La planification SAS sous forme de tri topologique

G. Prévost[1], S. Cardon[1], T. Cazenave[2], C. Guettier[3], É. Jacopin[1]

[1] Centre de Recherche de l'Académie Militaire de Saint-Cyr Coëtquidan, CReC
[2] Université Paris-Dauphine - PSL, LAMSADE
[3] SAFRAN Electronics and Defense

Résumé

Nous abordons la question de comment fournir en temps-réel un plan d'actions pour plusieurs millions de personnages non joueurs (PNJ) dans les mondes virtuels. Suite à une étude approfondie des plans générés par des jeux vidéo commerciaux utilisant la planification, nous présentons une nouvelle classe de problèmes au domaine de planification Structure d'Action Simplifiée (SAS) applicable à ces jeux vidéo. Nous présentons également un nouvel algorithme de complexité linéaire pour cette classe de problèmes qui, contrairement aux autres planificateurs SAS, nous permet effectivement de considérer la gestion de millions de PNJs par image.

Mots-clés

Intelligence Artificielle, planification, temps-réel, actions, SAS, tri topologique.

Abstract

We address the question of how to provide an action plan in real-time to several million non-player characters (NPCs) in virtual worlds. After a close study of the plans generated in commercial video games using planning, we present a new class of problems in the Simplified Action Structure (SAS) planning framework applicable to these video games. We also present a new linear-time algorithm for this class of problems which, unlike previous SAS planners, effectively allows us to consider the management of millions of NPCs per frame.

Keywords

Artificial Intelligence, planning, real-time, actions, SAS, topological sort.

1 Introduction

F.E.A.R. , un jeu de tir à la première personne (FPS) sorti en 2005, a été le premier jeu vidéo à utiliser la planification pour générer les comportements des personnages en temps réel [21, 22]. Le succès de F.E.A.R. [20] a été tel qu'il a conduit à une large diffusion de l'utilisation de la planification dans les FPS [14, 17, 24, 27] ; cette diffusion a notamment été facilitée par la publication l'année suivante d'un kit de développement (SDK) contenant le code du planificateur [19]. Aujourd'hui, non seulement le succès de F.E.A.R. est toujours reconnu [16], mais les plus grandes productions, touchant des millions de joueurs, n'hésitent pas à utiliser la planification et à la faire connaître [13, 8, 12], et ce malgré les contraintes de temps réel de plus en plus exigeantes.

En 2005, un moteur de jeu affichait environ 30 images par seconde, soit 33 ms pour toute la logique du jeu : entre les graphismes, la physique et les mécaniques de gameplay, cela laissait moins d'une milliseconde au planificateur pour générer des plans pour les quelques personnages qui l'appelaient [26] ; dix ans plus tard, le passage à 60 images par seconde n'a fait que réduire le budget de traitement disponible pour la planification. Aujourd'hui, pour satisfaire le budget alloué au planificateur, les studios limitent explicitement le nombre d'appels à quelques dizaines tout au plus [6, 13, 12] malgré l'augmentation des performances matérielles. De plus, les situations de jeu sont conçues de manière à ce que le nombre de personnages soit également limité, réduisant ainsi le nombre d'appels au planificateur. Cependant, la dynamique du marché du jeu vidéo pousse les productions à simuler des univers de plus en plus vastes avec, pour l'instant, des dizaines de milliers de personnages en vue [15] et demain des millions, en particuliers pour les jeux massivement multijoueurs. Les résultats de [5] suggèrent que les GPU sont une solution potentielle pour de tels univers dans le cloud gaming ; mais qu'en est-il des PC ou des consoles de jeu pour lesquels les GPU sont dédiés au graphisme ?

La complexité temporelle des problèmes de planification dépend d'une part des restrictions du langage pour représenter le problème de planification [4] et d'autre part de la partie de l'entrée qui est fixée [11]. À partir du code dans le SDK de F.E.A.R., nous pouvons observer que les états sont des vecteurs de valeurs discrètes et qu'à certains moments, comme les combats ou les tâches routinières, les actions sont fixées et seuls les états initiaux et finaux font partie de l'entrée du problème de planification. De plus, la grande majorité des actions sont unaires [10], c'est-à-dire qu'elles ne changent la valeur que d'une seule variable d'état. Enfin, les actions sont post-uniques par type (attaque, défense, inspection, ...), c'est-à-dire qu'un type d'actions est le seul à modifier une variable d'état donnée ; ceci permet d'insérer dans le plan un opérateur de défense générique, par exemple, et de retarder le choix du type de défense au moment de l'exécution du plan. Les réponses à un questionnaire rempli par plusieurs développeurs d'IA de jeux corro-

borent que notre approche est toujours d'actualité : la modélisation des problèmes de planification dans les jeux commerciaux correspond au domaine SAS avec actions unaires et post-uniques. Ces jeux appartiennent ainsi à la classe de problèmes SAS-PU [3] qui est NP-Hard.

L'analyse des données de planification récoltées en jeu par [18] nous montre que les plans dans les jeux de tir à la première personne ont deux caractéristiques spécifiques au domaine : (1) ils sont totalement ordonnés, (2) ils n'ont qu'une seule occurrence d'un type d'action donné : par exemple, une seule action pour menacer, recharger, se mettre à couvert, esquiver, etc. Par conséquent, dans la section suivante, nous présentons deux nouvelles restrictions SAS correspondant à "Totalement ordonné" (T) et à "isomorphisme de type" (T_1) que nous combinons en $\mathbb{T}_1$; nous présentons également un algorithme de complexité en temps linéaire pour la classe de problèmes SAS-PU$\mathbb{T}_1$. Dans la section suivante, nous présentons et discutons des tests visant à illustrer les propriétés linéaires de notre algorithme qui est capable de générer suffisamment de plans pour des millions de personnages tout en respectant le budget de traitement que le moteur de jeu alloue au planificateur à chaque image.

2 SAS-PU$\mathbb{T}_1$ en temps linéaire

2.1 Contexte

Nous utilisons les notations de [1] tout au long de ce document. La structure d'action simplifiée (SAS) et sa version étendue (SAS$^+$) représentent des *états* avec un ensemble $\mathcal{M}$ de m variables dont chacune peut prendre au plus n valeurs discrètes ou être *indéfinie* [1] ; nous notons $\mathcal{D}_v$ l'ensemble des valeurs de la variable d'état v. Trois ensembles de variables d'état sont utilisés dans SAS pour représenter les conditions d'une *action* : (1) les *post-conditions* (post) définissent de nouvelles valeurs pour certaines variables d'état, (2) les *pré-conditions* (pre) définissent les valeurs des variables d'état avant leur modification par les post-conditions, et (3) les *prevail-conditions* (prv) sont des pré-conditions qui ne seront pas modifiées par les post-conditions. Une action est *applicable* dans un état si ses pré- et prevail-conditions sont toutes satisfaites dans cet état ; *appliquer* une action dans un état change les valeurs de toutes les variables d'état définies dans les pré-conditions par celles définies dans les post-conditions, les prevail-conditions quant à elles restent inchangées. SAS diffère de SAS$^+$ par l'ajout de deux restrictions pertinentes pour notre domaine d'application : une action ne peut changer une variable d'état que d'une valeur définie à une autre valeur définie (S6), et aucune variable d'état *initial* ou *but* ne peut être indéfinie (S7).

Un *plan* est une séquence d'actions telle que tout état résultant de l'application d'une action de la séquence est cohérent avec l'action suivante de la séquence ; un plan résout l'*instance d'un problème de planification*, constitué des états initial et but $(s_0, s_\star)$ et d'un ensemble de types d'action, si : (1) toute action du plan est une instance distincte d'un *type d'action* du problème de planification, (2) la première action est applicable dans s_0, et (3) l'application de la dernière action de la séquence *conduit* à $s_\star$. Nous introduisons une *première nouvelle restriction* qui exige que le nombre d'instances d'actions distinctes du même type apparaissant dans un plan soit au maximum $k \in \mathbb{N}$ (T_k).

