SAS Planning As Topological Sorting

Guillaume Prévost[1*] , **Stéphane Cardon**[1] , **Tristan Cazenave**[2] , **Christophe Guettier**[3] and **Éric Jacopin**[1]

[1]CReC Saint-Cyr, Académie Militaire de Saint-Cyr Coëtquidan, F-56380

[2]LAMSADE, Université Paris-Dauphine - PSL, Place de Lattre de Tassigny, F-75775 Paris Cedex 16

[3]Safran Electronics and Defense, 12 Rue Colbert, F-35300

guillaume.prevost22@orange.fr

Abstract

We 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. A close study of the plans generated in first-person shooters shows that: (1) states are vectors of enumerated values, (2) both initial and final states can be totally defined, (3) operators are both post-unique and unary, (4) plans are totally ordered, and (5) operators occur only once in plans. (1) to (3) satisfy the Simplified Action Structure (SAS) polynomial time planning framework but neither (4) nor (5): we thus present two new restrictions to the SAS planning framework and a new linear-time algorithm which dramatically outperforms previous SAS planners and indeed allow us to consider the management of millions of NPCs per frame.

1 Introduction and Motivation

F.E.A.R., a first-person shooter (FPS) released in 2005, was the first video game to use planning to generate character behaviors in real time [Orkin, 2005; Orkin, 2006]. The success of F.E.A.R. [Ocampo, 2007] was such that it led to a wide diffusion of the use of planning in FPSes [Hillburn, 2013; Humphreys, 2013; Straatman *et al.*, 2013; van der Sterren, 2013]; this diffusion was notably facilitated by the publication the following year of a development kit (SDK) that contained the planner code [Monolith Productions, 2006]. Today, not only is the success of F.E.A.R. still recognized [Horti, 2017], but the biggest productions, reaching millions of players, do not hesitate to use planning and to make it known [Higley, 2015; Conway, 2015; Girard, 2021], and this despite the real time constraints which are more and more demanding.

In 2005, a game engine was running at about 30 frames per second, that is 33 ms for the whole game logic: among graphics, physics and gameplay mechanics, this leaves less than a millisecond for the planner to generate plans for the few characters that called the planner [van der Leeuw, 2009]; ten years later, the move to 60 frames per second has only led to the reduction of the available processing budget for planning. And today, to satisfy the processing budget allocated to the planner, studios explicitly limit the number of calls to at most a few dozen [Champandard *et al.*, 2009; Higley, 2015; Girard, 2021] despite the increase in hardware performance. Furthermore, the game situations are designed in such a way that the number of characters is also limited, thus reducing the number of calls to the planner. However, the dynamics of the video game market pushes productions to simulate increasingly large universes with, for the moment, tens of thousands of characters in sight [Hollister, 2021] and tomorrow millions of them. The results of [Cardon and Jacopin, 2020] suggest that GPUs are a potential solution for such universes in cloud gaming; but what about PCs or game consoles for which GPUs are dedicated to graphics?

The time complexity of planning problems depends on the one hand on language restrictions for representing the planning problem [Bylander, 1991] and on the other hand on which part of the input is fixed [Erol *et al.*, 1995]. From the F.E.A.R. SDK code, we can observe that states are vectors of discrete values and that at certain times, such as fights or routine tasks, the actions are fixed and only the initial and final states are part of the input of the planning problem. Furthermore, the vast majority of the actions are unary [Domshlak and Brafman, 2002], i.e., they change the value for only one state variable. Finally, the actions are post-unique by type (attack, defense, deletion, ...), i.e. one type of actions is the only one to modify a given state variable; this makes it possible to insert in the plan a generic defense operator, for example, and to delay the choice of the type of defense at the time of the plan execution. Answers to a questionnaire filled up by several game AI developers validate that our approach is still up to date: the modeling of planning problems in commercial games is such that SAS with unary and post-unique actions corresponds to the strong NP-Hard SAS-PU class of problems [Bäckström and Nebel, 1995].

Accessing the in-game planning data analyzed in [Jacopin, 2014], we observe that plans in first-person shooters have two domain-specific features: (1) they are totally ordered, (2) they have only one occurrence of a given type of action: e.g., only one action for threatening, reloading, taking cover, dodg-

[*]Contact Author

ing, etc. Therefore, in the next section we present two new SAS restrictions matching "Totally ordered" (T) and "Type set isomorphism" (T_1) which we combine into $\mathbb{T}_1$; we further present a linear time complexity algorithm for the SAS-PU$\mathbb{T}_1$ problem classes. In the following section, we report and discuss tests to illustrate the linear-time properties of our algorithm which is able to generate enough plans for millions of characters while satisfying the processing budget that the game engine allocates to the planner at each frame.

2 SAS-PU$\mathbb{T}_1$ Planning in Linear Time

2.1 Background

We use notations from [Bäckström, 1992a] throughout this paper. The Simplified Action Structure (SAS) and its Extended (SAS^+) version represent *states* with a set $\mathcal{M}$ of m variables each of which can take at most n discrete values or be *undefined*; we note $\mathcal{D}_v$ the set of values of state variable v. Three sets of state variables are used to represent the conditions of an *action*: (1) *postconditions* (post) define new values for some state variables, (2) *preconditions* (pre) requires specific values for all the state variables which values are modified in the postconditions, and (3) *prevail* conditions (prv) requires specific values for some state variables which are not preconditions and therefore are not postconditions either. An action is *applicable* in a state iff its preconditions are consistent with their values in that state; *applying* an action in a state changes the values of all the state variables of its postconditions while prevail conditions remain unchanged. SAS differs from SAS^+ by adding two restrictions which are relevant to our application domain: an action can only change a state variable from one defined value to another defined value (S6), and no *initial* or *goal* state variable can be undefined (S7).

