Topological Planning with Post-unique and Unary Actions

Guillaume Prévost[1] , **Stéphane Cardon**[1] , **Tristan Cazenave**[2] , **Christophe Guettier**[3] and **Éric Jacopin**[4]
[1]Académie Militaire de Saint-Cyr Coëtquidan, CReC Saint-Cyr, France
[2]Université Paris Dauphine - PSL, LAMSADE, CNRS, France
[3]Safran Electronics and Defense, France
[4]Hawkswell Studios, France
guillaume.prevost@st-cyr.terre-net.defense.gouv.fr

Abstract

We are interested in realistic planning problems to model the behavior of Non-Playable Characters (NPCs) in video games. Search-based action planning, introduced by the game F.E.A.R. in 2005, has an exponential time complexity allowing to control only a dozen NPCs between two frames. A close study of the plans generated in first-person shooters shows that: (1) actions are unary, (2) actions are contextually post-unique and (3) there is no two instances of the same action in an NPC's plan. By considering (1), (2) and (3) as restrictions, we introduce new classes of problems with the Simplified Action Structure formalism which indeed allow to model realistic problems and whose instances are solvable by a linear-time algorithm. We also experimentally show that our algorithm is capable of managing millions of NPCs per frame.

1 Introduction

In 2005, F.E.A.R. was released and it is the first video game to use *action planning* for the control in *real-time* of the *Non-Playable Characters* (NPCs). The planning is executed through the well-known Goal-Oriented Action Planning (GOAP) system [Orkin, 2005]. The behavior of these NPCs were so relevant that reviews praised the approach when released [Ocampo, 2007], and it is still recognized as such today [Horti, 2017]. This *Artificial Intelligence* (AI) technique has then be implemented in several other video games such as Rise of the Tomb Raider [Conway, 2015], Middle Earth: Shadow of Mordor [Higley, 2015], Immortals Fenyx rising and the Assassin's Creed series since Odyssey [Girard, 2021].

Action planning is a sub-field of AI that aims to give agents, or NPCs in our case, the capability to build sequences of actions to plan and behave in their environment. Given the description of an initial state, the description of a goal state and the description of an action set, the purpose of a *planner* is to find a sequence of actions, known as *plan*, to reach the goal state and that is applicable in the initial state, or to return that no such plan exists. GOAP uses finite domain variables to represent the *planning problem* of the NPCs. These problems are decidable but planning remains intractable in general [Bylander, 1994]. In particular, the planning algorithm of GOAP is based on A* whose worst-case time complexity is exponential with the number of actions and the size of the plan. To give a plan to as many NPCs as possible while respecting the real-time constraint, which represents less than 10% of the time between two frames[1], and avoiding triggering the worst-case scenarios, the GOAP developers of these game companies have relied on tricks[2]. Among these tricks are:

1. The action representation is simple. Each action has few pre- and post-conditions. In F.E.A.R., almost all actions are unary, i.e. each action only has one post-condition [Monolith Productions, 2006].

2. There are pruning techniques to avoid exploring the entire action search space while planning [Orkin, 2003; Girard, 2021]. In F.E.A.R., each action has a *Context Precondition* method whose role is to check whether the action is contextually viable. If not, the action is merely withdrawn from the search. It means that even if several actions may have the same post-condition, only some of them (sometimes none) will be considered.

3. Plans are short [Jacopin, 2014]. In Middle Earth: Shadow of Mordor, to keep plans short, Higley explains that some actions are just animations [Higley, 2015]. It means these actions will never be considered by the planner but they will still be animated to make the player believes NPCs have long plans.

Even though game companies use these tricks, the planning problems of their NPCs remain intractable. Points 1., 2. and 3. can be used to define assumptions, however:

1. Actions are unary. Each action only has one post-condition.
2. Actions are (contextually) post-unique. Given a situation, no two actions have the same post-condition.
3. Each action has at most one occurrence in the plan.

In this paper, we propose to use these assumptions as restrictions so as to create classes of planning problems that are

[1]It represents $1,67ms$ for a 60 frames per second video game.

[2]The Software Development Kit of F.E.A.R. is freely available online.

tractable, and with an algorithm capable of solving each instance. The remainder of the paper is organized as follows: we first give a background on action planning restrictions. We then define an NPC planning problem with post-unique and unary actions to introduce the SAS formalism and the issues with these two restrictions. It follows the definition of new tractable classes of planning problems and the introduction of our linear time algorithm capable of solving the problem instances of these classes, along with complexity and correctness theorems. We eventually present a concrete experiment performed on abstract settings to show that our classes of problems allow the creation of realistic problems and to highlight the potential of our planner.

2 Background

The use of restrictions to create tractable classes of problems is not new [Cooper *et al.*, 2012]. C. Bäckström, in his thesis, has developed the *Simplified Action Structure* (SAS) formalism and has applied restrictions on his action representation to create the first tractable class of problems: SAS-PUS [Bäckström, 1992]. PUS are the restrictions and stand for Post-uniqueness, Unariness, and Single-Valuedness, which fits two of our assumptions. In SAS, actions have post-conditions and two types of preconditions: the pre- and prevail-conditions. The pre-conditions define what must be true before the action execution and what will be changed by the post-conditions. Whereas the prevail-conditions define what must be true before the action execution and what must hold during the entire action execution. If an NPC fills a bucket with water, the bucket is previously empty (pre-condition), then filled (post-condition). And the bucket must remain in hands while being filled (prevail-condition). The (S) restriction implies that if a prevail-condition is defined for an action, then all the other actions must either have the same prevail-condition or not be affected by it. In our example, if an NPC has an action requiring the bucket in hands, all the other actions of the set must either require the bucket in hands or not care about having it. In other words, (S) implies that there is no action in the NPC's set whose prevail-condition is to not have the bucket in hands. (S) is very restrictive and prevents the creation of some realistic problems. To generalize our bucket example, one cannot create On/Off situations with the SAS-PUS class of problems.

Considering our assumptions 1. and 2., it would be great to get rid of the single-valuedness. (S) cannot be simply removed, however, C. Bäckström proved that the resulting class SAS-PU is intractable [Bäckström, 1992]. The reason for the intractability is the exponentially-sized minimal[3] solution plans of some problem instances. An underlying conclusion is that some actions have an exponential number of occurrences inside these plans, which, in addition to making the SAS-PU class intractable, does not respect our third assumption. According to **Theorem 4.4** [Bäckström, 1992, p.76], the class SAS-PUS does not respect our third assumption either: the minimal solution plan of each solvable SAS-PUS problem instance contains actions with at most two occurrences. Thus, the question is: does there exist a restriction which, once combined with (P) and (U), creates a tractable class whose solvable problem instances are solved by a minimal solution plans containing actions with at most one occurrence?