Trois restrictions supplémentaires sont pertinentes pour notre étude : (*Post-unicité*) deux types d'action distincts ne peuvent pas changer la même variable d'état à la même valeur (P), (*Unaire*) chaque type d'action change exactement la valeur d'une variable d'état (U), et (*Single-valuedness*) si plusieurs types d'action ont la même prevail-condition de définie alors la valeur sur cette prevail-condition doit être la même pour tous ces types d'action (S). Comme conséquence à la fois du **théorème 4.4** [1, p. 76], qui stipule que toute solution de plan minimal d'une instance de problème SAS$^+$-PUS contient au plus deux actions de chaque type, et de notre nouvelle restriction (T_k), toute instance de problème SAS$^+$-PUS est également une instance de problème SAS$^+$-PUST$_2$.

Nous imposons finalement comme *seconde nouvelle restriction* que les plans soient totalement ordonnés (T), et comme ils correspondent à notre domaine d'application, nous cherchons à résoudre les instances de problème SAS-PUTT$_1$ que nous notons SAS-PU$\mathbb{T}_1$. Nous observons que le problème qui consiste à allumer de façon répétée une série de lumières dans un tunnel [1, pp. 16-17] puis à éteindre ces lumières, appartient à la classe de problèmes SAS-PUS$\mathbb{T}_1$ qui est un sous-ensemble de la classe de problèmes SAS-PU$\mathbb{T}_1$; à notre connaissance, ces classes de problèmes ne sont pas nouvelles mais il n'était pas nécessaire d'expliciter les restrictions (T_k) et (T) qui y conduisent. La restriction (T) également n'est pas totalement nouvelle [3] mais elle n'a jamais été explicitement énoncée. Le problème du tunnel n'est autre que le problème MultiPrv_2_Cycle de la section 3. Enfin, comme nos plans solutions sont T_1, nous ne ferons plus la différence en une action et un type d'action par la suite.

2.2 Exemple d'un problème SAS-PU$\mathbb{T}_1$

Avant de présenter un problème concret de la classe SAS-PU$\mathbb{T}_1$, l'`Éleveur de Chevaux` (Tab. 1), nous présentons des notions et notations utiles pour la suite. Premièrement, du fait de la restriction (P) et (U), chaque action a est identifiable par la paire (v_i, p), où p est la postcondition de a sur v_i : $v_i \in \mathcal{M}, p \in \mathcal{D}_{v_i}$ tel que $post(a)[v_i] = p$. Nous dénotons dorénavant l'*identifiant* de chaque action (PU) avec le format $a^p_{v_i}$. De plus, $post(a^p_{v_i}) \equiv post(a^p_{v_i})[v_i]$ et $pre(a^p_{v_i}) \equiv pre(a^p_{v_i})[v_i]$. Ces identifiants nous permettent de définir trois ensemble de *prédécesseurs* pour une action $a^p_{v_i}$: (1) $\mathcal{N}_{pre}(a^p_{v_i}) = \{a^q_{v_j} \mid v_j = v_i \wedge q = pre(a^p_{v_i})\}$, l'ensemble des prédécesseurs de $a^p_{v_i}$ par *dépendance post-pré*. Du fait de (P), cet ensemble est un singleton. (2) $\mathcal{N}_{prv}(a^p_{v_i}) = \{a^q_{v_j} \mid v_j \neq v_i \wedge q = prv(a^p_{v_i})[v_j]\}$, l'ensemble des prédécesseurs de $a^p_{v_i}$ par *dépendance post-prv*. (3) $\mathcal{N}_m(a^p_{v_i}) = \{a^q_{v_j} \mid v_i \neq v_j \wedge a^q_{v_j} \notin \mathcal{N}_{prv}(a^p_{v_i}) \wedge pre(a^p_{v_i}) = prv(a^q_{v_j})[v_i]\}$, l'ensemble des actions dont la

1. Les valeurs indéfinies sont notées u pour *undefined*.

prevail-condition sur v_i est *menacée* par $a^p_{v_i}$. Les ensembles $\mathcal{N}_{pre}$ et $\mathcal{N}_{prv}$ sont intuitifs à comprendre, $\mathcal{N}_m$ en revanche découle de la deuxième condition [2] du *Modal Truth Criterion* (MTC) développé par David Chapman [7, p.340]. Soit $a^p_{v_i}, a^q_{v_j} \in \mathcal{A}$, l'égalité $pre(a^p_{v_i}) = prv(a^q_{v_j})[v_i]$ signifie que l'action $a^p_{v_i}$ peut "clobber" [3], i.e. *menace*, la prevail-condition de $a^q_{v_j}$ sur v_i car, par définition de la pré-condition, $pre(a^p_{v_i})$ est la valeur de v_i qui sera changée, ou "clobbered", par $a^p_{v_i}$. Par conséquent, pour respecter la deuxième condition du MTC et sachant que les plans solutions sont T_1, l'action $a^q_{v_j}$ doit être ordonnée avant $a^p_{v_i}$ dans le plan solution. $a^q_{v_j}$ devient donc un prédécesseur de $a^p_{v_i}$, d'où l'ensemble $\mathcal{N}_m(a^p_{v_i})$. On dénote $\mathcal{N}(a^p_{v_i}) = \mathcal{N}_{pre}(a^p_{v_i}) \cup \mathcal{N}_{prv}(a^p_{v_i}) \cup \mathcal{N}_m(a^p_{v_i})$ l'ensemble des prédécesseurs ($\mathcal{N}$eighbors) de $a^p_{v_i}$. C'est un ensemble partiellement ordonné d'actions.

La post-unicité implique que $\mathcal{N}_{pre}(a^p_{v_i})$ est un singleton, i.e. $a^p_{v_i}$ n'a qu'un seul prédécesseur par dépendance post-pré. En revanche, ce prédécesseur de $a^p_{v_i}$ n'a pas nécessairement que $a^p_{v_i}$ comme *successeur* via la dépendance post-pré. En effet, la post-unicité n'implique pas la pré-unicité comme on peut le voir avec les actions *Stocker botte de foin* et *Remplir mangeoire* de l'Éleveur de Chevaux (Tab. 1) qui partagent la même pré-condition malgré la post-unicité du problème. Elles ont toutes les deux comme unique prédécesseur *Prendre botte de foin* mais il en résulte que *Prendre botte de foin* a deux successeurs via la dépendance post-pré. Cette implication a son importance pour la recherche des actions menaçantes car une prevail-condition menacée est définie par l'égalité $pre(a^p_{v_i}) = prv(a^q_{v_j})[v_i]$ ($a^p_{v_i}, a^q_{v_j} \in \mathcal{A}$) mais l'action menaçante $a^p_{v_i}$ n'est pas identifiable par la pair $(v_i, pre(a^p_{v_i}))$ en temps constant. Pour pallier ce problème, on introduit deux accesseurs par action : *Next* et *NextInCycle*, qui pointent vers un et un seul successeur si elles sont définies [4]. L'accesseur *NextInCycle* est définissable lors d'un pré-processing. En effet, plusieurs actions dépendantes entre elles par dépendances post-pré peuvent boucler entre elles, auxquels cas on peut facilement montrer par post-unicité que ce *cycle* est unique [5]. Comme le cycle est unique, s'il existe, alors chaque action du cycle n'a qu'un seul prédécesseur et qu'un seul successeur dans ce cycle. S'il n'y a pas de cycle, $NextInCycle = \emptyset$. Pour tout $v_i \in \mathcal{M}$, on note $\mathrm{Cycle}[v_i]$ l'ensemble des actions appartenant à ce cycle. L'accesseur *Next*, quant à lui, est définie dynamiquement et une seule fois par la procédure 1 *BuildChain* qui retourne une *chaîne d'actions*, i.e. une séquence d'actions totalement ordonnée par dépendance post-pré. Une action menaçante est donc identifiable en temps constant une fois la chaîne d'actions construite (avec *Next*) ou si elle appartient à un cycle (avec *NextInCycle*). Soit $a^x_{v_i}, a^q_{v_j} \in \mathcal{A}$ telles que $Next(a^x_{v_i})$ pointe vers une action qui menace $a^q_{v_j}$, on a $pre(Next(a^x_{v_i})) = prv(a^q_{v_j})[v_i]$, et donc $post(a^x_{v_i}) = prv(a^q_{v_j})[v_i]$. Ce qui implique que si $a^q_{v_j} \in \mathcal{N}_m(Next(a^x_{v_i}))$, alors $a^x_{v_i} \in \mathcal{N}_{prv}(a^q_{v_j})$.