A *plan* is a sequence of actions such that any state resulting from applying an action of the sequence is consistent with the next action in the sequence; a plan solves the *planning problem* made of the initial and goal states $(s_0, s_\star)$ and a set of action types iff (1) any action in the plan is a distinct instance of an *action type* of the planning problem, (2) the first action is applicable in s_0, (3) and applying the last action of the sequence *results* in $s_\star$. We introduce a *first new restriction* which requires that the number of distinct action instances of the same type occurring in a plan is at most $k \in \mathbb{N}$ (T_k).

Three additional restrictions are relevant to our framework: (*Post-uniqueness*) no two distinct action types can change the same state variable to the same value (P), (*Unaryness*) each action type changes the value of exactly one state variable (U), and (*Single-valuedness*) the defined value of any prevail condition required by any two distinct action types must be the same (S).

As a consequence of both **Theorem 4.4** [Bäckström, 1992a, p. 76], which states that any minimal plan solution of a SAS^+-PUS problem instance contains at most two actions of each type, and our new restriction (T_k), any SAS^+-PUS problem instance is also an SAS^+-PUST_2 problem instance.

We eventually require as a *second new restriction* that plans are totally ordered set of distinct action instances (T), and as they correspond to our application domain, we seek to solve SAS-PUTT_1 problem instances which we note SAS-PU$\mathbb{T}_1$. We observe that the problem which consists in repeatedly switching on a series of lights in a tunnel [Bäckström, 1992a, pp. 16-17] and then switching off these lights, belongs to the SAS-PUS$\mathbb{T}_1$ problem class which is a subset of the SAS-PU$\mathbb{T}_1$ problem class; to our knowledge, these problem classes are not new but no specific need had required to spell out the restrictions (T_k) and (T) that lead to them. The tunnel problem is none other than the MultiPrv_2_Cycle problem in section 3.

2.2 Planning confined to topological sorting

SAS^+-PUS [Bäckström, 1992b] is a tractable class of SAS^+ planning problems for which the best algorithm [Bäckström, 1992a, pp. 84-90], which we henceforth note $\mathcal{P}$, runs in $O(m^2n)$ (cf. **Theorem 4.11** [Bäckström, 1992a, p. 89]). In a first phase, $\mathcal{P}$ iterates over each of the m goal state variables to build *chains* of at most n actions towards the initial state thanks to the restriction (P). In a second phase, $\mathcal{P}$ orders couples of $O(mn)$ actions instances occurring in distinct chains thanks to restriction (S) to produce a partially ordered set of actions instances; to achieve this second phase, $\mathcal{P}$ first iterates over $O(mn)$ actions and then over $O(m)$ variables . In a third and final phase, if the partially ordered set of action instances contains no cycle then it is returned as a solution.

Our algorithm, which we henceforth note $\mathbb{P}$, follows the three phases of $\mathcal{P}$ while simplifying the time complexity of the second phase. The key point of $\mathbb{P}$ to achieve linear time complexity is to represent the partially ordered set of action instances through $\mathcal{N}$eighboring actions according to the ordering constraints implied by both restrictions (P) and (U). In fact, enough action ordering can be achieved dynamically thanks to $\mathcal{N}$eighboring actions thus making topological sorting [Cormen *et al.*, 2009] the dominant phase of $\mathbb{P}$.

We now turn to give details on how $\mathbb{P}$ works to solve SAS-PU$\mathbb{T}_1$ problem instances in linear time. The restrictions (P) and (U) allow an action a to be identified by the pair (v_i, p), where p is the postcondition of a on v_i: $v_i \in \mathcal{M}, p \in \mathcal{D}_{v_i}$ st. $post(a)[v_i] = p$. Using an array with (action) index $i \times n + p$, we can retrieve an action instance $a^p_{v_i}$ in $O(1)$ time while planning. With these indexes we initialize two lists of neighbor actions when calling $\mathbb{P}$: $\mathcal{N}_{pre}(a^p_{v_i}) = \{a^q_{v_j} \mid v_j = v_i \wedge q = pre(a^p_{v_i})\}$ which is a singleton due to the restriction (P), and $\mathcal{N}_{prv}(a^p_{v_i}) = \{a^q_{v_j} \mid v_j \neq v_i \wedge q = prv(a^p_{v_i})[v_j]\}$, which will allow $\mathbb{P}$ to only work with defined pre- and prevail conditions thus reducing time complexity.

Actions also have 2-index array which is initially empty, called Next, that is dynamically set such that $\forall a^p_{v_i} \forall a^x_{v_i} \in$ Next$(a^p_{v_i})$, $p = pre(a^x_{v_i})$; Next$(a^p_{v_i})$ has at most 2 values thanks to restriction (T_1). Next$(a^p_{v_i})$ returns the action instance ordered after $a^p_{v_i}$ in the chain for state variable v_i thus allowing to ordering action instances in $O(1)$ time in phase 2. When there are 2 action instances after $a^p_{v_i}$ then Next$(a^p_{v_i})$ has 2 values (hence the 2-index array); this happens when a given value of a state variable occurs both as a precondition of several action instances and in the initial state. The actions of Horse Breeder NPC of the Western problem described in the next section may generate such a case during planning.