$\mathcal{A}$	pre	post	prv	Name
a_1	$v_0 = 1$	$v_0 = 0$	$\langle \mathsf{u}, \mathsf{u}, \mathsf{u} \rangle$	DropHaystack
a_2	$v_0 = 0$	$v_0 = 1$	$\langle \mathsf{u}, 0, \mathsf{u} \rangle$	TakeHaystack
a_3	$v_0 = 1$	$v_0 = 2$	$\langle \mathsf{u}, \mathsf{u}, \mathsf{u} \rangle$	FillHorseFeeder
a_4	$v_1 = 1$	$v_1 = 0$	$\langle \mathsf{u}, \mathsf{u}, \mathsf{u} \rangle$	DropBucket
a_5	$v_1 = 0$	$v_1 = 1$	$\langle 0, \mathsf{u}, \mathsf{u} \rangle$	PickUpBucket
a_6	$v_2 = 0$	$v_2 = 1$	$\langle \mathsf{u}, 1, \mathsf{u} \rangle$	FillBucketWithWater
a_7	$v_2 = 1$	$v_2 = 2$	$\langle \mathsf{u}, 1, \mathsf{u} \rangle$	FillHorseTrough

$$\mathcal{M} = \{v_0 : \text{Haystack},\ v_1 : \text{Bucket},\ v_2 : \text{Water}\}$$
$$\mathcal{D}_{v_0} = \{0 : none,\ 1 : inHands,\ 2 : inFeeder\}$$
$$\mathcal{D}_{v_1} = \{0 : none, 1 : inHands\}$$
$$\mathcal{D}_{v_2} = \{0 : inSource,\ 1 : inBucket,\ 2 : inTrough\}$$

Table 1: The SAS action set of the Horse Breeder. Actions are (P) and (U). Variable u means *undefined*.

(P), (U) and (S) are *syntactical* restrictions because they affect the action representation. There also exists *structural* restrictions that restrict the structure of a planning problem [Jonsson and Bäckström, 1998]. Most of the time, the structure of a problem is represented as a graph: [Domshlak and Brafman, 2002; Helmert, 2006] have used the *causal graph* which focuses on the post-prevail dependencies; [Jonsson and Bäckström, 1998] have introduced the *domain-transition graph* which, unlike the causal graph, focuses on the post-pre dependencies. The idea is then to take advantage of this representation to find a structural restriction. The causal graph and the domain-transition graph, however, respectively ignore the post-pre dependencies or the post-prevail dependencies. To fully captures the intractability of SAS-PU problems and to define a new structural restriction that respect our 3^{rd} assumption, we need to consider both the post-pre and the post-prevail dependencies. To this end, we define in this paper two other types of graph: the *domain action graph* presented in Section 3 and the *action graph* presented in Section 4. They focus on SAS-PU actions and the relations between them.

3 SAS-PU Planning Problems

A SAS *planning problem* is defined as a couple $(\mathcal{M}, \mathcal{A})$, with $\mathcal{M}$ a set of state variables and $\mathcal{A}$ a set of actions. Each state variable $v_i \in \mathcal{M}$ has a domain of values denoted $\mathcal{D}_{v_i}$. A list of size $|\mathcal{M}|$ of such values defines a *state* and the i^{th} element of a state is a value of $\mathcal{D}_{v_i}$ assigned to the state variable v_i. Then, each action of $\mathcal{A}$ is defined with post-conditions (post) and two types of preconditions: the pre-conditions (pre) and the prevail-conditions (prv). The unariness (U) implies each action only has one post-condition. The post-uniqueness (P) implies there are no two actions with the same post-condition. In SAS, the pre- and post-conditions both define the same state variables, so, due to (U), they both only affect one state variable. We thus define them with the state variable af-

[3] A minimal solution plan is a plan with the lowest number of actions.

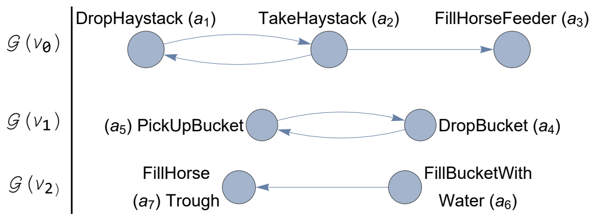

Figure 1: The domain-action graphs of the Horse Breeder.

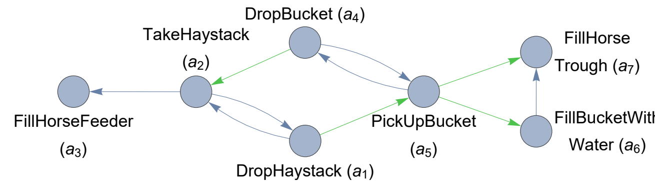

Figure 2: The action graph ($\mathcal{G}$) of the Horse Breeder. The blue edges are the post-pre dependencies ($E_{pre}(\mathcal{A})$) while the green edges are the post-prevail ones ($E_{prv}(\mathcal{A})$).

fected (v_i) and the value assigned ($p, q \in \mathcal{D}_{v_i},\ p \neq q$): $\forall a \in \mathcal{A},\ pre(a) = (v_i = q) \ \wedge \ post(a) = (v_i = p)$. The prevail-conditions, on the contrary, are represented as a partially-defined state: $\forall a \in \mathcal{A},\ prv(a) = \langle v_i = p \mid v_i \in \mathcal{M},\ p \in \mathcal{D}_{v_i} \cup \{\mathsf{u}\}\rangle$, and u is the *undefined* value. Table 1 gives an example of a SAS-PU planning problem with the Horse Breeder, a NPC from the game Red Dead Redemption 2 [DefendTheHouse, 2018].

The purpose of a SAS planner is then to solve a *problem instance* defined as the quadruple $(\mathcal{M}, \mathcal{A}, s_0, s_\star)$, with s_0 the start state and $s_\star$ the goal state. These two states are totally defined in this paper as we work with SAS and not with SAS$^+$ nor SAS* [Jonsson and Bäckström, 1998]. Let I_{hb} denote a problem instance of the Horse Breeder, $I_{hb} = (\mathcal{M}, \mathcal{A}, \langle 0,0,0\rangle, \langle 2,0,2\rangle)$ is a possible problem instance to define the daily routine: feed the horses. His goal is to have the water in the horse trough ($s_\star[v_2] = 2$) and the horse feeder filled with a haystack ($s_\star[v_0] = 2$). At the beginning of his daily routine, he carries nothing ($s_0[v_0] = 0$, $s_0[v_1] = 0$) and the water is in the source ($s_0[v_2] = 0$). To solve a problem instance, the planner will eventually build a minimal solution plan Δ. Such plans are composed of several *chains of actions*, one (possibly empty) for each state variable from the start to the goal state [Bäckström, 1992]. A chain of actions on v_i from $s_0[v_i]$ to $s_\star[v_i]$, denoted $\mathrm{chain}_{v_i}(s_0[v_i], s_\star[v_i])$, is a linear sequence of actions that focus on the relation between the pre- and post-condition. Once all the chains are known, the actions of these chains are ordered among themselves according to their prevail-conditions to create Δ. For example, $\Delta = \langle a_5, a_6, a_7, a_4, a_2, a_3\rangle$ is a linear and minimal solution plan that solves I_{hb}, and Δ is composed of: $\mathrm{chain}_{v_0}(0,2) = \langle a_2, a_3\rangle$, $\mathrm{chain}_{v_1}(0,0) = \langle a_5, a_4\rangle$ and $\mathrm{chain}_{v_2}(0,2) = \langle a_6, a_7\rangle$.