2. La première condition du MTC est satisfaite par $\mathcal{N}_{pre}$ et $\mathcal{N}_{prv}$.

3. *Clobber* est un terme utilisé et défini par D. Chapman.

4. *Next* et *NextInCycle* peuvent pointer vers le même successeur.

5. Les actions (PU) (noeuds) et leurs dépendances post-pré (arcs) forment un graphe dirigé faiblement connecté. Par contraposition, s'il y a deux cycles dans le graphe alors il y a deux actions avec la même post-condition.

$\mathcal{A}$	pre	post	prv	Description
$a^0_{v_0}$	$v_0 = 1$	$v_0 = 0$	$\langle u, u, u\rangle$	Stocker botte de foin
$a^1_{v_0}$	$v_0 = 0$	$v_0 = 1$	$\langle u, 0, u\rangle$	Prendre botte de foin
$a^2_{v_0}$	$v_0 = 1$	$v_0 = 2$	$\langle u, u, u\rangle$	Remplir mangeoire
$a^0_{v_1}$	$v_1 = 1$	$v_1 = 0$	$\langle u, u, u\rangle$	Poser seau
$a^1_{v_1}$	$v_1 = 0$	$v_1 = 1$	$\langle 0, u, u\rangle$	Prendre seau
$a^1_{v_2}$	$v_2 = 0$	$v_2 = 1$	$\langle u, 1, u\rangle$	Remplir seau d'eau
$a^2_{v_2}$	$v_2 = 1$	$v_2 = 2$	$\langle u, 1, u\rangle$	Remplir abreuvoir

TABLE 1 – Actions de l'Éleveur de Chevaux dont le but est de nourrir les chevaux : $s_\star = \langle 2, 0, 2\rangle$ où v_0 représente la *Botte de foin* avec $\mathcal{D}_{v_0} = \{0 : stock,\ 1 : enMain,\ 2 : dansMangeoire\}$, v_1 représente un *Seau* avec $\mathcal{D}_{v_1} = \{0 : auSol,\ 1 : enMain\}$, et v_2 représente de l'*Eau* avec $\mathcal{D}_{v_2} = \{0 : dansFontaine,\ 1 : dansSeau,\ 2 : dansAbreuvoir\}$.

Tab. 1 présente le problème de l'Éleveur de Chevaux, un problème de classe SAS-PU$\mathbb{T}_1$. Toutes les actions sont post-uniques et unaires car chaque action ne modifie qu'une seule variable d'état et il n'y en a pas deux modifiant une même variable d'état à la même valeur. Chaque action est donc identifiable avec le format $a^p_{v_i}$ et ces identifiants se trouvent dans la colonne $\mathcal{A}$. Ce problème n'est pas (S) car les actions *Remplir Seau d'Eau* ($a^1_{v_2}$) et *Remplir abreuvoir* ($a^2_{v_2}$) ont une prevail-condition différente que *Prendre botte de foin* ($a^1_{v_0}$) sur la variable d'état *Seau* (v_1) : $prv(a^2_{v_2})[v_1] = prv(a^1_{v_2})[v_1] \neq prv(a^1_{v_0})[v_1]$. Le problème aurait été (S) si : $prv(a^1_{v_0})[v_1] = u$ ou $prv(a^1_{v_0})[v_1] = 1$, par exemple. Ce problème est T_1 car, pour chaque instance (s_0, $s_\star$) du problème, le plan solution minimal, i.e. le plan solution possédant le moins d'actions, s'il existe, entre s_0 et $s_\star$, ne contient pas deux fois la même action. Considérons maintenant l'action *Remplir abreuvoir* identifiée par $a^2_{v_2}$ pour illustrer $\mathcal{N}(a^2_{v_2})$. On a $\mathcal{N}_{pre}(a^2_{v_2}) = \{a^1_{v_2}\}$ et $\mathcal{N}_{prv}(a^2_{v_2}) = \{a^1_{v_1}\}$. En fonction de l'instance (s_0, $s_\star$), *Remplir abreuvoir* peut avoir sa prevail-condition sur v_1 menacée par l'action *Poser seau*, auquel cas $a^2_{v_2} \in \mathcal{N}_m(a^0_{v_1})$ afin que *Remplir abreuvoir* soit ordonnée avant *Poser seau* dans le plan solution minimal. Enfin, considérons les actions $a^0_{v_0}$, $a^1_{v_0}$ et $a^2_{v_0}$ pour illustrer le comportement des dépendances post-pré et comment sont définis *Next* et *NextInCycle*. On constate que *Stocker botte de foin* ($a^0_{v_0}$) et *Prendre botte de foin* ($a^1_{v_0}$) bouclent entre elles. On a $\mathrm{Cycle}[v_0] = \{a^0_{v_0}, a^1_{v_0}\}$, $a^1_{v_0} \in \mathcal{N}_{pre}(a^0_{v_0}) \wedge NextInCycle(a^0_{v_0}) = a^1_{v_0}$, et $a^0_{v_0} \in \mathcal{N}_{pre}(a^1_{v_0}) \wedge NextInCycle(a^1_{v_0}) = a^0_{v_0}$. En revanche, *Remplir mangeoire* ($a^2_{v_0}$) n'est pas dans le cycle, on a $a^1_{v_0} \in \mathcal{N}_{pre}(a^2_{v_0}) \wedge NextInCycle(a^2_{v_0}) = \emptyset$. $a^1_{v_0}$ a deux successeurs possibles en fonction de l'instance du problème, on peut avoir $Next(a^1_{v_0}) = a^2_{v_0}$ ou $Next(a^1_{v_0}) = NextInCycle(a^1_{v_0}) = a^0_{v_0}$.

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

Input : $v_i \in \mathcal{M}$, $s, g \in \mathcal{D}_{v_i}$ deux variables du domaine de v_i ; $\mathcal{D}$, l'ensemble des actions jaunes ; $E_{\mathcal{D}}$, l'ensemble des ordres entre les actions de $\mathcal{D}$; $\mathcal{A}$, L'ensemble des actions du problème.
Parameters : x, y, deux valeurs de $\mathcal{D}_{v_i}$.
Output : Chaque action a de la chaîne est jaune, ajouté à $\mathcal{D}$ et $(\mathcal{N}_{pre}(a), a)$ est ajouté à $E_{\mathcal{D}}$.

$x \leftarrow \emptyset$; $y \leftarrow \emptyset$
if $a^g_{v_i} \notin \mathcal{A}$ **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($a^y_{v_i}$) $= \emptyset$ **then**
  Next($a^y_{v_i}$) $\leftarrow a^g_{v_i}$
end if
while $y \neq s$ **do**
  if $a^y_{v_i} \notin \mathcal{A}$ **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($a^y_{v_i}$) $= \emptyset$ **then**
    Next($a^y_{v_i}$) $\leftarrow a^x_{v_i}$
  end if
end while

2.3 Planification réduite à un tri topologique

SAS$^+$-PUS [2] est une classe abordable de problèmes de planification SAS$^+$ pour lesquels le meilleur algorithme [1, pp. 84-90], que nous notons désormais $\mathcal{P}$, s'exécute en $O(m^2 n)$ (cf. **Théorème 4.11** [1, p. 89]). Dans une première phase, $\mathcal{P}$ itère sur chacune des m variables d'état du but pour construire des *chaînes* d'au plus n actions vers l'état initial grâce à la restriction (P). Dans une deuxième phase, $\mathcal{P}$ ordonne les couples de $O(mn)$ actions apparaissant dans des chaînes distinctes grâce à la restriction (S) pour produire un ensemble partiellement ordonné d'actions ; pour réaliser cette deuxième phase, $\mathcal{P}$ itère d'abord sur $O(mn)$ actions et ensuite sur $O(m)$ variables. Dans une troisième et dernière phase, si l'ensemble partiellement ordonné d'actions ne contient aucun cycle, il est retourné comme solution.