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

Input: $v_i \in \mathcal{M}$, $s, g \in \mathcal{D}_{v_i}$; $\mathcal{D}$, the set of yellow action instances; $E_\mathcal{D}$, the set of orderings of yellow action instances; $\mathcal{A}$, the entire set of action instances.
Parameters: x, y, two values of $\mathcal{D}_{v_i}$.
Output: Each action instance a of the chain is yellow, added to $\mathcal{D}$ and $(\mathcal{N}_{pre}(a), a)$ added to $E_\mathcal{D}$.

```
 1: $x \leftarrow \emptyset$; $y \leftarrow \emptyset$
 2: if $a^g_{v_i} \notin \mathcal{A}$ then fail
 3: end if
 4: Color($a^g_{v_i}$) $\leftarrow$ $yellow$; $\mathcal{D} \leftarrow \mathcal{D} \cup \{a^g_{v_i}\}$; $y \leftarrow pre(a^g_{v_i})$
 5: $E_\mathcal{D} \leftarrow E_\mathcal{D} \cup \{(a^y_{v_i}, a^g_{v_i})\}$
 6: Next($a^y_{v_i}$) $\leftarrow$ Next($a^y_{v_i}$) $\cup \{a^g_{v_i}\}$
 7: while $y \neq s$ do
 8:   if $a^y_{v_i} \notin \mathcal{A}$ then fail
 9:   end if
10:   if Color($a^y_{v_i}$) $= yellow$ then fail
11:   end if
12:   Color($a^y_{v_i}$) $\leftarrow$ $yellow$; $\mathcal{D} \leftarrow \mathcal{D} \cup \{a^y_{v_i}\}$
13:   $x \leftarrow y$; $y \leftarrow pre(a^y_{v_i})$
14:   $E_\mathcal{D} \leftarrow E_\mathcal{D} \cup \{(a^y_{v_i}, a^x_{v_i})\}$
15:   Next($a^y_{v_i}$) $\leftarrow$ Next($a^y_{v_i}$) $\cup \{a^x_{v_i}\}$
16: end while
```

Finally, each action instance has 4 different *colors*: (white) the initial color when $\mathbb{P}$ is called, (yellow) the action is useful to solve the problem, (blue) the action is being topologically sorted, (green) the action is topologically sorted and inserted in the solution plan.

When solving SAS-PU$\mathbb{T}_1$ problem instances, $\mathbb{P}$ considers the yellow actions and their pre- and prevail conditions as a graph $G = (\mathcal{D}, E_\mathcal{D})$, with $\mathcal{D}$ as a list of yellow actions and $E_\mathcal{D}$ as their ordering; G is built during both phases 1 and 2, while phase 3 consists of ordering G to provide a totally ordered action plan.

$\mathbb{P}$ backwardly builds a sequence of action instances changing the value of v_i from s_0 to $s_\star$ (cf. Procedure 1 BuildChain). Each action instance in the sequence shall be colored in yellow and then added to $\mathcal{D}$ while only the directed edges between their pre-neighbor and themselves are added to $E_\mathcal{D}$. At the end of phase 2, all the action instances of $\mathcal{D}$ have their prevail conditions satisfied, either by s_0 or by another action instance in $\mathcal{D}$, and $E_\mathcal{D}$ has all the directed edges required for the topological sort (phase 3) to provide a totally ordered action plan.

Not taking single-valuedness into account, $\mathbb{P}$ has to deal with situations where prevail conditions are both satisfied in the initial and the goal states but values in between are now possibly required by the prevail conditions of other action instances. We thus use a boolean matrix, called Flag (cf. Procedure 3), in order to decide which edges to add in such situations.

Theorem 1. *$\mathbb{P}$ is correct and complete.*

Proof. (*Sketch*) If the goal state is different from the initial state, there must exist actions that solve the problem instance; $\mathbb{P}$ shall find these actions thanks to their indexes or yields a failure otherwise. According to the above details $\mathbb{P}$ builds G

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

Input: $a^p_{v_i}$, a yellow action instance of index v_i and postcondition $p \in \mathcal{D}_{v_i}$; $s_0 \in \mathcal{S}$, the initial state; P, the solution plan.
Output: $a^p_{v_i}$ is colored green after all its neighbors have been considered; it is then enqueued at the tail of P.

```
 1: Color($a^p_{v_i}$) $\leftarrow$ $blue$
 2: for $a^q_{v_j} \in E_\mathcal{D}(a^p_{v_i})$ do
 3:   if $a^q_{v_j} \notin \mathcal{N}_{pre}(a^p_{v_i}) \vee pre(a^p_{v_i}) \neq s_0[v_i] \vee$
      $a^p_{v_i} \neq$ Last(Next($a^q_{v_j}$)) then
 4:     if Color($a^q_{v_j}$) $= blue$ then fail {Cycle detected.}
 5:     end if
 6:     if Color($a^q_{v_j}$) $= yellow$ then
 7:       DFSTopo($a^q_{v_j}, \mathcal{D}, E_\mathcal{D}, s_0, \mathtt{P}$)
 8:     end if
 9:   end if
10: end for
11: Color($a^p_{v_i}$) $\leftarrow$ $green$; P $\leftarrow$ P + $\{a^p_{v_i}\}$;
```

during both phase 1 and phase 2, ordering (demoting) an action instance after others ($(\mathcal{N}_{pre}(a), a)$ and $(a^q_{v_j}, a^p_{v_i})$ edges) or ordering (promoting) an action instance before another ($(a^p_{v_i}$,Next$(a^q_{v_j})$ edges) [Chapman, 1987] to remove threats on pre- and prevail conditions. Consequently, the depth-first search topological sort shall return a totally ordered plan with no threat on all these conditions making the successive application of the totally ordered actions instances eventually result in the goal state of the problem. □