In section 2, we explained that there exist problem instances in SAS that can be solved by minimal solution plans containing actions with several occurrences. In SAS, planners are allowed to instantiate several times the same action to build a plan. This is feasible if and only if these actions are looping with some other actions via the post-pre dependencies. The Horse Breeder can pick up and drop the bucket as many times as he wants because DropBucket and PickUpBucket are looping together. On the contrary, in this problem representation, the planner can only instantiate FillBucketWithWater, FillHorseFeeder or FillHorseTrough once to solve a problem instance. The *domain-action graph* enables the visualization of such cycles:

Definition 1 (Domain-action graph)**.** *Let $v_i \in \mathcal{M}$, a domain-action graph of v_i, denoted $\mathcal{G}(v_i)$, is a weakly connected and directed graph. $\mathcal{G}(v_i) = (\mathcal{A}(v_i), E_{pre}(\mathcal{A}(v_i)))$ s.t.:*

- *$\mathcal{A}(v_i) = \{a \mid a \in \mathcal{A},\ post(a) \in \mathcal{D}_{v_i}\}$ is the vertex set.*
- *$E_{pre}(\mathcal{A}(v_i)) = \{(a,b) \mid a,b \in \mathcal{A}(v_i),\ post(a) = pre(b)\}$ is the directed edge set.*

The Horse Breeder has three domain-action graphs, one for each state variable (cf. Figure 1). $\mathcal{G}(v_0)$ and $\mathcal{G}(v_1)$ have one cycle with two actions while $\mathcal{G}(v_2)$ has no cycle. It can be highlighted that, due to (P), there are at most $|\mathcal{D}_{v_i}|$ actions in $\mathcal{A}(v_i)$. Then, if there are less than $|\mathcal{D}_{v_i}| - 1$ actions in $\mathcal{A}(v_i)$ then $\mathcal{G}(v_i)$ is disconnected and the actions of each of the disconnected part will never end up in the same action plan. Although this is theoretically feasible, with a SAS structure, i.e. with totally defined start and goal states, this is equivalent to considering that there are as many state variables as there are disconnected parts. Hence the weakly connected characteristic of the domain-action graphs. Eventually, if there are exactly $|\mathcal{D}_{v_i}| - 1$ actions in $\mathcal{A}(v_i)$, then there is one value in $\mathcal{D}_{v_i}$ that cannot be set by an action but only by the environment. In our example, we consider that (Water $= inSource$) can only be set by the environment and not by an action of the Horse Breeder.

Lemma 1. *Let $v_i \in \mathcal{M}$,*

- *If $|\mathcal{A}(v_i)| = |\mathcal{D}_{v_i}| - 1$, then $\mathcal{G}(v_i)$ is a directed tree.*
- *If $|\mathcal{A}(v_i)| = |\mathcal{D}_{v_i}|$, then $\mathcal{G}(v_i)$ has a unique cycle.*

Proof. (Idea of proof) This lemma is proven by recursion and by using the previous observations[4]. □

This Lemma is significant as it highlights that actions responsible for the intractability of SAS-PU problems are contained in identifiable cycles:

Definition 2. *We denote $\mathcal{C}ycle(v_i)$ the set of actions that are inside the unique, possibly empty, cycle of $\mathcal{G}(v_i)$.*

For the Horse Breeder, we have: $\mathcal{C}ycle(v_0) = \{a_1, a_2\}$, $\mathcal{C}ycle(v_1) = \{a_4, a_5\}$, $\mathcal{C}ycle(v_2) = \emptyset$. Although, for each $v_i \in \mathcal{M}$, $\mathcal{C}ycle(v_i)$ contains actions that are likely to be instantiated several times, if none of them satisfy one defined prevail-condition of another action, then the planner does not need to pass through $\mathcal{C}ycle(v_i)$ several times. Let consider that the Horse Breeder only has two actions: DropBucket and PickUpBucket: the minimal solution plan to solve $s_0[v_1] = s_\star[v_1]$ is $\Delta = \emptyset$; and, the minimal solution plan to

[4]Feel free to contact the authors to get the technical appendices.

solve $s_0[v_1] \neq s_\star[v_1]$ is of size 1 and contains either an instance of DropBucket or an instance of PickUpBucket depending on which one satisfies $s_\star[v_1]$. Now, no matter how many actions the planning problem is composed of, if none of them requires DropBucket or PickUpBucket to satisfy their prevail-condition, the planner will never loop through $\mathcal{C}ycle(v_1)$. In fact, this is crucial to identify what we called the requestable actions:

Definition 3 (Requestable Action)**.** *A requestable action is an action that solves the prevail-condition of at least another action of $\mathcal{A}$. We denote $\mathcal{R}eq$ the set of requestable actions and $\mathcal{R}eq(v_i)$ the set of requestable actions affecting $v_i \in \mathcal{M}$. $\forall v_i \in \mathcal{M}$, $\mathcal{R}eq(v_i) = \{a \mid a \in \mathcal{A}(v_i),\ \exists b \in \mathcal{A},\ post(a) = prv(b)[v_i]\}$ and $\mathcal{R}eq = \bigcup\limits_{v_i \in \mathcal{M}} \mathcal{R}eq(v_i)$*

For the Horse Breeder, the requestable actions are: $\mathcal{R}eq = \{a_1, a_4, a_5\}$. **Definitions 2** and **3** allow us to define a new structural restriction in function of the number of actions per cycle:

Definition 4. *Let $v_i \in \mathcal{M},\ k \in \mathbb{N}$. Given a SAS-PU problem, we denote C_k the structural restriction that limits to at most k the number of actions inside each $\mathcal{C}ycle(v_i)$ having at least one requestable action. If $\mathcal{R}eq(v_i) \cap \mathcal{C}ycle(v_i) \neq \emptyset$, then $|\mathcal{C}ycle(v_i)| \leq k$.*

In the next section, we introduce the new classes of problems SAS-PUC$_k$. In particular, with respect to some $k \in \mathbb{N}$, we introduce new classes of tractable problems and we indicate from which k our 3^{rd} assumption is no longer respected.