Notre algorithme, que nous notons désormais $\mathbb{P}$, suit les trois phases de $\mathcal{P}$ tout en simplifiant la complexité temporelle de la deuxième phase. Le point clé de l'algorithme $\mathbb{P}$ pour obtenir une complexité temporelle linéaire est d'utiliser les dépendances post-prv entre actions pour vérifier les prevail-conditions plutôt que de parcourir l'ensemble des variables d'état. Les prevail-conditions sont des états *partiellement* définis et, couplées aux restrictions (P) et (U), elles représentent également un ensemble partiellement ordonné d'actions ($\mathcal{N}_{prv}$, cf. sous-section 2.2), que $\mathbb{P}$ utilise

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

Input : $a^p_{v_i}$, une action jaune identifiée ; s_0, l'état initial ; P, le plan solution.
Output : $a^p_{v_i}$ est coloré en vert une fois que tous ses prédécesseurs ont été topologiquement triés ; Elle est insérée en queue de la liste P.

Color($a^p_{v_i}$) $\leftarrow blue$
for $a^q_{v_j} \in \mathcal{N}(a^p_{v_i})$ **do**
  if $a^q_{v_j} \notin \mathcal{N}_{pre}(a^p_{v_i}) \vee a^p_{v_i}$ n'est pas la 1^{re} action à modifier $s_0[v_i]$ **then**
    if Color($a^q_{v_j}$) $= blue$ **then** fail {Cycle détecté.}
    end if
    if Color($a^q_{v_j}$) $= yellow$ **then**
      DFSTopo($a^q_{v_j}, \mathcal{D}, E_{\mathcal{D}}, s_0, \mathtt{P}$)
    end if
  end if
end for
Color($a^p_{v_i}$) $\leftarrow green$; P $\leftarrow$ P $+ \{a^p_{v_i}\}$;

lors des phases 2 et 3.

Soit $\mathcal{A}$ l'ensemble des actions d'un problème de planification et $(s_0, s_\star)$ une instance de ce problème, $\mathbb{P}$ planifie en arrière en utilisant l'identification des actions $a^p_{v_i} \in \mathcal{A}$ et leurs ensembles de voisinage $\mathcal{N}_{pre}(a^p_{v_i})$ et $\mathcal{N}_{prv}(a^p_{v_i})$ afin de trouver un plan solution minimal et totalement ordonné entre s_0 et $s_\star$. Un pré-processing est ainsi nécessaire pour (1) construire une table de *Hashing* des actions $a^p_{v_i}$ de $\mathcal{A}$ en fonction de (v_i, p) et pour (2) construire leurs ensembles $\mathcal{N}_{pre}(a^p_{v_i})$ et $\mathcal{N}_{prv}(a^p_{v_i})$. Pour toute instance d'un problème de planification, (1) permet à $\mathbb{P}$ d'accéder à n'importe quelle action en un temps constant ($O(1)$) tandis que (2) permet de réduire la complexité temporelle de $\mathbb{P}$ par rapport à $\mathcal{P}$ lors de la phase 2. Toutes les actions du problème ne sont pas utiles à la résolution d'une instance, on note ainsi $\mathcal{D}$ l'ensemble des actions nécessaires à la résolution. Il en découle que l'ensemble les prédécesseurs d'une action a ne sont pas nécessairement tous utiles. En effet, l'état initial peut satisfaire la pré-condition de a ainsi que certaines de ses prevail-conditions, auxquels cas les éventuels prédécesseurs concernés ne sont pas utiles. Si l'`Éleveur de Chevaux` ne porte initialement pas de *botte de foin* ($s_0[v_0] = 0$) et que son objectif est d'en porter une ($s_\star[v_0] = 1$), alors l'action *Prendre botte de foin* est nécessaire et réalisable dans s_0, et l'action qui la précède, *Stocker botte de foin*, n'est pas utile. Ainsi, la résolution d'une instance (s_0, $s_\star$) par $\mathbb{P}$ consiste à trouver l'ensemble $\mathcal{D}$ et à construire un ensemble suffisant des ordres entre les actions de $\mathcal{D}$, on dénote $E_{\mathcal{D}}$ cet ensemble d'ordres. Cette recherche et construction des ordres s'effectue lors des phases 1 et 2. La troisième et dernière phase est un tri topologique du plan partiellement ordonné $\langle \mathcal{D}, E_{\mathcal{D}} \rangle$ afin de retourner un plan totalement ordonnée (restriction (T)). Enfin, on introduit 4 *couleurs* pour les actions : (blanc) la couleur initiale lorsque $\mathbb{P}$ est appelé, (jaune) l'action est utile pour résoudre le problème, (bleu) l'action est en cours de tri, (vert) l'action est triée topologiquement et insérée dans le plan solu-

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

Input :$\mathcal{M}$; $\mathcal{A}$; s_0, $s_\star$: états initial et final totalement définis.

Parameters : $\mathcal{D}$, l'ensemble des actions jaunes ; $E_\mathcal{D}$, l'ensemble des ordres entre les actions jaunes ;

Output : P : un plan d'action totalement ordonné qui relie s_0 à $s_\star$; produit un échec si l'instance n'est pas solvable.

1: $\mathtt{P} \leftarrow \emptyset$; $\mathcal{D} \leftarrow \emptyset$; $E_\mathcal{D} \leftarrow \emptyset$
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 et $s_\star$ sont égaux.}
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** $q \neq s_0[v_j]$ **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$(v_j, s_0[v_j], q, \mathcal{D}, E_\mathcal{D}, \mathcal{A})$
16: **end if**
17: $E_\mathcal{D} \leftarrow E_\mathcal{D} \cup \{(a^q_{v_j}, a^p_{v_i})\}$
18: **end if**
19: **if** $q \neq s_\star[v_j]$ **then**
20: **if** Next$(a^q_{v_j}) = \emptyset$ **then**
21: BuildChain$(v_j, q, s_0[v_j], \mathcal{D}, E_\mathcal{D}, \mathcal{A})$
22: **end if**
23: **if** prv(Next$(a^q_{v_j}))[v_i] \neq p$ **then**
24: $E_\mathcal{D} \leftarrow E_\mathcal{D} \cup \{(a^p_{v_i}$,Next$(a^q_{v_j}))\}$
25: **end if**
26: **end if**
27: **if** $q = s_0[v_j] \wedge q \in$ Cycle$[v_j]$ **then**
28: **if** $a^p_{v_i}$ est s'ordonne avant modification de $s_0[v_j]$ **then** {cf. section 2.3}
29: $E_\mathcal{D} \leftarrow E_\mathcal{D} \cup \{(a^p_{v_i}$,NextInCycle$(a^{s_0[v_j]}_{v_j}))\}$
30: **else** {$a^p_{v_i}$ s'ordonne après.}
31: $E_\mathcal{D} \leftarrow E_\mathcal{D} \cup \{(a^{s_0[v_j]}_{v_j}, a^p_{v_i})\}$
32: **end if**
33: **end if**
34: **end for**
35: **end for**
36: **for** $a \in \mathcal{D}$ **do** {Phase 3}
37: **if** Color$(a) = yellow$ **then**
38: DFSTopo$(a, \mathcal{D}, E_\mathcal{D}, s_0, \mathtt{P})$
39: **end if**
40: **end for**
41: **return** P

tion. Les actions de $\mathcal{D}$ sont toutes colorées en jaune. Blanc et jaune sont utilisées dans les 3 phases alors bleu et vert ne sont utilisées que dans la phase 3.

(Phase 1, 3.2 à 3.6) $\mathbb{P}$ va d'abord chercher les différences entre s_0 et $s_\star$ pour y construire des chaînes d'actions. Ces chaînes sont construites par la procédure 1 *BuildChain* via les dépendances post-pré des actions (1.5 et 1.13). Dynamiquement, cette procédure colore en jaune, ajoute dans $\mathcal{D}$ (1.4 et 1.13) et définit la variable *Next* (1.7 et 1.16) des actions parcourues. Également, pour chaque action parcourue, l'ordre avec son prédécesseur est ajouté dans $E_\mathcal{D}$ (1.6 et 1.15).