Theorem 2. *$\mathbb{P}$ has a worst case time complexity of $O(|\mathcal{A}| + |E_\mathcal{A}|)$.*

Proof. Let $\mathcal{N}_{demo}(a^p_{v_i}) = \{a^q_{v_j} \mid v_i \neq v_j \wedge a^q_{v_j} \notin \mathcal{N}_{prv}(a^p_{v_i}) \wedge \exists a^r_{v_i} \in \mathcal{N}_{pre}(a^p_{v_i}) \cup \mathcal{N}_{prv}(a^q_{v_j})\}$ which stores indexes of all action instances whose prevail condition for state variable v_i is threatened by $a^p_{v_i}$; therefore, $a^p_{v_i}$ is ordered after all actions in $\mathcal{N}_{demo}(a^p_{v_i})$. We have $E_\mathcal{A} = \{(b, a) | a \in \mathcal{A} \wedge b \in \mathcal{N}_{pre}(a) \cup \mathcal{N}_{prv}(a) \cup \mathcal{N}_{demo}(a)\}$ with $\mathcal{N}_{prv}$ indexes allowing $\mathbb{P}$ to only navigate through the defined prevail conditions of the action instances of $\mathcal{D}$. The phase 1 has at most $|\mathcal{A}| = O(mn)$ steps: m steps via the for loop (1.2) times n steps via the *BuildChain* procedure. Due to restriction (T$_1$), $\mathcal{D}$ cannot be greater than $\mathcal{A}$, so the for loop in 3.10 requires at most $|\mathcal{A}| = O(mn)$ steps. The two for loops in 3.10 and in 3.11 therefore require $|\mathcal{A}| + |E_{prv}|$ steps with $E_{prv} = \{(b, a) | a \in \mathcal{A} \wedge b \in \mathcal{N}_{prv}(a)\}$. The phase 3 topologically sorts the yellow actions of the graph G in $O(|\mathcal{D}| + |E_\mathcal{D}|)$; in the worst case this is equivalent to $O(|\mathcal{A}| + |E_\mathcal{A}|)$ which in turns dominates the whole and eventually proves the linear time complexity of $\mathbb{P}$. □

Theorem 3. *$\mathbb{P}$ requires at most $O(m^2 n)$ space.*

Proof. A SAS-PUT$_1$ action takes $O(m)$ space: the pre- and postcondition can be reduced to one variable each, plus a variable for the index of the state variable affected, and the set $\mathcal{N}_{pre}$ is a singleton due to restrictions (P) and (U); the prevail conditions, on the contrary, are lists of m elements, and

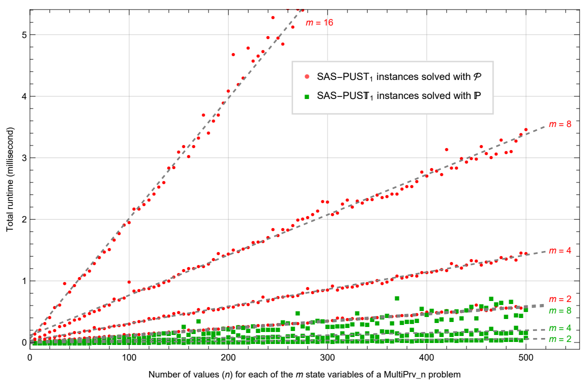

Figure 1: $\mathbb{P}$ and $\mathcal{P}$ runtimes are linear in the number of values (n) per state variable (m) although the MultiPrv_n problem is designed to generate $O(m^2)$ orders between mn actions.

the set $\mathcal{N}_{prv}$ has at most m elements. The matrix of action instances, which stores action indexes, takes $O(mn)$ space. Finally, $\mathbb{P}$ creates a graph $G = (\mathcal{D}, E_{\mathcal{D}})$ which takes at most $(m^2 n)$ space: $|\mathcal{D}| \leq |\mathcal{A}| \leq mn$ due to the restriction (T$_1$) and each action instance can have at most $2(m-1)$ neighbors: at most $(m-1)$ neighbors via $\mathcal{N}_{prv}$ + at most $(m-1)$ neighbors via $\mathcal{N}_{demo}$. □

3 Benchmarking $\mathbb{P}$ vs $\mathcal{P}$

We here describe the three problems that the we use in this paper to illustrate the details of the time complexities of both $\mathbb{P}$ and $\mathcal{P}$. In the Figures 1,2, and 3, green squares, resp. red disks, represent problem instances solved by $\mathbb{P}$, resp. $\mathcal{P}$. Tests were performed on the following configuration: AMD Ryzen 2700X (8-Core) CPU (3.7GHz), 32Gb of RAM and Windows 10 (64 bits); both planners are written in C++14 with default settings for Microsoft Visual Studio 2019. The supplementary materials of this paper contain all the files that are necessary to compile and run the tests, including the project file.