4 The Classes of Problems SAS-PUC$_k$

Our 3^{rd} assumption is: there is no two times the same action in an NPC's plan. This assumption is an output restriction, however, and does not give information on how to design a SAS planning problem. The purpose of this section is thus to study SAS-PUC$_k$ problems for some $k \in \mathbb{N}$ so as to find problems whose solvable instances are necessarily solved by a solution plan respecting our 3^{rd} assumption. If such classes of problems exist, then we will say that they respect the 3^{rd} assumption.

In the following, we only consider $k = 0$, $k = 2$ and $k \geq 3$. More precisely, we prove that the class SAS-PUC$_0$ respects our 3^{rd} assumption but not the class SAS-PUC$_2$ nor the class SAS-PUC$_k$ when $k \geq 3$. With additional restrictions on SAS-PUC$_2$ problems, however, we created two sub-classes, namely SAS-PUC$_2^S$ and SAS-PUC$_2^*$, that respect our 3^{rd} assumption. Concerning $k = 1$, this case is meaningless as it implies there is an action whose pre-condition is equal to its post-condition. This is not possible due to inner restrictions of the SAS formalism [Bäckström, 1992, p.52].

Theorem 1. *SAS-PUC$_0$ problems respect our 3^{rd} assumption.*

Proof. (Idea of proof) In these problems, for each $v_i \in \mathcal{M}$, there is no $\mathcal{C}ycle(v_i)$ containing a requestable action. If the planner instantiates some actions from these cycles, it is just to link the start to the goal state. The planner will never loop through these cycles to seek for a requestable action. For the actions outside such cycles, they can obviously only be instantiated once. The theorem is then proved by recursion. □

Theorem 2. *SAS-PUC$_2$ problems are intractable.*

Proof. (Idea of proof) The Gray Code problem defined in **Proof 6.14** [Bäckström, 1992, p.138] is also of the class SAS-PUC$_2$. The conclusion is that it exists problem instances with an exponentially-sized minimal solution plan. □

Lemma 2. *For $k \geq 3$, every SAS-PUC$_k$ problem has at least one problem instance whose minimal solution plan has at least one action with two occurrences in it.*

Proof. (Idea of proof) For each of these problems, we can build a problem instance similar to the one presented in **Figure 4.3** [Bäckström, 1992, p.75]. It results the statement of this Lemma. □

According to our 3^{rd} assumption and **Lemma 2**, we cannot model a NPC problem with the class SAS-PUC$_k$ with $k \geq 3$. The Horse Breeder problem is of the class SAS-PUC$_2$. It is (P) and (U) and both $\mathcal{C}ycle(v_0)$ and $\mathcal{C}ycle(v_1)$ are concerned by the restriction (C$_2$). $|\mathcal{C}ycle(v_0)| = |\mathcal{C}ycle(v_1)| = 2$ and they both contain at least one requestable action: $\mathcal{R}eq(v_0) \cap \mathcal{C}ycle(v_0) = \{a_1\}$ and $\mathcal{R}eq(v_1) \cap \mathcal{C}ycle(v_1) = \{a_4, a_5\}$. It can be proved by hands that the Horse Breeder problem respects our 3^{rd} assumption. So there exists sub-classes of SAS-PUC$_2$ problems that respect our 3^{rd} assumption: SAS-PUC$_2^S$ and SAS-PUC$_2^*$ are two of them.

Definition 5. *SAS-PUC$_2^S$ is a sub-class of SAS-PUC$_2$ and there is at most one requestable action per domain-action graph cycle: $\forall v_i \in \mathcal{M},\ |\mathcal{R}eq(v_i) \cap \mathcal{C}ycle(v_i)| \leq 1$.*

The letter S in (C$_2^S$) recalls (S), the single-valuedness restriction.

Theorem 3. *SAS-PUC$_2^S$ problems respect our 3^{rd} assumption.*

Proof. (Idea of proof) Let $v_i \in \mathcal{M}$, consider $\mathcal{C}ycle(v_i) = \{a^-, a^+\}$ such that $post(a^+) = +$ and $post(a^-) = -$, and a^+ is the requestable action. If $s_0[v_i] = +$, then the planner does not need to pass through $\mathcal{C}ycle(v_i)$ as $+$ is satisfied by $s_0[v_i]$ and a^- is not requestable. If $s_0[v_i] = -$, then the planner can search for a^+ in $\mathcal{C}ycle(v_i)$. it does not need to do it more than once, however, thus respecting the 3^{rd} assumption. If $s_0[v_i] \in \mathcal{D}_{v_i} \setminus \{-, +\}$, then the planner cannot reach the actions in $\mathcal{C}ycle(v_i)$. So the problem instance is not solvable if a^+ is requested while planning. The theorem is proved by recursion using this idea of proof. □

SAS-PUC$_2^S$ problems suffer from the same issue as the SAS-PUS problems: On/Off situations cannot be modeled. The Horse Breeder is not of the class SAS-PUC$_2^S$ as DropBucket and PickUpBucket are both requestable and looping together: $|\mathcal{R}eq(v_1) \cap \mathcal{C}ycle(v_1)| = 2$. A corollary of both **Theorems 2** and **3** is that a domain-action graph cycle, in which both actions are requestable, is the underlying cause of intractability in certain SAS-PUC$_2$ problems. **Proof 6.14** [Bäckström, 1992, p.138] is a very good example

$\mathcal{A}$	ID	$\mathcal{N}_{pre}$	$\mathcal{N}_{prv}$	Name
a_1	$a^0_{v_0}$	$\{a_2\}$	$\emptyset$	DropHaystack
a_2	$a^1_{v_0}$	$\{a_1\}$	$\{a_4\}$	TakeHaystack
a_3	$a^2_{v_0}$	$\{a_2\}$	$\emptyset$	FillHorseFeeder
a_4	$a^0_{v_1}$	$\{a_5\}$	$\emptyset$	DropBucket
a_5	$a^1_{v_1}$	$\{a_4\}$	$\{a_1\}$	PickUpBucket
a_6	$a^1_{v_2}$	$\{ghost\}$	$\{a_5\}$	FillBucketWithWater
a_7	$a^2_{v_2}$	$\{a_6\}$	$\{a_5\}$	FillHorseTrough

Table 2: Identifiers and predecessor sets for the Horse Breeder.

to understand how a planner can loop through such a cycle: the two requestable actions must be instantiated several times to alternatively satisfy the prevail-condition of some others action instances who already have a specific ordering.

