mispredictions. Each element of my C++ implementation of $P$ is above 1.0 which is fine. The key point of this Figure is the linear growth in the number of RI for all the methods and with respect to the number of NPCs. It ensures that solution plans are not cached, that the processor fully applies the planner to solve all given problem instances.

With such performances, the use of the cache undeniable, however. In light of these last results, we can conclude that the cache is not used to store solution plans but that it may be used to cache the entire planner and all operators in the problem being evaluated to not loose time while executing the code, i.e. while planning.

5.2.5 Discussion

In this subsection, I discuss the creation of the benchmarks and the runtime performance difference between $P$ and $P$.

- Although relaxing the syntactical restriction (S) with the structural restriction ($C_2^n$) gives more freedom to the representation of operators, there are still concessions to be made. For instance, it is not possible to have the following situation: let consider two switches On/Off where the two On operators are required by an operator A and the two Off required by another operator B. All the operators are concerned by the restriction ($C_2^n$) and the two switches must be removed one after another from the operator graphs with the red directed edges to see whether ($C_2^n$) is respected. It is not respected because there is a path between A and B and a path between B and A in the resulting operator graph with the red directed edges (linked to threatening actions) after removing one switch. In such situation, the trick is to merge the two switches in one. This is what I did with the Native Hunter, with the operators Unequip and GetReadyToHunt. Initially, I provided the Native hunter with the operators EquipBow, UnequipBow, EquipQuiver, UnequipQuiver. To GoHunting the Native Hunter had to EquipBow and EquipQuiver and, then, to StoreTheGame the Native Hunter had to UnequipBow and UnequipQuiver. This model is equivalent to the two switches On/Off situation previously described and does not respect ($C_2^n$). Therefore, EquipBow and EquipQuiver were merged into GetReadyToHunt and UnequipBow and Un-equip Quiver were merged into Unequip. ($C_2^n$) is thus now respected. Eventually, I decided to create a SAS-PUSC$_2$ version by swapping the order between Store the Game and Unequip. Initially, the Native Hunter had to Unequip before StoringTheGame, in the final version (cf. Table F.7) this is the opposite.

- To collect the data presented in Figures 5.16, 5.17 and 5.18, I have set up experiments with a fixed start for each benchmark (a fixed goal for HZD), and with no more than 10 goals (or 6 starts for HZD). in other words, there is no more than 10 different problem instances per benchmark. Which implies for 3, 4 millions NPCs that problem instances were repeated at least 340 thousands times. I showed in subsection 5.2.4 that my AMD
Ryzen\textsuperscript{TM} 7 2700X does not cache the solution plans of the problem instances, however, but applies the planner completely every time despite the many repetitions. We did this internal analysis of the processor because the use of the function \textit{rand()} from the C library to randomly choose a goal (ACO and RDR-2) or a start (HZD) is not relevant in these experiments due to the very short time interval \mathbb{P} has to work with. Even if the function \textit{rand()} is called at each initialization of the start and the goal states, the time seed remains the same which implies the function \textit{rand()} always return the same number, therefore \textit{rand()} was not relevant. Then, the use of a for loop to choose one goal (or start for HZD) after another, or the use of a list of ten thousands unsorted number between 0 and the number of goals (or starts for HZD) to try to choose randomly a goal (or a start), produced eventually the same results.

- Data from \mathbb{P}, shown in Figures 5.16 and 5.17 are a bit noisy compared to those of \mathcal{P}, shown in Figure 5.18. The reason is that the two algorithms provide plans to a number of NPCs with a different order of magnitude. These data from \mathbb{P}, although noisy, were not easy to obtain because of the very short time interval implied by the real-time constraint, however. Compare to the C++ implementation of \mathcal{P}, I have not used the C++ Standard Library vector class to store operators, actions, plans and other useful internal lists for \mathbb{P}. The reason is that the vector class has an internal container with a fixed-size that may allocate some extra storage to accommodate for possible growth [49]. Even though it is possible to allocate some extra storage by hand with the function \textit{reserve}, it is not possible to \textit{reserve} the size of the container in a vector of vector. When \mathbb{P} was first coded with vectors, the dynamic allocation for the extra storage of the vectors were visible on the collected data. It created “jump” in the set of runtime data and the slope of each subset framed by two jumps was varying as well. In other words, the use of vectors with \mathbb{P} made the collected data unusable. As a result, my implementation of \mathbb{P} is coded with C functions like \textit{malloc} or \textit{memset} to create and allocate useful lists, variables, hashing tables and so on. Then, the internal functions (\textit{BuildChain}, \textit{SearchExtension}, \textit{TopoPlan}, \textit{DFSTopo}) use pointers and references as much as possible to avoid the manipulation of heavy objects, like actions. Besides, compared to \mathcal{P}, actions are not instantiated while planning in my C++ implementation of \mathbb{P}. Indeed, \mathbb{P} simply uses references to actions/operators from the hashing table that was built in the setup. On the contrary, \mathcal{P} plans for problems of the class SAS\textsuperscript{+}-PUS and, for some problem instances, the minimal solution plans have two times the same actions in it. As a result, the planning algorithm \mathcal{P} instantiates actions when building chain of actions and manipulate them during the entire planning process. In my C++ implementation of \mathcal{P} I tried to manipulate actions as little as possible and use references instead. \mathcal{P} returns a partial-ordered action plan \langle A, R \rangle, with \mathcal{A} the set of actions and \mathcal{R} the set of orders between actions. Instantiations of actions in my C++ implementation of \mathcal{P} occur only when building chains of actions, in order to insert them into \mathcal{A}, which I represent as a vector of actions. Finally, \mathcal{R} is a vector of integer vectors and it stores the orders between the references of the actions in \mathcal{A}. Eventually, although I am convinced the way I have implemented \mathbb{P} played a role in its performances, the quality of the AMD Ryzen\textsuperscript{TM} 7 2700X processor played a role as well [50]. In particular, the AMD Ryzen\textsuperscript{TM} 7 2700X has the \textit{Sense MI Technology} which means this processor uses the
Neural Net Prediction, a neural network whose purpose is to understand the applications and anticipate the next steps in the activity flow in realtime. This technology combined with many others, like pipelining, the use of dispatchers, the use of threads, etc. may have played a role as well.

• It seems that many factors explain the huge differences in performance between $\mathcal{P}$ and $\mathbb{P}$. Obviously, the complexity reduction played a role, especially when $m$ and $n$ are high. However, when $m$ is low ($< 100$), the difference in performance is huge and seems to grow, i.e. the lower $m$, the higher the difference in performance. The scalable problems highlight this point. $\mathbb{P}$ is 10 times faster than $\mathcal{P}$ in MultiPrv_n_Cycle when $m$ is low, and about 2 times faster when $m$ is high. In OnePrv_n_Cycle, $\mathbb{P}$ is 65 times faster than $\mathcal{P}$ when $m$ is low, and 178 times faster when $m$ is high. Even though these are significant differences in the runtime performance between the two algorithms, they are not as significant as the one on the realistic benchmarks ($\mathbb{P}$ is $> 34{,}000$ times faster than $\mathcal{P}$). I performed an “AMD experiment”, i.e. I checked the behavior of my processor when solving the realistic benchmarks, to ensure that plans were not cached. It was not the case, so they may be other explanations explaining the huge difference in performance. My intuition is that, when $m$ is small, the problem is small enough for my AMD processor to cache all the $\mathbb{P}$ functions and the problem inside the L1 cache.

• Most of the time, there are several different topological sorts for a given partial-order. Adding some randomization while selecting a node (action) to sort can bring further diversity for solution action plans. For ACOrigins, if the Citizen must feed himself, he can: Buy Food, Eat, Buy Drink, Drink. But he also can: Buy Drink, Buy Food, Drink and Eat. And if one wants to reduce the possibilities, then additional prevail-conditions must be considered.

• Concerning the knowledge representation, there are some actions that must be executed before any other action. For instance with how I represent the Acquisition Machines of Horizon Zero Dawn: as they only have one goal: Harvest, if they are threatened, the first action that must be executed is obviously Attack(?target); therefore, ($\text{isThreatened} = false$) is a prevail-condition for any other action in my representation. We can do that with as many actions as we want, it will ensure that, no matter the topological sort in $\mathbb{P}$, the result is a valid and totally ordered plan. The problem may become very constrained, however, as shown in Figure 5.14. So, I doubt this representation is easily sustainable for large projects, although it allows planners to create well-ordered action plans with more than 10 actions. The alternative is to rely on the Goal system to help to handle the behavior consistency. This is the case in F.E.A.R., you definitely do not want your NPCs to patrol when the player is spotted, so the Attack action must be executed before the Patrol action. The state ($\text{TargetIsDead} = true$) is not a precondition of Patrol, however. The Goal system handles the situation instead, by giving the goal KillEnemy, then by giving the goal Patrol when the scene is safe again.

• Given any SAS problem ($\Phi = (\mathcal{M}, \mathcal{S}, \mathcal{O})$), the start $s_0$ and the goal $s_\star$ states create instances of the problem ($\prod = (\Phi, s_0, s_\star)$). $s_0$ and $s_\star$ are a collection of defined state variables (SAS-structure), these states variables are said to model the world of the NPCs but they can be used for the sole purpose of planning. Let consider the Horse Breeder
problem from Table F.5, the state variables *Haystack* and *Water* and their respective domain are useful to create plans in order to fill the feeder and the water trough, respectively. But, to decide whether the feeder or the water trough need to be filled, other state variables, like *FeederFilled* and *TroughFilled*, can be considered. These are sensing state variables that the *Horse Breeder* may use to decide whether he needs to $s_*(\text{Feed the horses})$, or just $s_*(\text{Fill the feeder})$, or just $s_*(\text{Fill the trough})$. Furthermore, in subsection 5.2.1, I mentioned that *Haystack* and *Water* can be seen as targetable items, so that the *Horse Breeder* can have item variables like $\text{targetHaystack} : \?\text{haystackItem}$ or $\text{targetWater} : \?\text{sourceOfWater}$ that store positions of the last seen items or the closest ones if the *Horse Breeder* knows his surroundings for instance. This discussion allows to highlight some differences between a decision-making system and a planning system. An NPC may be equipped with many different variables: state ones, item ones, sensing ones, etc. Some of these variables may be used by the decision-making system while the planning system may use others. Some variables can obviously be shared by both systems. According to the belief, the needs, the will, etc. of an NPC, the decision-making system will define a goal $s_*$ and a start $s_0$ for the planning system to return an action plan. Concerning targetable items, either the NPC is equipped with item variables that keep track of the useful items that surround him or, while executing the plan, the NPC seeks the required items. In any case, the search for an item, as well as the search for a path, must not be done while planning. This last point is about the temporal order of magnitudes. The planner works in real-time, a plan can be returned in few nanoseconds according to the results of section 5.2 whereas the execution time of an action is at least as long as its animation, i.e. several milliseconds or seconds. As a consequence, the planning system must not be burdened by any variable updates or searches. An NPC must either anticipate by updating as often as possible its variables or find what it needs while executing its plan. The second method may be less safe because it may lead to plan interruptions if requirements are not met while executing the plan. Whereas the first method can influence the decision-making system. For the *Horse Breeder*, let consider the following situation: the feeder is empty as well as his hay stock and he needs to wait until tomorrow for the supplier to restock, then $\text{targetHaystack} = \?\text{none}$ and $\text{FeederFilled} = \text{false}$. In this case, the *Horse Breeder* needs to $s_*(\text{Fill the feeder})$ but he cannot because he does not have haystacks anymore. Hence, the decision-making system will not give the goal $s_*(\text{Fill the feeder})$ to the *Horse Breeder*. On the contrary, with the second method, the decision-making system will give the goal $s_*(\text{Fill the feeder})$. Then, the execution of *Take Haystack* actions is equivalent to a search for an haystack, which will fail in this situation, thus forcing the *Horse Breeder* to interrupt the on-going plan.
6.1 How to Reduce Time Complexity to Scale up Real-time Planning?

Goal-Oriented Action Planning I conducted a study on the GOAP planner of F.E.A.R. and found that action costs, which were used by the cost function to evaluate the consistency of a plan, are in practice overshadowed by context preconditions. So much so that the actions become contextually post-unique (1). Then, I also highlight from the source code of F.E.A.R. that its developers tried to keep the action representation simple, that is each action only has few preconditions and only one post-condition. Which means actions are unary (2) in F.E.A.R. Finally, I highlighted that action plans are totally ordered (3) and there is no two times the same action in them (4). I have finally turned these four points into assumptions that I have used throughout my thesis.

Simplified Action Structure I recall the theory behind the Simplified Action Structure (SAS) formalism in this chapter. Even though GOAP is based on STRIPS, it is in fact closer to the SAS formalism. Furthermore, (1) and (2), two of my assumptions, are restrictions that C. Bäckström have used to create SAS\(^+\)-PUS problems, which are problems that can be solved by a polynomial time algorithm. As I aimed to reduce the complexity for scaling up realtime planning, I therefore implemented the SAS\(^+\)-PUS algorithm in C++, denoted \(\mathcal{P}\), and I tried it on realistic benchmarks, based on video games, which form SAS\(^+\)-PUS problems. The results were encouraging as \(\mathcal{P}\) is able to plan for hundreds of NPCs in real-time and by providing them longer action plans that current planners in GOAP system can do. The restriction (S), however, is very restrictive.

Topological Planning In this chapter, I relaxed (S) and presented new structural restrictions to be combined with (P) and (U) so as to create tractable problems for video games. As one of my assumption is no two times the same action in a solution plan, I introduced an output restriction denoted \(\Upsilon_1\) and which I use to create the \(\Upsilon_1\)-Truth Crite-
rion, a truth criterion which, if respected, ensures the solution plan is correct. \( \Upsilon_1 \) does not provide instructions to produce tractable problems, however. I therefore introduce a new restriction, denoted \( C_k \), whose role is to restrict to at most \( k \) the number of requestable operators inside the unique cycle of each domain operator graph. Indeed, I demonstrated in this chapter that the reason of the intractability of SAS-PU problems is operators affecting the same state variable and looping together. I therefore closely study and restrict, with \( C_k \), the cycles formed by operators looping together. In particular, I present the classes of problems SAS-PUC\(_0\), SAS-PUC\(_2^S\) and SAS-PUC\(_2^*\), and I proved that these problems are tractable because their solvable instances are solved by minimal solution plans that are \( \Upsilon_1 \), i.e. they do not have twice the same action in them. Eventually, I present TopoPlan, denoted \( P \), a linear time algorithm that solves instances of a SAS-PUC\(_0\), SAS-PUC\(_2^S\) or SAS-PUC\(_2^*\) problem. I proved that this algorithm is correct with respect to the \( \Upsilon_1 \)-Truth Criterion, and complete. I also proved that \( P \) as a worst-case space complexity of \( O(n^2 \cdot m^2) \), i.e. quadratic in the number of state variables times the size of the variable domain. Finally, I proved that \( P \)'s worst-case time complexity is \( O(|O| + |E_O|) \), i.e. linear in function of the number of operators plus the number of orders between the operators. I therefore succeeded in answering one part of my main question, the one on how to reduce time complexity.

**Benchmarks and Results** In these chapter, I present the runtime performance of both \( P \) and \( P \) on different benchmarks. I first test \( P \) and \( P \) on scalable problems, i.e. problems whose resolution difficulty grows in function of the number of state variables or in function of the size of each state variable domain. These benchmarks allow me to show empirically the time complexity of both \( P \) and \( P \). It also shows that, overall, \( P \) is faster than \( P \).

Then, I introduced realistic benchmarks: RDR-2, ACO and HZD, which are based on the commercial video games Red Dead Redemption, Assassin’s Creed: Origins and Horizon Zero Dawn respectively. These benchmarks allow me to show that my tractable classes of problems are applicable to realistic problems. In particular, they allow the creation of problems that were not possible with SAS\(^+\)-PUS and that are closer to real-world problems. The runtime performance of \( P \) on some instances of these realistic problems were astounding, \( P \) was able to produce millions of plans in real-time. With adapted instances that can be solved by both \( P \) and \( P \), I show that \( P \) is way faster than \( P \) to solve them.

The creation of operators remains tricky with my classes of problems, however, as some concessions may be required to stay within the restrictions. It is worth the effort, however, because instances of such problems are solved by \( P \), my linear-time algorithm with which I have proven that once optimized it is possible to plan in real-time for millions of NPCs. Eventually, the main conclusion in light of the performances of \( P \) is that planning in real-time on CPU is feasible on a large scale with current technologies.

### 6.2 Future Work

In this section, I give some ideas to pursue the researches presented in my PhD manuscript.
SAS-PUC<sub>k</sub> problems that are Υ<sub>2</sub>  The next step after SAS-PUC<sub>k</sub> problems that are Υ<sub>1</sub> are obviously those that are Υ<sub>2</sub>, then Υ<sub>3</sub> and so on. In my opinion, the study of the unique cycle of the domain operator graph remains crucial for those who want to explore these problems.

Another step is to consider SAS<sup>*</sup>- or SAS<sup>+</sup>-structures with the restrictions (P), (U) and C<sub>k</sub>. These two structures allow state variables to be undefined in the goal state, while the SAS<sup>+</sup> also allows start state to be undefined. It adds a notion of uncertainty which can be of interest for some people.

Hierarchical Topological Planning  In the first paragraph of subsection 5.2.5, concerning the Native Hunter, I discussed how I have merged the pair of operators EquipBow/UnequipBow and EquipQuiver/UnequipQuiver into the pair GetReadyToHunt/Unequip because the two first pairs of operators did not respect the restriction C<sup>*</sup><sub>2</sub>. But, GetReadyToHung and Unequip can in fact be seen as sub-goals whose minimal solution plans are ⟨EquipBow, EquipQuiver⟩ and ⟨UnequipBow, UnequipQuiver⟩ respectively. This observation paves the way to hierarchical planning with P. One can creates problems of sub-problems of sub-sub-problems... and then applies P to solve each problem instance after another in order to get a minimal solution plan for a given agent.

Apply TopoPlan in GOAP systems  It is a matter of maturity in order to increase the Technology Readiness level of P. I have proved theoretically and empirically my algorithm and I have applied it on realistic benchmarks and it has astonishing result. Integrating it into a project while ensuring that it works and interacts well with other subsystems is certainly a next step.
Appendices
The idea is to use the identification technique of Definition 4.1.1 so as to store operators in an hashing table, which can be done with an array (or a vector of vector in C++) for instance. Any action can then be instantiated in a constant time ($O(1)$) via the procedure $\text{FindPost}$ by looking after the corresponding operator in the pre-built hashing table. It requires a preprocessing (l.xx to yy), however, but it allows C.Bäckström to reduce the worst-case time complexity of his algorithm from $O(n^2 \cdot m^2)$ to $O(n \cdot m^2)$.
APPENDIX A. MODIFIED PLANNING ALGORITHM FOR SAS⁺-PUS PROBLEMS

Procedure 5 FindPost($\mathcal{H}_V, x, i$)

Input: $\mathcal{H}_V$, vector implementation of $\mathcal{H}$; $x \in D_v$.
Parameters: $a$, action.
Output: $a$, instantiated action from $\mathcal{H}$.

1: $a \leftarrow \text{Instantiate}(\mathcal{H}_V[i][x])$
2: return $a$

Procedure 6 BuildChain($\mathcal{H}_V, x, g, v_i, A, T, R, U_V$)

Input: $\mathcal{H}_V, A, T, R, U_V$ are called by reference; $x, g \in D_v$.
Parameters: $C$, chain of actions; $a$, an action.
Output: $C$.

1: if $x \sqsubseteq g$ then return $\langle \rangle$
2: end if
3: $a \leftarrow \text{FindPost}(\mathcal{H}_V, g, v_i)$
4: if $a = \emptyset$ or $U_V[g]$ then Fail.
5: end if
6: $U_V[g] \leftarrow \text{true}$
7: $C \leftarrow \text{BuildChain}(\mathcal{H}_V, x, \text{pre}(a)[v_i], v_i, A, T, R, U_V)$
8: $U_V[g] \leftarrow \text{false}$
9: $A \leftarrow A \cup \{a\}; T \leftarrow T \cup \{a\}$
10: if $C \neq \langle \rangle$ then
11: Order Last($C$) before $a$ in $R$
12: end if
13: return $C \oplus \langle a \rangle$
Algorithm 7 Plan($\mathcal{M}, \langle D_{v_1}, ..., D_{v_{|\mathcal{M}|}} \rangle, \mathcal{H}, s_0, s_\star$) - Part 1

**Input:** $\mathcal{M}$, set of state variables; $\mathcal{H}$, set of operators; $\langle D_{v_1}, ..., D_{v_{|\mathcal{M}|}} \rangle$, Domains of state variables; $s_0$ and $s_\star$, start and goal states respectively.

**Parameters:** $\mathcal{A}, \mathcal{T}$, sets of actions; $\mathcal{H}_V$ hashing table of operators; $\mathcal{R}$, set of relations; $\mathcal{R}_1, \mathcal{R}_2$, boolean matrices; $F$, boolean list; $\bar{\alpha}_{v_i}$, $\bar{\beta}_{v_i}$, $\bar{\gamma}_{v_i}$, chains of actions.

**Output:** $\langle \mathcal{A}, \mathcal{R} \rangle$, non-linear action plan. Tuple with $\mathcal{A}$ the set of actions and $\mathcal{R}$ the relations between them.

