Germain Bregeon
Emma Covili
Mathilde Da Cruz

La planification classique

Introduction à l'intelligence artificielle symbolique : Projet

Choix : Enseignant

Licence 3 - IM2D
Université Paris Dauphine
2019-2020

Table des matières

1 Introduction

1.1 Présentation

1.1.1 La planification

Une personne peut tout à fait décider de parler ou bien d'aller faire ses courses sans réfléchir à la façon dont ses actions vont influer sur le monde qui l'entoure. Chaque personne fait cela très naturellement et ceci semble même assez simple. En réalité, cette capacité humaine est très difficile à formaliser. Comment un robot, ou plus généralement une intelligence artificielle, va-t-il choisir la prochaine action à effectuer ? Être capable de choisir ce qui va être fait ensuite est central dans ce qu'on appelle le comportement intelligent. La question de la prochaine action à effectuer entre tout à fait dans le domaine de l'intelligence artificielle, considérant que l'un de ses objectifs est de saisir le fonctionnement de l'intelligence et plus particulièrement ses aspects informatiques.

En IA, on peut distinguer trois différentes approches pour choisir l'action à suivre. Une première dans laquelle l'action est donnée directement par le programmeur (*programming-based*), une seconde dans laquelle l'action est induite par l'expérience, c'est ce qu'on trouve aujourd'hui notamment dans le domaine du *machine learning* (*learning-based*) et enfin une troisième dans laquelle l'action est déduite automatiquement par un modèle d'**actions**, d'**états** et d'**objectifs** (*model-based*). C'est cette dernière approche qui est l'objet de la planification et ces trois concepts en gras sont centraux dans la représentation des problèmes de planification. En effet, la spécification d'un problème de classification passe par la description précise de ces éléments.

En résumé, **la planification est le domaine de l'intelligence artificielle qui traite de l'organisation et du raisonnement concernant les actions**. Son objectif est de développer des planificateurs qui pourront générer une séquence d'actions dans le but d'atteindre un ou plusieurs objectifs définis en amont en partant d'un ou plusieurs états initiaux.

1.1.2 La planification classique

C'est souvent au Stanford Research Institute (SRI) qu'on attribue les premières tentatives en planification. En 1963, Charles Rosen, chercheur en réseaux de neurones au SRI, propose le développement d'un automate mobile qui combinera plusieurs domaines de l'intelligence artificielle tels que la reconnaissance de formes et les réseaux de neurones. Il fit en avril 1964 une proposition au Département de la Défense, en Recherche et Ingénierie (DDR&E) pour « la recherche pour un automate intelligent » qui devrait « finalement aboutir au développement de machines qui effectueront des tâches qui sont actuellement considérées comme nécessitant une intelligence humaine »[1]. C'est ainsi qu'a été développé entre 1966 et 1972 le robot Shakey. D'après le site Internet du SRI, Shakey pouvait «

[1] "ultimately lead to the development of machines that will perform tasks that are presently considered to require human intelligence" , Nils J. Nilsson (2009, section 12.1)

accomplir des tâches ayant recours à la planification, à la recherche d'itinéraire, ou encore au réarrangement d'objets simples. » [2]

Ces premiers travaux ont dominé jusqu'au milieu des années 1980 et ont été la base de la majorité des recherches en planification. C'est pour cette raison que les chercheurs réfèrent cette approche comme celle de **la planification classique**. Bien que depuis les premières recherches au début des années 60, les ambitions aient été revues à la baisse, on a observé de considérables progrès en planification classique, aussi bien concernant la richesse de modélisation des problèmes, que l'efficacité des planificateurs.

Nir Lipovetzky (2014, p.3) définit la planification classique comme « le problème de trouver une séquence d'action qui fait coïncider un état initial donné à un objectif, dans lequel l'environnement et les actions sont déterministes »[3].

En effet la planification classique repose sur plusieurs **hypothèses simplificatrices** :

1. L'environnement est représenté de manière étatique. On peut voir cela comme si l'on prenait une photo à un instant précis et que l'environnement était décrit exactement et entièrement tel que sur cette photo (et donc ne change pas sans action de l'agent). Ces **états** sont décrits par des phrases, des axiomes, des contraintes. Cette hypothèse trouve ses limites lorsqu'il s'agit de décrire des processus continus.
2. Les actions sont déterministes. Effectivement l'agent qui va effectuer les actions doit savoir exactement quels seront les nouveaux états de l'environnement après chaque action qui peut être effectuée. Ainsi, il connaît parfaitement son environnement à n'importe quel moment de son plan d'action. Une **action** peut nécessiter des préconditions, *i.e.* des propriétés qui doivent être vérifiées pour pouvoir effectuer cette action, et peut avoir des effets. Ces préconditions et effets sont décrits au préalable.
3. On se donne un ou plusieurs **objectifs** "binaires" définis *a priori*. Cet objectif est en fait un ensemble d'états qui doivent être vérifiés lorsque toutes les actions du plan auront été effectuées. A un instant donné, un objectif est soit vérifié, soit non vérifié.