SAS-PUC$^*_2$ problems, on the contrary, allow the use of domain-action graph cycles with two requestable actions, which is essential to model some On/Off situations such as $\mathcal{C}ycle(v_1)$ in the Horse Breeder problem, and still respect our 3^{rd} assumption. To define this class, we need to introduce the *action graph*, which is a graph that captures both the post-pre and the post-prevail dependencies between the actions. Figure 2 gives the action graph of the Horse Breeder.

Definition 6 (Action graph)**.** *An action graph is the directed graph $\mathcal{G} = (\mathcal{A}, E_{\mathcal{A}})$, with $\mathcal{A}$ the vertex set and $E_{\mathcal{A}}$ the directed edge set. $E_{\mathcal{A}} = E_{pre}(\mathcal{A}) \cup E_{prv}(\mathcal{A})$ such that:*

- $E_{prv}(\mathcal{A}) = \{(a, b) \mid a \in \mathcal{R}eq(v_i),\ \exists b \in \mathcal{A},\ post(a) = prv(b)[v_i]\}$*, stores the post-prevail dependencies.*
- $E_{pre}(\mathcal{A}) = \bigcup_{v_i \in \mathcal{M}} (E_{pre}(\mathcal{A}(v_i)))$*, stores the post-pre dependencies.*

Definition 7. *SAS-PUC$^*_2$ is a sub-class of SAS-PUC$_2$ and for each $\mathcal{C}ycle(v_i) = \{a^+, a^-\}$ with two requestable actions, the actions requesting a^+ must not be related to the actions requesting a^- in the action graph $\mathcal{G} \setminus \mathcal{G}(v_i)$.*

The Horse Breeder is of the class SAS-PUC$^*_2$: $a_4, a_5 \in \mathcal{C}ycle(v_1)$ are both requestable and $a_2, a_6, a_7 \in \mathcal{A}$ are such that $\{(a_4, a_2), (a_5, a_6), (a_5, a_7)\} \subset E_{prv}(\mathcal{A})$ and a_2 is not related to either a_6 nor a_7 in $\mathcal{G} \setminus \mathcal{G}(v_1)$. That is, if a_4 and a_5 are removed from $\mathcal{G}$, a_2 is in a different subgraph as a_6 and a_7. Let $\mathcal{C}ycle(v_i) = \{a^+, a^-\}$, the idea of SAS-PUC$^*_2$ is that, in any instance of these problems, the actions requesting a^+ can be ordered independently of the actions requesting a^-. Thus, the planner can instantiate a^+ and ordered all the actions requesting a^+ before instantiating a^- and ordered all the actions requesting a^-.

Theorem 4. *SAS-PUC$^*_2$ problems respect our 3^{rd} assumption.*

Proof. (Idea of the proof) When $\mathcal{G}(v_i)$ is removed from $\mathcal{G}$, the results is a disconnected graph with two or more parts. And each part is an action graph that models either a SAS-PUC$_0$ problem, a SAS-PUC$^S_2$ problem or a SAS-PUC$^*_2$ problem. The theorem is then proved by recursion. □

Procedure 1 BuildChain($v_i, s, g, \mathcal{D}, E_{\mathcal{D}}, \mathcal{A}$)

Input: $v_i \in \mathcal{M}$, $s, g \in \mathcal{D}_{v_i}$, $s \neq g$ and $a^g_{v_i}$ is white; $\mathcal{D}$, a set of yellow actions; $E_{\mathcal{D}}$, a set of orders between the actions of $\mathcal{D}$; **Parameters**: x, y, two values of $\mathcal{D}_{v_i}$. **Output**: Each browsed action a is colored yellow, added to $\mathcal{D}$ and the order $(\mathcal{N}_{pre}(a), a)$ is added to $E_{\mathcal{D}}$.

$x \leftarrow \emptyset$; $y \leftarrow \emptyset$
if $a^g_{v_i}$ is a ghost action **then** fail
end if
Color($a^g_{v_i}$) $\leftarrow yellow$; $\mathcal{D} \leftarrow \mathcal{D} \cup \{a^g_{v_i}\}$; $y \leftarrow pre(a^g_{v_i})$; $E_{\mathcal{D}} \leftarrow E_{\mathcal{D}} \cup \{(a^y_{v_i}, a^g_{v_i})\}$
if Next($a^y_{v_i}$) $= \emptyset$ $\{a^y_{v_i}$ *can be a ghost action.*$\}$ **then**
  Next($a^y_{v_i}$) $\leftarrow a^g_{v_i}$
end if
while $y \neq s$ **do**
  if $a^y_{v_i}$ is a ghost action **then** fail
  end if
  if Color($a^y_{v_i}$) $= yellow$ **then** fail
  end if
  Color($a^y_{v_i}$) $\leftarrow yellow$; $\mathcal{D} \leftarrow \mathcal{D} \cup \{a^y_{v_i}\}$; $x \leftarrow y$; $y \leftarrow pre(a^y_{v_i})$; $E_{\mathcal{D}} \leftarrow E_{\mathcal{D}} \cup \{(a^y_{v_i}, a^x_{v_i})\}$
  if Next($a^y_{v_i}$) $= \emptyset$ $\{a^y_{v_i}$ *can be a ghost action.*$\}$ **then**
    Next($a^y_{v_i}$) $\leftarrow a^x_{v_i}$
  end if
end while

5 Topological Planning

In this section, we present our algorithm *TopoPlan* and two procedures that composes it: *BuildChain* and *DFSTopo*. The specification of our algorithm is the following:

Definition 8. (TopoPlan's specification)
***Input**: $(\mathcal{M}, \mathcal{A}, s_0, s_\star)$, a problem instance of the class SAS-PUC$_0$, SAS-PUC$^S_2$ or SAS-PUC$^*_2$.*
***Output**: If the instance is solvable, then* TopoPlan *returns Δ, a linear and minimal solution plan with at most one occurrence of each action of $\mathcal{A}$. If the instance is not solvable,* TopoPlan *yields a failure.*

5.1 Pre-processing

First of all, there is a pre-processing phase for our algorithm in which each action is associated with the state variable it affects, the value of its pre- and post-condition, and its prevail-conditions. (P) and (U) allow the use of the post-condition of each action as an identifier (ID) for creating a hashing table. Let $v_i \in \mathcal{M}$, $p \in \mathcal{D}_{v_i}$, $a \in \mathcal{A}$: $a^p_{v_i}$ identifies a iff $post(a) = (v_i = p)$. We also created two sets of predecessors: $\mathcal{N}_{pre}(a) = \{b \mid (b, a) \in E_{pre}(\mathcal{A})\}$ is the set of predecessors that establish the pre-condition of a. Due to (P), this set is a singleton. $\mathcal{N}_{prv}(a) = \{b \mid (b, a) \in E_{prv}(\mathcal{A})\}$ is the set of predecessors that establish the defined prevail-conditions of a. Eventually, if $|\mathcal{A}(v_i)| = |\mathcal{D}_{v_i}| - 1$, then there is exactly one ID that identifies a *ghost action*. A ghost action is an action with an ID but that does not exist in the action set. The concatenation of Table 1 and Table 2 is an example of this pre-processing applied to the Horse Breeder.