1: $\mathcal{A} \leftarrow \emptyset$; $\mathcal{T} \leftarrow \emptyset$; $\mathcal{R} \leftarrow \emptyset$
2: $\mathcal{R}_1 \leftarrow$ size $|\mathcal{M}| \times 2|\mathcal{H}|$ boolean matrix. All entries set to false
3: $\mathcal{R}_2 \leftarrow$ size $2|\mathcal{H}| \times |\mathcal{M}|$ boolean matrix. All entries set to false
4: $F \leftarrow$ size $|\mathcal{M}|$ boolean list. All entries set to false
5: $\mathcal{H}_V \leftarrow$ size $|\mathcal{M}|$ vector of vector pointers
6: for $v_i \in \mathcal{M}$ do
7: \hspace{1em} $\bar{\alpha}_{v_i} \leftarrow \langle \rangle$; $\bar{\beta}_{v_i} \leftarrow \langle \rangle$; $\bar{\gamma}_{v_i} \leftarrow \langle \rangle$;
8: \hspace{1em} $\mathcal{H}_V[v_i] \leftarrow$ size $|D_{v_i}|$ vector of action types with entries initialized to $\emptyset$
9: end for
10: for $h \in \mathcal{H}$ do
11: \hspace{1em} $v_i \leftarrow$ Affects($h$)
12: \hspace{1em} $\mathcal{H}_V[v_i][post(h)[v_i]] \leftarrow h$
13: end for
14: $U_V \leftarrow$ size $\max_{v_i \in \mathcal{M}}(D_{v_i})$ boolean vector. All entries set to false
15: for $v_i \in \mathcal{M}$ do
16: \hspace{1em} $\bar{\gamma}_{v_i} \leftarrow BuildChain(\mathcal{H}_V, s_0[v_i], s_\star[v_i], v_i, \mathcal{A}, \mathcal{T}, \mathcal{R}, U_V)$
17: end for
APPENDIX A. MODIFIED PLANNING ALGORITHM FOR SAS⁺-PUS PROBLEMS

18: while $T \neq \emptyset$ do
19:   Select arbitrary $a \in T$ and let $T \leftarrow T \setminus \{a\}$
20:   for $v_i \in M$ do
21:     if $prv(a)[v_i] \neq u$ then
22:       {If $a$ is the first action to be inserted into $F[i]$, then build $\alpha_{v_i}$ and $\beta_{v_i}$}
23:         if not($F[v_i]$) then
24:           $F[v_i] \leftarrow true$
25:           if $InSeq(\gamma_{v_i}, prv(a)[v_i], v_i)$ then
26:             SplitSeq($\gamma_{v_i}, \alpha_{v_i}, \beta_{v_i}, prv(a)[v_i], v_i$)
27:           else
28:             $\alpha_{v_i} \leftarrow BuildChain(H_V, s_0[v_i], prv(a)[v_i], v_i, A, T, R, U_V)$
29:         end if
30:       end if
31:       if $s_2[i] = u$ then
32:         $\beta_{v_i} \leftarrow \langle \rangle$
33:       else
34:         $\beta'_{v_i} \leftarrow BuildChain(H_V, prv(a)[v_i], s_0[v_i], v_i, A, T, R, U_V)$
35:         $\beta_{v_i} \leftarrow \beta'_{v_i} \oplus \gamma_{v_i}$
36:         if $\beta'_{v_i} \neq \langle \rangle$ and $\gamma_{v_i} \neq \langle \rangle$ then
37:           Order Last($\beta'_{v_i}$) before First($\gamma_{v_i}$) in $R$
38:         end if
39:       end if
40:     end if
41:     if $\alpha_{v_i} \neq \langle \rangle$ then
42:       $R_1[v_i, label(a)] \leftarrow true$
43:     end if
44:     if $\beta_{v_i} \neq \langle \rangle$ then
45:       $R_2[label(a), v_i] \leftarrow true$
46:     end if
47:   end for
48: end while
APPENDIX A. MODIFIED PLANNING ALGORITHM FOR SAS⁺-PUS PROBLEMS

50: \{Copy $R_1$ and $R_2$ into $R$ without duplicate entries.\}
51: for $a \in A$ do
52: $v_j \leftarrow \text{Affects}(\text{ope}(a))$
53: for $v_i \in A$ do
54: if $R_2[\text{label}(a), v_i]$ then
55: Order $a$ before $\text{First}(\bar{\beta}_{v_i})$ in $R$
56: end if
57: if $R_1[v_i, \text{label}(a)]$ and (First($\bar{\beta}_{v_j}$) $\neq a$ or not($R_2[\text{label}(\text{Last}(\bar{\alpha}_{v_i}), v_j)$) then
58: Order $\text{Last}(\bar{\alpha}_{v_i})$ before $a$ in $R$
59: end if
60: end for
61: end for
62: \{Test whether $R$ is irreflexive\}
63: if $R$ is not irreflexive then
64: Fail.
65: else
66: return $\langle A, R \rangle$
67: end if
Domain Operator Graphs with (PU) Operators

Figure B.1: The Mathematica source code to produce the domain operator graphs presented in the appendix, or with higher size.

```mathematica
MakeChains[v_] :=
  Gather[ParallelMap[AdjacencyGraph[#, DirectedEdges -> True, VertexSize -> Medium, ImageSize -> Tiny] &,
    ParallelMap[Transpose,
      Tuples[Map[(Append[v, Table[0, v]]) &], Map[(IntegerDigits[v, 2, v]) &], Subsets[Table[2^i, {i, 0, v - 1}], {{v - 1}}] &]]]
  IsomorphicGraphQ]

(chains1, chains2, chains3, chains4, chains5, chains6) = 
  {MakeChains[1], MakeChains[2], MakeChains[3], MakeChains[4], MakeChains[5], MakeChains[6];
chList = Flatten[v, 1] & (chains1, chains2, chains3, chains4, chains5, chains6);

(isoChains1, isoChains2, isoChains3, isoChains4, isoChains5, isoChains6) = 
  #[(All, 1)] & /@ (chains1, chains2, chains3, chains4, chains5, chains6);

isolist = {isoChains1, isoChains2, isoChains3, isoChains4, isoChains5, isoChains6};

wklList = Map[Select[#, WeaklyConnectedGraphQ] &, isolist];

idDOPg[n_] := ParallelMap[Flatten, MapThread[List, {Range[Length[wklList[[n]]], wklList[[n]]]}]]; idDOPgMat[n_] :=
  ParallelMap[Flatten, 
    MapThread[List, {Range[Length[wklList[[n]]]], 
      wklList[[n]], ParallelMap[MatrixForm, ParallelMap[AdjacencyMatrix, wklList[[n]]]]}]])]
```

Figure B.1: The Mathematica source code to produce the domain operator graphs presented in the appendix, or with higher size.

\[ \{\{1, \quad \}, \{2, \quad \}\} \]

Figure B.2: \(|\mathcal{D}_v| = 2\). All possible shapes of domain operator graphs with 2 operators. The number on the left of each graph is an identifier for referencing.
Figure B.3: \(|D_{w}| = 3\). All possible shapes of domain operator graphs with 3 operators. The number on the left of each graph is an identifier for referencing.

Figure B.4: \(|D_{w}| = 4\). All possible shapes of domain operator graphs with 4 operators. The number on the left of each graph is an identifier for referencing.
Figure B.5: $|\mathcal{D}_v| = 5$. All possible shapes of domain operator graphs with 5 operators. The number on the left of each graph is an identifier for referencing.
I present four structures in this appendix: 2 SAS-PUC\textsubscript{2} and 2 SAS-PUC\textsubscript{2}\textsuperscript{*} ones. Compared to the Gray Code or the Horse Breeder structures used to illustrate my thesis, the structures I present in this appendix show more simply how a domain operator graph cycle can be expanded by a planner to produce a minimal solution plan. Concerning the minimal solution plans presented here: all of them are $\Upsilon_1$ except 2: one is $\Upsilon_2$ (C.3c) and the other one is $\Upsilon_3$ (C.4c). Concerning the last two exceptions, I admitted the $\Upsilon_1$-Truth Criterion (cf. Theorem 4.3.1) to be valid for $\Upsilon_2$ and $\Upsilon_3$ and have applied the same planning strategy to suggest the two solution plans (C.3c and C.4c). It can be seen by hand, however, that these two solution plans are minimal and valid. In Figures C.3 and C.1 are dashed nodes and edges. I added these dashed parts symbolically to illustrate that C\textsubscript{2} and C\textsubscript{2}\textsuperscript{*} are a local restriction on SAS-PU structures, and therefore the presented structures can be “bigger” or “extended”. The dashed nodes denoted $\delta_v(+/-,s_\ast)$ represent external chains of actions (cf. section 4.3) in Figures that graphically represent non-linear plans, such as C.3c or C.1c. In Figures representing operator graphs, these dashed nodes symbolically represent possible trees of domain operator graphs (cf. subsection 4.2.4). Figures C.1 and C.2 present two SAS-PUC\textsubscript{2}\textsuperscript{*} structures with four different problems. These two examples visually show the idea behind the C\textsubscript{2} restriction: the ordering of a green (resp. blue) action after a yellow one is not dictated by a blue (resp. green) action. That is, the planner can order a green (resp. blue) action right after its yellow establisher and/or promote it right before its yellow clobberer. If a blue operator is related to a green one, however, then the planner must respect the order in some problems, which can lead to cases whose minimal solution plan is not $\Upsilon_1$. Figures C.3 and C.4 present such structures having relations between blue and green operators without passing through the yellow parts. In such cases, the planner must clobber the prevail-condition of some blue (resp. green) actions to establish a green (resp. blue) one that must be ordered before, due to the relation in the operator graph. The planner must then establish previously clobbered prevail-conditions to complete the solution plan and so on, until the goal state is reached with a correct and minimal plan. It results in minimal solution plans that are not $\Upsilon_1$, such as C.3c and C.4c.
Figure C.1: A SAS-PUC$^*_2$ structure: $o_2$ is not related to $o_1$ when the yellow part of the declobbered operator graph is removed. Let $v_i \in \mathcal{M}$, (C.1c) to (C.1f) are $\Upsilon_1$ minimal solution plans of problems s.t. $s_0 = \langle s_0[v_i], \text{pre}(o_1), \text{pre}(o_2) \rangle$ and $s_* = \langle s_*[v_i], \text{post}(o_1), \text{post}(o_2) \rangle$. The problems differ with $s_0[v_i]$ and $s_*[v_i]$ defined in the sub-caption. Finally, we have: \begin{align*}
pre(o_1^+) = \text{post}(o_{1i}) = - ; \quad \text{pre}(o_1^-) = \text{post}(o_{1i}^+) = + ; \quad \text{prv}(o_1)[v_i] = + \quad \text{and} \quad \text{prv}(o_2)[v_i] = -.
\end{align*}
Figure C.2: Let \( v_i, v_j \in \mathcal{M} \). (C.2c) to (C.2f) represent graphically non-linear, \( \Upsilon_1 \) and minimal action plans solving problems such that, for \( j \in [1, 6] \), \( s_0[v_j] = \text{pre}(o_j) \) and \( s_\star[v_j] = \text{post}(o_j) \). Instances of \( o_j \) are denoted \( a_j \). The problems only differ via \( s_0[v_i] \) and \( s_\star[v_i] \), and their respective definition is given in each sub-caption. We have \( \text{pre}(o_{v_i}^+) = \text{post}(o_{v_i}^-) = - \), and \( \text{pre}(o_{v_i}^-) = \text{post}(o_{v_i}^+) = + \), and \( a_+ \) and \( a_- \) are resp. the instance of \( o_{v_i}^+ \) and \( o_{v_i}^- \). Finally, the blue operators/actions have +, and the green ones have −, as a prevail-condition on \( v_i \). (C.2c) and (C.2e) have two minimal, \( \Upsilon_1 \) and non-linear solution plans. It should be noted that, as for the Horse Breeder structure, green (resp. blue) operators can be related between each others.
Figure C.3: A SAS-PUC$_2$ structure: $o_1$ is related to $o_2$ in the declobbered operator graph when the yellow part is removed, via the dashed green directed edge. The notation is equivalent to the previous Figures. The minimal and non-linear solution plan of Figure C.3c is $Y_2$. In this problem, $a_2$ must be demoted after $a^+$ because $a^+$ establishes $a_1$ which itself establishes $a_2$. Then, $v_i = -$ is not the goal state, so $a_2$ is promoted before $a^+$ which satisfies the pre-condition of $\delta_{v_1}(+, s_*[v_i])$. Finally, $\delta_{v_1}(+, s_*[v_i])$ satisfies $s_*[v_i]$.
APPENDIX C. RESTRICTIONS $C_2$, $C_2^g$ AND $C_2^w$

(a) Operator Graph.

(b) Declobbered Operator Graph.

(c) The minimal non-linear solution plan solving the problem: $\Pi = (s_0 = (\text{pre}(a_1), \text{pre}(a_6^+)), s_1 = (\text{post}(a_{-1}), \text{post}(a_{-1}^-)))$

Figure C.4: A SAS-PUC structure: each blue operator is related to one or two green operators via the blue directed edges, even when the yellow nodes are removed from the declobbered operator graphs. I just provide one problem to illustrate how a cycle of a domain operator graph can be expanded in such a structure. The solution plan of Figure C.4c is $\Upsilon_3$: $a_1^+$ and $a_{-1}^-$ have both been instantiated three times to provide a minimal and non-linear solution plan to the problem.
C.1 Proof of SAS-PUC$^S_2$

Theorem C.1.1. SAS-PUC$^S_2$ problems are $\Upsilon_1$.

Proof. Any SAS problem is at least $\Upsilon_1$ (Lemma 4.1.1). I prove that any SAS-PUC$^S_2$ problem is at most $\Upsilon_1$ by recursion.

Let $m, k \in \mathbb{N}$ such that $m = |\mathcal{M}|$ and $k \leq m$. The proposition at rank $k$, denoted $P(k)$, refers to: “any SAS-PUC$^S_2$ problem instance with a structure that has $k$ distinct domain operator graph cycles restricted by $C^S_2$ has a minimal solution plan that is $\Upsilon_1$.”

Base case: $P(1)$, $v_1 \in \mathcal{M}$, let denote $\text{Cycle}(v_1) = \{o_{v_1}^-, o_{v_1}^+\}$ the unique domain operator graph cycle affected by the restriction $C^S_2$. Let suppose $o_{v_1}^+$ is the unique requestable operator of $\text{Cycle}(v_1)$. Let $\Pi = (\Phi, s_0, s_*)$ a problem instance.

If $s_0[v_1] \in \mathcal{D}_{v_i} \setminus \{-, +\}$, then every operator $o \in \mathcal{O}$ such that $(\text{prv}(o)[v_1] = +)$ cannot have an action instance in the solution plan. A meaningful minimal chain to solve $(\text{prv}(o)[v_1] = +)$ is $\delta_{v_1}(s_0[v_1], +)$ and $\delta_{v_1}(s_0[v_1], +) = \emptyset$ because $(v_1 = +)$ cannot be reached from $s_0[v_1] \in \mathcal{D}_{v_i} \setminus \{-, +\}$. Such problem instances are equivalent to solving an instance of a SAS-PUC$^S_0$ problem. So the solution plan, if it exists, is necessarily $\Upsilon_1$.

In the following, I consider $s_0[v_1] \in \{-, +\}$ and $s_*[v_1] \in \{-, +, x\}$ with $x \in \mathcal{D}_{v_i} \setminus \{-, +\}$, which correspond to every remaining combination of the pair $(s_0[v_1], s_*[v_1])$:

- $(s_0[v_1], s_*[v_1]) = (+, +)$, there is no meaningful minimal chain between $+$ and $+$. Therefore, $\text{chain}_{v_1}(+, +) = \emptyset$

- $(s_0[v_1], s_*[v_1]) = (+, -)$, the only meaningful minimal chain is $\delta_{v_1}(s_0[v_1], s_*[v_1])$, therefore $\text{chain}_{v_1}(+, -) = \delta_{v_1}(+,-)$.

- $(s_0[v_1], s_*[v_1]) = (+, x)$, the first meaningful minimal chain is either $\delta_{v_1}(+, p_1)$ or $\delta_{v_1}(+, x)$. If $\delta_{v_1}(+, p_1)$, it is followed by $\delta_{v_1}(p_1, p_2)$ or $\delta_{v_1}(p_2, x)$, etc. It results: $\text{chain}_{v_1}(+, x) = \delta_{v_1}(+, x)$.

- $(s_0[v_1], s_*[v_1]) = (-, +)$, the first and only meaningful chain is: $\delta_{v_1}(-, +)$. Therefore, $\text{chain}_{v_1}(-, +) = \delta_{v_1}(-, +)$.

- $(s_0[v_1], s_*[v_1]) = (-, -)$, $\delta_{v_1}(-, -) = \delta_{v_1}(-, +) \oplus \delta_{v_1}(+, -)$ is meaningful because $\delta_{v_1}(-, +)$ and $\delta_{v_1}(+, -)$ are both meaningful. $\text{chain}_{v_1}(-, -) = \emptyset$, or $\delta_{v_1}(-, -)$

- $(s_0[v_1], s_*[v_1]) = (-, x)$, a first meaningful minimal chain is $\delta_{v_1}(-, +)$ followed by $\delta_{v_1}(+, x)$. It results: $\text{chain}_{v_1}(-, x) = \delta_{v_1}(-, x)$

No matter $s_0[v_1], s_*[v_1] \in \mathcal{D}_{v_i}$, the minimal chain of actions from $s_0[v_1]$ to $s_*[v_1]$ is at most $\Upsilon_1$. It results that the minimal chain of actions that could have created a minimal solution plan more than $\Upsilon_1$ is always $\Upsilon_1$. Eventually, SAS-PUC$^S_2$ problems with one cycle affected by $C^S_2$ are $\Upsilon_1$.

Induction Hypothesis: Let $k \in \mathbb{N}$ such that $k \leq |\mathcal{M}|$, $P(k)$ is true.
APPENDIX C. RESTRICTIONS C₂, C₂⁺ AND C₂*  

Induction Step: Does any SAS-PUC²⁺ problem instance having a structure with \((k + 1) \leq |\mathcal{M}|\) cycles affected by the restriction \(C₂⁺\) has a minimal solution plan that is \(\Upsilon_1\)? Let \(v_{k+1} \in \mathcal{M}\) such that \(\text{Cycle}(v_{k+1})\) is affected by \(C₂⁺\) and \(o^+_{v_{k+1}}\) is the requestable operator of \(\text{Cycle}(v_{k+1})\). If \(s_0[v_{k+1}] \in \mathcal{D}_{v_{k+1}} \setminus \{-, +\}\), then every operator \(o \in \mathcal{O}\) such that \((\text{prv}(o))[v_{k+1}] = +\) cannot have an action instance in the solution plan. A meaningful minimal chain to solve \((\text{prv}(o))[v_{k+1}] = +\) is \(\delta_{\text{prv}}(s_0[v_{k+1}], +)\) and \(\delta_{\text{prv}}(s_0[v_{k+1}], +) = ()\) because \((v_{k+1} = +)\) cannot be reached from \(s_0[v_{k+1}] \in \mathcal{D}_{v_{k+1}} \setminus \{-, +\}\). Such problem instances are equivalent to solving an instance of a SAS-PUC²⁺ problem with \(k\) cycles affected by the restriction \(C₂⁺\). So the solution plan, if it exists, is necessarily \(\Upsilon_1\) according to the induction hypothesis. If we consider \(s_0[v_1] \in \{-, +\}\) and \(s_\ast[v_1] \in \{-, +, x\}\), the different forms of the chain of actions \(\text{chain}_{v_{k+1}}\) follow the same explanation as in \(P(1)\). It results that the only chain of actions that could have been at least \(\Upsilon_2\) is not minimal if not \(\Upsilon_1\). Eventually, \(P(k + 1)\) is proved and it concludes the proof. \(\square\)

C.2 Proof of SAS-PUC²⁺

Theorem C.2.1. SAS-PUC²⁺ problems are \(\Upsilon_1\).

Proof. Every SAS problem is at least \(\Upsilon_1\) (Lemma 4.1.1). I prove that SAS-PUC²⁺ problems are at most \(\Upsilon_1\) by recursion.

Base case: Let consider any SAS-PUC²⁺ problem instance \(\Pi_{(1)} = (\Phi_{(1)}, s_0, s_\ast)\) with \(\Phi_{(1)}\) a SAS-structure that has only one cycle restricted by \(C₂⁺\). \(v_1 \in \mathcal{M}\), let denote \(\text{Cycle}(v_1)\) that single cycle restricted by \(C₂⁺\), we have \(\mathcal{RO}(v_1) \cap \text{Cycle}(v_1) = \text{Cycle}(v_1) = \{o^-_{v_1}, o^+_{v_1}\}\).

Due to \(C₂⁺\), we therefore have:

- Let \(\mathcal{O}(v_1)' = (\mathcal{O}(v_1) \setminus \text{Cycle}(v_1))\), \((\mathcal{O}(v_1)', \mathcal{E}_{\text{prv}}(\mathcal{O}(v_1)'))\) is a domain operator graph with two distincts, possibly empty, directed trees (cf. subsection 4.2.4)

- \(\forall o_1 \in \mathcal{O} \setminus \mathcal{O}(v_1)\) s.t. \(o_1^+ \prec_{\text{prv}} o_1\), \(o_1\) is part of \(\mathcal{G}_1 = (\mathcal{O}_1, \mathcal{E}^+_{\mathcal{O}_1}) \subset \mathcal{G}_{\mathcal{O}\setminus\text{Cycle}(v_1)}\) which is a declobbered operator graph of a SAS-PUC₀ or SAS-PUC²⁺ structure.

- \(\forall o_2 \in \mathcal{O} \setminus \mathcal{O}(v_1)\) s.t. \(o_2^- \prec_{\text{prv}} o_2\), \(o_2\) is part of \(\mathcal{G}_2 = (\mathcal{O}_2, \mathcal{E}^+_{\mathcal{O}_2}) \subset \mathcal{G}_{\mathcal{O}\setminus\text{Cycle}(v_1)}\) which is a declobbered operator graph of a SAS-PUC₀ or SAS-PUC²⁺ structure.

Indeed, due to \(C₂⁺\), \(o_1\) is not related to \(o_2\) in \(\mathcal{G}_{\mathcal{O}\setminus\text{Cycle}(v_1)}\). Which means both operators are in two distinct subgraphs of \(\mathcal{G}_{\mathcal{O}\setminus\text{Cycle}(v_1)}\), namely \(\mathcal{G}_1\) and \(\mathcal{G}_2\). Only the cycle \(\text{Cycle}(v_1)\) has two requestable operators in it, which implies that, \(\forall v_i \in \mathcal{M} \setminus \{v_1\}\), \(\text{Cycle}(v_i)\) is either empty (SAS-PUC₀), or has no requestable operator (SAS-PUC₀), or only have one requestable operator in it (SAS-PUC²⁺).