MultiPrv_n is designed to check that the runtime of both planners is linear in (n) (cf. Figure 1); it generates $O(m^2)$ orders between mn actions with $0 < p < n$; $\forall v_i \in \mathcal{M}$, we have:

- $pre(a^p_{v_i}) = p - 1$ and $post(a^p_{v_i}) = p$,
- $prv(a^p_{v_i})[v_j] = 1$, for $i < j \leq m$,
- $prv(a^p_{v_i})[v_j] = u$, for $1 \leq j \leq i$.

[Bäckström, 1992a] reports a Lisp implementation of $\mathcal{P}$ with runtimes superlinear in (n), on the contrary to theoretical results (cf. **Theorem 4.10**, p. 89); because, among other things, Lisp did not provide control over memory management, it was suspected to alter the practical results. We implemented both $\mathbb{P}$ and $\mathcal{P}$ in C++ with no specific optimization but we indeed had to carefully manage memory to achieve runtimes linear in (n) for both $\mathcal{P}$ and $\mathbb{P}$.

MultiPrv_2_Cycle is a specific case of MultiPrv_n such that $0 \leq p < n = 2$: an action type can set each state variable to their smallest value, thus introducing a cycle in these values; it is nothing more than the tunnel problem we presented

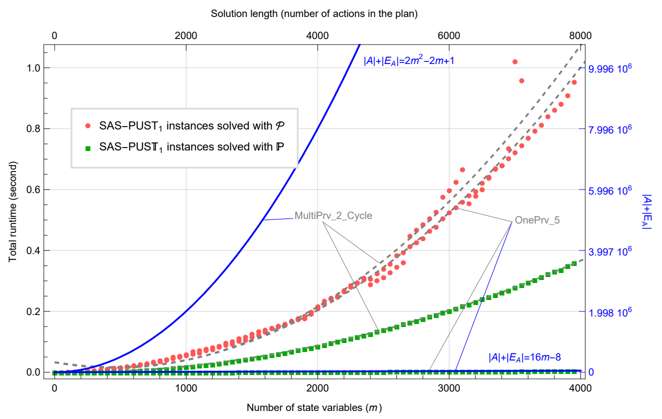

Figure 2: When the number of orderings is linear (resp. quadractic) in the number of state variables (m) then $\mathbb{P}$ runtimes are linear (resp. quadratic) in the number of state variables (m) which illustrates the $O(|A| + |E_A|)$ time complexity, whereas $\mathcal{P}$ runtimes are always quadratic in the number of state variables (m) despite the differences in the number of orderings between MultiPrv_2_Cycle and OnePrv_5.

in section 2.1. As it generates $O(m^2)$ orders between $2m$ actions, we can easily scale this problem to show that worst case the runtime of $\mathbb{P}$ is quadratic in the number of variables m (cf. Figure 2).

OnePrv_5 is designed to show that the runtime of $\mathbb{P}$ can be linear in the number of variables (m) while that of $\mathcal{P}$ is quadratic (cf. Figure 2) despite the $O(m)$ number of action orderings to process. OnePrv_5 generates $O(m)$ orders between $4m$ actions with $0 < p < n = 5$; $\forall v_i \in \mathcal{M}$, we have:

- $pre(a^p_{v_i}) = p - 1$ and $post(a^p_{v_i}) = p$,
- $prv(a^p_{v_i})[v_{i+1}] = \lfloor (n = 5)/2 \rfloor = 2$, with $v_i \neq v_m$,
- $prv(a^p_{v_i})[v_j] = u, \forall v_j \in \mathcal{M} \setminus \{v_{i+1}\}$.

$\mathcal{A}$	Pre	Post	Prevail	Description
$a^0_{v_0}$	$v_0 = 1$	$v_0 = 0$	$\langle u, 0, u \rangle$	Store a haystack
$a^1_{v_0}$	$v_0 = 0$	$v_0 = 1$	$\langle u, 0, u \rangle$	Take a haystack
$a^2_{v_0}$	$v_0 = 1$	$v_0 = 2$	$\langle u, 0, u \rangle$	Fill the horse feeder
$a^0_{v_1}$	$v_1 = 1$	$v_1 = 0$	$\langle 0, u, u \rangle$	Drop the bucket
$a^1_{v_1}$	$v_1 = 0$	$v_1 = 1$	$\langle 0, u, u \rangle$	Pick up the bucket
$a^1_{v_2}$	$v_2 = 0$	$v_2 = 1$	$\langle u, 1, u \rangle$	Fill with water
$a^2_{v_2}$	$v_2 = 1$	$v_2 = 2$	$\langle u, 1, u \rangle$	Fill the horse trough

Table 1: Actions for the Horse Breeder NPC whose goal is to feed horses: $s_\star = \langle 2, 0, 2 \rangle$ where v_0 represents a *HayStack* with $\mathcal{D}_{v_0} = \{stored(0),\ inHands(1),\ inFeeder(2)\}$, v_1 represents a *Bucket* with $\mathcal{D}_{v_1} = \{onFloor(0),\ inHands(1)\}$, and v_2 represents *Water* with $\mathcal{D}_{v_2} = \{inFountain(0),\ inBucket(1),\ inTrough(2)\}$. These actions are both post-unique (P) and unary (U); however, prevail conditions for the state variable v_1 requires 2 values ($v_1 = 0$ and $v_1 = 1$) which violates the single-valuedness restriction (S).