The post-uniqueness does not mean pre-uniqueness. For example, the Horse Breeder's actions FillHorseFeeder and

Procedure 2 DFSTopo($a^p_{v_i}, \mathcal{D}, E_{\mathcal{D}}, s_0, \Delta$)

Input: $a^p_{v_i}$, a yellow action affecting v_i with the value $p \in \mathcal{D}_{v_i}$; $s_0 \in \mathcal{S}$; Δ, a linear sequence of green actions.
Output: $a^p_{v_i}$ is colored in green once all its neighbors have been topologically sorted; It is then enqueued to Δ.

1: Color($a^p_{v_i}$) $\leftarrow blue$
2: **for** $a^q_{v_j} \in E_{\mathcal{D}}(a^p_{v_i})$ {i.e. $(a^q_{v_j}, a^p_{v_i}) \in E_{\mathcal{D}}$} **do**
3:   **if** $a^q_{v_j} \notin \mathcal{N}_{pre}(a^p_{v_i}) \vee a^p_{v_i}$ 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($a^q_{v_j}, \mathcal{D}, E_{\mathcal{D}}, s_0, \Delta$)
8:     **end if**
9:   **end if**
10: **end for**
11: Color($a^p_{v_i}$) $\leftarrow green$; $\Delta \leftarrow \Delta + \{a^p_{v_i}\}$;

TakeHaystack are post-unique but both have the same precondition. *TopoPlan* plans backwards to benefit from the post-uniqueness, but we gave actions a pointer, called *Next*, that *TopoPlan* will dynamically set (1.6, 1.15) and use (3.20, 3.23, 3.24) while planning to refer in constant time to the successor of an action in a chain[5].

Eventually, each action can have four different colors: (white) the action is not in the solution plan, (yellow) the action is in the solution plan, (blue) the action is being topologically sorted, (green) the action is topologically sorted.

5.2 How TopoPlan Works

According to **Theorems 1**, **3** and **4**, solvable instances of a SAS-PUC$_0$, SAS-PUC$_2^S$ or SAS-PUC$_2^*$ problem are solved by a minimal solution plan that respects our 3^{rd} assumption. We previously explained that such action plans are composed of chains of actions. The strategy of our algorithm, *TopoPlan*, is therefore to find and create every required chain of actions, then to order the actions of these chains via their post-prevail dependencies. The building of the chains is done backwards by the procedure *BuildChain* (3.4, 3.15 and 3.21). The post-prevail orders are done by browsing each action a colored yellow by *BuildChain* (3.9, a is therefore in a chain) and by looking after the predecessors of a via $\mathcal{N}_{prv}(a)$ (3.10).

Due to the fact that we work with totally defined start and goal states, and due to the shape of a domain-action graph with (P) and (U) actions, i.e. a directed tree with a (possibly empty) unique cycle (cf. **Lemma 1**), a chain of actions derived from this graph that respects our 3^{rd} assumption can only have 3 different forms. The first form (σ_1) consists of a unique path from the start value to the goal value. For instance with the Horse Breeder, $\text{chain}_{v_0}(0, 2) = \langle a_2, a_3 \rangle$ and $\text{chain}_{v_2}(0, 2) = \langle a_6, a_7 \rangle$ are of the form of (σ_1). In our algorithm, these chains are built during Phase 1 (3.2 to 3.6) by the *BuildChain* procedure (3.4) for each state variable whose value in the start and goal states is different. The

[5]We refer to the procedures with the notation (procedure.line).

Procedure 3 TopoPlan($\mathcal{M}, \mathcal{A}, s_0, s_\star$)

Input/Output: (**Definition 8**). **Parameters**: $\mathcal{D}$, $\mathcal{T}$, two yellow action sets; $E_{\mathcal{DT}}$, an order set for the actions of $\mathcal{D} \cup \mathcal{T}$.

1: $\Delta \leftarrow \emptyset$; $\mathcal{D} \leftarrow \emptyset$; $\mathcal{T} \leftarrow \emptyset$; $E_{\mathcal{DT}} \leftarrow \emptyset$
2: **for** $v_i \in \mathcal{M}$ **do** {Phase 1}
3:   **if** $s_0[v_i] \neq s_\star[v_i]$ **then**
4:     BuildChain($v_i, s_0[v_i], s_\star[v_i], \mathcal{D}, E_{\mathcal{DT}}, \mathcal{A}$)
5:   **end if**
6: **end for**
7: **if** $\mathcal{D} = \emptyset$ **then return** $\emptyset$ {s_0 and $s_\star$ are equal.}
8: **end if**
9: **for** $a^p_{v_i} \in \mathcal{D} \cup \mathcal{T}$ **do** {Phase 2}
10:   **for** $a^q_{v_j} \in \mathcal{N}_{prv}(a^p_{v_i})$ **do**
11:     **if** $q \neq s_0[v_j]$ **then**
12:       **if** $a^q_{v_j} \notin \mathcal{A}$ **then** fail
13:       **end if**
14:       **if** Color($a^q_{v_j}$) $= white$ **then**
15:         BuildChain($v_j, s_0[v_j], q, \mathcal{T}, E_{\mathcal{DT}}, \mathcal{A}$)
16:       **end if**
17:       $E_{\mathcal{DT}} \leftarrow E_{\mathcal{DT}} \cup \{(a^q_{v_j}, a^p_{v_i})\}$
18:     **end if**
19:     **if** $q \neq s_\star[v_j]$ **then**
20:       **if** Next($a^q_{v_j}$) $= \emptyset$ **then**
21:         BuildChain($v_j, q, s_0[v_j], \mathcal{T}, E_{\mathcal{DT}}, \mathcal{A}$)
22:       **end if**
23:       **if** prv(Next($a^q_{v_j}$))$[v_i] \neq p$ **then**
24:         $E_{\mathcal{DT}} \leftarrow E_{\mathcal{DT}} \cup \{(a^p_{v_i}$,Next($a^q_{v_j}$))}
25:       **end if**
26:     **end if**
27:     **if** $q = s_0[v_j] \wedge a^q_{v_j} \in \mathcal{C}ycle(v_j) \wedge \mathcal{C}ycle(v_j)$ is concerned by (C$_2^*$) **then**
28:       $E_{\mathcal{DT}} \leftarrow E_{\mathcal{DT}} \cup \{(a^{s_0[v_j]}_{v_j}, a^p_{v_i})\}$
29:     **end if**
30:   **end for**
31: **end for**
32: **for** $a \in \mathcal{D} \cup \mathcal{T}$ **do** {Phase 3}
33:   **if** Color(a) $= yellow$ **then**
34:     DFSTopo($a, \mathcal{D} \cup \mathcal{T}, E_{\mathcal{DT}}, s_0, \Delta$)
35:   **end if**
36: **end for**
37: **return** Δ

second form (σ_2) consists of the unfolding of the domain-action graph cycle. It happens when the start value is equal to the post-condition of one of the two actions inside that cycle and the other action of the cycle is requested to solve the problem instance. This action chain (σ_2) therefore leaves and returns to the start value[6]. For instance with the Horse Breeder, $\text{chain}_{v_1}(0, 0) = \langle a_5, a_4 \rangle$ is of the form of (σ_2). These chains can be built during Phase 2 (3.9 to 3.31). In this phase, actions instantiated in a chain are ordered in relation to each other according to their post-prevail dependencies (3.17, 3.24 and 3.28). But some requested actions