Therefore, for all solvable problem instances \(\Pi_{(1)}\) such that \(s_0[v_1] \notin \{-, +\}\), then \(\Pi_{(1)}\) is a SAS-PUC₀ or a SAS-PUC²⁺ problem instance which implies that the minimal solution plan is \(\Upsilon_1\) (Theorems 4.4.3 and C.1.1). I will now consider \(s_0[v_1] \in \{-, +\}\) and \(s_\ast[v_1] \in \{\delta^-, \delta^+\}\) with \(\delta^- = \text{post}(\delta_{v_1}(-, \delta^-))\) and \(\delta^+ = \text{post}(\delta_{v_1}(+, \delta^-))\) such that \(a^+_{v_1}, a^-_{v_1} \notin \).
\( \delta_{v_i}(-, \delta^-) \) and \( a_{v_i}^+, a_{v_i}^- \notin \delta_{v_i}(+, \delta^+) \), i.e. no action of these two minimal chains are instantiated from an operator of \( \text{Cycle}(v_i) \) (cf. Appendices C.1a and C.3a to illustrate \( \delta^- \) and \( \delta^+ \)). These minimal chains are respectively empty if \( \delta^- = - \) and \( \delta^+ = + \) respectively. These starts and goals correspond to every remaining combination of the pair \((s_0[v_i], s_\ast[v_i])\). I denote \( \alpha_1 \) the sequence of actions such that: \( \exists a \in \alpha_1, \text{prv}(a)[v_i] \subseteq + \land \forall a' \in \alpha_1, \text{prv}(a') \neq + \). I denote \( \alpha_2 \) the sequence of actions such that: \( \exists b \in \alpha_2, \text{prv}(b)[v_i] \subseteq - \land \forall b' \in \alpha_2, \text{prv}(b') \neq - \). \( \alpha_1 \) and \( \alpha_2 \) are necessarily \( \Upsilon_1 \) as they are minimal solution plans of SAS-PUC\(_2^S\) or SAS-PUC\(_0\) subproblem instances. Depending on the combination of \((s_0[v_i], s_\ast[v_i])\), \( \Pi_{(1)} \) can only have the following types of plan \( \Delta \):

- \((s_0[v_i], s_\ast[v_i]) = (-, \delta^-)\), the minimal solution plan is:
  \[
  \Delta = \langle a_{v_i}^+ \rangle \oplus \alpha_1 \oplus \langle a_{v_i}^- \rangle \oplus \alpha_2, \quad \text{with:}
  \text{chain}_{v_i}(-, \delta^-) = \langle a_{v_i}^+, a_{v_i}^- \rangle \oplus \delta_{v_i}(-, \delta^-) \land \delta_{v_i}(-, \delta^-) \subseteq \alpha_2.
  \]
  or,
  \[
  \Delta = \alpha_1 \oplus \alpha_2, \quad \text{with:}
  \text{chain}_{v_i}(-, \delta^-) = \delta_{v_i}(-, \delta^-) \subseteq \Delta \land \forall a \in \Delta, \text{prv}(a)[v_i] \neq +.
  \]
- \((s_0[v_i], s_\ast[v_i]) = (-, \delta^+)\), the minimal solution plan is:
  \[
  \Delta = \alpha_2 \oplus \langle a_{v_i}^+ \rangle \oplus \alpha_1, \quad \text{with:}
  \text{chain}_{v_i}(-, \delta^+) = \langle a_{v_i}^+ \rangle \oplus \delta_{v_i}(+, \delta^+) \land \delta_{v_i}(+, \delta^+) \subseteq \alpha_1.
  \]
- \((s_0[v_i], s_\ast[v_i]) = (+, \delta^-)\), the minimal solution plan is:
  \[
  \Delta = \alpha_1 \oplus \langle a_{v_i}^- \rangle \oplus \alpha_2 \quad \text{with:}
  \delta_{v_i}(-, \delta^-) \subseteq \alpha_2 \land \text{chain}_{v_i}(-, \delta^-) = \langle a_{v_i}^- \rangle \oplus \delta_{v_i}(-, \delta^-).
  \]
- \((s_0[v_i], s_\ast[v_i]) = (+, \delta^+)\), the minimal solution plan is:
  \[
  \Delta = \oplus \langle a_{v_i}^- \rangle \oplus \alpha_2 \oplus \langle a_{v_i}^+ \rangle \oplus \alpha_1 \quad \text{with:}
  \text{chain}_{v_i}(+, \delta^+) = \langle a_{v_i}^-, a_{v_i}^+ \rangle \oplus \delta_{v_i}(+, \delta^+) \land \delta_{v_i}(+, \delta^+) \subseteq \alpha_1.
  \]
  or,
  \[
  \Delta = \alpha_2 \oplus \alpha_1, \quad \text{with:}
  \text{chain}_{v_i}(+, \delta^+) = \delta_{v_i}(+, \delta^+) \subseteq \Delta \land \forall a \in \Delta, \text{prv}(a)[v_i] \neq -.
  \]

Every problem instance \( \Pi_{(1)} \) have therefore a minimal solution plan that is \( \Upsilon_1 \), which concludes the base case.

**Induction hypothesis:** Let \( k \in \mathbb{N} \) such that \( k \leq |\mathcal{M}| \). The proposition at rank \( k \) is: “Any \( \Pi_{(k)} = (\Phi_{(k)}, s_0, s_\ast) \), a SAS-PUC\(_2\) problem instance with \( \Phi_{(k)} \) a SAS-structure that has exactly \( k \) cycles restricted by \( C_2^\ast \), has a minimal solution plan that is \( \Upsilon_1 \)”.

Let consider \( \{v_1, ..., v_k\} \subseteq \mathcal{M} \) the state variables such that, for each \( v_i \in \{v_1, ..., v_k\} \), \( \text{Cycle}(v_i) \) is restricted by \( C_2^\ast \), we have \( \mathcal{R}\mathcal{O}(v_i) \cap \text{Cycle}(v_i) = \text{Cycle}(v_i) = \{o_{v_i}^-, o_{v_i}^+\} \).

**Induction step:** Let consider \( v_{k+1} \), the \((k+1)^{th}\) state variable restricted by \( C_2^\ast \). We have: \( \mathcal{R}\mathcal{O}(v_{k+1}) \cap \text{Cycle}(v_{k+1}) = \text{Cycle}(v_{k+1}) = \{o_{v_{k+1}}^-, o_{v_{k+1}}^+\} \). The restriction \( C_2^\ast \) implies that:
• Let $O(v_{k+1})' = (O(v_{k+1}) \setminus \text{Cycle}(v_{k+1})), (O(v_{k+1})', E_{\text{pre}}(O(v_{k+1})'))$ is a domain operator graph with two distincts, possibly empty, directed trees (cf. subsection 4.2.4)

• $\forall o_1 \in O \setminus O(v_{k+1})$ s.t. $o_{v_{k+1}}^+ \prec_{\text{prv}} o_1, o_1$ is part of $G_1 = (O_1, E_{O_1}^+) \subset G_{O \setminus \text{Cycle}(v_{k+1})}$ which is a declobbered operator graph of a SAS-PUC$_0$, SAS-PUC$_S$ structure or it is a declobbered operator graph of the structure of a SAS-PUC$_2^*$ problem with exactly $k$ cycles restricted by $C_2^*$ (Induction hypothesis).

• $\forall o_2 \in O \setminus O(v_1)$ s.t. $o_{v_1}^- \prec_{\text{prv}} o_2, o_2$ is part of $G_2 = (O_2, E_{O_2}^+) \subset G_{O \setminus \text{Cycle}(v_1)}$ which is a declobbered operator graph of a SAS-PUC$_0$, SAS-PUC$_S$ structure or it is a declobbered operator graph of the structure of a SAS-PUC$_2^*$ problem with exactly $k$ cycles restricted by $C_2^*$ (Induction hypothesis).

With an explanation similar to the base case, i.e. via the enumerated possible minimal solution plans with respect to all the combinations of $(s_0[v_{k+1}], s_\star[v_{k+1}])$, I prove that chain$_{v_{k+1}}(s_0[v_{k+1}], s_\star[v_{k+1}])$ is always $\Upsilon_1$. This chain is then concatenated with smaller sequence of actions that are minimal and $\Upsilon_1$ because they solve smaller problem instances that are either from SAS-PUC$_0$ problems (Theorem 4.4.3), or from SAS-PUC$_S$ problems (Theorem C.1.1), or from SAS-PUC$_2^*$ problems with exactly $k$ cycles restricted by $C_2^*$ (Induction hypothesis).

This concludes the proof, SAS-PUC$_2^*$ problems are $\Upsilon_1$. □
This appendix presents the proof of BuildChain and TopoPlan 4. In the following, $O$ represents the set of (PU) actions stored in the hashing table that was built in the setup (cf. subsection 4.5.1). Each action of $O$ has a color:

- (White) the action is not instantiated, i.e. it is not in $A$. Every action of $O$ is white at the beginning of the resolution of any problem instance;
- (Yellow) the action is instantiated, i.e. it is in $A$;
- (Blue) the action is being topologically sorted;
- (Green) the action is topologically sorted, i.e. added in a linear solution plan $\Delta$.

$\Delta = (A, E_A)$ is a solution plan made up of: $A$ a set of (PU) actions, $E_A$ a set of orders between the actions of $A$.

### D.1 Proof of BuildChain

**Definition D.1.1.** The specification of BuildChain is:

- **Input:** $O$, $A$, $E_A$, $v_i \in M$, $s, g \in D_{v_i}$ s.t. $s \neq g$ and $a_{v_i}^g \in O$ is white.
- **Output:** a minimal chain of actions from $s$ to $g$, $\delta_{v_i}(s, g)$. Each action $a$ of $\delta_{v_i}(s, g)$ is colored yellow, added to $A$ and each order $(b, a)$, s.t. $b \in N_{pre}(a)$, is added to $E_A$.

**Theorem D.1.1.** BuildChain satisfies its specification, so it is correct.

Before proving the theorem, I provide some prerequisites and explanations. Let $v_i \in M$, $s, g \in D_{v_i}$, a chain of actions on $v_i$ from $s$ to $g$ is a linear sequence of actions such that:

- $\text{chain}_{v_i}(s, g) = (D, E_D) = \langle a_{|D|}, \ldots, a_2, a_1 \rangle$
The procedure BuildChain adds an additional order when building the chain of actions. Let denote $E_D^+$ this set with the additional order. It is defined as follows:

$$E_D^+ = \{(a_{|D|+1}, a_{|D|}) \mid a_{|D|+1} \in N_{pre}(a_{|D|})\} \cup E_D$$

It should be noticed that $a_{|D|+1}$ is either (1) a ghost action, or (2) it is white (i.e. not in a chain of actions), or (3) it is yellow. (1) and (2) can be used by the procedure DFSTopo as a valid stop condition to break the recursivity (2.3), because, in cases (1) or (2), $a_{|D|}$ is obviously the first action to modify the start on $v_i$. In case (3), then the additional order acts as a concatenation between two minimal chains. Case (3) means $a_{|D|+1}$ belongs to another action chain than chain $v_i(s,g)$. $a_{|D|+1}$ is necessarily the last action of this other chain. Indeed, due to the coloring system it cannot exist two different chains of actions with similar actions in them (cf. Lemma D.1.2). Therefore, $a_{|D|+1}$ is the last action of a chain on $v_i$ and $a_{|D|}$ is the first action of chain $v_i(s,g)$, it concludes the fact the additional order $(a_{|D|+1}, a_{|D|})$ acts as a concatenation between two chains on $v_i$.

It results that the procedure BuildChain could have had $D$ and $E_D^+$ as local parameters so that the instructions:

- (1.4) $A \leftarrow A \cup \{a^g_{v_i}\}$ ; $E_A \leftarrow E_A \cup \{(a^y_{v_i}, a^g_{v_i})\}$
- (1.13) $A \leftarrow A \cup \{a^y_{v_i}\}$ ; $E_A \leftarrow E_A \cup \{(a^y_{v_i}, a^x_{v_i})\}$

could have respectively been replaced by:

- (1.4) $D \leftarrow D \cup \{a^g_{v_i}\}$ ; $E_D^+ \leftarrow E_D^+ \cup \{(a^y_{v_i}, a^g_{v_i})\}$
- (1.13) $D \leftarrow D \cup \{a^y_{v_i}\}$ ; $E_D^+ \leftarrow E_D^+ \cup \{(a^y_{v_i}, a^x_{v_i})\}$

And after “(1.17): end while”, in a new line, the instructions: $A \leftarrow A \cup D$; $E_A \leftarrow E_A \cup E_D^+$, could have been added.

I will use this equivalence to prove that the procedure BuildChain 2 is correct.

**Proof. of Theorem D.1.1**

**Loop Invariant:** Let $k \in \mathbb{N}^*$, $P(k)$ “at the end of the $k^{th}$ loop, chain$_{v_i}(k,0) = \delta_{v_i}(k,0)$, i.e. it is minimal such that:

- $\delta_{v_i}(k,0) = (D(k), E_D^+(k-1)) = (a^k_{v_i}, ..., a^0_{v_i})$ ;
- $E_D^+(k) = \{(a^k_{v_i}, a^{k-1}_{v_i})\} \cup E_D^+(k-1)$ and $a^k_{v_i} \in N_{pre}(a^{k-1}_{v_i})$ ;
• Each action of $D_k = \{a_{v_i}^{k-1}, \ldots, a_{v_i}^0\}$ is colored yellow.

1. **Initialisation**: Let consider $P(1)$, we have:

- $\text{chain}_{v_i}(1, 0) = (D(1) = \{a_{v_i}^0\}, E_{D(0)}^+ = \{\}) = (a_{v_i}^0)$ ;
- $E_{D(1)}^+ = \{(a_{v_i}^0, a_{v_i}^0)\}$ and $a_{v_i}^1 \in \mathcal{N}_{\text{pre}}(a_{v_i}^0)$ (1.4);
- $a_{v_i}^0 \in D(1)$ is colored yellow (1.4).

$\mathcal{N}_{\text{pre}}(a_{v_i}^0)$ is a singleton due to (P) and we have $a \in \mathcal{N}_{\text{pre}}(a_{v_i}^0)$ such that $\text{post}(a) = \text{pre}(a_{v_i}^0) = 1$, hence the identification $a_{v_i}^1$ (cf. **Definition 4.1.1**). $\text{chain}_{v_i}(1, 0)$ is obviously minimal, therefore: $\text{chain}_{v_i}(1, 0) = \delta_{v_i}(1, 0)$, and $P(1)$ holds.

2. **Maintenance**: Let consider $P(k)$ to be true, and, at the end of the $k$th loop, we have $y = \text{pre}(a_{v_i}^{k-1}) = k$. So $y = k$ at the beginning of the $(k + 1)$th loop. At the end of the $(k + 1)$th loop, we have:

- $a_{v_i}^k$ is colored yellow
- $a_{v_i}^k$ is added to $D$. That is, $D_{k+1} = D_k \cup \{a_{v_i}^k\}$
- $x = y = \text{pre}(a_{v_i}^{k-1}) = k$, i.e. $x = k$
- $y = \text{pre}(a_{v_i}^k) = \text{pre}(a_{v_i}^k) = k + 1$
- $E_{D_{k+1}}^+ = E_{D_k}^+ \cup \{(a_{v_i}^k, a_{v_i}^k)\} = E_{D_k}^+ \cup \{(a_{v_i}^{k+1}, a_{v_i}^k)\}$

Therefore, $\text{chain}_{v_i}(k + 1, 0) = (D_{k+1}, E_{D_{k+1}}^+) = (a_{v_i}^k) \oplus \delta_{v_i}(k, 0)$ and it is minimal because $a_{v_i}^k$ exists (otherwise it is a ghost operator and the procedure would have failed (1.9)) and it was not in $\delta_{v_i}(k, 0)$ at the beginning of the $(k + 1)$th loop (otherwise, $a_{v_i}^k$ would have been yellow and the procedure would have failed (1.11)). The reasoning about the minimality of the chain is similar to the **Proof 3.2.1**. Therefore, the loop invariant holds at the end of each iteration of the **while loop**.

3. **Termination**: First of all, $s, g \in D_{v_i}$ and $D_{v_i}$ is a finite set of value, so if it exists a chain of actions from $s$ to $g$, then the size of the chain is finite and the **while loop** terminates with the stop condition: $\text{pre}(\text{chain}_{v_i}(s, g)) = s$.

The **while loop** does not terminate iff the domain operator graph $\mathcal{G}_{\mathcal{O}}(v_i)$ is not weakly connected and $s$ and $g$ are the post-condition of two operators that do not belong to the same graph. I demonstrated in subsection 4.2.2 that in such cases, it cannot exist a chain of actions between $s$ and $g$. **BuildChain** builds in a backward manner the chain of actions, thus starting with $a_{v_i}^0$. So **BuildChain** navigates through the domain operator graph that contains $\text{oper}(a_{v_i}^0)$. **Lemma 4.2.2** implies that, the **while loop** either (1) ends up with a ghost action, or (2) ends up cycling through the unique cycle of the domain operator graph because it cannot reach $s$. the if condition (1.9) handles the case (1) while the if condition (1.11) handles the case (2). A verification of the weakly connected condition can
be done during the Setup. If so, (1.9) and (1.11) are no longer required. In particular, Lemma 3.2.1 says that a minimal chain is \( \Upsilon_1 \), so if the minimal chain exists the while loop cannot fail with (1.11).

Eventually, the finite size of \( D_{v_i} \) and the two if conditions (1.9) and (1.11) ensure that the while loop will always terminate.

4. Correctness: The input specification imposes that \( s \) and \( g \) are in \( D_{v_i} \), so if the domain operator graph \( G_O(v_i) \) is weakly connected, then it exists a chain from \( s \) to \( g \). In particular, it exists a minimal chain from \( s \) to \( g \). Lemma 3.2.1 implies the minimal chain is \( \Upsilon_1 \) so the while loop will always terminate. Eventually, the exit condition and the loop invariant both hold so the procedure BuildChain meets its specification.

Corollary D.1.1. The procedure BuildChain is complete.

Proof. If there exists a chain of actions between \( s \) and \( g \), then it exists a minimal chain of actions between \( s \) and \( g \) and it is unique (cf. Lemma 3.2.2). If BuildChain fails, (1.9) or (1.11), then there is no solution (cf. Termination of the proof D.1.1). So it contradicts the fact that there exists a chain of action. If it does not fail, Theorem D.1.1 implies BuildChain returns a correct solution, i.e. the unique minimal chain of actions between \( s \) and \( g \).

Corollary D.1.2. The procedure buildChain yields a failure if an action instance belongs to at least two different minimal chains.

In other words, this corollary says that BuildChain yields a failure if the solution plan being built is \( \Upsilon_k \) with \( k \geq 2 \).

Proof. Suppose that BuildChain has built the minimal chain \( \delta_1 \). For all \( a \in \delta_1 \), \( a \) is colored yellow. Suppose that BuildChain is building \( \delta_2 \) and, at the \( k^{th} \) loop, \( a \) is the next action to be added to \( \delta_2 \). Because \( a \) is yellow, BuildChain yields a failure with (1.11). Therefore, BuildChain prevents the creation of minimal chains having the same actions.

D.2 Proof of TopoPlan \( \mathbb{P} \)

I will now prove the correctness of TopoPlan \( (\mathbb{P}) \).

Definition D.2.1. The specification of TopoPlan is:

- Input: \( \Pi = (\mathcal{M}, \mathcal{O}, s_0, s^*_s) \), a problem instance of the class SAS-PUC\(_0\), SAS-PUC\(_2^S\) or SAS-PUC\(_2^\ast\).

- Output: If \( \Pi \) is solvable, then \( \mathbb{P} \) returns \( \Delta \) a linear, \( \Upsilon_1 \) and minimal solution plan that solves \( \Pi \) while respecting the \( \Upsilon_1 \)-Truth Criterion. If \( \Pi \) is not solvable, \( \mathbb{P} \) yields a failure.

Theorem D.2.1. The algorithm TopoPlan satisfies its specification, so it is correct.
APPENDIX D. PROOF OF THEOREMS IN CHAPTER 4

Proof. TopoPlan works in three phases. From (3.2) to (3.6) is Phase 1, from (3.9) to (3.31) is Phase 2 and, finally, from (3.32) to (3.36) is Phase 3.

In Phase 1, there is a for loop (3.2). Let \( k \in [1, |\mathcal{M}|] \), the loop invariant of (3.2) is:

\[
P_{(3.2)}(k) : \text{At the end of the } k^{th} \text{ loop,}
\]

\[
\forall i \in [1, k], \text{chain}_{v_i}(s_0[v_i], s_*[v_i]) = \begin{cases} \delta_{v_i}(s_0[v_i], s_*[v_i]) & \text{if } s_0[v_i] \neq s_*[v_i] \\ \langle \rangle & \text{otherwise.} \end{cases}
\]

In this phase, TopoPlan aims at building a minimal chain of actions for each state variable whose start is different from the goal. The building of these chains is made by BuildChain which is correct (cf. Theorem D.1.1). Each minimal chain is unique (cf. Lemma 3.2.2) and independent from one state variable to another. It results that if \( P_{(3.2)}(0) \) is obviously true and, for \( k \in [1, |\mathcal{M}|] \), if \( P_{(3.2)}(k) \) is true, \( P_{(3.2)}(k+1) \) is also true by independence and because BuildChain is correct no matter the state variable. Concerning the termination, \( \mathcal{M} \) is a finite set so, if BuildChain does not fail, the for loop (3.2) always terminates. If BuildChain fails, then it means a chain of actions between the start and the goal does not exist, so the problem instance is not solvable. If the problem instance is solvable, the loop invariant holds and the loop always terminates. So \( P_{(3.2)}(|\mathcal{M}|) \) holds. At the end of Phase 1, we therefore have:

\[
\Delta_1 = (\mathcal{A}_1, E_{A_{1, pre}}) = \bigcup_{v_i \in \mathcal{M}} \begin{cases} \delta_{v_i}(s_0[v_i], s_*[v_i]) & \text{if } s_0[v_i] \neq s_*[v_i] \\ \langle \rangle & \text{otherwise.} \end{cases}
\]

\( \Delta_1 \) is a non-linear action plan with:

\[
\{ \mathcal{A}_1 = \{ a \mid v_i \in \mathcal{M}, a \in \delta_{v_i}(s_0[v_i], s_*[v_i]) \} \\
E_{A_{1, pre}} = \{ (b, a) \mid a \in \mathcal{A}_1, b \in \mathcal{O}, b \in \mathcal{N}_\text{pre}(a) \}
\]

And \( \Delta_1 \) is obviously minimal. Each action of \( \mathcal{A}_1 \) are yellow due to BuildChain.

The specification of Phase 2 is:

- Input: \( \Pi = (\mathcal{M}, \mathcal{O}, s_0, s_*) \) and: \( \begin{cases} \mathcal{A} = \mathcal{A}_1 \\ E_A = E_{A_{1, pre}} \end{cases} \)
- Output: \( \Delta_2 = (\mathcal{A}, E_A) \) a non-linear sequence of actions that is \( \Upsilon_1 \) where each action is correctly ordered according to the \( \Upsilon_1 \)-Truth Criterion.

There are two for loops ((3.9) and (3.10)), so I described two loop invariants. Let \( l \in \mathbb{N} \) s.t. \( 1 \leq l \leq |\mathcal{A}| \) and \( \mathcal{A} = \{ a_1, ..., a_{|\mathcal{A}|} \} \), the notation \( \mathcal{A}[1..l] \) is the set of actions: \( \mathcal{A}[1..l] = \{ a_1, ..., a_l \} \). And so, \( \mathcal{A}[1..|\mathcal{A}|] = \mathcal{A} \).

\( P_{(3.9)}(l) : \) At the end of the \( l^{th} \) loop, \( \forall a_{v_i}^p \in \mathcal{A}[1..l], a_{v_i}^p \) is correctly ordered in \( \Delta_2 = (\mathcal{A}, E_A) \).