Notons qu'une description précise de l'environnement est très importante puisque le résultat réel (c'est à dire l'état de l'environnement après que toutes les actions aient été effectuées) de la planification ne sera conforme à l'objectif que si la représentation de l'environnement reflète suffisamment bien cet environnement réel. Aujourd'hui, plusieurs domaines de la planification permettent de pallier les problèmes engendrés par les simplifications en planification classique. La planification dans l'incertain notamment, permet d'intégrer des actions à effet probabiliste et permet donc une modélisation plus précise de l'ensemble des actions possibles. Elle permet aussi d'intégrer des fonctions d'utilité sur les buts, ce qui fera intervenir la théorie de la décision. Tout cela permet effectivement une modélisation plus fidèle du monde réelle, mais aussi bien plus complexe à étudier. Plus le degré de prise en compte des détails de l'environnement sera élevé, plus le processus de

[2] "Shakey could perform tasks that required planning, route-finding, and the rearranging of simple objects"

[3] "Classical planning is the problem of finding a sequence of actions that maps a given initial state to a goal state, chere the environment and the actions are deterministic."

planification sera difficile. D'autres paramètres influent aussi sur la difficulté du problème, tels que le temps et le coût (en matière première, en argent, etc.).

La solution d'un problème de planification classique, **le plan**, est une séquence **ordonnée** d'actions générées par le planificateur. Chaque action pouvant potentiellement modifier l'état de l'environnement, l'ordre donné par le planificateur est important. Ce plan d'action doit pouvoir être effectué depuis l'état initial et mener à l'état objectif. Ce plan est **correct** si tous les buts sont satisfaits.

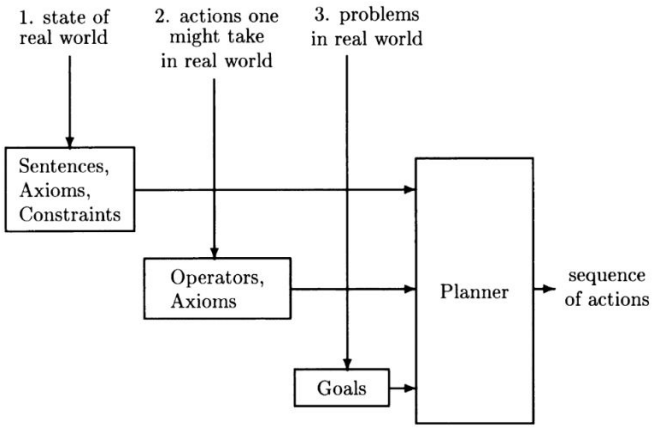

Figure 1.1 "The classical planning problem" extrait de l'ouvrage *Practical Planning: Extending the Classical AI Planning Paradigm(1988, p.4)*.
Ici, le terme *real world* pour désigner l'environnement indique l'éventuel objectif de représenter le monde réel dans son entière complexité. Les flèches verticales indiquent une relation de représentation.

1.2 Motivation

La recherche en planification automatique est motivée par des aspects aussi bien pratiques que théoriques.

Elle peut tout d'abord être très utile pour des tâches complexes qui impliquent des exigences changeantes - notamment en terme de sécurité ou d'efficacité - ou encore des tâches qui implique l'interaction de plusieurs agents (à moins d'une coordination explicitée en amont, il est nécessaire pour chaque agent de prendre en considération les autres agents ainsi que les effets de leurs actions pour pouvoir faire leur propre plan d'action). La planification permet de bénéficier de plus d'autonomie, d'adaptabilité et de capacités de contrôle pour ces tâches complexes.

Au début, les outils de planification se concentrent plutôt sur des problèmes imaginaires (l'exemple le plus courant est celui du Blocks world) mais plus récemment on trouve divers domaines d'utilisation pratique pour des problèmes réels, notamment concernant la robotique, la gestion de projets, la navigation sur internet, la logistique, le domaine médical ou encore l'aérospatial et la défense - ces derniers étant les premiers domaines à avoir bénéficié de la planification. C'est par ailleurs grâce à la planification qu'un ordinateur est devenu champion du monde de Bridge en juillet 1997.[4]

La planification a aussi des motivations théoriques qui sont en fait celles de l'intelligence artificielle, en particulier saisir le fonctionnement de l'intelligence, comme

[4] Stephen J. J. Smith, Dana Nau, et Tom Throop, " Computer Bridge : A Big Win for AI Planning", *AI Magazine*, 19 : 93-106, 1998

évoqué plus haut. L'étude de la planification cherche à répondre à des questions concernant la rationalité individuelle et collective, et la capacité à délibérer. On peut aussi trouver une motivation plus philosophique à la planification, qui consiste simplement à mieux se comprendre en comprenant mieux nos manières d'agir et donc le raisonnement derrière nos actions.

Finalement, les utilisations pratiques et théoriques de la planification sont assez liées et on peut même voir cela d'une manière circulaire : les observations réelles de ce qui fonctionne bien peuvent être une source de travail et d'inspiration pour développer les théories de la planifications, et ces théories peuvent ensuite servir à leur tour à améliorer la planification lors d'utilisations pratiques, et ainsi de suite.

1.3 Notre exemple

Figure 1.2 Illustration de notre exemple.