[6]It can be highlighted that if the cycle is unfolded more than once, the 3^{rd} assumption is no longer respected.

may not be instantiated yet because they are in a domain-action graph cycle. The two *BuildChain* procedures (3.15) and (3.21) thus build the missing chains. Let consider the Horse Breeder instance $I_{hb} = (\mathcal{M}, \mathcal{A}, \langle 0,0,0\rangle, \langle 2,0,2\rangle)$, at the end of Phase 1, the chains $\mathrm{chain}_{v_0}(0,2) = \langle a_2, a_3\rangle$ and $\mathrm{chain}_{v_2}(0,2) = \langle a_6, a_7\rangle$ are built but $\mathrm{chain}_{v_1}(0,0) = \langle\rangle$. Yet a_5 is required by a_6 and a_7, so the first *BuildChain* procedure (3.15) builds $\mathrm{chain}_{v_1}(0,1) = \langle a_5\rangle$ and the other one (3.21) builds $\mathrm{chain}_{v_1}(1,0) = \langle a_4\rangle$. The concatenation of the two results in $\mathrm{chain}_{v_1}(0,0) = \langle a_5, a_4\rangle$ which is a chain of the form of (σ_2). The last possible form (σ_3) is the concatenation of (σ_2) followed by (σ_1). In the Horse Breeder problem, it can happen for the state variable v_0: if $s_0[v_0] = 1$ and $s_\star[v_0] = 2$, then $\mathrm{chain}_{v_0}(1,2) = \langle a_1, a_2, a_3\rangle$ is feasible such that: $\sigma_1 = \langle a_3\rangle$ and $\sigma_2 = \langle a_1, a_2\rangle$. It should be noticed that, in this case, the first action of (σ_1) and the first action of (σ_2) both have their pre-condition equal to the start value, hence the second stop condition (2.3) in the *DFSTopo* procedure. Eventually, in Phase 3 (3.32 to 3.36), *TopoPlan* topologically sorts the partially ordered sequence of actions $\langle \mathcal{D} \cup \mathcal{T}, E_{\mathcal{DT}}\rangle$ returned by Phase 2.

Theorem 5. *TopoPlan satisfies its specification so it is correct and complete.*

Proof. (Idea of the proof) We prove that the three phases of *TopoPlan* are correct and complete. To do so, we define a loop invariant for each phase and we show that each loop terminates. □

Theorem 6 (Time Complexity)**.** *TopoPlan worst-case time complexity is $O(|\mathcal{A}| + |E_\mathcal{A}|)$, with $\mathcal{A}$ the set of actions and $E_\mathcal{A}$ the set of orders between the actions of $\mathcal{A}$.*

Proof. We explained that the *BuildChain* calls (3.4), (3.15) and (3.21) never build the same chain of actions: (3.4) builds σ_1 chains, (3.15) and (3.21) builds two different parts of σ_2. This is also ensured by the if statements (3.14), (3.20) and (1.11): they check the *BuildChain* procedure only builds chains with white actions, i.e. actions that are not yet in the plan. It results $\mathcal{D} \cap \mathcal{T} = \emptyset$. It also results that, between (3.2) and (3.31) the three *BuildChain* calls execute $O(|\mathcal{D} \cup \mathcal{T}|) \equiv O(|\mathcal{A}|)$ instructions: $|\mathcal{D}|$ instructions in Phase 1 (3.2 to 3.6) plus $|\mathcal{T}|$ instructions in Phase 2 (3.9 to 3.31).

Now, for the two for loops (3.9) and (3.10), assume that all the necessary chains have been built, then the core of the second for loop (from 3.11 to 3.29) has a constant c_{st} number of instructions. The first for loop (3.9) only take 1 instruction to execute. It results the formula:

$$\begin{aligned}\sum_{a \in \mathcal{D}\cup\mathcal{T}} (1 + \sum_{b \in \mathcal{N}_{prv}(a)} c_{st}) &= \sum_{a\in\mathcal{A}} (1 + c_{st} \cdot \sum_{b \in \mathcal{N}_{prv}(a)} 1) \\ &= |\mathcal{A}| + c_{st} \cdot \sum_{(b,a) \in E_{prv}(\mathcal{A})} 1 \\ &= O(|\mathcal{A}| + |E_{prv}(\mathcal{A})|) \qquad (1)\end{aligned}$$

Eventually, at the end of Phase 2 (3.31), *TopoPlan* has executed $O(|\mathcal{A}|) + O(|\mathcal{A}| + |E_{prv}(\mathcal{A})|) \equiv O(|\mathcal{A}| + |E_{prv}(\mathcal{A})|)$ instructions. Phase 3 (3.32 to 3.36), performs a topological sort to sort the non-linear plan $\langle \mathcal{D} \cup \mathcal{T}, E_{\mathcal{DT}}\rangle$. It takes $O(|\mathcal{D} \cup \mathcal{T}| + |E_{\mathcal{DT}}|) \equiv O(|\mathcal{A}| + |E_\mathcal{A}|)$ instructions which dominates the whole. □

Theorem 7 (Space Complexity)**.** *TopoPlan requires at most $O(|\mathcal{A}|^2)$ space.*

Proof. A (PU) action a takes $O(|\mathcal{M}|)$ space: the pre- and post-condition can be reduced to one variable each, plus a variable for the ID; the set $\mathcal{N}_{pre}(a)$ is a singleton due to restrictions (P) and (U); the prevail-conditions, on the contrary, are stored on a list of $|\mathcal{M}|$ elements and the set $\mathcal{N}_{prv}(a)$ has at most $|\mathcal{M}|$ elements. Then, the hashing table that stores IDs takes $O(|\mathcal{A}|)$ space. Finally, the pre-processing phase creates an action graph $\mathcal{G}$ which takes at most $O(|\mathcal{A}|^2)$ space: The set of actions takes $O(|\mathcal{A}|)$ space and each action can have at most $O(|\mathcal{A}|)$ predecessors. Hence $O(|\mathcal{A}|^2)$, which dominates the whole. □

6 Experiments

We have carried out experiments on abstract settings to test *TopoPlan*. Given the description of three different realistic SAS-PUC$_2^*$ problems (Different NPCs from Red Dead Redemption 2 (including the Horse Breeder), citizens from Assassin's Creed: Origins [Ubisoft, 2017] and the acquisition machines from Horizon Zero Dawn [Games, 2017]), we wanted to test how many of these NPCs can get a plan in real time by our C++ implementation of *TopoPlan*? With the following configuration: AMD Ryzen™ 7 2700X (8-Core) CPU (3.7GHz), 32Gb of RAM and Windows 10 (64 bits), *TopoPlan* was able to provide more than 3.000.000 plans with up to 10 actions in it in less than $1.67ms$, which is 10% of the time between two frames in a 60FPS video game[7].