(Phase 2, 3.9 à 3.35), $\mathbb{P}$ vérifie les prevail-conditions de toutes les actions de $\mathcal{D}$ (3.9) en parcourant les prédécesseurs de l'ensemble $\mathcal{N}_{prv}$ (3.10). Soient $a^p_{v_i}, a^q_{v_j} \in \mathcal{A}$ telles que $a^p_{v_i}$ est l'action en cours de vérification et $a^q_{v_j}$ un prédécesseur par dépendance post-prv. La prevail-condition sur v_j de $a^p_{v_i}$, $prv(a^p_{v_i})[v_j]$, est satisfaite soit par $s_0[v_j]$ soit par $a^q_{v_j} \in \mathcal{N}_{prv}(a^p_{v_i})$ (restriction (P)), sinon l'instance de problème n'est pas solvable. Si le prédécesseur $a^q_{v_j}$ est jaune, cela signifie que la chaîne d'actions v_j entre $s_0[v_j]$ et $s_\star[v_j]$ est déjà construite, alors $\mathbb{P}$ se contente d'ordonner l'action en cours de vérification $a^p_{v_i}$ avec son prédécesseur $a^q_{v_j}$ (3.17). Si $a^q_{v_j}$ n'est pas jaune, elle est blanche (3.14) et la chaîne d'actions v_j entre $s_0[v_j]$ et $prv(a^p_{v_i})[v_j]$ est manquante. Elle est donc construite (3.15), puis l'ordre entre $a^p_{v_i}$ et $a^q_{v_j}$ est construit et ajouté dans $E_\mathcal{D}$ (3.17). De 3.19 à 3.26, $\mathbb{P}$ traite les actions menaçantes, ou "clobberer". Comme expliqué dans la sous-section 2.2, si $a^q_{v_j}$ satisfait la prevail-condition de $a^p_{v_i}$ sur v_j et si $q \neq s_\star[v_j]$, alors le successeur de $a^q_{v_j}$ ($Next(a^q_{v_j})$), s'il existe, menace la prevail-condition sur v_j de $a^p_{v_i}$. Pour cela $Next(a^q_{v_j})$ doit être définie (3.20). $Next(a^q_{v_j}) = \emptyset$ équivaut à dire que le successeur de $a^q_{v_j}$, s'il existe, est blanc. La chaîne d'actions v_j entre $prv(a^p_{v_i})[v_j]$ et $s_0[v_j]$ est donc manquante et doit être construite (3.21). (3.23) $Next(a^q_{v_j})$ est nécessairement défini, et si $Next(a^q_{v_j}) \notin \mathcal{N}_{prv}(a^p_{v_i})$, i.e. $Next(a^q_{v_j})$ n'est pas un prédécesseur par dépendance post-prv de $a^p_{v_i}$, alors $a^p_{v_i} \in \mathcal{N}_m(Next(a^q_{v_j}))$, i.e. $a^p_{v_i}$ est un prédécesseur de $Next(a^q_{v_j})$ pour respecter le MTC (cf. sous-section 2.2), d'où la construction de l'ordre (3.24). De 3.27 à 3.33, $\mathbb{P}$ traite des cas qui ne pouvaient exister avec la restriction (S). En assouplissant (S), il est dorénavant possible d'avoir des prevail-conditions pour une même variable d'état v_j ayant des valeurs différentes (cf. Tab. 1, $prv(a^1_{v_0})[v_1] \neq prv(a^1_{v_2})[v_1]$). En particuliers, il est possible qu'une action ait sa prevail-condition en v_j satisfaite par $s_0[v_j] \in \mathcal{D}_{v_j}$ et qu'une autre action nécessaire à la résolution du problème ait sa prevail-condition sur v_j satisfaite par une autre valeur $x \in \mathcal{D}_{v_j}$. Cette situation est problématique si $a^{s_0[v_j]}_{v_j}$ et $a^x_{v_j}$ appartiennent à Cycle$[v_j]$. Dans ce cas la valeur $s_0[v_j]$ peut apparaître deux fois, malgré la restriction T_1, une fois avec l'état initial $s_0[v_j]$ et une fois après l'exécution de l'action $a^{s_0[v_j]}_{v_j}$. Il faut donc déterminer si l'action ayant $s_0[v_j]$ comme prevail-condition s'ordonne avant la modification de v_j (3.29) ou si elle s'ordonne après $a^{s_0[v_j]}_{v_j}$ qui rétablit $s_0[v_j]$ (3.31). On peut déterminer (3.29) ou (3.31) lors du pré-traitement en cherchant si deux actions qui ont une prevail-condition définie différemment sur v_j sont reliées par un chemin dirigé qui ne passe pas par leur dépendance post-prv en v_j. Soit $a^p_{v_i}, a^q_{v_j} \in \mathcal{A}$, si $prv(a^p_{v_i})[v_k] = s_0[v_k]$, $prv(a^q_{v_j})[v_k] = x \neq s_0[v_k]$ et il existe un chemin dirigé entre $a^p_{v_i}$ et $a^q_{v_j}$

ne passant pas par les actions affectant v_k où $a^p_{v_i}$ est avant $a^q_{v_j}$, alors $a^p_{v_i}$ doit être exécutée avant la modification de l'état initial en v_j (3.29), sinon, si $a^p_{v_i}$ est après $a^q_{v_j}$ ou $a^p_{v_i}$ et $a^q_{v_j}$ ne sont pas reliées, on est dans le cas (3.31). Prenons l'instance ($s_0 = \langle 0,0,0\rangle$, $s_\star = \langle 2,0,2\rangle$) pour l'`Éleveur de Chevaux`, on a $prv(a^1_{v_0})[v_1] = s_0[v_1]$ et $prv(a^1_{v_2})[v_1] = prv(a^2_{v_2})[v_1] = 1 \neq s_0[v_1]$. Les actions $a^0_{v_1}$ et $a^1_{v_1}$ seront utiles pour résoudre l'instance, elles seront colorées en jaune lors de la phase 2 car lors de la phase 1, $s_0[v_1] = s_\star[v_1]$ et donc la condition (3.3) n'est pas respectée. Sans la partie 3.27 à 3.33 de l'algorithme, l'action $a^1_{v_0}$ n'a pas d'ordre spécifique relatif à sa prevail-condition en v_1 car elle est satisfaite par $s_0[v_1]$. Or, elle est menacée par $a^1_{v_1}$, mais également rétablie par $a^0_{v_1}$. Sans ordre spécifique, il existe des tris topologiques où $a^1_{v_0}$ est ordonnée après $a^1_{v_1}$ et avant $a^0_{v_1}$, ce qui est incompatible avec sa prevail-condition sur v_1. Il n'existe pas de chemin dirigé entre $a^1_{v_0}$ et $a^1_{v_2}$, ni entre $a^1_{v_0}$ et $a^1_{v_2}$, sans passer par les actions affectant v_1, $a^1_{v_0}$ s'ordonne donc après $a^0_{v_1}$ (3.30).
(Phase 3, 3.36 à 3.40) Enfin, $\mathbb{P}$ trie topologiquement avec la Procédure 2 *DFSTopo* toutes les actions de $\mathcal{D}$ en fonction des ordres de l'ensemble $E_\mathcal{D}$. Concernant la condition d'arrêt (2.3) "$a^p_{v_i} \in \mathcal{A}$ est la première action à modifier $s_0[v_i]$", la condition $pre(a^p_{v_i}) = s_0[v_i]$ n'est pas suffisante. Prenons une instance de l'`Éleveur de Chevaux` où $s_0[v_0] = 1$ et $s_\star[v_0] = 2$, il y a deux chaînes d'actions possible entre $s_0[v_0]$ et $s_\star[v_0]$: $\langle a^2_{v_0}\rangle$ et $\langle a^0_{v_0}, a^1_{v_0}, a^2_{v_0}\rangle$. Si l'instance complète du problème est ($s_0 = \langle 1,0,0\rangle, s_\star = \langle 2,0,0\rangle$), alors $\langle a^2_{v_0}\rangle$ est le plan solution minimal et $a^2_{v_0}$ est la première action à modifier $s_0[v_0]$. En revanche, si l'instance complète du problème est ($s_0 = \langle 1,0,0\rangle, s_\star = \langle 2,0,2\rangle$), $\langle a^0_{v_0}, a^1_{v_0}, a^2_{v_0}\rangle$ fera partie du plan solution minimal et la première action à modifier $s_0[v_0]$ sera $a^0_{v_0}$.