Western is designed to evaluate the scalability of both $\mathbb{P}$ and $\mathcal{P}$ for our video-games application domain. It implements daily routines of NPCs [DefendTheHouse, 2018] in

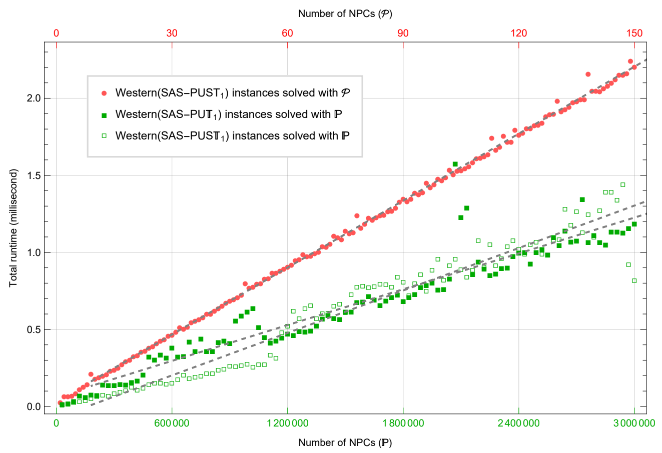

Figure 3: Solution plans for instances of the Western problem are 3 to 10-action long: the impact of the number of actions and the number of their orderings is negligible on the generation of each plan. Consequently, both $\mathbb{P}$ and $\mathcal{P}$ runtimes are linear in the number of NPCs. However, $\mathbb{P}$ builds each plan dramatically faster than $\mathcal{P}$ and is thus able to scale up to generating plans for two millions NPCs in less than 1 millisecond.

the western setting of the very successful [Take Two Interactive, 2021] commercial video-game Red Dead Redemption 2 [Rockstar Studios, 2018]. It is as close as possible to the game and is of class SAS-PU$\mathbb{T}_1$ which $\mathcal{P}$ cannot handle, with $m = 29$, $n = 5$, $|\mathcal{A}| = 58$, and 7 NPC classes with 9 specific goals. This domain is fully described in the supplementary materials of this paper; we only detail the Horse Breeder NPC action types in Table 1 which are both post-unique (P) and unary (U) but not single-valued (S). The Horse Breeder is the only NPC that can cause the use of the 2 indexes for Next as described in subsection 2.2. This situation occurs if $s_\star[v_0] = 2$, $s_0[v_0] = 1$, and the value $v_0 = 0$ is required by other actions as a prevail condition while planning, which is the case for $a^1_{v_1}$, for example. In this situation, BuildChain on v_0 during phase 1 will add $a^2_{v_0}$ to Next$(a^1_{v_0})$. During phase 2, if $a^1_{v_1}$ is in $\mathcal{D}$, the chain of actions from $s_0[v_0]$ to $prv(a^1_{v_1})[v_0]$ will also add $a^0_{v_0}$ to Next$(a^1_{v_0})$. We designed a restricted SAS-PUST$_1$ version of Western which both $\mathcal{P}$ and $\mathbb{P}$ can handle, with $m = 19$, $n = 2$, $|\mathcal{A}| = 37$, and 5 NPC Classes with 6 specific goals. Figure 3 shows that $\mathbb{P}$ is twenty thousand times faster than $\mathcal{P}$ and can generate plans for about two millions NPCs in 1 millisecond: planning can be used to control large cities in commercial video-games [Hollister, 2021].

4 Discussion

As we mentioned in subsection 2.1, $\mathcal{P}$ solves SAS$^+$-PUST$_2$ problem instances as a consequence of **Theorem 4.4** [Bäckström, 1992a, p. 76]. Consequently we could expect $\mathbb{P}$ to be about two times faster than $\mathcal{P}$ on specifically designed problems by avoiding to consider the insertion of two action instances rather than one. $\mathbb{P}$ obviously does less work than $\mathcal{P}$, but what kind of work? Restriction (T$_1$) alone certainly does not explain why $\mathbb{P}$ is several orders of magnitude faster than $\mathcal{P}$. Figure 2 shows that $\mathcal{P}$ does not take advantage of the linear growth of action orderings as it iterates over state variables and not over orderings of action instances as $\mathbb{P}$ does. In phase 2 of algorithm 3, $\mathbb{P}$ iterates from one prevail condition neighboring of a (yellow) action instance to another prevail condition neighboring so that the topological sorting visits only necessary orderings. This is key to explain the very high runtime efficiency of $\mathbb{P}$ for all the problems we tested. We eventually observed that the topological sorting of our graphs is deterministic as it is for the planner in [Domshlak and Brafman, 2002] which uses unary actions without single-valuedness but with binary state variables[1]. Minor key explanations are about allocating and filling data structures as much in advance as possible; however, a rigorous memory management only explains the regularity of the curves (both $\mathbb{P}$ and $\mathcal{P}$) in Figures 1, 2, and 3.

On the contrary to SAS$^+$ restrictions, the restriction (T-1) does not restrict the input of the planning algorithm but its output. In video-gaming application domain such as Western plans are less than 10-action long: it is quite simple to check, even by hand, that a problem instance satisfies the T-1 restriction; it is much more complex when plan contain thousands of actions. In the state of this paper, the only solution is to run $\mathcal{P}$ as membership testing of the problem class. $\mathcal{P}$ runtimes are fortunately not a hindrance for this purpose.