7 Conclusion

Based on three assumptions made by studying commercial video games using planning systems, we have introduced three new tractable classes of planning problems (SAS-PUC$_0$, SAS-PUC$_2^S$ and SAS-PUC$_2^*$) that allow the design of realistic NPCs. The instances of these problems are all solvable by a correct, complete and a linear-time planning algorithm, called *TopoPlan*, that we provide in this paper.

We used the SAS formalism which imposes the start and the goal states to be totally defined. In a future work, we can study other structures such as SAS* [Jonsson and Bäckström, 1998] that allows $s_\star$ to be partially-defined, or SAS$^+$ [Bäckström, 1992] that allows both s_0 and $s_\star$ to be partially defined. Such structures may be easier to use to define NPC problem instances for example. Among our assumptions are the post-uniqueness (P), although it is an acceptable restriction to create NPC planning problems, as shown in this paper, it remains a restriction GOAP developers would like to relieve. Our new structural restriction defined in this paper can surely help to study how (P) can be relieved to create new tractable classes of problems without (P).

[7]More details can be found in [Prévost *et al.*, 2022] about the benchmarks, the experiments and the results. There is no mention of the classes SAS-PUC$_0$, SAS-PUC$_2^S$ and SAS-PUC$_2^*$ in this paper which, instead, groups them under the class SAS-PU$\mathbb{T}_1$, where $\mathbb{T}_1$ is the concatenation of $\mathtt{T}$ (totally-ordered) and $\mathtt{T}_1$ (our 3^{rd} assumption).

Acknowledgments

Musical thanks to Aline Hufschmitt for her reviews and constant support during this work.

We also would like to thank Gabriel Robert and Simon Girard, AI experts from Ubisoft, for sharing their game development experience which definitely helped us to define our assumptions and create our NPC models.

Eventually, we would like to express our sincere gratitude to Peter Jonsson and Aurélie Beynier, the reviewers of Guillaume's thesis, for their invaluable appreciation and their encouragement to publish in this conference.

References

[Bäckström, 1992] Christer Bäckström. *Computational Complexity of Reasoning about Plans*. PhD thesis, Department of Computer and Information Science, Linköping University, september 1992.

[Bylander, 1994] Tom Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2):165–204, 1994.

[Conway, 2015] Chris Conway. GOAP in Tomb Raider. https://youtu.be/gm7K68663rA?t=1200, March 2015. Accessed on May 12^{th}, 2023.

[Cooper *et al.*, 2012] Martin Cooper, Frédéric Maris, and Pierre Régnier. Tractable monotone temporal planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 22, pages 20–28, 2012.

[DefendTheHouse, 2018] DefendTheHouse. NPC daily life in Read Dead Redemption 2. https://youtu.be/MrUJJgppMn4?t=434, November 2018. Accessed on May 12^{th}, 2023.

[Domshlak and Brafman, 2002] Carmel Domshlak and Ronen Brafman. Structure and complexity in planning with unary operators. In *Proceedings of the* 6^{th} *International Conference on Artificial Intelligence Planning Systems*, pages 34–43. AAAI Press, 2002.

[Games, 2017] Guerilla Games. Horizon Zero Dawn. https://www.guerrilla-games.com/games, December 2017. Accessed on May 12^{th}, 2023.

[Girard, 2021] Simon Girard. Postmortem: AI action planning on Assassin's Creed Odyssey and Immortals Fenyx Rising. https://www.gamedeveloper.com/programming/postmortem-AI-action-planning-on-Assassins-Creed-Odyssey-and-Immortals-Fenyx-Rising-, November 2021. Accessed on May 12^{th}, 2023.

[Helmert, 2006] Malte Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[Higley, 2015] Peter Higley. GOAP at monolith productions. https://youtu.be/gm7K68663rA, March 2015. Accessed on May 12^{th}, 2023.

[Horti, 2017] Samuel Horti. Why F.E.A.R.'s AI is still the best in first-person shooters – Flank, cover and run away. https://www.rockpapershotgun.com/why-fears-ai-is-still-the-best-in-first-person-shooters, April 2017. Accessed on May 12^{th}, 2023.

[Jacopin, 2014] Éric Jacopin. Game AI planning analytics: The case of three first-person shooters. In *Proceedings of the* 10^{th} *AIIDE*, pages 119–124. AAAI Press, 2014.

[Jonsson and Bäckström, 1998] Peter Jonsson and Christer Bäckström. State-variable planning under structural restrictions: algorithms and complexity. *Artificial Intelligence*, 100(1-2):125–176, April 1998.

[Monolith Productions, 2006] Monolith Productions. F.E.A.R. public tools. https://www.gamefront.com/games/f-e-a-r/file/f-e-a-r-v1-08-sdk, June 2006. Accessed on May 12^{th}, 2023.

[Ocampo, 2007] Jason Ocampo. F.E.A.R. review. https://www.gamespot.com/reviews/fear-review/1900-6169771/, April 2007. Accessed on May 12^{th}, 2023.

[Orkin, 2003] Jeff Orkin. Applying goal-oriented action planning to games. In Steve Rabin, editor, *AI Game Programming Wisdom 2*, volume 2, chapter 3.4, pages 217–227. Charles River Media, 2003.

[Orkin, 2005] Jeff Orkin. Agent architecture considerations for real-time planning in games. In *Proceedings of the* 1^{st} *AIIDE*, pages 105–110, 2005.

[Prévost *et al.*, 2022] Guillaume Prévost, Stéphane Cardon, Éric Jacopin, Tristan Cazenave, and Christophe Guettier. Planning for millions of NPCs in real-time. In *2022 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 330–336. IEEE, 2022.

[Ubisoft, 2017] Ubisoft. Assassin's Creed: Origins. https://store.ubi.com/fr/assassin-s-creed-origins-all-games, Octobre 2017. Accessed on May 12^{th}, 2023.