Théorème 1. $\mathbb{P}$ *est correct et complet.*

Démonstration. (*Esquisse*) $\mathbb{P}$ est correct car il satisfait les deux conditions du MTC lors de la construction de $\mathcal{D}$ et de $E_\mathcal{D}$ (Phases 1 et 2) (cf. sous-section 2.2 et la description ci-dessus). La première condition du MTC est satisfaite par (1.6, 1.17, 3.4, 3.15, 3.17, 3.21), la deuxième condition du MTC est satisfaite par (3.24, 3.29, 3.31). Il en découle un tri topologique correct lors de la phase 3, i.e. un plan solution minimal et totalement ordonné dont l'application successive des actions est possible et aboutit finalement à l'état but de l'instance du problème.
(*Esquisse*) $\mathbb{P}$ est complet car peu importe l'instance du problème il retourne une solution. Pour savoir si une instance est solvable, pour tout $v_i \in \mathcal{M}$ il doit exister une chaîne d'actions entre $s_0[v_i]$ et $s_\star[v_i]$. Cette existence se prouve à l'aide de la post-unicité. Si au moins une n'existe pas, $\mathbb{P}$ retourne un échec (1.2, 1.11, 1.13, 3.12). Si les chaînes d'actions existent mais que l'instance n'est pas solvable avec un plan T_1, $\mathbb{P}$ retourne un échec (1.13, 2.4). Enfin si les chaînes d'actions existent mais que l'instance n'est pas solvable car il y a des cycles via les dépendances post-prv ou via les relations entre actions menacées et menaçantes, alors $\mathbb{P}$ retourne un échec (2.4). Dans tous les autres cas, $\mathbb{P}$ termine et retourne une solution. Pour la terminaison de $\mathbb{P}$, aucune boucle n'est infinie : (1.10) a $O(n)$ itérations, (2.2) a $O(m)$ itérations, (3.2) a exactement m itérations, (3.9) a $O(\mathcal{A})$ itérations, (3.10) a $O(m)$ itérations et (3.36) a $O(\mathcal{A})$ itérations. Pour finir, la récursivité de *DFSTopo* se termine soit sur un échec, soit sur la condition d'arrêt (2.3). □

Théorème 2. $\mathbb{P}$ *a une complexité temporelle de* $O(|\mathcal{A}| + |E_\mathcal{A}|)$ *dans le pire des cas.*

Démonstration. Nous avons $E_\mathcal{A} = \{(b,a) | a \in \mathcal{A} \wedge b \in \mathcal{N}_{pre}(a) \cup \mathcal{N}_{prv}(a) \cup \mathcal{N}_m(a)\}$ avec l'ensemble $\mathcal{N}_{prv}$ qui permet à $\mathbb{P}$ de ne naviguer qu'à travers les prevail-conditions définies des actions de $\mathcal{D}$. La phase 1 a au plus $|\mathcal{A}| = O(mn)$ étapes : m étapes via la boucle for (3.2) fois n étapes via la procédure *BuildChain* (3.4). En raison de la restriction (T_1), $\mathcal{D}$ ne peut pas être supérieur à $\mathcal{A}$, donc la boucle for (3.9) nécessite au plus $|\mathcal{A}| = O(mn)$ étapes. Les deux boucles for (3.9) et (3.10) nécessitent $O(|\mathcal{A}| + |E_{\mathcal{A},prv}|)$ étapes avec $E_{\mathcal{A},prv} = \{(b,a) | a \in \mathcal{A} \wedge b \in \mathcal{N}_{prv}(a)\}$. En effet, lors de cette étape, les deux boucles for servent à vérifier les prevail-conditions de l'ensemble des actions de $\mathcal{D}$: il y a donc $|\mathcal{D}|$ étapes pour parcourir toutes les actions de $\mathcal{D}$, plus $|E_{\mathcal{D},prv}|$ étapes pour visiter tous les prédécesseurs de toutes les actions de $\mathcal{D}$. Les procédures *BuildChain* (3.15) et (3.21) peuvent être déclenchées ssi l'action en cours d'évaluation possède une prevail-condition qui n'est satisfaite ni par l'état initial, ni par une action de $\mathcal{D}$. Dans ce cas, les deux procédures cherchent des actions qui sont dans un ensemble $\mathcal{T} \subseteq \mathcal{A} \setminus \mathcal{D}$. Les procédures *BuildChain* rajoutent donc $|\mathcal{T}|$ étapes pour l'ensemble de leurs appels. Ces actions de $\mathcal{T}$ sont ajoutées à l'ensemble $\mathcal{D}$ par la procédure *BuildChain* car il faut dorénavant vérifier leurs prevail-conditions, il y a donc de nouveau $|\mathcal{T}|$ étapes de plus via la boucle for (3.9) ainsi que $|E_{\mathcal{T},prv}|$ de plus pour la boucle for (3.10). Finalement, la phase 2 a une complexité de $O(|Dset| + |E_{\mathcal{D},prv}| + 2 \cdot |\mathcal{T}| + |E_{\mathcal{T},prv}|) \equiv O(|Dset| + |E_{\mathcal{D},prv}| + 2 \cdot |\mathcal{A} \setminus \mathcal{D}| + |E_{\mathcal{A}\setminus\mathcal{D},prv}|) \equiv O(|\mathcal{A}| + |E_{\mathcal{A},prv}|)$. Toutes les instructions de (3.27) à (3.33) se font en ($O(1)$) grâce à un pré-traitement décrit dans le paragraphe (Phase 2) de la sous-section 2.3. Enfin, la phase 3 trie topologiquement les actions jaunes de l'ensemble $\mathcal{D}$ à l'aide de l'ensemble des ordres $E_\mathcal{D}$ en $O(|\mathcal{D}|+|E_\mathcal{D}|)$; dans le pire des cas, cela équivaut à $O(|\mathcal{A}| + |E_\mathcal{A}|)$ qui finalement domine l'ensemble et prouve la complexité linéaire en temps de $\mathbb{P}$. □

Théorème 3. $\mathbb{P}$ *nécessite au plus* $O(m^2 n)$ *d'espace.*

Démonstration. Une action (PU) occupe $O(m)$ d'espace : les pré- et post-conditions peuvent être réduites à une variable chacune, plus une variable pour l'indice de la variable d'état affectée, et l'ensemble $\mathcal{N}_{pre}$ est un singleton en raison des restrictions (P) et (U) ; les prevail-conditions, au contraire, sont des listes de m éléments, et l'ensemble $\mathcal{N}_{prv}$ a au plus m éléments. La table de *Hashing* des actions prend $O(mn)$ d'espace. Enfin, L'ensemble $E_\mathcal{D}$ peut avoir au plus $(m^2 n)$ éléments : on a $|\mathcal{D}| \leq |\mathcal{A}| \leq mn$ en raison de la restriction (T_1), il y a donc $O(mn)$ actions, et chaque action

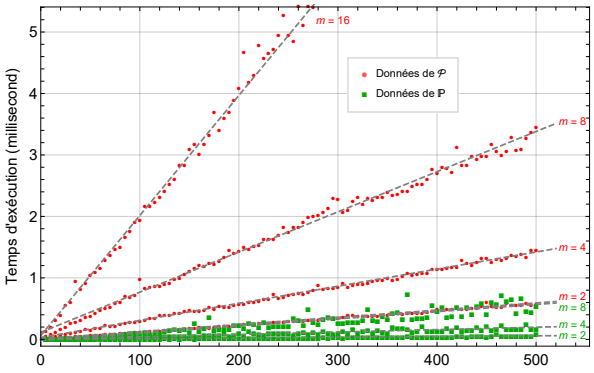

FIGURE 1 – Les temps d'exécution de $\mathbb{P}$ et $\mathcal{P}$ sont linéaires en fonction du nombre de valeurs (n) par variable d'état (m) bien que le problème MultiPrv_n_Cycle est conçu pour générer $O(m^2)$ ordres entre les mn actions. m est constant lors de cette expérience.