Pour illustrer les notions abordées tout au long du cours, nous décrivons ci dessous un exemple d'un modèle qui sera utilisé pour nos problèmes de planification. Nous utilisons ici notre propre langage. L'idée est simplement de comprendre les différents concepts. Ceux-ci seront repris de manière plus formelle dans les différents exemples qui suivront.

Prenons un tramway qui peut se déplacer entre différentes stations, et ouvrir ou fermer ses portes à chaque station. Il sera donc notre agent.

Les prédicats disponibles sont les suivants : `Marche`, `Vers(A,B)`, `Vers(B,C)`, `Vers(C,B)`, `Vers(B,A)`, `ArretStation(A)`, `ArretStation(B)`, `ArretStation(C)`, `PortesFermees`, `PortesOuvertes`, `Station(A)`, `Station(B)`, `Station(C)`

L'état initial est alors : `ArretStation(A)`, `PortesFermees`, `Vers(A,B)`, `Vers(B,C)`, `Vers(C,B)`,`Vers(B,A)`

L'état objectif est : `ArretStation(C)`,`PortesFermees`

Les actions possibles sont :

- `ouvrirPortes`
  - Précondition : Il faut que les portes soient fermées et que le tramway soit arrêté à une des stations du réseau.
  - Effet : Les portes s'ouvrent.
- `fermerPortes`
  - Précondition : Il faut que les portes soient ouvertes et que le tramway soit arrêté à une station.
  - Effet : Les portes se ferment.

- `avancer(X,Y)`
    - Précondition : Il faut que les portes soient fermées, que le tramway soit arrêté à une station et il faut que les deux stations soient adjacentes (à l'aide du prédicat vers(X,Y)).
    - Effet : Le tramway est en marche et va vers la station Y.
- `arreter(X)`
    - Précondition : Il faut que les portes soient fermées, que le tramway soit en marche et qu'il aille vers la station X.
    - Effet : Le tramway est arrêté à la station X.

2 Représenter un problème de planification classique

Les problèmes de planification classique sont exprimés dans un langage concis d'opérateurs et de transformateurs d'état, souvent proche de la logique propositionnelle.

La difficulté pour représenter un problème de planification est de trouver le juste équilibre entre l'efficacité du planificateur, et la complexité du modèle utilisé : il faut que la modélisation soit suffisamment précise pour que le planificateur soit efficace, tout en étant la moins complexe possible pour qu'une mise en oeuvre viable soit possible. C'est d'ailleurs là tout l'enjeu de la planification classique et de ses restrictions vues précédemment.

Il existe aujourd'hui de nombreux langages permettant de représenter un problème de planification classique. Dans cette partie, nous ne pourrons explorer que trois langages en particulier - le STRIPS, le PDDL et le situation calculus - mais parmi les plus connus nous pouvons citer notamment le NOAH (Sacerdoti, 1977) pour le contrôle en ingénierie mécanique, le NONLIN (Tate, 1977), DEVISER (Vere, 1990) utilisé pour la planification du satellite Voyager, et SIPE (Wilikins).

2.1 La planification STRIPS

2.1.1 Présentation

Le système STRIPS, **ST**anford **R**esearch **I**nstitute **P**roblem **S**olver, formalisé par Richard Fikes et Nils Nilsson au début des années 1970, est considéré comme fondateur en planification classique puisque c'est celui utilisé pour le contrôle du robot Shakey, et que l'essentiel des recherches en planification classique repose encore aujourd'hui sur ces travaux. Ce planificateur fonctionne en exécutant un domaine et un problème pour trouver un objectif. Le langage se compose de trois éléments, les états, les buts et les actions ; qui sont des éléments communs à beaucoup d'autres langages.

- Les états de l'environnement sont décrits par une série de prédicats utilisant la logique du premier ordre. Un état est défini comme une conjonction de littéraux positifs (état initial de notre tramway : `Marche(T)`, `Vers(A,B)`, `Vers(B,C)`, `Vers(C,B)`, `Vers(B,A)`, `ArretStation(T, A)`, `PortesFermees`, `PortesOuvertes(T)`, `Station(A)`, `Station(B)`, `Station(C)`, `Tramway(T)`). Les différents littéraux décrivant les états, doivent être instanciés (il

ne peut pas y avoir de variable dans les littéraux) et ils ne doivent pas contenir de fonction.

- Les buts sont représentés comme des états partiellements spécifiés, donc qui doivent être composés de littéraux instanciés positifs et sans fonction. L'état peut être partiellement spécifié, ce qui veut dire que dès lors qu'un état *e* contient tous les atomes d'un but *b* alors on peut dire que *e* satisfait le but *b*.
- Une action est définie sous forme d'opérateur en deux parties, d'abord la précondition de l'action et l'effet de l'action sur l'état. Ces deux parties forment une proposition. Comme pour un état, la précondition est une conjonction de littéraux positifs dénués de fonction et qui doivent être vérifiés dans un état avant que l'on puisse exécuter l'action. Toute variable que l'on trouve dans la liste des conjonctions de la précondition ou de l'effet, doit aussi apparaître dans la liste des paramètres de l'action. Pareillement, l'effet est une conjonction de littéraux, libre de fonctions, décrivant comment l'état change quand l'action est exécutée. Un littéral positif L^+ dans l'effet sera vrai dans l'état que l'on obtient après exécution de l'action, symétriquement, un littéral négatif L^- dans l'effet sera faux dans l'état suivant. L'action `ouvrirPortes` est décrit de la manière suivante :

    - `ouvrirPortes(X,T)` :
        - **Précondition :** `Tramway(T)` ∧ `Station(X)` ∧ `PortesFermees` ∧ `ArretStation(X)`
        - **Effet :** `PortesOuvertes(T)`

Avec STRIPS tous les littéraux qui ne sont pas mentionnés dans l'effet ne changent pas d'un état à un autre (les littéraux positifs restent positifs et les littéraux négatifs restent négatifs).