Let \( a_{v_i}^p \in \mathcal{A}, r \in \mathbb{N} \) s.t. \( 1 \leq r \leq |\mathcal{N}_\text{pre}(a_{v_i}^p)| \):
\[ P_{(3.10)}(r): \text{ At the end of the } r^{th} \text{ loop, } \forall a^q_{v_j} \in \mathcal{N}_{\text{prev}}(a^p_{v_i})[1..r] \text{ the proposition } q = \text{prev}(a^p_{v_i})[v_j] \text{ has been established and declobbered with respect to the } \Upsilon_1-\text{Truth Criterion.} \]

I starts by proving that the for loop (3.10) is correct. That is, whenever the loop invariant \( P_{(3.10)} \) holds and the loop (3.10) terminates, then \( P_{(3.10)}(\mathcal{N}_{\text{prev}}(a^p_{v_i})) \) holds. And if \( P_{(3.10)}(\mathcal{N}_{\text{prev}}(a^p_{v_i})) \) holds, then it means \( a^p_{v_i} \) has been ordered with respect to the \( \Upsilon_1-\text{Truth Criterion.} \)

\[ \mathcal{N}_{\text{prev}}(a^p_{v_i}) = \{ a^q_{v_j} \mid v_j \neq v_i \land q = \text{prev}(a^p_{v_i})[v_j] \} \text{ is the set of actions that establish the defined prevail-conditions of } a^p_{v_i}. \text{ Due to (P), there is exactly one action per defined prevail-condition of } a^p_{v_i}, \text{ and therefore there is no two actions in } \mathcal{N}_{\text{prev}}(a^p_{v_i}) \text{ that affect the same state variable. Therefore, at each loop, the establishment of a prevail-condition of } a^p_{v_i} \text{ is independent of the others. In other words, if } P_{(3.10)}(r) \text{ holds for } r \in \llbracket 1, \mathcal{N}_{\text{prev}}(a^p_{v_i}) \rrbracket, \text{ then } P_{(3.10)}(r) \text{ holds for every } r \in \llbracket 1, \mathcal{N}_{\text{prev}}(a^p_{v_i}) \rrbracket \text{ by independence of each loop.} \]

Let \( a^p_{v_i} \) be the action to assert and \( a^q_{v_j} \in \mathcal{N}_{\text{prev}}(a^p_{v_i}) \) one of its establisher (by definition of \( \mathcal{N}_{\text{prev}} \)), that is: \( \text{prev}(a^p_{v_i})[v_j] = \text{post}(a^q_{v_j}) = q \). Let consider the cases, \( s_0[v_i] \neq q \) and \( s_1[v_i] \neq q \), then \( \mathcal{P} \) respects the equation 4.3 of the \( \Upsilon_1-\text{Truth Criterion:} \)

\[ \exists T, C \in \text{chain}_{v_i}(s_0[v_i], s_1[v_i]) \text{ s.t. } T \prec C, \ (T \prec a \land a \prec C) \]

with \( T = a^q_{v_j} \) and \( C = \text{Next}(a^q_{v_j}) \) via the instructions between (3.11) and (3.18) for \( s_0[v_i] \neq q \), and between (3.19) and (3.26) for \( s_1[v_i] \neq q \). If \( a^q_{v_j} \) is yellow, then it is already inside \( \mathcal{A} \) so \( \mathcal{P} \) only needs to order \( a^q_{v_j} \) before \( a^p_{v_i} \) (3.17). If \( a^q_{v_j} \) is not yellow, then it is white and therefore not in \( \mathcal{A} \). In other words, \( a^q_{v_j} \in \mathcal{O} \setminus \mathcal{A} \) so \( \mathcal{P} \) builds a chain of actions from \( s_0[v_j] \) to \( q \), with the procedure \( \text{BuildChain} \) (3.15) proven correct, in order to establish the prevail-condition of \( a^p_{v_i} \). The actions of this chain are thus colored yellow and added to \( \mathcal{A} \), \( a^q_{v_j} \) included and it can now be ordered before \( a^p_{v_i} \) (3.17) to respect the \( \Upsilon_1-\text{Truth Criterion.} \)

If \( \text{Next}(a^q_{v_j}) = \emptyset \), then the successor of \( a^q_{v_j} \) has not yet been defined, which means that it is white and not in \( \mathcal{A} \). Indeed, any action of \( \mathcal{A} \) are added by \( \text{BuildChain} \) which, for any action \( a \) of the chain, assigns \( a \) as successor to \( b \in \mathcal{N}_{\text{prev}}(a) \). Therefore, an action chain from \( q \) to \( s_0[v_j] \) is missing, hence \( \text{BuildChain} \) (3.21)\(^1\). At the end of the procedure \( \text{BuildChain} \), \( \text{Next}(a^q_{v_j}) \) is defined and according to the \( \Upsilon_1-\text{Truth Criterion it is the clobberer, therefore } a^p_{v_i} \text{ must be promoted before } \text{Next}(a^q_{v_j}) \) (3.24). Due to the restriction (S4), no action can both clobber and establish a proposition, hence the if condition (3.23).

Then, if \( q = s_1[v_i] \), then the proposition \( q \) has no clobberer, so \( \mathcal{P} \) does not need to promote \( a^p_{v_i} \).

If \( q = s_0[v_i] \), then \( q \) has two possible establishers: the start \( s_0[v_i] \) or the action whose post-condition is \( q = s_0[v_i] \) (cf. Corollary 4.3.1). This situation cannot occur with SAS-PUC\(_0\) or SAS-PUC\(_2\)\(^\star\) problem instances (cf. Proofs 4.4.3, 4.4.5), if \( q = s_0[v_j] \), then the start is the only establisher. This situation can in fact only occur for problem instances whose structure has domain operator cycles with at least two requestable operators, i.e. it can only occur with SAS-PUC\(_2\) problem instances. Hence the if condition (3.27). In

\(^{\dagger}\)It should be noticed that these two \( \text{BuildChain} \), (3.17) and (3.21), build the minimal chain: \( \sigma_2 = \delta_{v_i}(s_0[v_j], s_0[v_j]) = \delta_{v_i}(s_0[v_j], q) \oplus \delta_{v_i}(q, s_0[v_j]). \)
that situation, the $\Upsilon_1$-Truth Criterion allows two possibilities: an establishment order ($a^p_{v_j} < a^p_{v_i}$) or an promotion order ($a^p_{v_i} < \text{Next}(a^p_{v_j})$). I decided to use exclusively the establishment order (3.28) and it was possible only because of the structural properties induced by $C^*_2$. Let $q = +$ and $-$ the other requestable value of $\text{Cycle}(v_j), C^*_2$ insures that actions having $q = +$ as a prevail-condition (like $a^p_{v_i}$) are in a different sequence of actions that the actions having $-$ as a prevail-condition (cf. \textbf{Proof 4.4.6}). The two orders are therefore both correct.

In addition to that, if an establisher does not exist (3.12) or if $\text{BuildChain}$, (3.15) or (3.21), fails, then the problem instance is not solvable.

Eventually, each prevail-condition $q$ of $a^p_{v_i}$ on $v_j$ is properly asserted. It means $P(3.10)(r)$ holds for at least one $r \in [1, |\mathcal{N}_{\text{prev}}(a^p_{v_i})|]$, then $P(3.10)(r)$ holds for every $r \in [1, |\mathcal{N}_{\text{prev}}(a^p_{v_i})|]$ by independence of each loop. Then $|\mathcal{N}_{\text{prev}}(a^p_{v_i})| < |\mathcal{M}|$, so the loop (3.10) always terminates. Then, it may exist actions with no defined prevail conditions, however (cf. $\text{DropHaystack}$ of the $\text{HorseBreeder}$ problem for example). In that case, $a^p_{v_i}$ has no establisher via post-prv dependency and it is therefore obviously asserted with respect to the $\Upsilon_1$-Truth Criterion. Therefore the for loop (3.10) is correct and any action of $\mathcal{A}$ is correctly established and declobbered by (3.10) in function of the $\Upsilon_1$-Truth Criterion.

Due to $\Upsilon_1$ and the post-uniqueness, the ordering of an action $a^p_{v_i} \in \mathcal{A}$ in $\Delta_2$ is independent of other actions of $\mathcal{A}$. And because (3.10) is correct, i.e. it correctly establishes and declobbers every defined prevail-condition of $a^p_{v_i} \in \mathcal{A}$, it results that every action of $\mathcal{A}$ is correctly ordered in $\Delta_2$. Therefore, $P(3.9)(l)$ holds for every $l \in [1, |\mathcal{A}|]$. Then, $|\mathcal{A}| \leq |\mathcal{O}|$ due to $\Upsilon_1$ and the if condition (1.11) of $\text{BuildChain}$ which ensures a chain of actions cannot be $\Upsilon_2$ of higher (cf. \textbf{Corollary D.1.2}). Therefore, the for loop (3.9) always terminates if (3.10) does not fail. And if (3.10) fails, it means the problem instance was not solvable. Eventually, $P(3.9)(|\mathcal{A}|)$ holds, therefore the for loop (3.9) is correct with respect to the $\Upsilon_1$-Truth Criterion: each action $a \in \mathcal{A}$ is properly ordered in $\Delta_2$.

\textbf{Phase 3}: Although $\Delta_2$ respects the $\Upsilon_1$-Truth Criterion, it is possible that the problem instance is not solvable due to actions looping between each other in $\Delta_2$. That is, if $\Delta_2$ is not irreflexive, then the problem instance is not solvable. This verification is done during Phase 3 with the topological sort. If Phase 3 succeeds, then $\Delta_2$ is irreflexive. A topological sort is a well-known algorithm, and it has been proven correct. Therefore, Phase 3 is correct and if it does not fail, then the result is: $\Delta$ a $\Upsilon_1$, linear, minimal solution plan that solve the problem instance.

Eventually, $\text{TopoPlan}$ satisfies its specification and it is therefore correct and complete.

$\square$
In this Appendix, I present step by step how \( \mathcal{P} \) works to solve two different problem instances of the Horse Breeder (\( \Phi_{HB} \), cf. Table 4.2).

- \( \Pi_1 = (\Phi_{HB}, s_0 = \langle 0, 0, 0 \rangle, s_* = \langle 2, 0, 2 \rangle) \)
- \( \Pi_2 = (\Phi_{HB}, s_0 = \langle 1, 0, 0 \rangle, s_* = \langle 2, 0, 2 \rangle) \)

In both instances, the horse breeder has to fill both the trough and the horse feeder. The difference between \( \Pi_1 \) and \( \Pi_2 \) lies at the start: in \( \Pi_1 \) the horse breeder carries nothing while in \( \Pi_2 \) he already has a haystack in hands.

Resolution of \( \Pi_1 = (\Phi_{HB}, s_0 = \langle 0, 0, 0 \rangle, s_* = \langle 2, 0, 2 \rangle) \) by \( \mathcal{P} \):

**Phase 1:**

- \( s_0[v_0] \neq s_*[v_0] \), \textit{BuildChain} builds backwards (3.4):
  - \textit{FillHorseFeeder} satisfies \( s_*[v_0] \), it is colored yellow and added to \( \mathcal{A} \).
    \( \text{(TakeHaystack, FillHorseFeeder)} \) is added to \( E_A \).
    \( \text{Next}(\text{TakeHaystack}) = \text{FillHorseFeeder} \) is defined.
    \( \text{pre}(\text{FillHorseFeeder}) \neq s_0[v_0] \).
  - \textit{TakeHaystack} satisfies \( \text{pre}(\text{FillHorseFeeder}) \), it is colored yellow and added to \( \mathcal{A} \).
    \( \text{(DropHaystack, TakeHaystack)} \) is added to \( E_A \).
    \( \text{Next}(\text{DropHaystack}) = \text{TakeHaystack} \) is defined.
    \( \text{pre}(\text{TakeHaystack}) = s_0[v_0] \).
  - \textit{BuildChain} ends, \( \delta_{v_i}(s_0[v_0], s_*[v_0]) = \langle \text{TakeHaystack, FillHorseFeeder} \rangle \) has been built.

- \( s_0[v_1] = s_*[v_1] \), \( \delta_{v_i}(s_0[v_0], s_*[v_0]) = \langle \rangle \).
- \( s_0[v_2] \neq s_*[v_2] \), \textit{BuildChain} builds backwards (3.4):
APPENDIX E. STEP BY STEP EXAMPLES OF \( \mathcal{P} \)

- **FillHorseTrough** satisfies \( s_*(v_2] \), it is colored yellow and added to \( \mathcal{A} \).
  (FillBucketWithWater, FillHorseTrough) is added to \( E_A \).
  \( Next(FillBucketWithWater) = FillHorseTrough \) is defined.
  \( pre(FillHorseFeeder) \neq s_0[v_2] \).

- **FillBucketWithWater** satisfies \( pre(FillHorseTrough) \), it is colored yellow and added to \( \mathcal{A} \).
  (GhostV2, FillBucketWithWater) is added to \( E_A \).
  \( Next(GhostV2) = FillBucketWithWater \) is defined.
  \( pre(FillHorseFeeder) = s_0[v_2] \).

- **BuildChain** ends, \( \delta_{v_1}(s_0[v_2], s_*(v_2]) = \langle FillBucketWithWater, FillHorseTrough \rangle \) has been built.

\( \mathcal{A} = \{ FillHorseFeeder, TakeHaystack, FillHorseTrough, FillBucketWithWater \} \)

**Phase 2:** For all actions of \( \mathcal{A} \) (3.9), check all their prevail-conditions via the set \( \mathcal{N}_{prev} \) (3.10).

- \( \mathcal{N}_{prev}(FillHorseFeeder) = \emptyset \)

- \( \mathcal{N}_{prev}(TakeHaystack) = \{ DropBucket \} \).
  - \( DropBucket \) affects \( v_1 \). \( \text{prv}(TakeHaystack)[v_1] = 0 = s_0[v_1] = s_*(v_1] \), so (3.11) and (3.19) are false. \( DropBucket \in \text{Cycle}(v_1] \) that is concerned by \( C_2^* \), so (3.27) is true. \( \langle DropBucket, TakeHaystack \rangle \) is added to \( E_A \) (3.28) (post-prevail dependency, possible promotion order). \( DropBucket \) remains white.

- \( \mathcal{N}_{prev}(FillHorseTrough) = \{ PickUpBucket \} \).
  - \( PickUpBucket \) affects \( v_1 \). \( \text{prv}(FillHorseTrough)[v_1] = 1 \neq s_0[v_1] \), so (3.11) is true. \( PickUpBucket \) is white (3.14), so \( BuildChain \) builds backwards (3.15):
    * \( PickUpBucket \) establishes \( \text{prv}(FillHorseTrough) \), it is colored yellow and added to \( \mathcal{A} \).
      (DropBucket, PickUpBucket) is added to \( E_A \).
      \( Next(DropBucket) = PickUpBucket \) is defined.
      \( pre(PickUpBucket) = s_0[v_1] \).
    * \( BuildChain \) ends, \( \delta_{v_1}(s_0[v_1], \text{prv}(FillHorseTrough)[v_1]) = \langle PickUpBucket \rangle \) has been built.

\( \langle PickUpBucket, FillHorseTrough \rangle \) (3.17, establishment order) is added to \( E_A \).
\( \text{prv}(FillHorseTrough)[v_1] = 1 \neq s_*(v_1] \), so (3.19) is true. \( Next(PickUpBucket) = \emptyset \) (3.20), so \( BuildChain \) builds backwards (3.21):
  * \( DropBucket \) satisfies \( s_0[v_1] \), it is colored yellow and added to \( \mathcal{A} \).
    (PickUpBucket, DropBucket) is added to \( E_A \).
    \( Next(PickUpBucket) = DropBucket \) is defined.
    \( pre(PickUpBucket) = \text{prv}(FillHorseTrough)[v_1] \).
APPENDIX E. STEP BY STEP EXAMPLES OF \( \mathbb{P} \)

* BuildChain ends, \( \delta_{v_i}(\text{prv}(\text{FillHorseTrough})[v_1], s_0[v_1]) = \langle \text{DropBucket} \rangle \) has been built.

Next(PickUpBucket) = DropBucket is now defined so, (FillHorseTrough, DropBucket) is added to \( E_A \) (3.24 promotion order).

- \( N_{\text{prv}}(\text{FillBucketWithWater}) = \{ \text{PickUpBucket} \} \).
  - PickUpBucket affects \( v_1 \). \( \text{prv}(\text{FillBucketWithWater})[v_1] = 1 \neq s_0[v_1] \), so (3.11) is true. PickUpBucket is yellow (3.14), so (PickUpBucket, FillBucketWithWater) is added to \( E_A \) (3.17 establishment order).
    - \( \text{prv}(\text{FillBucketWithWater})[v_1] = 1 \neq s_0[v_1] \), so (3.19) is false. Next(DropBucket) = TakeHaystack, so (PickUpBucket, TakeHaystack) is added to \( E_A \) (3.24 promotion order).

- \( N_{\text{prv}}(\text{PickUpBucket}) = \{ \text{DropHaystack} \} \).
  - DropHaystack affects \( v_0 \). \( \text{prv}(\text{PickUpBucket})[v_0] = 0 = s_0[v_0] \), so (3.11) is false. \( \text{prv}(\text{PickUpBucket})[v_0] \neq s_0[v_0] = 2 \), so (3.19) is true. Next(DropHaystack) = TakeHaystack, so (PickUpBucket, TakeHaystack) is added to \( E_A \) (3.24 promotion order).
    - \( \text{prv}(\text{PickUpBucket})[v_1] = s_0[v_1] \) and DropHaystack \( \in \text{Cycle}(v_0) \) but Cycle\((v_0)\) is not concerned by \( C_2^* \) (it is \( C_2^S \)), so (3.27) is false.

- \( N_{\text{prv}}(\text{DropBucket}) = \emptyset \).

Phase 3: Topological sort of \( \Delta = (\mathcal{A}, E_A) \) previously built.

\( \mathcal{A} = \{ \text{FillHorseFeeder}, \text{TakeHaystack}, \text{FillHorseTrough}, \text{FillBucketWithWater}, \text{PickUpBucket} \) DropBucket\)

- FillHorseFeeder is yellow, DFSTopo is launched (2.7):
  FillHorseFeeder turns blue (2.1).
  \( E_A(\text{FillHorseFeeder}) = \{ \text{TakeHaystack} \} \)
  FillHorseFeeder is not the first action to modify \( s_0[v_0] \) (2.3).
  TakeHaystack is yellow, DFSTopo is launched (2.7):
  TakeHaystack turns blue (2.1).
  \( E_A(\text{TakeHaystack}) = \{ \text{DropHaystack}, \text{DropBucket} \} \)
  TakeHaystack is the first action to modify \( s_0[v_0] \), DropHaystack is not considered.
  DropBucket \( \notin N_{\text{prv}}(\text{TakeHaystack}) \) (2.3).
  DropBucket is yellow, DFSTopo is launched (2.7):
  DropBucket turns blue (2.1).
  \( E_A(\text{DropBucket}) = \{ \text{PickUpBucket}, \text{FillHorseTrough}, \text{FillBucketWithWater} \} \)
  DropBucket is the first action to modify \( s_0[v_1] \), PickUpBucket is not considered.
  FillHorseTrough \( \notin N_{\text{prv}}(\text{DropBucket}) \) (2.3).
  FillHorseTrough is yellow, DFSTopo is launched (2.7):
  FillHorseTrough turns blue (2.1).
$E_A(\text{FillHorseTrough}) = \{\text{FillBucketWithWater, PickUpBucket}\}$
\text{FillHorseTrough} is not the first action to modify $s_0[v_2]$ (2.3), \text{FillBucketWithWater} is considered.
\text{FillBucketWithWater} is yellow, \text{DFSTopo} is launched (2.7):
\text{FillBucketWithWater} turns blue (2.1).
$E_A(\text{FillBucketWithWater}) = \{\text{GhostV2, PickUpBucket}\}.$
\text{FillBucketWithWater} is the first action to modify $s_0[v_2]$, so \text{GhostV2} is not considered.
\text{PickUpBucket} $\notin N_{pre}(\text{FillBucketWithWater})$ (2.3).
\text{PickUpBucket} is yellow, \text{DFSTopo} is launched (2.7):
\text{PickUpBucket} turns blue.
$E_A(\text{PickUpBucket}) = \{\text{DropBucket}\}.$
\text{PickUpBucket} is the first action to modify $s_0[v_1]$, \text{DropBucket} is not considered.
\text{PickUpBucket} turns green (2.11) and is added to $\Delta$. \text{DFSTopo} succeeds.
\text{FillBucketWithWater} turns green (2.11) and is added to $\Delta$. \text{DFSTopo} succeeds.
\text{PickUpBucket} is green (2.4) and (2.6) are false.
\text{FillHorseTrough} turns green (2.11) and is added to $\Delta$. \text{DFSTopo} succeeds.
\text{FillBucketWithWater} is green (2.4) and (2.6) are false.
\text{DropBucket} turns green (2.11) and is added to $\Delta$. \text{DFSTopo} succeeds.
\text{TakeHaystack} turns green (2.11) and is added to $\Delta$. \text{DFSTopo} succeeds.
\text{FillHorseFeeder} turns green (2.11) and is added to $\Delta$. \text{DFSTopo} succeeds.

- \text{TakeHaystack} is green.
- \text{FillHorseTrough} is green.
- \text{FillBucketWithWater} is green.
- \text{PickUpBucket} is green.
- \text{DropBucket} is green.

The solution plan is: $\Delta = \langle \text{PickUpBucket, FillBucketWithWater, FillHorseTrough, DropBucket, TakeHaystack, FillHorseFeeder} \rangle$

Resolution of $\Pi_2 = (\Phi_{HB}, s_0 = \langle 1, 0, 0 \rangle, s_* = \langle 2, 0, 2 \rangle)$ by $\mathbb{P}$:

Phase 1:
- $s_0[v_0] \neq s_*[v_0]$, \text{BuildChain} builds backwards (3.4):
  - \text{FillHorseFeeder} satisfies $s_*[v_0]$, it is colored yellow and added to $\mathcal{A}$.
    \text{(TakeHaystack, FillHorseFeeder)} is added to $E_A$.
    Next(TakeHaystack) = FillHorseFeeder is defined.
    $pre(FillHorseFeeder) = s_0[v_0]$.
  - \text{BuildChain} ends, $\delta_{v_0}(s_0[v_0], s_*[v_0]) = \langle \text{FillHorseFeeder} \rangle$ has been built.
• $s_0[v_1] = s_*(v_1)$, $\delta_{v_1}(s_0[v_0], s_*(v_0)) = \langle \rangle$.

• $s_0[v_2] \neq s_*(v_2)$, $\text{BuildChain}$ builds backwards (3.4):
  – $\text{FillHorseTrough}$ satisfies $s_*(v_2)$, it is colored yellow and added to $\mathcal{A}$.
    $\text{(FillBucketWithWater, FillHorseTrough)}$ is added to $E_A$.
    $\text{Next}(\text{FillBucketWithWater}) = \text{FillHorseTrough}$ is defined.
    $\text{pre}(\text{FillHorseFeeder}) \neq s_0[v_2]$.
  – $\text{FillBucketWithWater}$ satisfies $\text{pre}(\text{FillHorseTrough})$, it is colored yellow and added to $\mathcal{A}$.
    $\text{(GhostV2, FillBucketWithWater)}$ is added to $E_A$.
    $\text{Next}(\text{GhostV2}) = \text{FillBucketWithWater}$ is defined.
    $\text{pre}(\text{FillHorseFeeder}) = s_0[v_2]$.
  – $\text{BuildChain}$ ends, $\delta_{v_2}(s_0[v_2], s_*(v_2)) = \langle \text{FillBucketWithWater, FillHorseTrough} \rangle$ has been built.

$\mathcal{A} = \{\text{FillHorseFeeder, FillHorseTrough, FillBucketWithWater}\}$

**Phase 2**: For all actions of $\mathcal{A}$ (3.9), check all their prevail-conditions via the set $N_{prv}$ (3.10).

• $N_{prv}(\text{FillHorseFeeder}) = \emptyset$

• $N_{prv}(\text{FillHorseTrough}) = \{\text{PickUpBucket}\}$.

  – $\text{PickUpBucket}$ affects $v_1$. $prv(\text{FillHorseTrough})[v_1] = 1 \neq s_0[v_1]$, so (3.11) is true. $\text{PickUpBucket}$ is white (3.14), so $\text{BuildChain}$ builds backwards (3.15):
    * $\text{PickUpBucket}$ establishes $prv(\text{FillHorseTrough})$, it is colored yellow and added to $\mathcal{A}$.
      $\text{(DropBucket, PickUpBucket)}$ is added to $E_A$.
      $\text{Next}(\text{DropBucket}) = \text{PickUpBucket}$ is defined.
      $\text{pre}(\text{PickUpBucket}) = s_0[v_1]$.
    * $\text{BuildChain}$ ends, $\delta_{v_1}(s_0[v_1], prv(\text{FillHorseTrough})[v_1]) = \langle \text{PickUpBucket} \rangle$ has been built.