The restriction (T) also restricts the output of the planning algorithm; however, searching for totally ordered plans is always possible. A plan is a solution to a planning problem when its application transforms the initial state into the goal state of the problem: each action is applied one by one to successive situations and thus the application of a plan corresponds to a total order on its actions. Therefore, the restriction (T) can be applied to any SAS$^+$ class of problems, which include any SAS class of problems. It would make sense to limit the use of the restriction (T) to classes of problems such as MultiPrv_2_Cycle in which all solutions are totally ordered plans of actions. However, it also makes sense to use the restriction (T) to specify that our goal is to generate totally ordered plans of actions, as it is currently the case in commercial video-games where NPCs perform one task after another.

The processing budget is probably the only constraint which would prevent a game engine from accessing the current game state and thus reading the exact value of game state variables. Consequently, totally defined initial and goal states make a realistic restriction in our video-game application domain. Game engines impose an update rate for game state variables, however, which may cause out of date values to be part of planning problems. This rudimentary form of uncertainty [Bäckström, 1992a, p. 64] should definitively be part of future works on scaling planning to manage large virtual worlds.

A more subtle use of undefined values has to do with restriction (S2) of the SAS$^+$ framework which is relaxed in the SAS framework (cf. **Definition 3.2** [Bäckström, 1992a, p. 52]). The restriction (S2) enables an action type to define a state variable that was previously undefined: the postcondi-

[1]Binary variables are also one of the restriction (B) of the SAS$^+$ planning framework (cf. **Definition 3.9** [Bäckström, 1992a, p. 62]).

tion of this action can define a state variable that is undefined in the precondition. This is a rudimentary solution to deal with the update rate of game state variables: we can design an action type whose purpose is to make sure that a given state variable has a value; when executing this action while playing, the game engine waits until the next update of this state variable. However, we have not investigating restriction (S2) any further as our main objective is now to design $\mathbb{P}^+$ to solve SAS$^+$-PU$\mathbb{T}_1$ problem instances.

5 Conclusion

Our objective was to generate totally ordered SAS-PU plans so that no two action instances have the same action type for millions of NPC in real-time. To this end, we made the following key contributions to the SAS framework:

- We defined two new restrictions: (1) the restriction (T) which limits solutions to totally ordered plans, and (2) the restriction (T_1) which limits the number of action instance to one in any solution; we noted $\mathcal{T}_1$ the combination of these two restrictions.
- We designed an algorithm, which we noted $\mathbb{P}$, to solve SAS-PU$\mathcal{T}_1$ problem instances, thus relaxing the single-valuedness (S) restriction.
- We designed several SAS-PU$\mathcal{T}_1$ problems to test various features and in particular various time complexities as our objective is planning in real-time; in particular, the MultiPrv_X_Cycle generalizes the tunnel example. We also designed a realistic Western domain with respect to our video-gaming application domain.
- The worst case time complexity of our algorithm is linear in the number of action instances plus their orderings; the runtimes of our implementation of $\mathbb{P}$ for the Western domain scales to two millions NPCs in about one millisecond thus achieving our objective.

Future work will first take into account partial initial and final states; then we wish to address SAS$^+$-PU$\mathcal{T}_2$: is it possible to design an algorithm as efficient as $\mathbb{P}$ for this class of problems?

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

Input: $\mathcal{M}$; $\mathcal{A}$; s_0, $s_\star$: totally defined initial and goal states.
Parameters: $\mathcal{D}$, the set of yellow action instances; $E_\mathcal{D}$, the set of orderings of yellow action instances; Flag: $m \times m$ boolean matrix with all entries set to false.
Output: P: a totally ordered action plan that link s_0 to $s_\star$; yields a failure if the instance is not solvable.

```
1:  P ← $\emptyset$; $\mathcal{D} \leftarrow \emptyset$; $E_\mathcal{D} \leftarrow \emptyset$
2:  for $v_i \in \mathcal{M}$ do {Phase 1}
3:    if $s_0[v_i] \neq s_\star[v_i]$ then
4:      BuildChain($v_i, s_0[v_i], s_\star[v_i], \mathcal{D}, E_\mathcal{D}, \mathcal{A}$)
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 $a^p_{v_i} \in \mathcal{D}$ do {Phase 2}
10:   for $a^q_{v_j} \in \mathcal{N}_{prv}(a^p_{v_i})$ do
11:     if $q \neq s_\star[v_j]$ then
12:       if First(Next($a^q_{v_j}$)) $= \emptyset$ then
13:         BuildChain($v_j, q, s_0[v_j], \mathcal{D}, E_\mathcal{D}, \mathcal{A}$)
14:       end if
15:       if $prv$(First(Next($a^q_{v_j}$)))$[v_i] \neq p$ then
16:         $E_\mathcal{D} \leftarrow E_\mathcal{D} \cup \{(a^p_{v_i}$,First(Next($a^q_{v_j}$)))$\}$
17:       end if
18:     end if
19:     if $q \neq s_0[v_j]$ then
20:       if $a^q_{v_j} \notin \mathcal{A}$ then fail
21:       end if
22:       if Color($a^q_{v_j}$) $= white$ then
23:         BuildChain($v_j, s_0[v_j], q, \mathcal{D}, E_\mathcal{D}, \mathcal{A}$)
24:       end if
25:       $E_\mathcal{D} \leftarrow E_\mathcal{D} \cup \{(a^q_{v_j}, a^p_{v_i})\}$
26:       Flag$[v_i][v_j] \leftarrow true$
27:     else
28:       if ¬Flag$[v_i][v_j] \wedge prv(a^q_{v_j})[v_i] \neq p$ then
29:         $E_\mathcal{D} \leftarrow E_\mathcal{D} \cup \{(a^q_{v_j}, a^p_{v_i})\}$
30:       else
31:         if Flag$[v_i][v_j] \wedge prv$(Last(Next($a^q_{v_j}$)))$[v_i] \neq p$
            then
32:           $E_\mathcal{D} \leftarrow E_\mathcal{D} \cup \{(a^p_{v_i}$,Last(Next($a^q_{v_j}$)))$\}$
33:         end if
34:       end if
35:     end if
36:   end for
37: end for
38: for $a \in \mathcal{D}$ do {Phase 3}
39:   if Color($a$) $= yellow$ then
40:     DFSTopo($a, \mathcal{D}, E_\mathcal{D}, s_0$, P)
41:   end if
42: end for
43: return P
```