2.1.2 Avantages et inconvénients

La terminologie STRIPS est intéressante de par sa simplicité permettant de représenter facilement de nombreux problèmes de planification et permet déjà de décrire des problèmes assez complexes. STRIPS est à l'origine d'autres langages de description tels que ADL (*Action Description Language*) qui permet notamment l'ajout de préconditions négatives, d'effets conditionnels, etc. Mais la partie résolution du problème (que l'on décrit plus loin) peut parfois être longue si le programme doit parcourir tous les états possibles, l'ensemble de ces derniers pouvant être immense. Les principaux défauts de STRIPS sont ceux de la planification classique cités précédemment (dus aux hypothèses simplificatrice). Depuis STRIPS, de nombreux progrès ont été réalisés en planification classique, notamment avec le planificateur GraphPlan (Blum et Furst, 1995) dont les performances ont permis d'envisager des applications réelles des algorithmes de planification STRIPS à moyen terme. Malheureusement, nous n'aurons pas le temps d'étudier GraphPlan dans ce travail.

2.2 PDDL

2.2.1 Présentation

Le PDDL est devenu au fil du temps un des standards pour la description de problème de planification. Il s'inspire notamment de STRIPS et a été développé à la fin des années 90 par Drew McDermott et ses collègues. Les composantes d'un problème de planification sont donc très similaires à ceux d'un problème STRIPS, dans le sens où l'on a : des objets (ce qui nous intéresse dans l'environnement dans lequel on se trouve) ; des prédicats (qui nous permettent de définir les propriétés des objets que l'on observe) ; l'état initial (ce que l'on sait au début du problème) : l'objectif (là où on veut arriver) ; et les actions (qui sont spécifiées comme en STRIPS avec une partie précondition et une partie effet). Les problèmes de planification que l'on décrit en PDDL sont séparés en deux fichier, un fichier *domain* qui contient tous les prédicats et les actions, et un fichier *file* qui contient tous les objets, l'état initial et l'objectif.

Le fichier *domain* se décompose ainsi :

```
1 (define (domain Tram)
2     ( :predicates (Tramway ?t) (Station ?s) (Vers ?t ?s1 ?s2)
3                   (ArretStation ?t ?s) (Marche ?t) (PortesFermees
4                   ?t) (PortesOuvertes ?t) )
5     ( :action avancer : parameters (?x ?y ?t)
6          :precondition (and (Station ?x) (Station ?y) (Tramway
?t)
7                         (PortesFermees ?t)
8                         (ArretStation ?t ?x)
9                         (Vers ?x ?y)
10         :effect (and (marche ?t) (not(ArretStation ?t ?x)))...)
```

Le but dans ce fichier est de donner tous les prédicats que l'on va rencontrer dans notre programme. Lignes 2-3, on observe le code utilisé pour décrire les prédicats, on note que le prédicat `(Tramway ?t)` est vrai si et seulement si l'objet t est un tramway. Puis, comme en langage STRIPS, on a la description de toutes les actions avec le partie précondition et la partie effet.

Le fichier problem se présente ainsi :

```
1 (define (problem Parcours-Tramway)
2     ( :domain Tram)
3     ( :objects A B C T)
4     ( :init (Station A) (Station B) (Station C) (Tramway T)
5         (Vers A B) (Vers B C) (Vers C B) (Vers B A)
6         (ArretStation T A) (PortesFermees T) )
```

```
7     ( :goal (ArretStation T C)) )
```

On retrouve donc bien les différents objets que l'on va utiliser, l'état initial du problème et l'objectif que l'on cherche à atteindre. Le nom de la partie domain doit être identique au domain du fichier domain.

2.2.2 Avantages et inconvénients

PDDL est un des langage les plus communs aujourd'hui dans son champ d'application, un de ses avantages principaux est qu'il rend le type des objets et des paramètres très explicite, ce qui évite les erreurs de modélisation. C'est également un langage très concis qu'il est facile de prendre en main. Mais d'un autre côté le fait que les prédicats soient statiques peut être gênant si on veut définir un objet comme ayant deux types, il n'est pas précisé comment le déclarer en PDDL ou tout simplement s'il est possible de le faire.