$\text{(PickUpBucket, FillHorseTrough)}$ (3.17, establishment order) is added to $E_A$.
$prv(\text{FillHorseTrough})[v_1] = 1 \neq s_*(v_1)$, so (3.19) is true. $\text{Next}(\text{PickUpBucket}) = \emptyset$ (3.20), so $\text{BuildChain}$ builds backwards (3.21):

  * $\text{DropBucket}$ satisfies $s_0[v_1]$, it is colored yellow and added to $\mathcal{A}$.
    $\text{(PickUpBucket, DropBucket)}$ is added to $E_A$.
    $\text{Next}(\text{PickUpBucket}) = \text{DropBucket}$ is defined.
    $\text{pre}(\text{DropBucket}) = prv(\text{FillHorseTrough})[v_1]$.
  * $\text{BuildChain}$ ends, $\delta_{v_1}(prv(\text{FillHorseTrough})[v_1], s_0[v_1]) = \langle \text{DropBucket} \rangle$ has been built.
Next(PickUpBucket) = DropBucket is now defined so, (FillHorseTrough, DropBucket) is added to \(E_A\) (3.24, promotion order).

- \(N_{\text{prev}}(\text{FillBucketWithWater}) = \{\text{PickUpBucket}\}\).
  - \(\text{PickUpBucket}\) affects \(v_1\). \(\text{prv}(\text{FillBucketWithWater})[v_1] = 1 \neq s_0[v_1]\), so (3.11) is true. \(\text{PickUpBucket}\) is yellow (3.14), so (\(\text{FillBucketWithWater}, \text{PickUpBucket}\)) is added to \(E_A\) (3.17, establishment order).
    - \(\text{prv}(\text{FillBucketWithWater})[v_1] = 1 \neq s[v_1]\), so (3.19) is true. Next(\(\text{PickUpBucket}\)) = DropBucket (3.20), so (\(\text{FillBucketWithWater}, \text{DropBucket}\)) is added to \(E_A\) (3.24, promotion order).

- \(N_{\text{prev}}(\text{PickUpBucket}) = \{\text{DropHaystack}\}\).
  - \(\text{DropHaystack}\) affects \(v_0\). \(\text{prv}(\text{PickUpBucket})[v_0] = 0 \neq s_0[v_0] = 1\), so (3.11) is true. \(\text{DropHaystack}\) is white, so (3.14) is true, \(\text{BuildChain}\) builds backwards (3.15):
    - \(\text{TakeHaystack}\) establishes \(\text{prv}(\text{PickUpBucket})[v_0]\), it is colored yellow and added to \(A\).
      - (\(\text{TakeHaystack}, \text{DropHaystack}\)) is added to \(E_A\).
      - Next(\(\text{TakeHaystack}\)) is already defined (1.14).
      - \(\text{pre}(\text{DropHaystack}) = s_0[v_0]\).
    - \(\text{BuildChain}\) ends, \(\delta_{v_0}(s_0[v_0], \text{prv}(\text{PickUpBucket})[v_0]) = \langle \text{DropHaystack} \rangle\) has been built.

(DropHaystack, PickUpBucket) is added to \(E_A\) (3.17, establishment order).
\(\text{prv}(\text{PickUpBucket})[v_0] \neq s[v_0] = 2\), so (3.19) is true. Next(\(\text{DropHaystack}\)) = \(\emptyset\) (3.20), so BuildChain builds backwards (3.21):
  - \(\text{TakeHaystack}\) satisfied \(s_0[v_0]\), it is colored yellow and added to \(A\).
    - (DropHaystack, TakeHaystack) is added to \(E_A\).
    - Next(DropHaystack) = TakeHaystack is defined.
    - \(\text{pre}(\text{TakeHaystack}) = \text{prv}(\text{PickUpBucket})[v_0]\).
    - \(\text{BuildChain}\) ends, \(\delta_{v_0}(\text{prv}(\text{PickUpBucket})[v_0], s_0[v_0])[v_0] = \langle \text{TakeHaystack} \rangle\) has been built.

(PickUpBucket, TakeHaystack) is added to \(E_A\) (3.24, promotion order).

- \(N_{\text{prev}}(\text{DropBucket}) = \emptyset\).
- \(N_{\text{prev}}(\text{DropHaystack}) = \emptyset\).
- \(N_{\text{prev}}(\text{TakeHaystack}) = \{\text{DropBucket}\}\).
  - \(\text{DropBucket}\) affects \(v_1\). \(\text{prv}(\text{TakeHaystack})[v_1] = 0 = s_0[v_1] = s[v_1]\), so (3.11) and (3.19) are false. \(\text{DropBucket} \in \text{Cycle}(v_1)\) and is concerned by \(C^*_2\), so (3.27) is true. (DropBucket, TakeHaystack) is added to \(E_A\) (3.28, promotion order). DropBucket remains yellow.
\[ \mathcal{A} = \{ \text{FillHorseFeeder}, \text{FillHorseTrough}, \text{FillBucketWithWater}, \text{PickUpBucket}, \text{DropBucket}, \text{DropHaystack}, \text{TakeHaystack} \} \]

**Phase 3**: Topological sort of \( \Delta = (\mathcal{A}, E_{\mathcal{A}}) \) previously built.

- **FillHorseFeeder** is yellow, \( DFSTopo \) is launched (2.7):
  - \( \text{FillHorseFeeder} \) turns blue (2.1).
  - \( E_{\mathcal{A}}(\text{FillHorseFeeder}) = \{ \text{TakeHaystack} \} \)
  - \( \text{FillHorseFeeder} \) is not the first action to modify \( s_0[v_0] \) (2.3), \( \text{TakeHaystack} \) is considered.

- **TakeHaystack** is yellow, \( DFSTopo \) is launched (2.7):
  - \( \text{TakeHaystack} \) turns blue (2.1).
  - \( E_{\mathcal{A}}(\text{TakeHaystack}) = \{ \text{DropHaystack}, \text{DropBucket} \} \)
  - \( \text{TakeHaystack} \) is not the first action to modify \( s_0[v_0] \) (2.3), \( \text{DropHaystack} \) is considered.

- **DropHaystack** is yellow, \( DFSTopo \) is launched (2.7):
  - \( \text{DropHaystack} \) turns blue (2.1).
  - \( E_{\mathcal{A}}(\text{DropHaystack}) = \{ \text{TakeHaystack} \} \)
  - \( \text{DropHaystack} \) is the first action to modify \( s_0[v_0] \) (2.3), \( \text{TakeHaystack} \) is not considered.
  - \( \text{DropHaystack} \) turns green and is added to \( \Delta \). \( DFSTopo \) succeeds.

- **DropBucket** is yellow, \( DFSTopo \) is launched (2.7):
  - \( \text{DropBucket} \) turns blue (2.1).
  - \( E_{\mathcal{A}}(\text{DropBucket}) = \{ \text{PickUpBucket}, \text{FillHorseTrough}, \text{FillBucketWithWater} \} \)
  - \( \text{DropBucket} \) is the first action to modify \( s_0[v_1] \) (2.3), \( \text{PickUpBucket} \) is not considered.
  - \( \text{FillHorseTrough} \) is yellow, \( DFSTopo \) is launched (2.7):
    - \( \text{FillHorseTrough} \) turns blue (2.1).
    - \( E_{\mathcal{A}}(\text{FillHorseTrough}) = \{ \text{FillBucketWithWater}, \text{PickUpBucket} \} \)
    - \( \text{FillHorseTrough} \) is not the first action to modify \( s_0[v_2] \) (2.3), \( \text{FillBucketWithWater} \) is considered.

- **FillBucketWithWater** is yellow, \( DFSTopo \) is launched (2.7):
  - \( \text{FillBucketWithWater} \) turns blue (2.1).
  - \( E_{\mathcal{A}}(\text{FillBucketWithWater}) = \{ \text{GhostV2}, \text{PickUpBucket} \} \).
  - \( \text{FillBucketWithWater} \) is the first action to modify \( s_0[v_2] \), so \( \text{GhostV2} \) is not considered.
  - \( \text{PickUpBucket} \) turns green (2.11) and is added to \( \Delta \). \( DFSTopo \) succeeds.

- **PickUpBucket** is yellow, \( DFSTopo \) is launched (2.7):
  - \( \text{PickUpBucket} \) turns blue.
  - \( E_{\mathcal{A}}(\text{PickUpBucket}) = \{ \text{DropBucket} \} \).
  - \( \text{PickUpBucket} \) is the first action to modify \( s_0[v_1] \), \( \text{DropBucket} \) is not considered.
  - \( \text{PickUpBucket} \) turns green (2.11) and is added to \( \Delta \). \( DFSTopo \) succeeds.

- **FillHorseTrough** is green (2.4) and (2.6) are false.
- **FillBucketWithWater** is green (2.4) and (2.6) are false.
DropBucket turns green (2.11) and is added to $\Delta$. DFSTopo succeeds.
TakeHaystack turns green (2.11) and is added to $\Delta$. DFSTopo succeeds.
FillHorseFeeder turns green (2.11) and is added to $\Delta$. DFSTopo succeeds.

- TakeHaystack is green.
- FillHorseTrough is green.
- FillBucketWithWater is green.
- PickUpBucket is green.
- DropBucket is green.

The solution plan is: $\Delta = \langle \text{DropHaystack, PickUpBucket, FillBucketWithWater, FillHorseTrough, DropBucket, TakeHaystack, FillHorseFeeder} \rangle$
Red Dead Redemption 2: Operator Sets

<table>
<thead>
<tr>
<th>$O$</th>
<th>Pre $v_{14}$</th>
<th>Post $v_{14}$</th>
<th>Prevail $\langle v_{14}, v_{15}, v_{16}, v_{17}, v_{18} \rangle$</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$o_{v_{14}}^{1}$</td>
<td>$v_{14} = 0$</td>
<td>$v_{14} = 1$</td>
<td>$\langle u, u, u, u, u \rangle$</td>
<td>Receive / Give the orders</td>
</tr>
<tr>
<td>$o_{v_{14}}^{2}$</td>
<td>$v_{14} = 1$</td>
<td>$v_{14} = 2$</td>
<td>$\langle u, 1, 2, 1, u \rangle$</td>
<td>Take Position</td>
</tr>
<tr>
<td>$o_{v_{15}}^{1}$</td>
<td>$v_{15} = 1$</td>
<td>$v_{15} = 0$</td>
<td>$\langle u, u, u, u, u \rangle$</td>
<td>Store Rifflle</td>
</tr>
<tr>
<td>$o_{v_{15}}^{2}$</td>
<td>$v_{15} = 0$</td>
<td>$v_{15} = 1$</td>
<td>$\langle u, u, u, u, u \rangle$</td>
<td>Equip Rifflle</td>
</tr>
<tr>
<td>$o_{v_{16}}^{1}$</td>
<td>$v_{16} = 0$</td>
<td>$v_{16} = 1$</td>
<td>$\langle u, u, u, u, u \rangle$</td>
<td>Get Ammo</td>
</tr>
<tr>
<td>$o_{v_{16}}^{2}$</td>
<td>$v_{16} = 1$</td>
<td>$v_{16} = 2$</td>
<td>$\langle u, 1, u, u, u \rangle$</td>
<td>Reload Rifflle</td>
</tr>
<tr>
<td>$o_{v_{17}}^{1}$</td>
<td>$v_{17} = 1$</td>
<td>$v_{17} = 0$</td>
<td>$\langle u, u, u, u, u \rangle$</td>
<td>Drop Bandit Bag</td>
</tr>
<tr>
<td>$o_{v_{17}}^{2}$</td>
<td>$v_{17} = 0$</td>
<td>$v_{17} = 1$</td>
<td>$\langle u, u, u, u, u \rangle$</td>
<td>Equip Bandit Bag</td>
</tr>
<tr>
<td>$o_{v_{18}}^{1}$</td>
<td>$v_{18} = 0$</td>
<td>$v_{18} = 1$</td>
<td>$\langle 2, 1, 2, 1, u \rangle$</td>
<td>Attack the Bank</td>
</tr>
</tbody>
</table>

$v_{14} : \text{AttackReadiness, } D_{v_{14}} = \{0 : \text{none}, 1 : \text{OrdersGiven}, 2 : \text{inPosition}\}$.

$v_{15} : \text{Riffle, } D_{v_{15}} = \{0 : \text{stored}, 1 : \text{equiped}\}$.

$v_{16} : \text{Ammo, } D_{v_{16}} = \{0 : \text{none}, 1 : \text{inHandbags}, 2 : \text{engaged}\}$.

$v_{17} : \text{BanditBag, } D_{v_{17}} = \{0 : \text{dropped}, 1 : \text{equiped}\}$.

$v_{18} : \text{BankAttacked, } D_{v_{18}} = \{0 : \text{false}, 1 : \text{true}\}$.

Table F.1: Operator set of the 

The leader \textbf{Give the Orders} while the rest of the group \textbf{Receive the Orders}. These actions can get a parameter for contextualization: \textbf{Give the Orders}(?toNPC), \textbf{Receive the Orders}(?fromNPC). The \textbf{Bandit} is a SAS-PUSC$_2$ problem, solvable by both $\mathcal{P}$ and $\mathcal{P}$. 

165
APPENDIX F. RED DEAD REDEMPTION 2: OPERATOR SETS

<table>
<thead>
<tr>
<th>$O$</th>
<th>Pre</th>
<th>Post</th>
<th>Preval $\langle v_{28}, v_{29} \rangle$</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$o^0_{v_{28}}$</td>
<td>$v_{28} = 1$</td>
<td>$v_{28} = 0$</td>
<td>$\langle u, u \rangle$</td>
<td>Go rest</td>
</tr>
<tr>
<td>$o^1_{v_{28}}$</td>
<td>$v_{28} = 0$</td>
<td>$v_{28} = 1$</td>
<td>$\langle u, u \rangle$</td>
<td>Hang out</td>
</tr>
<tr>
<td>$o^1_{v_{29}}$</td>
<td>$v_{29} = 0$</td>
<td>$v_{29} = 1$</td>
<td>$\langle 1, u \rangle$</td>
<td>Get a Drink</td>
</tr>
<tr>
<td>$o^2_{v_{29}}$</td>
<td>$v_{29} = 1$</td>
<td>$v_{29} = 2$</td>
<td>$\langle 1, u \rangle$</td>
<td>Drink</td>
</tr>
</tbody>
</table>

$v_{27} : \text{Position, } D_{v_{27}} = \{0 : \text{Home, } 1 : \text{Bar}\}$.

$v_{28} : \text{Drink, } D_{v_{28}} = \{0 : \text{none, } 1 : \text{hasOne, } 2 : \text{hasDrunk}\}$.

Table F.2: Operator set of the Drunk Citizen. He is just roaming in the streets: $s_*(\text{Forget}) = \langle 0, 2 \rangle$. The Drunk Citizen is a SAS-PUSC$_2$ problem, solvable by both $\mathbb{P}$ and $\mathcal{P}$.

<table>
<thead>
<tr>
<th>$O$</th>
<th>Pre</th>
<th>Post</th>
<th>Preval $\langle v_{23}, v_{24}, v_{25}, v_{26}, v_{27} \rangle$</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$o^0_{v_{23}}$</td>
<td>$v_{23} = 0$</td>
<td>$v_{23} = 1$</td>
<td>$\langle u, u, u, u, u \rangle$</td>
<td>Have a Meal</td>
</tr>
<tr>
<td>$o^1_{v_{24}}$</td>
<td>$v_{24} = 1$</td>
<td>$v_{24} = 0$</td>
<td>$\langle u, u, u, u, u \rangle$</td>
<td>Use Bathroom</td>
</tr>
<tr>
<td>$o^1_{v_{25}}$</td>
<td>$v_{25} = 0$</td>
<td>$v_{25} = 1$</td>
<td>$\langle u, u, u, u, u \rangle$</td>
<td>Relax</td>
</tr>
<tr>
<td>$o^1_{v_{26}}$</td>
<td>$v_{26} = 0$</td>
<td>$v_{26} = 1$</td>
<td>$\langle u, u, u, u, 3 \rangle$</td>
<td>Getting Clean</td>
</tr>
<tr>
<td>$o^2_{v_{26}}$</td>
<td>$v_{26} = 1$</td>
<td>$v_{26} = 2$</td>
<td>$\langle 1, 0, u, 3 \rangle$</td>
<td>Rest</td>
</tr>
<tr>
<td>$o^1_{v_{27}}$</td>
<td>$v_{27} = 0$</td>
<td>$v_{27} = 1$</td>
<td>$\langle u, u, u, u \rangle$</td>
<td>Prepare for Work</td>
</tr>
<tr>
<td>$o^2_{v_{27}}$</td>
<td>$v_{27} = 1$</td>
<td>$v_{27} = 2$</td>
<td>$\langle u, u, 1, u \rangle$</td>
<td>Introduce then Dance</td>
</tr>
<tr>
<td>$o^3_{v_{27}}$</td>
<td>$v_{27} = 2$</td>
<td>$v_{27} = 3$</td>
<td>$\langle u, u, u, u \rangle$</td>
<td>Get paid</td>
</tr>
</tbody>
</table>

$v_{23} : \text{hunger, } D_{v_{23}} = \{0 : \text{hungry, } 1 : \text{fed}\}$.

$v_{24} : \text{needBathroom, } D_{v_{24}} = \{0 : \text{false, } 1 : \text{true}\}$.

$v_{25} : \text{isRelaxed, } D_{v_{25}} = \{0 : \text{false, } 1 : \text{true}\}$.

$v_{26} : \text{endDayRoutine, } D_{v_{26}} = \{0 : \text{notStarted, } 1 : \text{bodyWashed, } 2 : \text{Rested}\}$.

$v_{27} : \text{workRoutine, } D_{v_{27}} = \{0 : \text{notStarted, } 1 : \text{readyToWork, } 2 : \text{hasDanced, } 3 : \text{hasWorked}\}$.

Table F.3: Operator set of the Dancer. His role is to entertain the crowd, the audience: $s_*(\text{Work}) = \langle 0, 0, 0, 3 \rangle$. The Dancer is a SAS-PUSC$_0$ problem, solvable by both $\mathbb{P}$ and $\mathcal{P}$.
### APPENDIX F. RED DEAD REDEMPTION 2: OPERATOR SETS

<table>
<thead>
<tr>
<th>$O$</th>
<th>Pre</th>
<th>Post</th>
<th>Prevail $\langle v_3, v_4, v_5, v_6, v_7, v_8, v_9 \rangle$</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_{v_3}^0$</td>
<td>$v_3 = 1$</td>
<td>$v_3 = 0$</td>
<td>$\langle u, u, u, u, 0, u, u \rangle$</td>
<td>Store Horse Cleaning Tools</td>
</tr>
<tr>
<td>$a_{v_3}^1$</td>
<td>$v_3 = 0$</td>
<td>$v_3 = 1$</td>
<td>$\langle u, u, u, 0, u, u \rangle$</td>
<td>Get Horse Cleaning Tools</td>
</tr>
<tr>
<td>$a_{v_4}^0$</td>
<td>$v_4 = 1$</td>
<td>$v_4 = 0$</td>
<td>$\langle u, u, u, u, u, u, u \rangle$</td>
<td>Untie the Horse</td>
</tr>
<tr>
<td>$a_{v_4}^1$</td>
<td>$v_4 = 0$</td>
<td>$v_4 = 1$</td>
<td>$\langle u, u, u, u, u, u, u \rangle$</td>
<td>Tie the Horse</td>
</tr>
<tr>
<td>$a_{v_5}^0$</td>
<td>$v_5 = 0$</td>
<td>$v_5 = 1$</td>
<td>$\langle 1, 1, u, u, u, u, u \rangle$</td>
<td>Roughly Clean the Horse Foot</td>
</tr>
<tr>
<td>$a_{v_5}^1$</td>
<td>$v_5 = 1$</td>
<td>$v_5 = 2$</td>
<td>$\langle 1, 1, u, u, u, u, u \rangle$</td>
<td>Take Horse Shoe Off</td>
</tr>
<tr>
<td>$a_{v_5}^2$</td>
<td>$v_5 = 2$</td>
<td>$v_5 = 3$</td>
<td>$\langle 1, 1, u, u, u, u, u \rangle$</td>
<td>Clean the Barefoot</td>
</tr>
<tr>
<td>$a_{v_5}^3$</td>
<td>$v_5 = 3$</td>
<td>$v_5 = 4$</td>
<td>$\langle 1, 1, u, u, u, u, u \rangle$</td>
<td>Put Horse Shoe On</td>
</tr>
<tr>
<td>$a_{v_6}^0$</td>
<td>$v_6 = 0$</td>
<td>$v_6 = 1$</td>
<td>$\langle u, u, u, 1, u, u \rangle$</td>
<td>Light the Forfe</td>
</tr>
<tr>
<td>$a_{v_6}^1$</td>
<td>$v_6 = 1$</td>
<td>$v_6 = 0$</td>
<td>$\langle 0, u, u, u, u, u \rangle$</td>
<td>Store Forge Tools</td>
</tr>
<tr>
<td>$a_{v_7}^0$</td>
<td>$v_7 = 1$</td>
<td>$v_7 = 0$</td>
<td>$\langle 0, u, u, u, u, u \rangle$</td>
<td>Take Forge Tools</td>
</tr>
<tr>
<td>$a_{v_7}^1$</td>
<td>$v_7 = 0$</td>
<td>$v_7 = 1$</td>
<td>$\langle 0, u, u, u, u, u \rangle$</td>
<td>Take Forge Tools</td>
</tr>
<tr>
<td>$a_{v_8}^0$</td>
<td>$v_8 = 0$</td>
<td>$v_8 = 1$</td>
<td>$\langle u, u, u, 1, u, u \rangle$</td>
<td>Put Old Shoes in the Forge</td>
</tr>
<tr>
<td>$a_{v_8}^1$</td>
<td>$v_8 = 1$</td>
<td>$v_8 = 2$</td>
<td>$\langle u, u, u, 1, u, u \rangle$</td>
<td>Wait for Shoes to be Hot</td>
</tr>
<tr>
<td>$a_{v_9}^0$</td>
<td>$v_9 = 0$</td>
<td>$v_9 = 1$</td>
<td>$\langle u, u, u, 1, u, u \rangle$</td>
<td>Shape Hot Shoes</td>
</tr>
</tbody>
</table>

$v_3 : \text{CleaningTools, } D_{v_3} = \{ 0 : \text{stored, } 1 : \text{equipped} \}.$
$v_4 : \text{HorseHarnessed, } D_{v_4} = \{ 0 : \text{false, } 1 : \text{true} \}.$
$v_5 : \text{HorseFoot, } D_{v_5} = \{ 0 : \text{oldShoes, } 1 : \text{cleaned}, 2 : \text{shoeOff}, 3 : \text{rubbed}, 4 : \text{newShoes} \}.$
$v_6 : \text{ForgeIsLit, } D_{v_6} = \{ 0 : \text{false, } 1 : \text{true} \}.$
$v_7 : \text{Hammer, } D_{v_7} = \{ 0 : \text{stored, } 1 : \text{equipped} \}.$
$v_8 : \text{inForge, } D_{v_8} = \{ 0 : \text{nothing, } 1 : \text{coldShoes, } 2 : \text{hotShoes} \}.$
$v_9 : \text{Shoes, } D_{v_9} = \{ 0 : \text{old, } 1 : \text{shaped} \}.$

Table F.4: Operator set of the **Farrier**. His goals are $s_*(\text{Shoeing the horse}) = \langle 0, 0, 4, s_0[v_6], s_0[v_7], s_0[v_8], s_0[v_9] \rangle$ and $s_*(\text{Forge horseshoes}) = \langle 0, s_0[v_4], s_0[v_5], 1, 0, 2, 1 \rangle$. The **Farrier** is a SAS-PUC$_2$ problem, solvable by Pony.