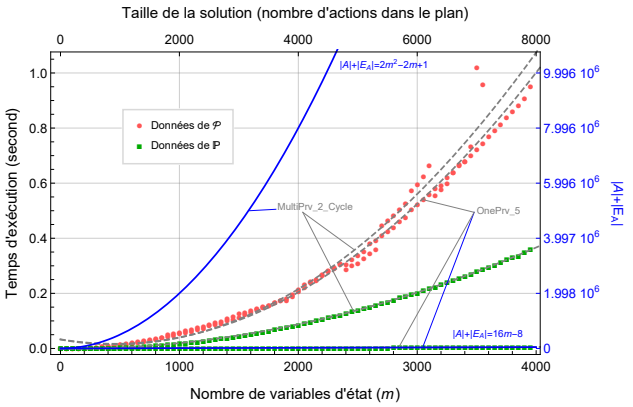

FIGURE 2 – Lorsque le nombre d'ordres est linéaire (resp. quadratique) par rapport au nombre de variables d'état (m), les temps d'exécution de $\mathbb{P}$ sont linéaires (resp. quadratiques) par rapport au nombre de variables d'état (m), ce qui illustre la complexité temporelle de $O(|A| + |E_A|)$, alors que les temps d'exécution de $\mathcal{P}$ sont toujours quadratiques par rapport au nombre de variables d'état (m) malgré les différences dans le nombre d'ordres entre MultiPrv_2_Cycle et OnePrv_5 .

peut avoir $O(m)$ prédécesseurs : au plus $(m-1)$ via $\mathcal{N}_{prv}$ et au plus $(m-1)$ via $\mathcal{N}_m$. D'où $O(E_{\mathcal{D}}) = O(m^2 n)$. □

3 Analyse comparative de $\mathbb{P}$ vs $\mathcal{P}$

Nous décrivons ici les trois problèmes que nous utilisons dans cet article pour illustrer les complexités temporelles de $\mathbb{P}$ et $\mathcal{P}$. Dans les figures 1,2 et 3 les carrés verts, resp. les disques rouges, représentent les instances de problèmes résolus par $\mathbb{P}$, resp. $\mathcal{P}$. Les tests ont été effectués avec la configuration suivante : CPU AMD Ryzen 2700X (8-Core) (3,7GHz), 32Gb de RAM et Windows 10 (64 bits) ; les deux planificateurs sont écrits en C++14 avec les paramètres par défaut de Microsoft Visual Studio 2019.

MultiPrv_n_Cycle est conçu pour vérifier que le temps d'exécution des deux planificateurs est linéaire en (n) (cf. Figure 1) ; il génère $O(m^2)$ d'ordres entre mn actions avec $0 \leq p < n$; $\forall v_i \in \mathcal{M}$, nous avons :

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

Pour l'expérience, les états initial et final sont de la forme :

- $\forall v_i \in \mathcal{M}$, $s_0[v_i] = 0$,
- $\forall v_i \in \mathcal{M} \setminus \{v_0\}$, $s_*[v_i] = 0 \wedge s_*[v_0] = n - 1$.

[1] indique avoir implémenté $\mathcal{P}$ avec LISP et rapporte des temps d'exécution superlinéaires en (n), contrairement aux résultats théoriques (cf. **Théorème 4.10**, p. 89) ; car, entre autres, LISP ne fournit pas de contrôle sur la gestion de la mémoire, il a été suspecté que ce soit la raison des résultats pratiques altérés. Nous avons implémenté $\mathbb{P}$ et $\mathcal{P}$ en C++ sans optimisation spécifique, mais nous avons effectivement dû gérer soigneusement la mémoire pour obtenir des temps d'exécution linéaires en (n) pour les deux algorithmes $\mathcal{P}$ et $\mathbb{P}$.

MultiPrv_2_Cycle est un cas spécifique de MultiPrv_n_Cycle avec $n = 2$; ce n'est rien de plus que le problème du tunnel que nous avons présenté dans la section 2.1. Comme il génère $O(m^2)$ d'ordres entre $2m$ actions, nous pouvons facilement étendre ce problème pour montrer que, dans le pire des cas, le temps d'exécution de $\mathbb{P}$ est quadratique en fonction du nombre de variables m (cf. Figure 2).

OnePrv_5 est conçu pour montrer que le temps d'exécution de $\mathbb{P}$ peut être linéaire avec le nombre de variables (m) alors que celui de $\mathcal{P}$ est quadratique (cf. Figure 2) malgré le nombre $O(m)$ d'ordres d'action à traiter. OnePrv_5 génère $O(m)$ ordres entre $4m$ actions avec $0 < p < n = 5$; $\forall v_i \in \mathcal{M}$, nous avons :

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

Pour l'expérience, les états initial et final sont de la forme :

- $\forall v_i \in \mathcal{M}$, $s_0[v_i] = 0$,
- $\forall v_i \in \mathcal{M}$, $s_*[v_i] = 5$.

Western est conçu pour évaluer l'évolutivité de $\mathbb{P}$ et $\mathcal{P}$ pour notre domaine d'application des jeux vidéo. Il met en œuvre les routines quotidiennes des PNJs [9] du très populaire [25] jeu vidéo commercial Red Dead Redemption 2 [23]. Il est aussi proche que possible du jeu et est de classe SAS-PUT$_1$ que $\mathcal{P}$ ne peut gérer, avec $m = 29$, $n = 5$, $|\mathcal{A}| = 58$, et 7 classes de PNJ avec 9 objectifs spécifiques. L'`Éleveur de Chevaux` décrit Table 1 est l'une de ces classes de PNJ.

Nous avons conçu une version SAS-PUST$_1$ de Western , qui est une version restreinte, que $\mathcal{P}$ et $\mathbb{P}$ peuvent traiter, avec $m = 19$, $n = 2$, $|\mathcal{A}| = 37$, et 5 classes NPC avec 6 objectifs spécifiques. La figure 3 montre que $\mathbb{P}$ est trente mille fois [6] plus rapide que $\mathcal{P}$ et peut générer des plans pour environ deux millions de NPCs en 1 milliseconde : la planification

6. (Fig. 3) Pour $1ms$, $\mathcal{P}$ produit un plan pour 70 NPCs (abscisse rouge) contre 2,4 millions pour $\mathbb{P}$ (abscisse verte).

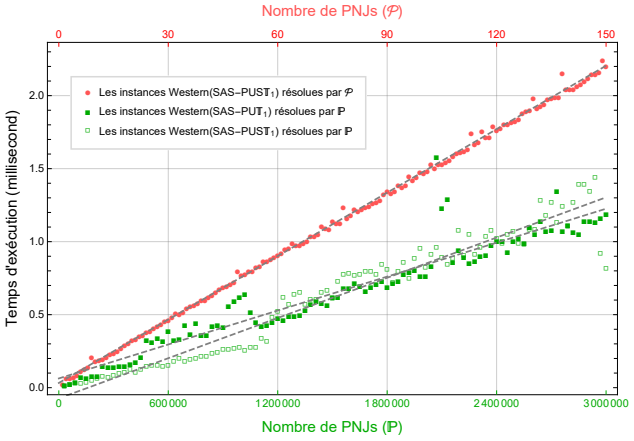

FIGURE 3 – L'abscisse rouge (haut) est l'abscisse du graphe rouge ($\mathcal{P}$) alors que l'abscisse verte (bas) est l'abscisse des deux graphes verts ($\mathbb{P}$). Les plans de solution des instances du problème Western sont longs de 3 à 10 actions : l'impact du nombre d'actions et du nombre de leurs ordonnancements est négligeable sur la génération de chaque plan. Par conséquent, les temps d'exécution de $\mathbb{P}$ et $\mathcal{P}$ sont linéaires par rapport au nombre de PNJs. Cependant, $\mathbb{P}$ construit chaque plan beaucoup plus rapidement que $\mathcal{P}$ et est donc capable de générer des plans pour deux millions de PNJs en moins d'une milliseconde.

avec $\mathbb{P}$ peut être utilisée pour contrôler de grandes villes dans les jeux vidéo commerciaux [15].