2.3 Situation Calculus

2.3.1 Présentation

Le langage de représentation situation calculus est un des formalismes les plus utilisés pour le raisonnement sur les modèles dynamiques de nos jours. Il a été introduit en 1991 par Ray Reiter. Le situation calculus se base sur la logique de premier ordre, il est composé de trois éléments de base qui sont:

- Les **actions** : elles représentent les différents changements d'état que l'on peut effectuer dans notre environnement comme l'action `ouvrirPortes` dans notre exemple. Une action peut prendre en argument des variables, des constantes, des objets...
- Les **situations** : elles permettent de décrire l'état de notre environnement. Une situation représente l'historique des actions qui ont été effectuées depuis notre situation initiale. La situation initiale, traditionnellement notée S_0, représente l'environnement avant qu'aucune action n'ait été effectuée. Après l'exécution d'une action, la nouvelle situation est dénotée : `do (action que l'on veut effectuer (argument(s) de l'action),` S_x`),` S_x représentant la situation de l'environnement avant l'action. Le résultat de "do" est la nouvelle situation après avoir effectué l'action sur S_x. Dans notre exemple, on a : S_0 = `ArretStation(A), PortesOuvertes, Vers(A,B), Vers(B,C), Vers(C,B), Vers(B,A)`; une autre situation est `do(fermerPortes,` S_0`),` qui est la situation que l'on obtient quand on applique l'action `fermerPortes` à la situation S_0 avec les fluents en plus ou en moins dû à l'action `fermerPortes`.
- Les **fluents** : ils sont utilisés pour décrire ce qui est vrai à une situation donnée. En effet au cours de l'exécution, certains prédicats et fonctions peuvent changer de valeur. Dans notre exemple, `PortesOuvertes(`S_x`)` peut être vrai ou faux en fonction d'où on en est dans le programme. Essentiellement, un fluent est un prédicat ou une fonction qui prend une situation en argument et dont la valeur change en fonction des situations.

En situation calculus, la description de la dynamique de l'environnement (ce qui est autorisé ou non, et quels sont les effets des actions) est décrit grâce à la logique du second ordre, et trois types de formules :

- Les formules à propos des actions : Certaines actions ne peuvent être effectuées, que si certaines conditions sont vérifiées. C'est pour cela que l'on utilise le prédicat spécial `Poss(a, S)` : qui nous dit que l'on peut exécuter l'action `a` dans la situation `S`. On donne les conditions que `Poss(a,S)` doit vérifier, pour pouvoir exécuter l'action `a` dans la situation `S` de la sorte :
  `Poss (a,S) ↔ (les différents fluents de notre environnement)`
  Par exemple pour l'action `avancer(X, Y)` de notre tramway, avant de l'exécuter, il faut vérifier la partie précondition décrit dans la partie [1.4] de ce document. En situation calculus cette précondition se formalise ainsi (à partir d'une situation S_x inconnue):
  `Poss(avancer(X, Y), S`$_x$`) ↔ [PortesFermées ∧ ArretStation(X) ∧ Vers(X,Y)]`
- Les formules à propos de l'état de l'environnement à une situation donnée : Les axiomes de successions des états permettent de donner l'état des fluents à une situation donnée en fonction de la situation précédente et de l'action que l'on effectue sur celle-ci. On les notes de la manière suivante :
  `Fluent (args du Fluent, do (a, S`$_x$`)) ↔ [conds(a, S`$_x$`)]`
  où `conds(a, S`$_x$`)` représente les conditions sur les fluents à l'état S_x qui doivent être vérifiés et avec quelle(s) action(s) ils doivent être exécuter pour que `Fluent (args du Fluent, do (a, S`$_x$`))` soit vrai à la situation `do (a, S`$_x$`)` ; `a` étant l'action que l'on effectue à la situation S_x pour arriver à la situation S_{x+1} = do (a, S_x). On prend comme exemple : `ArretStation(A, do (a, S`$_x$`))` (1) de notre tramway.
  `ArretStation(A, do (a, S`$_x$`)) ↔ [a = arreter(A)]` donc pour que le fluent (1) soit vrai, il faut que l'action que l'on effectue sur la situation précédente soit l'action `arreter(A)`. On décrit tous les fluents d'un problème de la sorte.
- Les axiomes fondamentaux : Les axiomes fondamentaux du situation calculus, que nous n'allons pas donner mais simplement décrire, permettent d'assurer que l'espace des situations est représenté par un arbre avec comme racine S_0, les noeuds représentant des situations et les arêtes les actions à effectuer pour passer d'une situation à une autre.

Le GOLOG qui est un langage de programmation logique de haut-niveau, développé par l'université de Stanford, se base entièrement sur le situation calculus, pour la spécification et l'exécution d'actions complexes dans un domaine dynamique.

2.3.2 Avantages et inconvénients

Le situation calculus est un des langages les plus connus pour le raisonnement sur les modèles dynamiques, ce qui le rend très courant de nos jours. Les situations partagent un grand nombre de propriétés avec un environnement possible, ces propriétés sont notées

de manière axiomatique sans sémantique particulière, ce qui est utile quand on est seulement intéressé par ce qu'une théorie implique.

Plusieurs scientifiques considèrent les situations de notre langage comme étant un arrêt sur image de notre monde à un moment donné, situation qui ne pourrait pas être décrite complètement. D'autres scientifiques décrivent une situation comme étant les noeuds d'un arbre avec pour racine la situation initiale, et d'autres encore considèrent S_0 comme étant juste un nom d'une situation dont les conséquences ont un intérêt. Le situation calculus se base donc sur une définition abstraite de la notion de situation. Cette définition peut entraîner des problèmes au moment de l'évolution de la situation. Si des nouveaux axiomes sont à définir, est ce que l'on peut les définir facilement, ou devons nous obligatoirement rester avec nos axiomes de départ, alors que l'environnement dans lequel nous vivons évolue tous les jours. Un autre problème du situation calculus est la représentation du temps : il n'est pas aisé d'introduire des opérateurs temporels.