References

[Bäckström and Nebel, 1995] Christer Bäckström and Bernhard Nebel. Complexity results for SAS planning. *Computational Intelligence*, 11(4):625–655, 1995.

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

[Bäckström, 1992b] Christer Bäckström. Equivalence and tractability results for SAS$^+$ planning. In *Proceedings of* 3^{rd} *Conference on the Principles of Knowledge Representation and Reasoning*, pages 126–137. Morgan Kaufmann, october 1992.

[Bylander, 1991] Tom Bylander. Complexity results for planning. In *Proceedings of* 12^{th} *IJCAI*, pages 274–279, August 1991.

[Cardon and Jacopin, 2020] Stéphane Cardon and Éric Jacopin. Binary GPU-planning for thousands of NPCs. In *IEEE Conference on Games*, pages 678–681. IEEE Press, August 2020.

[Champandard *et al.*, 2009] Alex Champandard, Tim Verweij, and Remco Straatman. Killzone 2 multiplayer bots. In *Paris Game AI Conference*. AIGameDev, https://www.guerrilla-games.com/read/killzone-2-multiplayer-bots (accessed on December 1^{st}, 2021), June 2009.

[Chapman, 1987] David Chapman. Planning for conjunctive goals. *Artificial intelligence*, 32(3):333–377, 1987.

[Conway, 2015] Chris Conway. GOAP in Tomb Raider. GDC AI Summit, March 2015.

[Cormen *et al.*, 2009] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[DefendTheHouse, 2018] DefendTheHouse. NPC Daily Life in Read Dead Redemption 2. https://www.youtube.com/watch?v=MrUJJgppMn4, November 2018. Accessed on january 10^{th}, 2022.

[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.

[Erol *et al.*, 1995] Kutluhan Erol, Dana Nau, and V.S. Subrahmanian. Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence*, 76(1-2):75–88, 1995.

[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-FenyxRising-, November 2021. Accessed on december 1st 2021.

[Higley, 2015] Peter Higley. GOAP at monolith productions. GDC AI Summit, March 2015.

[Hillburn, 2013] Daniel Hillburn. *Simulating Behavior Trees – A Behavior Tree/Hybrid Planner Approach*, volume 1, chapter 8, pages 99–111. CRC Press, 2013.

[Hollister, 2021] Sean Hollister. The matrix awakens didn't blow my mind, but it convinced me next-gen gaming is nigh. https://www.theverge.com/22828860/the-matrix-awakens-ps5-xbox-series-x-free-next-gen, december 2021. Accessed on January 12^{th}, 2022.

[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 December 3^{rd}, 2021.

[Humphreys, 2013] Troy Humphreys. *Exploring HTN Planners through Examples*, volume 1, chapter 12, pages 149–167. CRC Press, 2013.

[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.

[Monolith Productions, 2006] Monolith Productions. F.E.A.R. public tools, June 2006.

[Ocampo, 2007] Jason Ocampo. F.E.A.R. review. https://www.gamespot.com/reviews/fear-review/1900-6169771/, April 2007. Accessed on December 3^{rd}, 2021.

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

[Orkin, 2006] Jeff Orkin. Three States and a Plan: The A.I. of F.E.A.R. In *Proceedings of the Game Developer Conference*, page 17 pages, 2006.

[Rockstar Studios, 2018] Rockstar Studios. Red Dead Redemption 2, November 2018.

[Straatman *et al.*, 2013] Remco Straatman, Tim Verweij, Alex Champandard, Robert Morcus, and Hylke Kleve. *Hierarchical AI for Multiplayer Bots in Killzone 3*, chapter 29, pages 377–390. CRC Press, 2013.

[Take Two Interactive, 2021] Take Two Interactive. SEC Filing. https://ir.take2games.com/node/27706/html, July 2021. Accessed on January 13^{th}, 2022.

[van der Leeuw, 2009] Michiel van der Leeuw. The PS3's SPU in the real world – a Killzone 2 case study. Game Developer Conference, March 2009.

[van der Sterren, 2013] William van der Sterren. *Hierarchical Plan-Space Planning for Multi-Unit Combat Maneuvers*, volume 1, chapter 13, pages 169–183. CRC Press, 2013.