<table>
<thead>
<tr>
<th>$O$</th>
<th>Pre</th>
<th>Post</th>
<th>Prevail $\langle v_0, v_1, v_2 \rangle$</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$o_{v_0}^0$</td>
<td>$v_0 = 1$</td>
<td>$v_0 = 0$</td>
<td>$\langle u, u, u \rangle$</td>
<td>Drop Haystack</td>
</tr>
<tr>
<td>$o_{v_0}^1$</td>
<td>$v_0 = 0$</td>
<td>$v_0 = 1$</td>
<td>$\langle u, 0, u \rangle$</td>
<td>Take Haystack</td>
</tr>
<tr>
<td>$o_{v_0}^2$</td>
<td>$v_0 = 1$</td>
<td>$v_0 = 2$</td>
<td>$\langle u, u, u \rangle$</td>
<td>Pick up Bucket</td>
</tr>
<tr>
<td>$o_{v_1}^0$</td>
<td>$v_1 = 0$</td>
<td>$v_1 = 1$</td>
<td>$\langle u, u, u \rangle$</td>
<td>Drop Bucket</td>
</tr>
<tr>
<td>$o_{v_1}^1$</td>
<td>$v_1 = 0$</td>
<td>$v_1 = 1$</td>
<td>$\langle 0, u, u \rangle$</td>
<td>Pick up Bucket</td>
</tr>
<tr>
<td>$o_{v_1}^2$</td>
<td>$v_1 = 0$</td>
<td>$v_1 = 1$</td>
<td>$\langle 0, u, u \rangle$</td>
<td>Pick up Bucket</td>
</tr>
<tr>
<td>$o_{v_2}^0$</td>
<td>$v_2 = 0$</td>
<td>$v_2 = 1$</td>
<td>$\langle u, 1, u \rangle$</td>
<td>Fill Bucket with Water</td>
</tr>
<tr>
<td>$o_{v_2}^1$</td>
<td>$v_2 = 1$</td>
<td>$v_2 = 2$</td>
<td>$\langle u, 1, u \rangle$</td>
<td>Fill Horse Trough</td>
</tr>
</tbody>
</table>

$v_0 : \text{HayStack, } D_{v_0} = \{ 0 : \text{none, } 1 : \text{inHands, } 2 : \text{inFeeder} \}.$
$v_1 : \text{Bucket, } D_{v_1} = \{ 0 : \text{none, } 1 : \text{inHands} \}.$
$v_2 : \text{Water, } D_{v_2} = \{ 0 : \text{inSource, } 1 : \text{inBucket, } 2 : \text{inTrough} \}.$

Table F.5: Operator set of the **Horse Breeder**. His goal is to $s_*(\text{Feed the horses}) = \langle 2, 0, 2 \rangle$. The **Horse Breeder** is a SAS-PUC$_2$ problem, solvable by Pony.
### APPENDIX F. RED DEAD REDEMPTION 2: OPERATOR SETS

#### Table F.6: Operator set of the Innkeeper. He can work at the bar: $s_\ast(\text{Serve a drink}) = \langle 1, 2, s^0[v_{20}], 1 \rangle$; and manage the Inn: $s_\ast(\text{Assign a room}) = \langle s^0[v_{18}], s^0[v_{19}], 2, 1 \rangle$. The Innkeeper is a SAS-PUSC$_0$ problem, solvable by both $\mathcal{P}$ and $\mathcal{P}_\ast$.

<table>
<thead>
<tr>
<th>$\mathcal{O}$</th>
<th>Pre</th>
<th>Post</th>
<th>Preval $(v_{19}, v_{20}, v_{21}, v_{22})$</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$o^1_{v_{19}}$</td>
<td>$v_{19} = 0$</td>
<td>$v_{19} = 1$</td>
<td>$(u, u, u, u)$</td>
<td>Roll a Barrel</td>
</tr>
<tr>
<td>$o^1_{v_{20}}$</td>
<td>$v_{20} = 0$</td>
<td>$v_{20} = 1$</td>
<td>$(u, u, u, u)$</td>
<td>Wash the Glass</td>
</tr>
<tr>
<td>$o^2_{v_{20}}$</td>
<td>$v_{20} = 1$</td>
<td>$v_{20} = 2$</td>
<td>$(1, u, u, 1)$</td>
<td>Serve a Drink</td>
</tr>
<tr>
<td>$o^1_{v_{21}}$</td>
<td>$v_{21} = 0$</td>
<td>$v_{21} = 1$</td>
<td>$(u, u, u, u)$</td>
<td>Clean Room</td>
</tr>
<tr>
<td>$o^1_{v_{22}}$</td>
<td>$v_{21} = 1$</td>
<td>$v_{21} = 2$</td>
<td>$(u, u, 1, u)$</td>
<td>Assign Room</td>
</tr>
<tr>
<td>$o^1_{v_{22}}$</td>
<td>$v_{22} = 0$</td>
<td>$v_{22} = 1$</td>
<td>$(u, u, u, u)$</td>
<td>Collect Cash</td>
</tr>
</tbody>
</table>

$v_{19}:$ DrinksFountain, $\mathcal{D}_{v_{19}} = \{0: \text{empty}, 1: \text{filled}\}.$
$v_{20}:$ Glass, $\mathcal{D}_{v_{20}} = \{0: \text{dirty}, 1: \text{washed}, 2: \text{served}\}.$
$v_{21}:$ Room, $\mathcal{D}_{v_{21}} = \{0: \text{dirty}, 1: \text{cleaned}, 2: \text{assigned}\}.$
$v_{22}:$ MoneyCollected, $\mathcal{D}_{v_{22}} = \{0: \text{false}, 1: \text{true}\}.$

#### Table F.7: Operator set of the Native hunter. His goal is to $s_\ast(\text{Hunt}) = \langle 2, 0, 1, 1 \rangle).$ The Native hunter is a SAS-PUSC$_2$ problem, solvable by both $\mathcal{P}$ and $\mathcal{P}_\ast$.

<table>
<thead>
<tr>
<th>$\mathcal{O}$</th>
<th>Pre</th>
<th>Post</th>
<th>Preval $(v_{10}, v_{11}, v_{12}, v_{13})$</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$o^1_{v_{10}}$</td>
<td>$v_{10} = 0$</td>
<td>$v_{10} = 1$</td>
<td>$(u, 1, u, u)$</td>
<td>Go Hunting</td>
</tr>
<tr>
<td>$o^2_{v_{10}}$</td>
<td>$v_{10} = 1$</td>
<td>$v_{10} = 2$</td>
<td>$(u, u, u, u)$</td>
<td>Store the Game</td>
</tr>
<tr>
<td>$o^0_{v_{11}}$</td>
<td>$v_{11} = 1$</td>
<td>$v_{11} = 0$</td>
<td>$(2, u, u, u)$</td>
<td>Un-equip</td>
</tr>
<tr>
<td>$o^1_{v_{11}}$</td>
<td>$v_{11} = 0$</td>
<td>$v_{11} = 1$</td>
<td>$(u, u, 1, u)$</td>
<td>Get Ready to Hunt</td>
</tr>
<tr>
<td>$o^0_{v_{12}}$</td>
<td>$v_{12} = 0$</td>
<td>$v_{12} = 1$</td>
<td>$(u, u, u, u)$</td>
<td>Rest</td>
</tr>
<tr>
<td>$o^1_{v_{13}}$</td>
<td>$v_{13} = 0$</td>
<td>$v_{13} = 1$</td>
<td>$(u, u, u, u)$</td>
<td>Drink</td>
</tr>
</tbody>
</table>

$(v_{10})$HuntingRoutine, $\mathcal{D}_{v_{10}} = \{0: \text{notStarted}, 1: \text{onGoing}, 2: \text{done}\}.$
$(v_{11})$HuntingReadiness, $\mathcal{D}_{v_{11}} = \{0: \text{notReady}, 1: \text{Ready}\}.$
$(v_{12})$Energy, $\mathcal{D}_{v_{12}} = \{0: \text{low}, 1: \text{rested}\}.$
$(v_{13})$Thirst, $\mathcal{D}_{v_{13}} = \{0: \text{thirsty}, 1: \text{hydrated}\}.$
| NPC name - Goal                     | $v_0$ | $v_1$ | $v_2$ | $v_3$ | $v_4$ | $v_5$ | $v_6$ | $v_7$ | $v_8$ | $v_9$ | $v_{10}$ | $v_{11}$ | $v_{12}$ | $v_{13}$ | $v_{14}$ | $v_{15}$ | $v_{16}$ | $v_{17}$ | $v_{18}$ | $v_{19}$ | $v_{20}$ | $v_{21}$ | $v_{22}$ | $v_{23}$ | $v_{24}$ | $v_{25}$ | $v_{26}$ | $v_{27}$ | $v_{28}$ | $v_{29}$ |
|-----------------------------------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|----------|
| Horse breeder - Feeding the horse | 2     | 1     | 2     |       |       |       |       |       |       |       |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |
| Farrier - Shoeing the horse       |       |       |       | 0     | 0     | 4     |       |       |       |       |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |
| Farrier - Forge horseshoes        |       |       |       | 0     | 1     | 0     | 2     | 1     |       |       |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |
| Native hunter - Hunt              |       |       |       | 1     | 1     | 1     | 1     |       |       |       |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |
| Bandit - Rob a bank               |       |       |       | 2     | 1     | 2     | 1     | 1     |       |       |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |          |
| Innkeeper - Serve a drink         |       |       |       |       |       |       |       |       |       |       |          |          |          |          |          |          |          |          |          |          |          |          | 2         | 1         |          |          |          |          |          |          |          |          |          |          |          |
| Innkeeper - Assign a room         |       |       |       |       |       |       |       |       |       |       |          |          |          |          |          |          |          |          |          |          |          |          | 2         | 1         |          |          |          |          |          |          |          |          |          |          |          |          |
| Dancer - Work                     |       |       |       |       |       |       |       |       |       |       |          |          |          |          |          |          |          |          |          |          |          |          | 1         | 0         | 1         | 3         | 2         |          |          |          |          |          |          |
| Drunk citizen - Forget            |       |       |       |       |       |       |       |       |       |       |          |          |          |          |          |          |          |          |          |          |          |          | 0         | 2         |          |          |          |          |          |          |          |          |          |          |          |          |          |

Table F.8: The goals and their encoding. Empty box means the goal is equal to the start.
Cette thèse porte sur l’intelligence artificielle (IA) et la planification d’actions. En particulier, mes recherches portent sur la planification d’actions en temps réel et j’utilise les jeux vidéo pour tester, expérimenter et analyser mes résultats. L’IA est un domaine de connaissance et de recherche visant à concevoir des agents informatiques intelligents, c'est-à-dire des agents qui agissent intelligemment dans leur environnement et en fonction de leurs objectifs. À mon avis, la meilleure définition de l’IA, ou intelligence informatique, est celle de D. Poole, A. Mackworth et R. Goebel [1, p.1], traduite ici en Français :

L’intelligence informatique est l’étude de la conception d’agents intelligents. Un agent est une entité qui agit dans un environnement - il fait quelque chose. Les agents comprennent les vers, les chiens, les thermostats, les avions, les humains, les organisations et la société. Un agent intelligent est un système qui agit de manière intelligente : Ce qu’il fait est approprié aux circonstances et à son objectif, il est flexible face à des environnements et des objectifs changeants, il apprend par l’expérience et il fait des choix appropriés compte tenu des limitations perpectives et des capacités de calcul limitées. L’objectif scientifique central de l’intelligence computationnelle est de comprendre les principes qui rendent possible un comportement intelligent, dans des systèmes naturels ou artificiels. L’hypothèse principale est que le raisonnement est un calcul. L’objectif central de l’ingénierie est de spécifier des méthodes pour la conception d’artefacts utiles et intelligents.

Cette introduction est organisée comme suit : je donne d’abord une définition de la planification en temps réel, je donne ensuite ma motivation suivie de mon approche pour répondre à la question principale : comment réduire la complexité temporelle pour augmenter la planification en temps réel ? Je termine l’introduction par le plan de ma thèse.
G.1 Introduction

La Planification en Temps-Réel : Un problème de planification d’IA, ou plus précisément un problème de planification d’action d’IA (à distinguer du pathfinding et d’autres sous-domaines de la planification), est un problème d’IA où un agent intelligent peut planifier dans son environnement pour atteindre ses objectifs. Pour ce faire, l’environnement et les paramètres pertinents de l’agent sont symbolisés pour former des états que l’agent utilise pour évaluer une situation réelle à un moment donné de manière abstraite. La plupart du temps, un agent évalue sa situation actuelle ou ses objectifs. L’agent est ensuite équipé d’actions ou d’opérateurs que le planificateur utilise pour créer des plans d’action afin de se projeter dans le temps et d’effectuer des actions dans le monde. De plus, les actions et les plans d’action agissent tous deux comme une fonction de transition entre deux états. Ils peuvent avoir un coût afin de caractériser la qualité d’une transition, c’est-à-dire la pertinence d’une action ou d’un plan d’action pour une situation donnée.

La planification en temps réel concerne les systèmes de planification qui planifient dans un laps de temps très court, à savoir en une ou deux millisecondes tout au plus. Un bon exemple d’agents intelligents utilisant de tels planificateurs est celui des personnages non jouables (PNJ) qui peuplent certains jeux vidéo [2–5]. Pour ces jeux, le temps réel correspond à 10% du temps entre deux images, soit 3,33 ms pour un jeu vidéo avec un taux de rafraîchissement de 30 images par seconde (FPS), 1,67 ms pour un jeu à 60 FPS, 0,83 ms pour un jeu à 120 FPS, et ainsi de suite. Comme je m’intéresse aux PNJ contrôlés par des planificateurs, je présente le système Goal-Oriented Action Planning (GOAP) car il s’agit de la technologie utilisée dans les jeux vidéo actuels pour planifier en temps réel (cf. Chapitre 2 du manuscrit).

D’après une étude que j’ai menée sur GOAP, j’ai fait ressortir des usages qui peuvent être utilisés pour restreindre les problèmes de planification. Ces usages m’ont conduit à étudier le formalisme Simplified Action Structure [6] que je présente dans le Chapitre 3. J’utilise ensuite ce formalisme dans le Chapitre 4 et le Chapitre 5 comme base de travail.

Motivation : Je crois fermement que la planification est un domaine de l’IA qui représente l’avenir du comportement des PNJ. Les jeux vidéo sont de plus en plus réalistes et beaux grâce aux graphismes. Les environnements de jeu passent des couloirs (Call Of Duty, The Witcher 1 & 2, Pacman...) à d’immenses mondes ouverts (The Elder Scrolls : Skyrim, Gran Theft Auto, Assassin’s Creed, Red Dead Redemption,...), et même des open-worlds infinis grâce à la génération procédurale (Minecraft, No Man’s Sky...). Certaines histoires sont tellement immersives que vous avez l’impression d’être un acteur dans un film (The Witcher 3, Detroit : Become Human, Fable : The Lost Chapter,...). Les casques de réalité virtuelle sont proches de se retrouver dans tous les foyers, surtout avec l’annonce du Metaverse. Autant dire que les nouvelles technologies sont nombreuses pour créer des jeux vidéo toujours plus immersifs. Il est donc important de peupler ces environnements de haute qualité avec des PNJ crédibles, c’est-à-dire des PNJ qui ne se contentent pas de patrouiller ou d’interagir sans but avec le joueur.

Les PNJ crédibles sont des PNJ qui vivent dans l’environnement, qui semblent avoir
une vie réelle, quoi que fasse le joueur. Si les PNJ doivent intégrer avec le joueur, leurs actions doivent être cohérentes en fonction de leurs objectifs. C’est là que la planification est intéressante, en fonction de du but d’un PNJ, le planificateur va essayer de créer un plan d’action qui relie l’état actuel du PNJ à l’état but. Un plan d’action est plus pertinent qu’une seule action dans le sens où la première action du plan est utile pour une ou plusieurs actions qui viendront ensuite. Par exemple, un PNJ qui transporte une caisse peut le faire pour renforcer une position, et le même PNJ utilisera ensuite sa position renforcée pour se mettre à couvert afin de préparer une embuscade et ainsi de suite. Goal-Oriented Action Planning (GOAP) [7] est le planificateur le plus utilisé dans les jeux vidéo. F.E.A.R. [2], Shadow of Mordors [3], Assassin’s Creed [5], Rise of the Tomb Raider [4], tous ces jeux ont utilisé GOAP pour rendre leurs PNJ plus crédibles. Les PNJ de F.E.A.R sont, à ce jour, toujours considérées comme faisant partie des meilleurs PNJ de l’industrie du jeu vidéo [8]. Les piétons, la faune, les ennemis des récents Assassin’s Creed (Odyssey, Valhalla...) sont contrôlé via GOAP : il rend ces jeux plus dynamiques et immersifs pour les joueurs [9].

La contrainte temps réel signifie que le planificateur doit fournir autant de plans d’action que possible dans le temps imparti. Dans un jeu, le temps imparti est le temps entre deux images. Plus il y a d’agents à contrôler, moins il y a de temps pour générer un plan. De même pour le nombre d’actions dans un plan, plus il y a d’actions, plus il faut de temps pour générer un plan. La complexité temporelle est donc un critère important pour évaluer l’algorithme de planification. Malheureusement, le planificateur actuel utilisé dans GOAP est basé sur A*, un algorithme de recherche, et dont la complexité temporelle est exponentielle [7]. Cela conduit à des restrictions de développement pour respecter le budget temps tout en essayant de répondre aux attentes croissantes des joueurs. Ma question principale tout au long de cette thèse est donc : comment réduire la complexité temporelle pour permettre un passage à l’échelle de la planification en temps réel ?

**Approche** : Pour répondre à cette question, j’ai d’abord étudié GOAP dans un jeu vidéo commercial. F.E.A.R., qui reste une grande référence en termes de PNJ crédibles, est également l’un des rares jeux vidéo commerciaux utilisant GOAP et dont le code source est librement disponible en ligne [10]. Sur la base des résultats de mon étude sur la planification dans F.E.A.R., j’ai ensuite analysé plusieurs dizaines de projets GitHub pour voir si GOAP est toujours mis en œuvre de la même manière. Enfin, j’ai contacté certaines équipes de développeurs utilisant GOAP dans leurs jeux vidéo pour leur poser quelques questions sur leurs habitudes de développement. Tout cela m’a permis de tirer des conclusions que j’ai ensuite transformées en hypothèses.

Ces hypothèses m’ont ensuite conduit à étudier le formalisme SAS (Simplified Action Structure). En effet, deux de mes hypothèses sont en fait des restrictions d’entrée existantes du formalisme SAS. C. Bäckström, le fondateur de SAS, a restreint les problèmes de planification SAS pour trouver des sous-classes de problèmes abordables. Il a également développé un algorithme en temps polynomial qui résout les instances de ses problèmes abordables. J’ai réutilisé l’approche de C. Bäckström pour créer de nouvelles classes de problèmes adaptés aux PNJ dans les jeux vidéo et j’ai ensuite développé un algorithme
ayant une complexité temporelle linéaire capable de résoudre ces nouveaux problèmes. Enfin, j’ai réalisé plusieurs expériences en utilisant plusieurs benchmarks différents pour évaluer l’efficacité de mon algorithme et vérifier qu’il respecte bien la contrainte temps réel.

G.2 Goal-Oriented Action Planning


Dans ce chapitre, j’aborde l’utilisation des coûts d’action dans GOAP, et plus généralement la représentation des actions en pratique dans les jeux commerciaux utilisant GOAP. En effet, ce système gère les situations de jeu inattendues mieux que d’autres alternatives d’ingénierie comportementale ; mais les concepteurs de jeux ressentent un manque de contrôle [14] sur une architecture d’IA capable de produire des comportements inattendus pour les PNJ [7]. Par conséquent, les coûts d’action apparaissent comme un moyen de modifier ce qui se passe à l’écran, ce qui entraîne la question suivante pour une nouvelle équipe de développement de GOAP : comment concevoir les coûts d’action ?

L’IA de F.E.A.R. a été largement reconnue comme la meilleure de sa catégorie lors de sa sortie [16], et l’est toujours [8]. Aussi controversées que puissent être ces critiques, GOAP dans F.E.A.R. a néanmoins eu un grand impact parmi les développeurs d’IA de jeux et a définitivement suscité l’intérêt des planificateurs dans les jeux vidéo commerciaux en général et dans les FPS en particulier.

Grâce au SDK publié par Monolith en 2006 [10], la difficulté de développement de l’IA des jeux n’est plus un problème de codage : le code C++ de GOAP de F.E.A.R. peut être étudié et ses principales parties peuvent être dupliquées. Bien qu’aujourd’hui de nombreux jeux utilisent GOAP pour le comportement des PNJ, F.E.A.R. reste à ce jour le seul jeu dont le SDK permet d’accéder au code C++ du planificateur. Il existe également plus de 300 dépôts GOAP sur Github, implémentant des fonctionnalités de planification pour de
Le créateur de GOAP, Jeff Orkin, propose 'd’appliquer un coût aux actions pour forcer A* à considérer des actions plus spécifiques avant des actions plus générales'. Plus l’action est élevée dans la hiérarchie d’héritage C++, plus le coût de l’action est important ; et c’est effectivement ce que j’observe en étudiant le code source de F.E.A.R.

Conclusion : J’ai analysé les données de planification collectées en jouant à F.E.A.R. pour évaluer l’utilisation des coûts d’action dans GOAP afin de fournir une réponse à l’important problème de développement de l’IA du jeu, à savoir la conception de ces coûts d’action au début d’une nouvelle partie. J’ai découvert que les comportements des PNJs sont le résultat des préconditions contextuelles : les coûts d’action ont un impact négligeable sur les comportements des PNJs. Par conséquent, si votre implémentation de GOAP suit de près celle de F.E.A.R., les coûts d’action peuvent être supprimés de cette implémentation en faveur des préconditions contextuelles ; cela se fait au prix de quelques cycles CPU, cependant, et le temps d’exécution des préconditions contextuelles ne doit pas être surutilisé mais surveillé de près pour satisfaire le budget temps de GOAP.