3 Différents algorithmes de planification

3.1 La planification *State space*

La planification *State space* est utilisée dans la conception d'algorithme pour trouver des données ou des solutions aux problèmes posés. Elle permet de définir quelles parties de l'espace des états, ou *state space* en anglais, le programme va étudier et dans quel ordre.

Pour un agent, l'ensemble des états possibles s'appelle l'**espace des états**. Cela s'apparente à une description de toutes les combinaisons possibles de l'environnement. Ici, l'espace des états est {`Marche, Vers(A,B), PortesFermees; Marche, Vers(B,C), PortesFermees; ArretStation(A), PortesOuvertes`...}.

L'idée de la planification *State space* est de parcourir ou d'explorer l'espace des états pour trouver le meilleur plan afin d'arriver à l'état final désiré en fonction de l'espace des états initiaux. Les caractéristiques de l'espace des états impactent donc les actions possibles. Le tramway ne peut pas ouvrir ses portes s'il est en marche. En considérant ainsi l'ensemble des situations possibles pour l'agent, on peut déterminer les actions qu'il peut entreprendre ainsi que les effets de ces actions en terme de changement d'état.

Pour déterminer comment arriver à l'état final désiré, la planification State-space nécessite donc un espace des états, des actions et un test de but. En effet, pour s'arrêter, l'algorithme doit comparer l'état dans lequel il se trouve avec celui auquel il doit parvenir. La création du plan peut être implémentée à l'aide de différents algorithmes que nous allons présenter.

3.1.1 Chaînage avant

Le chaînage avant, ou *forward chaining*, est un algorithme récursif qui consiste en la génération successive à partir de l'état initial, des nouveaux états à partir des opérateurs dont les préconditions sont vraies dans l'état courant, jusqu'à obtenir un état correspondant à l'état but recherché. On parle d'un algorithme *goal-driven*, c'est à dire qu'on part des données pour accomplir notre but.

Une action A est **applicable** sur l'état S si et seulement si les préconditions de A sont satisfaites dans S.

Pour construire l'état S' après avoir appliquer l'action A, on utilise les listes d'ajout et de retrait de A, tout en tenant compte de la liaison des variables de A dans S. Les listes d'ajout de A correspondent aux littéraux de l'effet de A qui seront vrais après avoir appliqué A. Les listes de retraits de A correspondent aux littéraux de la précondition de A qui seront faux après avoir appliqué A.

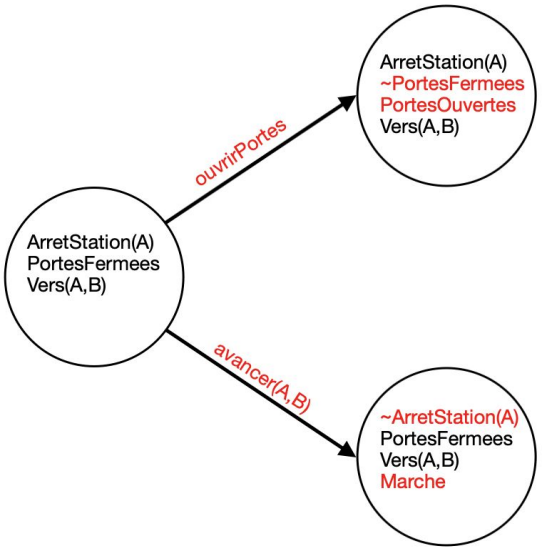

Figure 3.1 : Illustration du fonctionnement de l'algorithme de chaînage avant

Dans notre exemple, on a `S={ArretStation(A), PortesFermees, Vers(A,B)}`. Pour appliquer l'opérateur `ouvrirPortes` les préconditions sont `PortesFermees` et `ArretStation(X)`; ainsi on peut dériver de l'état S l'état S' en appliquant l'action `ouvrirPortes` car les préconditions de `ouvrirPortes` sont dans l'état S. On obtient ainsi l'état `S'={ArretStation(A), ~PortesFermees, PortesOuvertes, Vers(A,B)}`.

On peut aussi appliquer `avancer(A,B)` à l'état `S`. En effet, l'opérateur `avancer(X,Y)` nécessite les préconditions `ArretStation(X)`, `PortesFermees` et `Vers(X,Y)` toutes trois présentes dans l'état `S` (où `X` est lié à `A` et `Y` à `B`). On crée alors l'état `S'` en ajoutant l'état `Marche` et en négationnant l'état `ArretStation(A)`.