4 Discussion

Comme nous l'avons mentionné dans la sous-section 2.1, en conséquence du **Théorème 4.4** [1, p. 76] $\mathcal{P}$ résout les instances des classes de problèmes SAS$^+$-PUST$_2$. Par conséquent, nous pourrions nous attendre à ce que $\mathbb{P}$ soit environ deux fois plus rapide que $\mathcal{P}$ sur des problèmes spécifiquement conçus en évitant de considérer l'insertion de deux actions de même type plutôt qu'une. $\mathbb{P}$ fait évidemment moins de travail que $\mathcal{P}$, mais quel genre de travail ? La restriction (T$_1$) n'explique certainement pas à elle seule pourquoi $\mathbb{P}$ est plus rapide de plusieurs ordres de grandeur que $\mathcal{P}$. La figure 2 montre que $\mathcal{P}$ ne profite pas de la croissance linéaire des ordonnancements entre action car il itère sur les variables d'état et non sur les ordres d'actions comme le fait $\mathbb{P}$. Dans la phase 2 de l'algorithme 3, $\mathbb{P}$ ne vérifie que les prevail-conditions définies grâce au système de voisinage $\mathcal{N}$, réduisant ainsi considérablement la charge travail de $\mathbb{P}$ par rapport à $\mathcal{P}$. Ceci est la clé pour expliquer la très grande efficacité d'exécution de $\mathbb{P}$ pour tous les problèmes que nous avons testés. Nous avons finalement observé que le tri topologique de nos graphes est déterministe comme c'est le cas pour le planificateur de [10] qui utilise des actions unaires sans Single-Valuedness mais avec des variables d'état binaires [7]). Les explications clés mineures concernent l'allocation et le remplissage des structures de données autant que possible à l'avance ; cependant, une gestion rigoureuse de la mémoire n'explique que la régularité des courbes (à la fois $\mathbb{P}$ et $\mathcal{P}$.) dans les figures 1, 2, et 3.

Contrairement aux restrictions SAS$^+$, la restriction (T$_1$) ne limite pas l'entrée de l'algorithme de planification mais sa sortie. Dans le domaine d'application des jeux vidéo tels que Western les plans font moins de 10 actions : il est assez simple de vérifier, même à la main, qu'une instance du problème satisfait la restriction T$_1$; c'est beaucoup plus complexe lorsque les plans contiennent des milliers d'actions. En l'état de cet article, la seule solution est d'exécuter $\mathbb{P}$ en tant que test d'appartenance à la classe du problème. Les temps d'exécution de $\mathbb{P}$ ne sont heureusement pas un obstacle à cette fin.

La restriction (T) limite également la sortie de l'algorithme de planification ; cependant, la recherche de plans totalement ordonnés est toujours possible. Un plan est une solution à un problème de planification lorsque son application transforme l'état initial en l'état but du problème : chaque action est appliquée une par une aux situations successives et donc l'application d'un plan correspond à un ordre total sur ses actions. Par conséquent, la restriction (T) peut être appliquée à toute classe de problèmes SAS$^+$, qui inclut toutes les classes de problèmes de SAS. Il serait logique de limiter l'utilisation de la restriction (T) aux classes de problèmes telles que MultiPrv_2_Cycle dans lesquelles toutes les solutions sont des plans d'actions totalement ordonnés. Cependant, il est également logique d'utiliser la restriction (T) pour spécifier que notre objectif est de générer des plans d'actions totalement ordonnés, comme c'est actuellement le cas dans les jeux vidéo commerciaux où les PNJ exécutent une tâche après l'autre.

Le budget de traitement est probablement la seule contrainte qui empêcherait un moteur de jeu d'accéder à l'état actuel du jeu et donc de lire la valeur exacte des variables d'état du jeu. Par conséquent, les états initiaux et finaux totalement définis constituent une restriction réaliste dans notre domaine d'application des jeux vidéo. Les moteurs de jeu imposent toutefois un taux de mise à jour pour les variables d'état du jeu, ce qui peut faire que des valeurs périmées fassent partie des problèmes de planification. Cette forme rudimentaire d'incertitude [1, p. 64] devrait définitivement faire partie des futurs travaux sur la planification qui gère de grands mondes virtuels.

Une utilisation plus subtile des valeurs indéfinies est en lien avec la restriction (S2) du domaine SAS$^+$, mais qui est assouplie dans le domaine SAS (cf. **Définition 3.2** [1, p. 52]). La restriction (S2) permet à un type d'action de définir une variable d'état qui était auparavant indéfinie : la postcondition de cette action peut définir une variable d'état qui est indéfinie dans la précondition. Il s'agit d'une solution rudimentaire pour gérer le taux de mise à jour des variables d'état du jeu : nous pouvons concevoir un type d'action dont le but est de s'assurer qu'une variable d'état donnée a une valeur ; lors de l'exécution de cette action pendant le jeu, le moteur de jeu attend la prochaine mise à jour de cette variable d'état. Cependant, nous n'avons pas approfondi la restriction (S2) car notre objectif principal dorénavant est de concevoir $\mathbb{P}^+$ pour résoudre les instances de problème de la classe SAS$^+$-PU$\mathbb{T}_1$.

7. Les variables binaires sont également une des restrictions (B) du cadre de planification SAS$^+$ (cf. **Définition 3.9** [1, p. 62].

5 Conclusion

Notre objectif était de générer des plans SAS-PU totalement ordonnés pour des millions de PNJs en temps réel et de sorte que deux actions n'aient pas le même type. À cette fin, nous avons apporté les contributions majeures suivantes au domaine SAS :

- Nous avons défini deux nouvelles restrictions : (1) la restriction (T) qui limite les solutions aux plans totalement ordonnés, et (2) la restriction (T_1) qui limite le nombre de types d'action à un dans toute solution ; nous avons noté $\mathbb{T}_1$ la combinaison de ces deux restrictions.
- Nous avons conçu un algorithme, que nous avons noté $\mathbb{P}$, pour résoudre les instances du problème SAS-PU$\mathbb{T}_1$, en assouplissant ainsi la restriction Single-Valuedness (S).
- Nous avons conçu plusieurs problèmes SAS-PU$\mathbb{T}_1$ pour tester diverses caractéristiques et en particulier diverses complexités temporelles puisque notre objectif est la planification en temps réel ; en particulier, MultiPrv_n_Cycle qui généralise l'exemple du tunnel. Nous avons également conçu un domaine Western réaliste par rapport à notre domaine d'application des jeux vidéo.
- La complexité temporelle de notre algorithme dans le pire des cas est linéaire par rapport au nombre d'actions et à leur ordre entre elles ; les temps d'exécution de notre implémentation de $\mathbb{P}$ sont très rapides pour le domaine Western : $\mathbb{P}$ fournit un plan à deux millions de PNJ en environ une milliseconde, atteignant ainsi notre objectif.

Les travaux futurs prendront d'abord en compte les états initiaux et finaux partiels ; nous souhaitons ensuite aborder le problème SAS$^+$-PU$\mathbb{T}_2$: est-il possible de concevoir un algorithme aussi efficace que $\mathbb{P}$ pour cette classe de problèmes ?

Références

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

[2] 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.

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

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

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

[6] 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.

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

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

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

[10] 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.

[11] 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.

[12] 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.

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

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

[15] 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.

[16] 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.

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

[18] 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.

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

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

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

[22] 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.

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

[24] 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.

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

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

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