Finalement, même si les préconditions contextuelles dans F.E.A.R. diminuent les performances d’exécution du planificateur GOAP, je crois fermement que c’est la voie à suivre. En fait, parmi les réponses que nous avons obtenues en questionnant les équipes de développeurs avec la question : 'Utilisez-vous des préconditions de contexte ?', il y a la réponse de Troy Humphrey (Highmoon Studio) qui nous a dit que ce genre de préconditions est mauvais parce que les développeurs vont mettre trop d’instructions à l’intérieur, y compris des fonctions lourdes comme le pathfinding (C’est le cas pour certaines actions de F.E.A.R.), même si on leur dit de ne pas le faire. Le résultat est que ces instructions peuvent être exécutées plusieurs fois pendant la planification car il n’y a pas de variable dédiée.
pour elles. C’est la deuxième partie de la réponse de Troy Humphrey, la forme actuelle des préconditions contextuelles n’est pas du tout optimale, elles doivent être stockées d’une manière ou d’une autre car la détection et la planification sont deux abstractions différentes. Finalement, en considérant qu’il existe une manière plus efficace de gérer les préconditions contextuelles et le fait qu’elles éclipsent les coûts d’action en pratique, je suis convaincu que les coûts d’action peuvent être entièrement supprimés en faveur des préconditions contextuelles. La conséquence directe sera que les actions deviendront contextuellement post-unicles.

J’ai donc fait les hypothèses suivantes que j’utilise dans cette thèse pour produire des classes de problèmes abordables et fournir un algorithme de complexité temporelle linéaire, correct et complet pour résoudre toute instance de ces classes.

**Proposition G.2.1** (Working Assumptions). Les hypothèses suivantes sont compatibles avec la création d’un comportement réaliste pour les PNJ :

1. Les actions sont unaires.
2. Les actions sont contextuellement post-unique.
3. Les plans solutions sont totalement ordonnés.
4. Il n’y a pas deux fois la même action dans le plan.

La dernière hypothèse peut être surprenante à première vue. En fait, la taille des plans d’action est plutôt petite, par exemple 1 ou 2 actions au maximum dans F.E.A.R. Et elle est légèrement supérieure dans certains autres jeux vidéo utilisant GOAP. Et dans ces petits plans, en particulier dans F.E.A.R., il n’y a pas deux fois la même action.

**G.3 Simplified Action Structure**


La *Simplified Action Structure* (SAS), introduite par C. Bäckström [6] au début des années 90, est une version restreinte du formalisme *Action Structure* [26]. SAS est similaire au très connu formalisme propositionnel : *Stanford Research Institute Problem Solver* (STRIPS) [27], les opérateurs ont des post-conditions (post) mais : (i) les variables d’état
APPENDIX G. RÉSUMÉ EN FRANÇAIS

sont multivaluées et (ii) les opérateurs ont deux types de préconditions. (i) Le caractère multivalué est une caractéristique du système de planification GOAP (cf. Chapitre 2). C’est en effet plus pratique que les listes d’ajout et de suppression du formalisme STRIPS, surtout lorsqu’on utilise un langage de programmation comme C++ qui permet l’affectation de valeurs. (ii) Les deux types de préconditions sont : les pré-conditions (pré) et les prevail-conditions (prv) (décrites dans ma thèse), et sont facilement adaptables à la représentation actuelle des opérateurs GOAP. La façon dont ces préconditions sont décrites est pratique d’un point de vue algorithmique car leur description permet une définition simple d’un Critère de vérité [23] qui est un critère utile pour prouver la correction et la complétude d’un algorithme. Finalement, les opérateurs de GOAP sont plus proches de ceux de SAS que de ceux de STRIPS.

D. Chapman a prouvé que la planification est non-abordable en général [23] et, malheureusement, le système de planification GOAP ne fait pas exception (cf. Chapitre 2). L’approche de C. Bäckström pour faire face à l’intractabilité générale est d’appliquer des restrictions sur les opérateurs SAS afin de créer et d’étudier des sous-classes de problèmes dans l’espoir de trouver des problèmes utilisables et traitables [6,28]. Parmi ces restrictions figurent la (P)ost-uniqueness et la (U)nariness [29] qui concernent directement 2 de mes 4 hypothèses (cf. Proposition 2.4.1). Cependant, sans l’ajout d’autres restrictions, la classe de problèmes SAS-PU est NP-Hard [30]. L’ajout des restrictions (S)ingle-Valuedness et (B)nariness à (PU) conduit à SAS-PUBS et SAS⁺-PUS qui sont toutes deux des classes traitables et qui peuvent être résolus par les algorithmes en temps polynomial décrits dans la thèse de C. Bäckström [6].

Dans ce chapitre, je rappelle comment un problème de planification est décrit avec le formalisme SAS et ce qu’est un plan solution minimal pour une instance de problème SAS. Ensuite, je rappelle les restrictions existantes avant de me concentrer sur SAS-PU et SAS-PUS, deux classes de problèmes cohérentes pour GOAP selon mes hypothèses.

Conclusion : La restriction (S) est très restrictive [35]. Il est délicat, voir difficile, de représenter des problèmes réalistes avec elle. (S) ne permet pas les problèmes avec deux actions dont l’une nécessite qu’une pièce allumée et l’autre que la pièce soit éteinte, par exemple. Cette restriction ne permet uniquement de dire que la pièce est allumée (resp. éteinte) en tant que prevail-condition. Supprimer (S) de SAS-PUS revient à considérer la classe de problèmes SAS-PU, qui est non-abordable. Le caractère non-abordable est lié aux plans solution minimaux de taille exponentielle. Il existe des problèmes où le plan solution minimal requiert un nombre exponentiel d’instances d’opérateurs, alors qu’avec (S), les plans solution minimaux ont au plus deux fois la même instance. Une de mes hypothèses du chapitre 2 est que les plans d’action dans les jeux vidéo ont au plus une occurrence de la même action, c’est-à-dire au plus une instance par opérateur. Est-il possible de ne considérer que les problèmes SAS-PU dont les plans solution minimaux ont au plus une instance par opérateur ? Existe-t-il une restriction aux structures SAS dont la classe résultante de problèmes avec (P) et (U) n’a que des plans solution minimaux avec une occurrence de la même action ?
G.4 La Planification Topologique

Introduction

Ce chapitre est le cœur de cette thèse où je présente de nouvelles classes de problèmes et dont les instances sont résolues par mon algorithme, nommé TopoPlan et noté \( \mathbb{P} \), en temps linéaire. Dans ce chapitre, je présente d’abord les prérequis qui tirent avantage des restrictions (P) et (U). En particulier, je présente comment identifier les opérateurs et les actions en temps constant pendant la planification et comment les identifiants peuvent être utilisés pour créer de nouveaux ensembles de prédécesseurs. Ensuite, je présente les graphes d’opérateurs et les graphes d’actions qui sont des représentations graphiques de l’entrée et de la sortie d’un problème de planification SAS-PU. J’utilise ensuite ces graphes pour prouver le \( \Upsilon_1 \)-Truth Criterion, un critère de vérité qui prend en compte mes hypothèses. J’utilise également ces graphes et certaines de leurs propriétés pour introduire une nouvelle restriction structurale, notée \( C_k \), me permettant de créer de nouvelles classes de problèmes qui respectent mes hypothèses de travail et qui sont abordables. Je présente enfin mon algorithme en temps linéaire qui tire parti des prérequis et que je prouve correct et complet avec \( \Upsilon_1 \)-Truth Criterion.

Dans le chapitre 2, j’ai défini quatre hypothèses (Proposition 2.4.1) qui m’ont aidé à atteindre la planification en temps réel. Parmi elles figurent les restrictions (P) et (U) qui, une fois combinées avec la structure SAS, créent la classe de problèmes SAS-PU, prouvée NP-Hard [30]. Dans le chapitre 3, je rappelle la restriction (S) qui, avec (P) et (U), créent la classe abordables des problèmes SAS-PUS. (S) est cependant très restrictive et certaines instances des problèmes SAS-PUS ont des plans solution minimaux avec deux instances d’une même action (\( \Upsilon_2 \)). J’édite donc dans ma thèse la représentation SAS-PU et sa structure pour trouver des sous-classes SAS-PU qui sont abordables et \( \Upsilon_1 \), où \( \Upsilon_1 \) signifie qu’il n’y a pas deux fois la même action dans le plan solution minimal de chaque instance résoluble d’un problème.

La propriété non-abordable de la classe SAS-PU est due à des plans solution minimaux de taille exponentielle. L’observation suivante de C. Bäckström [6, p. 175], traduite ci-dessous en Français, l’explique partiellement :

L’une des sources de la planification non-abordable est que l’on autorise des instances qui nécessitent des solutions minimaux de taille exponentielle. [...] Ce n’est cependant pas trivial (de supprimer de telles instances). Il n’existe pas d’ensemble unique de restrictions ayant cet effet. [...] Bien qu’il ne soit pas suffisant de limiter les plans pour atteindre le caractère abordable, c’est une étape inévitable à considérer.

Ce qui manque dans cette observation est la raison pour laquelle certaines solutions minimales sont des plans d’action de taille exponentielle. Une action est une instance d’un opérateur et, par conséquent, une solution minimale de taille exponentielle est un plan d’action où certains opérateurs ont été instanciés un nombre exponentiel de fois. Proof 6.14 [6, p. 138] en est un excellent exemple. La preuve est basée sur le Code
Gray [37] représenté comme un problème SAS-PUB où chaque opérateur modifie un seul bit. Soit $\Phi = (\mathcal{M}, \mathcal{S}, \mathcal{O})$ une structure du problème Code Gray, l’instance $\Pi = (\Phi, s_0 = \langle 0, 0, \ldots, 0 \rangle, s_* = \langle 1, 0, \ldots, 0 \rangle)$ conduit à un plan minimal de taille exponentielle principalement parce que l’opérateur qui fait passer le bit de poids faible de 0 à 1 est au moins instancié $2^{(|\mathcal{M}| - 2)}$ fois. Je montre un exemple à 3 bits du Code Gray dans mon manuscrit. C’est un exemple qui illustre bien comment un plan peut croître rapidement en taille. Les opérateurs du Code Gray peuvent être instanciés de nombreuses fois car les opérateurs affectant le même bit bouclent ensemble.

Prérequis
Dans cette section, je présente des notions et définitions qui tirent avantage des restrictions (P) et (U). En particuliers, je rappelle ce qu’est un opérateur sollicitable :

**Définition G.4.1.** Soit $\mathcal{RO}$ l’ensemble des opérateurs sollicitables (requestable operator) de $\mathcal{O}$. Ces opérateurs résolvent une prevail-condition d’au moins un autre opérateur de $\mathcal{O}$.

$$\mathcal{RO} = \{ o \mid v_i \in \mathcal{M}, o \in \mathcal{O}(v_i), \text{ post}(o)[v_i] \in \mathcal{R}^o_{v_i} \}$$

Les notations $\Upsilon_1$ et $\Upsilon_2$ se généralise de la manière suivante :

**Définition G.4.2** (Restriction de sortie $\Upsilon_k$). Soit $k \in \mathbb{N}$, un plan d’action $\Delta$ est $\Upsilon_k$ ssi:

- Le nombre d’actions du même opérateur dans $\Delta$ est au plus $k$.
- Il y a au moins un opérateur instancié $k$ fois dans $\Delta$.

J’utilise $\Upsilon_k$ pour définir les classes de problèmes SAS. Si un problème est $\Upsilon_k$, alors toutes les instances de ce problème ont un plan solution minimal au plus $\Upsilon_k$. Si une classe est $\Upsilon_k$, alors n’importe quel problème de cette classe est au plus $\Upsilon_k$. Par exemple, la classe SAS-PUS est $\Upsilon_2$ et la classe SAS-PU est $\Upsilon_\infty$. Bien évidemment, tout problème est au moins $\Upsilon_1$. Au vu de ma dernière hypothèse de travail, la question est : “Existe t’il des classes de problèmes au plus $\Upsilon_1$ ?”

$\Upsilon_k$ est cependant une restriction de sortie. J’ai donc eu pour volonté de trouver une nouvelle restriction d’entrée qui, couplée à (P) et (U), permet de créer des classes de problèmes $\Upsilon_1$. Pour cela, j’ai décidé de modéliser les problèmes sous forme de graphe afin de les étudier et de trouver une nouvelle restriction structurelle.

Les Graphes d’Opérateur
De manière similaire aux state-transition graph et aux domain-transition graph [28], ou aux causal graphs [38,39], qui consistent à représenter un problème de planification sous forme de graphe, j’introduis trois nouveaux graphes qui se concentrent sur les opérateurs/actions et leurs relations entre eux : deux d’entre eux sont des graphes d’opérateurs (Domain Operator Graph et Operator Graph) et représentent le problème de planification, le troisième est
un graphe d’actions (Action Graph) et est une représentation graphique du plan solution. Ces graphes axés sur les opérateurs et les actions mettent en lumière les opérateurs et les actions au lieu des états, des valeurs d’état ou des variables d’état. De plus, la sortie d’un planificateur SAS étant des plans d’action, il est pratique de les représenter sous forme de graphe pour visualiser les relations entre les actions.

Comme mentionné ci-dessus, il existe des opérateurs (PU) qui bouclent ensemble et, par conséquent, pour certaines instances de problèmes, ils peuvent être instanciés un nombre exponentiel de fois, créant ainsi des plans de taille exponentielle. Ces opérateurs bouclant entre eux, bouclent via leur dépendance post-pré. Ces dépendances sont observables grâce aux Domain Operator Graph. Je prouve que dans ce graphe, lorsqu’il existe, le cycle formé par certains opérateurs est unique. Ce qui me permet de définir:

**Definition G.4.3.** Soit $v_i \in \mathcal{M}$, $\text{Cycle}(v_i)$ est l’ensemble des opérateurs de l’unique cycle du Domain Operator Graph sur $v_i$.

**Critère de Vérité ($\Upsilon_1$-Truth Criterion)**

Un plan solution minimal pour une instance de problème SAS-PU est composé de plusieurs chaînes d’actions entre l’état de départ et l’état but, et plus précisément il est composé d’une chaîne d’action (éventuellement vide) pour chaque variable d’état. Cette assertion de C. Bäckström tient lorsque (S) est relaxé, et vient de son critère de vérité dénommé Intended Minimal Plan (IMP). L’IMP, cependant, est adapté aux problèmes SAS$^+$-PUS qui sont au plus $\Upsilon_2$. Dans cette section de mon manuscrit, je vise à fournir un critère de vérité pour les instances de problèmes SAS-PU dont les plans solution minimaux sont $\Upsilon_1$. La section dans mon manuscrit est organisée comme suit : je présente la forme générale d’une chaîne d’actions, c’est-à-dire les formes possibles d’une chaîne d’actions qui est $\Upsilon_k$. Sur cette base, je présente les formes possibles lorsque $k = 1$. Ainsi, une chaîne d’actions $\Upsilon_1$ entre deux valeur $x$ et $z$ du domaine $\mathcal{D}_{v_i}$, noté $\text{chain}_{v_i}(x,z)$, a pour forme générique :

$$v_i \in \mathcal{M}, \ x, y, z \in \mathcal{D}_{v_i},$$

$$\text{chain}_{v_i}(x,z) = \delta_{v_i}(x,y) \oplus \delta_{v_i}(y,z)$$

$$\land \forall a \in \delta_{v_i}(x,y), \ ope(a) \in \text{Cycle}(v_i)$$

$$\land \forall b \in \delta_{v_i}(y,z), \ ope(b) \notin \text{Cycle}(v_i)$$

Avec $\delta_{v_i}(x,y)$ la chaîne minimale entre deux valeurs $x$ et $y$, prouvée $\Upsilon_1$ dans mon manuscrit.

En fonction de divers paramètres, je présente l’ensemble des situations de clobbering (i.e. menace) à prendre en compte. Cela me permet de définir mon critère de vérité: $\Upsilon_1$-Truth Criterion, le critère à respecter pour ordonner un plan solution minimal d’une instance de problèmes SAS-PU qui serait $\Upsilon_1$. Dans la section présentant TopoPlan, mon algorithme de planification en temps linéaire, je me sers également de $\Upsilon_1$-Truth Criterion pour prouver la correction et la complétude de TopoPlan.

**Theorem G.4.1 (Upsilon 1-Truth Criterion).** Soit $\Delta = (\mathcal{A}, \prec)$ un plan solution non-linéaire et $\Upsilon_1$ d’une instance de problème SAS-PU $\prod = (\Phi, s_0, s_*)$. Soit $\sigma_{v_i} = (\mathcal{A}(v_i), \prec)$ une chaîne d’actions qui affecte $v_i \in \mathcal{M}$. $\Delta$ est correct et $\Upsilon_1$ ssi:
Pour tout $v_i \in M$,

1. $\sigma_{v_i} = \text{chain}_{v_i}(s_0[v_i], s_4[v_i]) \subset \Delta \Rightarrow \sigma_{v_i}$ est $\Upsilon_1$ ou vide.

2. $\forall a \in A \setminus A(v_i)$, $p = \text{prv}(a)[v_i]$, $(\exists T \in \sigma_{v_i}, p = \text{post}(T)[v_i] \land T \prec a) \\
\land (\exists C \in \sigma_{v_i}, p = \text{pre}(C)[v_i] \land (T \prec C \lor p = s_0[v_i])) \Rightarrow (C \text{ est unique } \land a \prec C)$

Une Nouvelle Restriction Structurelle ($C_k$)

$\Upsilon_1$ est une restriction de sortie et tous les problèmes SAS sont au moins $\Upsilon_1$. Pour la conception de problèmes SAS-PU, il résulte que $\Upsilon_1$ ne donne pas de règles ou d’instructions pour créer un problème SAS-PU sans plans minimaux de taille exponentielle. La seule façon de s’assurer qu’un problème est au plus $\Upsilon_1$ est d’exécuter un planificateur qui respecte $\Upsilon_1$-Truth Criterion sur chaque instance du problème. Une autre solution consiste à trouver une nouvelle restriction qui, une fois combinée avec les restrictions syntaxiques (P) et (U), produit une classe où chaque problème est au plus $\Upsilon_1$. Dans cette section, je présente donc une nouvelle restriction structurelle, dénommée $C_k$, qui produit de tels problèmes.

$C_k$ est une restriction structurelle qui se concentre sur le cycle unique $\text{Cycle}(v_i)$ de chaque Domain Operator Graph. Elle est définie comme suit :

Definition G.4.4 (Restriction structurelle $C_k$). Soit $\Phi = (M, S, O)$ une structure de la classe SAS-PU, $v_i \in M$ et $k \in \mathbb{N}$. Je désigne par $C_k$ la restriction structurelle qui limite à au plus $k$ le nombre d’opérateurs à l’intérieur de chaque $\text{Cycle}(v_i)$ ayant au moins un opérateur sollicitable. Soit $o^e_{v_i} \in O$, si $o^e_{v_i} \in RO \land o^e_{v_i} \in \text{Cycle}(v_i)$, alors $|\text{Cycle}(v_i)| \leq k$.

$C_k$ est une restriction sur les structures de la classe SAS-PU, elle est donc nécessaire combinée avec (P) et (U). La restriction $C_k$ est évidemment compatible avec la restriction (S) également.

Dans mon manuscrit, je vais donc étudier les classes de problèmes SAS-PUC$_k$ en fonction des valeurs de $k$. En particuliers, je considère $k = 0$, $k = 2$ and $k \geq 3$. Le cas, $k = 1$, est inutile car cela signifie que les pre-conditions et les post-conditions d’une action sont égales, ce qui n’est pas possible d’après la restriction (S4) [6, p.52].

Theorem G.4.2. Les problèmes SAS-PUC$_0$ sont $\Upsilon_1$.

Proof. cf. manuscrit.

Theorem G.4.3. Les problèmes SAS-PUC$_2$ sont non-abordables.

Proof. cf. manuscrit.

Lemma G.4.1. Pour $k \geq 3$, chaque problème SAS-PUC$_k$ a au moins une instance dont le plan solution minimal est $\Upsilon_2$.

Proof. cf. manuscrit.
D’après mon hypothèse 4, je ne m’intéresse donc pas aux problèmes SAS-PUC\_k avec \( k \geq 3 \). Je montre cependant qu’il existe au moins des problèmes SAS-PUC\_2 qui sont \( \Upsilon_1 \). J’ai donc cherché à catégoriser certains de ces problèmes : Les catégories SAS-PUC\_2 et SAS-PUC\_2\* sont deux catégories de problèmes qui respectent \( \Upsilon_1 \), i.e. ma 4\textsuperscript{ème} hypothèse.

**Definition G.4.5.** La sous-catégories SAS-PUC\_2 est une sous-catégorie de SAS-PUC\_2 et \( \forall v_i \in \mathcal{M} \), \(|\mathcal{RO}(v_i) \cup \text{Cycle}(v_i)| \leq 1\).

La notation \((C_2^S)\) rappelle (S), la restriction Single-Valuedness. \( C_2^S \) signifie qu’il y a au plus un opérateur sollicitable par cycle \( \text{Cycle}(v_i) \).

**Theorem G.4.4.** Les problèmes SAS-PUC\_2 sont \( \Upsilon_1 \).

*Proof.* cf. manuscrit.

Les problèmes SAS-PUC\_2 souffrent de la même contrainte que les problèmes SAS-PUS : les situations On/Off ne sont pas modélisable.

Les problèmes SAS-PUC\_2, en revanche, permettent de modéliser des situations On/Off (cf. l’exemple du Horse Breeder dans ma thèse), c’est-à-dire des problèmes où les deux opérateurs d’un cycle \( \text{Cycle}(v_i) \) de taille 2 sont sollicitables. Ces problèmes sont définis comme suit :

**Definition G.4.6.** La catégorie SAS-PUC\_2 est une sous-catégorie de SAS-PUC\_2 et \( \exists \text{Cycle}(v_i) = \{o^+ , o^- \} \) t.q. \(|\mathcal{RO}(v_i) \cup \text{Cycle}(v_i)| \leq 2\). \( \forall o_x, o_y \in \mathcal{O} \setminus \mathcal{O}(v_i) \), si \( \text{post}(o^+) = \text{prv}(o_x)[v_i] \) et \( \text{post}(o^-) = \text{prv}(o_x)[v_i] \), alors \( o_x \) ne doit pas être relié à \( o_y \) dans l’Operator Graph moins \( \text{Cycle}(v_i) \).

**Theorem G.4.5.** Les problèmes SAS-PUC\_2 sont \( \Upsilon_1 \).

*Proof.* cf. manuscrit

Soit \( \text{Cycle}(v_i) = \{a^+, a^-\} \), L’idée de la classe SAS-PUC\_2 est que, pour chaque instance de ces problèmes, les actions sollicitant l’instance de \( o^+ \) peuvent être ordonnée indépendamment des actions sollicitant l’instance de \( o^- \). Ainsi, le planificateur peut instancier \( o^+ \) et ordonner toutes les actions sollicitant \( a^+ \), puis instancier \( o^- \) et ordonner toutes les actions sollicitant \( a^- \), ou inversement.

Maintenant que mes nouvelles classes de problèmes abordables et \( \Upsilon_1 \) sont introduites, je vais présenter mon algorithme \( \text{TopoPlan} \).