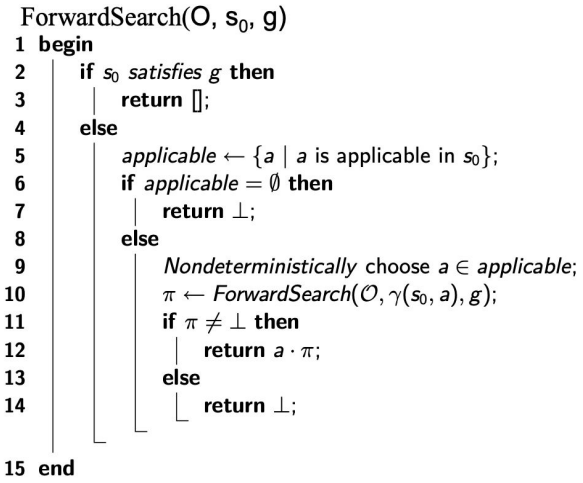

Figure 3.2 : Algorithme de chaînage avant

Dans l'algorithme général de chaînage avant (Figure 3.2), la fonction `ForwardSearch` prend en paramètre `O` la liste des opérateurs, s_0 l'état initial et `g` le but.

Si s_0 satisfait `g`, la fonction renvoie une liste vide, en effet il n'y a pas besoin de créer un chemin puisque l'état initial correspond déjà à l'état final. C'est la condition d'arrêt de la récursivité de l'algorithme.

Sinon, l'algorithme va créer un ensemble des actions `a` applicables dans s_0. Si cet ensemble est vide alors il n'y a aucun moyen de trouver un chemin, la fonction renvoie un échec. Sinon, l'algorithme choisit de manière non déterministe une action dans `applicable`, définit π, un plan d'action, en appelant de nouveau la fonction `ForwardSearch` avec comme nouvel état initial $\gamma(s_0, a)$ où γ correspond à l'application de l'action `a` sur l'espace de départ s_0, en effet on va créer s_1 en appliquant les listes d'ajout et de retraits de l'action `a` à l'espace s_0. Si π est trouvé, la fonction renvoie la concaténation de π et de l'action `a`. Sinon, elle renvoie un échec.

L'avantage du chaînage avant est que l'algorithme est complet c'est à dire qu'il explore toute les possibilités offertes par l'espace des états, par conséquent le facteur de ramification est très important, du fait de l'exploration systématique de toutes les possibilités, donc on a un algorithme avec une complexité très forte.

3.1.2 Chaînage arrière

Le chaînage arrière, ou *backward chaining*, est un algorithme récursif qui prend l'état final comme état initial, trouve les opérateurs qui peuvent ajouter cet état final à l'espace d'état et considère les préconditions de ces opérateurs comme les nouveaux états finaux à atteindre. On parle d'un algorithme *data-driven* : on part du but pour essayer de retrouver l'état de départ.

Une action A est dite **pertinente** pour une conjonction de buts G si elle satisfait l'un des buts de G. Une action A est dite **consistante** pour une conjonction de buts G si elle ne détruit pas l'un des buts de G.

Étant donné une conjonction de buts G, on va choisir une action A pertinente et consistante pour G, retirer de G tout effet positif de A et ajouter dans G tout littéral de la précondition de A s'il n'y est pas déjà.

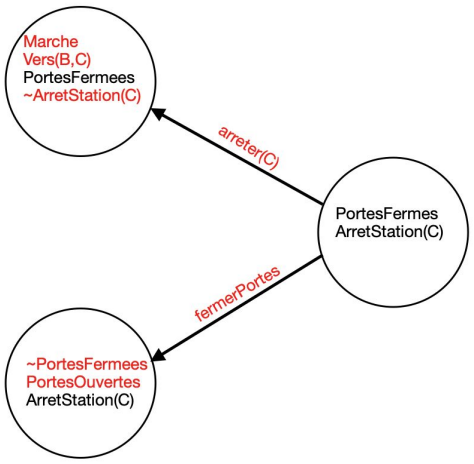

Figure 3.3 : Illustration du fonctionnement de l'algorithme de chaînage arrière

Dans notre exemple, on a `G={PortesFermees, ArretStation(C)}`. L'opérateur `arreter(C)` a comme effet `ArretStation(C)`, présent dans `G`, on peut donc utiliser cet opérateur. Sa précondition est `PortesFermees, Marche` et `Vers(B,C)`. Ainsi, ces littéraux vont être ajoutés à notre nouvel état et devenir les nouveaux sous-buts à atteindre.

À partir de `G`, on peut aussi utiliser l'opérateur `fermerPortes` qui a comme effet `PortesFermees`, présent dans `G`. Sa précondition est `PortesOuvertes` et `ArretStation(X)`. Ces littéraux vont donc être ajoutés à notre nouvel état et devenir le nouveau but à atteindre.

Figure 3.4: Algorithme de chaînage arrière

backwardSearch(O,s_0,g)

Action a is relevant for g iff $g \cap effect^+(a) \neq \emptyset$ et $g \cap effect^-(a) = \emptyset$:

$$\gamma^{-1}(g,a) = (g - effect(a)) \cup precond(a)$$

```
1  begin
2      if s0 satisfies g then
3          return [];
4      else
5          relevant ← {a | a is relevant in g};
6          if relevant = ∅ then
7              return ⊥;
8          else
9              Nondeterministically choose a ∈ relevant;
10             π ← BackwardSearch(O, s0, γ⁻¹(g, a));
11             if π ≠ ⊥ then
12                 return π · a;
13             else
14                 return ⊥;
15 end
```

Dans l'algorithme général du chaînage arrière (Figure 3.4) la fonction `BackwardSearch` prend en paramètre `O` la liste des opérateurs, s_0 l'état initial et `g` le but.