**Mon Algorithme \( \textbf{TopoPlan} \) (\( \mathbb{P} \))**

Cette section dans mon manuscrit présente l’algorithme \( \text{TopoPlan} \), noté \( \mathbb{P} \), qui résout les problèmes SAS-PUC\_k qui sont au plus \( \Upsilon_1 \). Chaque instance de problème est notée \( \Pi = (\Phi, s_0, s_\star) \). \( \mathbb{P} \) est basé sur \( \mathbb{P} \) : il planifie en arrière, pour profiter de la post-unicité, et le fait en trois phases. La phase 1 construit les chaînes d’actions minimales entre le début \( s_0 \) et le but \( s_\star \), pour chaque variable d’état \( v_i \in \mathcal{M} \). Toutes les actions de ces chaînes sont
colorées en jaune ; la phase 2 vérifie les prevail-conditions de toutes les actions jaunes. Elle crée des ordres entre les actions jaunes des différentes chaînes afin d’établir et/ou de promouvoir correctement chaque prevail-condition définie de chacune des actions jaunes. Soit $v_i \in \mathcal{M}$ et $p \in \mathcal{R}_{v_i}^O$ une valeur sollicitable, deux chaînes d’actions minimales, une entre $s_0[v_i]$ et $p$, et une entre $p$ et $s_0[v_i]$ peuvent être construites pour assurer l’établissement de $p$ tout en respectant le $\Upsilon_1$-Truth Criterion (cf. mon manuscrit). Finalement, un plan non linéaire minimal et $\Upsilon_1$ est retourné ; la phase 3 trie topologiquement le plan non linéaire. $\mathbb{P}$ diffère cependant de $\mathcal{P}$ de plusieurs façons : j’ai ajouté des conditions supplémentaires pour prendre en compte la relaxation de la restriction (S) ; je suis allé plus loin dans le pre-processing et j’ai créé des ensembles supplémentaires d’opérateurs (cf. predecessor sets dans mon manuscrit) pour réduire la complexité de $\mathbb{P}$ ; Le pre-processing de $\mathbb{P}$ n’est pas effectué pendant la planification par rapport à $\mathcal{P}$, mais pendant la compilation ; Enfin, $\mathbb{P}$ n’instancie pas les actions, il colore les opérateurs et renvoie des plans d’identifiants.

Dans cette section de mon manuscrit, je présente mon algorithme puis prouve sa complexité temporelle et spatiale. $\mathbb{P}$ est correct et complet, et il a une complexité linéaire en fonction du nombre d’opérateurs ($|\mathcal{O}|$) et du nombre de liens entre opérateur ($|E_O|$) du problème de planification considéré : $O(|\mathcal{O}| + |E_O|)$. Enfin, il a une complexité spatiale quadratique ($O(n^2 \cdot m^2)$) en fonction du nombre de variables d’état ($m = \mathcal{M}$) et du nombre de valeur maximal qu’un domaine de variable peut avoir ($n = \max_{v_i \in \mathcal{M}} |\mathcal{D}_{v_i}|$).

### G.5 Benchmarks et Résultats

Ce chapitre est divisé en deux parties suivies d’une discussion. La première section présente les benchmarks évolutifs qui sont des benchmarks conçus pour croître par rapport au nombre de variables d’état ($m$) ou par rapport au nombre maximum de valeurs dans un domaine de variables d’état ($n$). Parmi ces benchmarks figurent les exemples BlockWorld et Tunnel [6]. J’utilise ces benchmarks pour renforcer empiriquement mes résultats théoriques. Le but de la deuxième section est de montrer que des benchmarks réalistes peuvent être représentés avec mes nouvelles classes de problèmes abordables. Parmi ces benchmarks réalistes figurent RDR-2, ACO et HZD qui sont respectivement basés sur les jeux vidéo commerciaux Red Dead Redemption [11], Assassin’s Creed : Origins [12] et Horizon Zero Dawn [13]. J’utilise ensuite ces benchmarks pour comparer mon implémentation C++ de $\mathbb{P}$ et $\mathcal{P}$. Les résultats sont impressionnants : $\mathbb{P}$ est environ $\approx 34.000$ fois plus rapide que $\mathcal{P}$ pour résoudre les instances de ces problèmes réalistes. Je termine le chapitre par une discussion qui explique pourquoi $\mathbb{P}$ est beaucoup plus rapide que $\mathcal{P}$ ainsi que d’autres différences intéressantes entre les deux. Je donne aussi des exemples et explications pour représenter des problèmes réalistes avec mes nouvelles classes de problèmes abordables. Voici quelques points en question :

- Bien que la relaxation de la restriction syntaxique (S) avec la restriction structurelle ($C_2^*$) donne plus de liberté à la représentation des opérateurs, il y a encore des concessions à faire. Par exemple, il n’est pas possible d’avoir la situation suivante : considérons deux commutateurs On/Off où les deux opérateurs On sont requis par un opérateur A et les deux
Off requis par un autre opérateur B. Tous les opérateurs sont concernés par la restriction \((C_2^*)\) et il faut enlever les deux commutateurs l’un après l’autre de l’Operator Graph pour voir si \((C_2^*)\) est respecté. Il n’est pas respecté parce qu’il y a un chemin entre A et B et un chemin entre B et A dans l’Operator Graph après avoir retiré un commutateur. Dans une telle situation, l’astuce consiste à fusionner les deux commutateurs en un seul. C’est ce que j’ai fait avec le Native Hunter, avec les opérateurs Unequip et GetReadyToHunt. Initialement, j’ai fourni au Native Hunter les opérateurs EquipBow, UnequipBow, EquipQuiver, UnequipQuiver. Pour GoHunting, le Native Hunter devait EquipBow et EquipQuiver et, ensuite, pour StoreTheGame, le Native Hunter devait UnequipBow et UnequipQuiver. Ce modèle est équivalent à la situation des deux commutateurs On/Off décrite précédemment et ne respecte pas \((C_2^*)\). Par conséquent, EquipBow et EquipQuiver ont été fusionnés en GetReadyToHunt et UnequipBow et UnequipQuiver ont été fusionnés en Unequip. \((C_2^*)\) est donc maintenant respecté. Finalement, j’ai décidé de créer une version SAS-PUSC2 en intervertissant l’ordre entre Store the Game et Unequip. Initialement, le Native Hunter devait Unequip avant StoreTheGame, dans la version finale c’est finalement l’inverse.

- Pour collecter les données présentées dans le chapitre Benchmarks and Results, j’ai mis en place des expériences avec un départ fixe pour chaque benchmark (un but fixe pour HZD) et une dizaine de buts (une demi-douzaine de départs pour HZD). En d’autres termes, il n’y a pas plus de 10 instances de problème différentes par benchmark. Ce qui implique que pour 3, 4 millions de PNJs, les instances du problème ont été répétées au moins 340 milliers de fois. Je montre dans ce chapitre que mon processeur, un AMD Ryzen™ 7 2700X, ne met pas en cache les plans solutions des instances du problème, cependant, mais applique le planificateur complètement à chaque fois malgré les nombreuses répétitions. J’ai effectué cette analyse interne du processeur parce que l’utilisation de la fonction \(\text{rand}()\) de la bibliothèque C pour choisir aléatoirement un but (ACO et RDR-2) ou un départ (HZD) n’est pas pertinente dans ces expériences en raison de l’intervalle de temps très court avec lequel \(P\) doit travailler. Même si la fonction \(\text{rand}()\) est appelée à chaque initialisation des états de départ et de but, la seed de temps reste la même. Ce qui implique que la fonction \(\text{rand}()\) renvoie toujours le même nombre, donc \(\text{rand}()\) n’est pas pertinente. Ensuite, l’utilisation d’une boucle for pour choisir un but (ou un départ pour HZD) après l’autre, ou l’utilisation d’une liste de dix mille nombres non triés entre 0 et le nombre de buts (ou de départs pour HZD) pour essayer de choisir aléatoirement un but (ou un départ), a finalement produit les mêmes résultats.
- Les données récoltées de \(P\) et présentées dans la thèse sont un peu bruitées par rapport à celles de \(P\). La raison en est que les deux algorithmes fournissent des plans à un certain nombre de PNJ avec un ordre de grandeur différent. Ces données de \(P\), bien que bruyantes, n’ont cependant pas été faciles à obtenir en raison de l’intervalle de temps très court impliqué par la contrainte de temps réel. Par rapport à l’implémentation C++ de \(P\), je n’ai pas utilisé la classe \textit{vector} de la bibliothèque standard C++ pour stocker les opérateurs, actions, plans et autres listes internes utiles pour \(P\). La raison en est que la classe \textit{vector} possède un conteneur interne de taille fixe qui peut allouer un espace supplémentaire pour faire face à une éventuelle croissance [49]. Même s’il est possible d’allouer un peu de stockage supplémentaire à la main avec la fonction reserve, il n’est pas possible de réserver
APPENDIX G. RÉSUMÉ EN FRANÇAIS

la taille du conteneur dans un vecteur de vecteur. Lorsque $P$ a été codé pour la première fois avec des vecteurs, l’allocation dynamique pour le stockage supplémentaire des vecteurs était visible sur les données collectées. Elle créait des "sauts" dans l’ensemble des données d’exécution et la pente de chaque sous-ensemble encadré par deux sauts était également variable. En d’autres termes, l’utilisation de vecteurs avec $P$ rendait les données collectées inutilisables. Par conséquent, mon implémentation de $P$ est codée avec des fonctions C comme $malloc$ ou $memset$ pour créer et allouer des listes utiles, des variables, des tables de hachage et ainsi de suite. Ensuite, les fonctions internes ($BuildChain$, $SearchExtension$, $TopoPlan$, $DFSTopo$) utilisent autant que possible des pointeurs et des références pour éviter la manipulation d’objets lourds, comme les actions. D’ailleurs, par rapport à $P$, les actions ne sont pas instantiées pendant la planification dans mon implémentation C++ de $P$. En effet, $P$ utilise simplement les références aux actions/opérateurs de la table de hachage qui a été construite dans la configuration. À l’inverse, $P$ planifie pour des problèmes de la classe SAS$^+$-PUS et, pour certaines instances de problèmes, les plans solution minimaux contiennent deux fois les mêmes actions. Par conséquent, l’algorithme de planification $P$ instancie les actions lors de la construction de la chaîne d’actions et les manipule tout au long du processus de planification. Dans mon implémentation C++ de $P$ j’ai essayé de manipuler les actions aussi peu que possible et d’utiliser des références à la place. $P$ renvoie un plan d’action partiellement ordonné $(A, R)$, avec $A$ l’ensemble des actions et $R$ l’ensemble des ordres entre les actions. Les instanciations d’actions dans mon implémentation C++ de $P$ ne se produisent que lors de la construction de chaînes d’actions, afin de les insérer dans $A$, que je représente comme un vecteur d’actions. Enfin, $R$ est un vecteur de vecteurs entiers et il stocke les ordres entre les références des actions dans $A$. Finalement, bien que je sois convaincu que la façon dont j’ai implémenté $P$ ait joué un rôle dans ses performances, la qualité du processeur AMD RyzenTM 7 2700X a également joué un rôle [50]. En particulier, le AMD Ryzen™ 7 2700X possède le Sense MI Technology, ce qui signifie que ce processeur utilise le Neural Net Prediction, un réseau neuronal dont le but est de comprendre les applications et d’anticiper les prochaines étapes du flux d’activité en temps réel. Cette technologie, combinée à de nombreuses autres, comme le pipelining, l’utilisation de répartiteurs, l’utilisation de threads, etc. peut également avoir joué un rôle.

- Il semble que de nombreux facteurs expliquent les énormes différences de performances entre $P$ et $P$. De toute évidence, la réduction de la complexité a joué un rôle, surtout lorsque $m$ et $n$ sont élevés. Cependant, lorsque $m$ est faible ($< 100$), la différence de performance est énorme et semble croître, c’est-à-dire que plus $m$ est faible, plus la différence de performance est élevée. Les problèmes évolutifs mettent ce point en évidence. $P$ est 10 fois plus rapide que $P$ avec le benchmark MultiPrv_n_Cycle lorsque $m$ est faible, et environ 2 fois plus rapide lorsque $m$ est élevé. Avec le benchmark OnePrv_n_Cycle, $P$ est 65 fois plus rapide que $P$ lorsque $m$ est faible, et 178 fois plus rapide lorsque $m$ est élevé. Bien qu’il s’agisse de différences significatives dans les performances d’exécution entre les deux algorithmes, elles ne sont pas aussi significatives que celles des benchmarks réalisistes ($P$ est $> 34.000$ fois plus rapide que $P$). J’ai réalisé une "expérience AMD", c’est-à-dire que j’ai vérifié le comportement de mon processeur lors de la résolution des benchmarks réalisistes, pour m’assurer que les plans n’étaient pas mis en cache. Ce n’était pas le cas, donc il peut y
avoir d’autres explications à cette énorme différence de performance. Mon intuition est que, lorsque \( m \) est petit, le problème est suffisamment petit pour que le processeur AMD mette en cache toutes les fonctions \( P \) et le problème dans le cache L1.

### G.6 Conclusion

**Comment Réduire la Compléxité Temporelle pour un Passage à l’Échelle en Temps-Réel ?**

**Goal-Oriented Action Planning** J’ai mené une étude sur le planificateur GOAP de F.E.A.R. et j’ai découvert que les coûts des actions, qui étaient utilisés par la fonction de coût pour évaluer la cohérence d’un plan, sont en pratique éclipsés par les context preconditions. A tel point que les actions deviennent contextuellement post-uniques (1). Ensuite, je souligne, à l’aide du code source de F.E.A.R., que ses développeurs ont essayé de garder la représentation des actions simple, c’est-à-dire que chaque action a seulement quelques préconditions et une seule post-condition. Ce qui signifie que les actions sont unaires (2). Enfin, j’ai souligné que les plans d’action sont totalement ordonnés (3) et qu’il n’y a pas deux fois la même action dans ceux-ci (4). J’ai enfin transformé ces quatre points en hypothèses que j’ai utilisées tout au long de ma thèse.

**Simplified Action Structure** Je rappelle dans ce chapitre la théorie derrière le formalisme Simplified Action Structure(SAS). Même si GOAP est basé sur STRIPS, il est en fait plus proche du formalisme SAS. De plus, (1) et (2), deux de mes hypothèses, sont des restrictions que C. Bäckström a utilisées pour créer des problèmes SAS+ PUS, qui sont des problèmes qui peuvent être résolus par un algorithme en temps polynomial. Comme mon objectif était de réduire la complexité de la planification en temps réel, j’ai donc implémenté l’algorithme SAS+ PUS en C++, appelé \( P \), et je l’ai essayé sur des benchmarks réalistes, basés sur des jeux vidéo, qui forment des problèmes SAS+ PUS. Les résultats étaient encourageants car \( P \) est capable de planifier pour des centaines de PNJ en temps réel et en leur fournissant des plans d’actions plus longs que ce que les planificateurs actuels de GOAP sont capable de faire. La restriction (S) est cependant très restrictive.

**Topological Planning** Dans ce chapitre, j’ai relâché (S) et présenté de nouvelles restrictions structurelles à combiner avec (P) et (U) de manière à créer des problèmes abordables pour les jeux vidéo. Comme l’une de mes hypothèses est qu’il n’y a pas deux fois la même action dans un plan de solution, j’ai introduit une restriction de sortie dénotée \( \Upsilon_1 \) et que j’utilise pour créer le \( T_1 \)-Truth Criterion, un critère de vérité qui, s’il est respecté, assure que le plan de solution est correct. Cependant, \( \Upsilon_1 \) ne fournit pas d’instructions permettant de produire des problèmes abordables. J’introduis donc une nouvelle restriction, notée \( C_k \), dont le rôle est de restreindre à au plus \( k \) le nombre d’opérateurs sollicitables à l’intérieur du cycle unique de chaque Domain Operator Graph. En effet, j’ai montré dans
ce chapitre que la raison de la non-abordabilité des problèmes SAS-PU est liée aux opérateurs affectant la même variable d’état et bouclant ensemble. J’étudie donc de près et je restrains, avec C_k, les cycles formés par les opérateurs bouclant ensemble. En particulier, je présente les classes de problèmes SAS-PUC_0, SAS-PUC_2^S et SAS-PUC_2^*, et j’ai prouvé que ces problèmes sont abordables car leurs instances solvables sont résolues par des plans solution minimaux qui sont Y_1, c’est-à-dire qu’ils ne contiennent pas deux fois la même action. Finalement, je présente TopoPlan, noté P, un algorithme en temps linéaire qui résout les instances d’un problème SAS-PUC_0, SAS-PUC_2^S ou SAS-PUC_2^*. J’ai prouvé, à l’aide d’un critère de vérité (Y_1-Truth Criterion), que cet algorithme est correct et complet. J’ai également prouvé que la complexité spatiale de P dans le pire des cas est de O(n^2 \cdot m^2), c’est-à-dire quadratique par rapport au nombre de variables d’état et la taille maximale du domaine des variables. Enfin, j’ai prouvé que la complexité temporelle de P dans le pire des cas est de O(|O| + |E_0|), c’est-à-dire linéaire en fonction du nombre d’opérateurs plus le nombre d’ordres entre les opérateurs. J’ai donc réussi à répondre à une partie de ma question principale, celle de savoir comment réduire la complexité temporelle.

Benchmarks and Results Dans ce chapitre, je présente les performances d’exécution de P et P sur différents benchmarks. Je teste d’abord P et P sur des problèmes évolutifs, c’est-à-dire des problèmes dont la difficulté de résolution croît en fonction du nombre de variables d’état ou en fonction de la taille de chaque domaine de variables d’état. Ces repères me permettent de montrer empiriquement la complexité temporelle de P et P. Ils montrent également que, globalement, P est plus rapide que P.


La création d’opérateurs reste cependant délicate avec mes classes de problèmes, certaines concessions peuvent être nécessaires pour rester dans les limites des restrictions. Le jeu en vaut cependant la chandelle, car les instances de ces problèmes sont résolues par P, mon algorithme en temps linéaire avec lequel j’ai prouvé qu’une fois optimisé, il est possible de planifier en temps réel pour des millions de PNJ. Finalement, la principale conclusion à la lumière des performances de P est que la planification en temps réel sur CPU est réalisable à grande échelle avec les technologies actuelles.

Travaux Futures

Dans cette section, je donne quelques idées pour poursuivre les recherches présentées dans mon manuscrit de thèse.
Problèmes SAS-PUC\textsubscript{k} qui sont $\Upsilon_2$ L’étape suivante après les problèmes SAS-PUC\textsubscript{k} qui sont $\Upsilon_1$ est évidemment ceux qui sont $\Upsilon_2$, puis $\Upsilon_3$ et ainsi de suite. À mon avis, l’étude du cycle unique dans les Domain Operator Graph reste cruciale pour ceux qui veulent explorer ces problèmes.

Une autre étape consiste à considérer les structures SAS\textsuperscript{*} ou SAS\textsuperscript{+} avec les restrictions (P), (U) et C\textsubscript{k}. Ces deux structures permettent aux variables d’état d’être indéfinies dans l’état but, et SAS\textsuperscript{+} permet également à l’état de départ d’être indéfini. Cela ajoute une notion d’incertitude qui peut être intéressante pour certaines recherches.

Planification topologique hiérarchique Pour le Native Hunter, j’ai expliqué pourquoi j’ai fusionné les paires d’opérateurs EquipBow/UnequipBow et EquipQuiver/UnequipQuiver dans la paire GetReadyToHunt/Unequip. Les deux premières paires d’opérateurs ne respectaient pas la restriction C\textsubscript{*}, alors que la fusion des deux oui. Mais, GetReadyToHung et Unequip peuvent en fait être vus comme des sous-buts dont les plans de solution minimale sont respectivement $\langle$ EquipBow, EquipQuiver $\rangle$ et $\langle$ UnequipBow, UnequipQuiver $\rangle$. Cette observation ouvre la voie à la planification hiérarchique avec $\mathbb{P}$. On peut créer des problèmes de sous-problèmes de sous-sous-problèmes... et ensuite appliquer $\mathbb{P}$ pour résoudre chaque instance de problème l’une après l’autre afin d’obtenir un plan de solution minimal pour un agent donné.

Appliquer TopoPlan dans les systèmes GOAP C’est une question de maturité technologique (TRL). J’ai prouvé théoriquement et empiriquement mon algorithme et je l’ai appliqué sur des benchmarks réalistes mais de manière abstraite. Les résultats sont impressionnant mais il faut maintenant tester $\mathbb{P}$ sur des mondes virtuels plus concrets, réaliste. L’intégrer dans un projet tout en s’assurant qu’il fonctionne et interagit bien avec les autres sous-systèmes est certainement une pertinente prochaine étape.


BIBLIOGRAPHY


RÉSUMÉ

Cette thèse aborde le problème de la génération de plans en temps-réel permettant le contrôle de plusieurs millions de personnages non-joueurs (PNJs) dans les jeux vidéo et les mondes virtuels. La planification d’actions basée sur des algorithmes de recherche, introduite dans le jeu F.E.A.R. en 2005, a une complexité temporelle exponentielle. Elle ne permet de ne gérer qu’au maximum plusieurs dizaines de PNJs par image et avec peu d’actions par plan. J’ai mené une étude approfondie des plans générés dans les jeux de tir à la première personne et cette étude montre que : (1) les états sont des vecteurs de valeurs énumérées, (2) les états initiaux et finaux peuvent être totalement définis, (3) les actions sont à la fois post-uniques et unaires, (4) les plans sont totalement ordonnés, et (5) les actions n’apparaissent qu’une seule fois dans les plans. (1) à (3) correspondent à des problèmes de planification polynomiaux du cadre Simplified Action Structure (SAS), mais pas (4) ni (5) : je présente donc de nouvelles restrictions satisfaisant (5) pour ces problèmes SAS et un nouvel algorithme de complexité linéaire satisfaisant (4). A l’aide d’expériences mettant en œuvre des problèmes de planification modélisés à partir de jeu-vidéo commerciaux, je montre que mon planificateur surpasse de façon spectaculaire les planificateurs SAS précédents et qu’il est en effet capable de construire des plans plus longs et pour plusieurs millions de PNJs en temps-réel.

MOTS CLÉS

Intelligence Artificielle, Planification d’actions, Planification SAS, Temps-réal, Post-unique, Unaire, Algorithme de complexité linéaire.

ABSTRACT

This thesis address the problem of scaling the generation of plans in real-time to control the behaviors of several millions of Non-Player Characters (NPCs) in video-games and virtual worlds. Search-based action planning, introduced in the game F.E.A.R. in 2005, has an exponential time complexity managing at most several tens of NPCs per frame and with short plans. I carried out a close study of the plans generated in first-person shooters and this study shows that: (1) states are vectors of enumerated values, (2) both initial and final states can be totally defined, (3) actions are both post-unique and unary, (4) plans are totally ordered, and (5) actions occur only once in plans. (1) to (3) satisfy the Simplified Action Structure (SAS) polynomial time planning framework but neither (4) nor (5): I thus present new restrictions satisfying (5) to this SAS planning framework and a new linear-time algorithm satisfying (4). Using experiments involving planning problems modelled from existing commercial video games, I show that my planner outperforms previous SAS planners and that it is indeed capable of building longer plans for several millions of NPCs in real-time.

KEYWORDS

Artificial Intelligence, Action Planning, SAS Planning, Real-Time, Post-unique, Unary, Linear Time Algorithm