Si s_0 satisfait `g`, la fonction renvoie une liste vide. En effet il n'y a pas besoin de créer un chemin puisque l'état initial correspond déjà à l'état final. C'est la condition d'arrêt de la récursivité de l'algorithme.
Sinon, l'algorithme va créer un ensemble des actions `a` pertinentes dans `g`. Si cet ensemble est vide alors il n'y a aucun moyen de trouver un plan, la fonction renvoie un échec. Sinon, l'algorithme choisit de manière non déterministe une action dans `relevant`, définit π, un plan d'action, en appelant de nouveau la fonction `BackwardSearch` avec comme nouvel état final γ^{-1}`(g , a)` où γ correspond à l'application inverse de l'action `a` sur l'espace de départ `g`. En effet on va créer `g'` en appliquant les listes d'ajout et de retrait de l'action `a` à l'espace `g`. Si π est trouvé, la fonction renvoie la concaténation de π et de l'action `a`. Sinon, elle renvoie un échec.

L'avantage du chaînage arrière est que le facteur de ramification est faible, bien plus faible que celui du chaînage avant, puisqu'on explore moins de possibilités. Cela garantie une baisse de complexité. Cependant l'algorithme n'est pas complet, on peut trouver des incohérences dans le développement de plans.

3.1.3 D'autres algorithmes

Ces deux algorithmes, bien que souvent utilisés notamment par le langage STRIPS, ne sont pas les seuls disponibles. En effet, on peut trouver une version de chaînage avant et arrière pour un plan totalement ordonné. Il existe aussi des algorithmes de recherche heuristique systématique avec garanties d'optimalité, par exemple l'algorithme A* et ses variantes IDA* et WA* ; et des algorithmes de recherche heuristique systématique sans garanties d'optimalité, par exemple l'algorithme de recherche standard "meilleur-premier" qui est comme A* mais qui ignore la composante "coût-jusque-là" de la fonction d'évaluation ou encore des algorithmes de recherche non systématiques incomplets, notamment les algorithmes de recherche stochastique.

3.1.4 Limites de la planification State space

La planification classique résout les sous-buts dans un ordre donné. Le problème est que cela amène parfois à détruire ce qui a déjà été construit. Ce phénomène est connu sous le nom d'**anomalie de Sussman**.

Supposons qu'un individu pieds nus doive se retrouver dans l'état où il porte sa chaussure droite, sa chaussure gauche, sa chaussette droite et sa chaussette gauche. S'il cherche à réaliser les buts dans l'ordre de l'énoncé, il échouera. En effet, il sera obligé de défaire sa chaussure droite pour pouvoir mettre sa chaussette droite. Idem pour la gauche. En revanche l'exécution du plan pour la gauche est indépendante de l'exécution pour la droite. Le plan global est donc partiellement ordonné.

Pour résoudre ce type de problème, on ne peut pas utiliser la planification State space car elle impose un ordre d'exécution des actions.

3.2 D’autres méthodes de planification

Devant les limites rencontrées avec la planification State space, d’autres algorithmes de planification ont été mis en place.

La méthode qui mélange le chaînage avant et arrière trouve un plan totalement ordonné avec une séquence d'actions strictements linéaires et un ordre chronologique de planification. Ce type de planification est trop rigide, comme on l’a vu précédemment, c’est pour cela qu’on va utiliser une méthode de recherche de plan plus souple : on va résoudre les sous-buts et combiner les sous-plans. C’est la planification **Plan space**. Dans la planification Plan space, un programme parcourt un espace de plans, à la recherche d'un plan qui le fera passer de son état initial à l’état final.

On peut utiliser la stratégie du **moindre engagement**, ou *least commitment*, qui consiste à se compromettre uniquement sur les aspects pertinents des actions, en laissant les autres aspects à plus tard. Ce genre de programmation est beaucoup utilisée dans certains jeux, notamment le bridge. Un planificateur qui peut placer deux actions dans un plan sans spécifier laquelle intervient en premier est appelé **planificateur en ordre partiel**, comme NOAH ou POP. La planification en ordre partiel (POP) permet à un planificateur de construire des plans qui ne sont que partiellement ordonnés et donc seulement assez complets pour atteindre son objectif. Un tel plan laisse à l'agent qui l'utilisera autant de souplesse que possible au “moment de l'exécution". Les POP répondent à trois idées clés : les états et les opérateurs sont décomposables, ils peuvent ajouter une action au plan à n'importe quel endroit, ils décomposent un problème en sous-tâches, ils résolvent séparément et ré-assemblent les solutions. La solution est présentée comme un graphe d’actions.

La planification en ordre partiel semble assez naturelle, puisqu’elle est très proche de la manière dont une personne peut établir un plan, petit à petit, en résolvant un par un les sous-problèmes. Cependant cette approche naturelle est très lente. La structure de l’espace de recherche reste assez compliquée à comprendre. C’est notamment pour cette raison que des chercheurs ont décidé de développer une méthode de résolutions dans les graphes avec les algorithmes qu’ils connaissaient déjà : le planificateur GraphPlan évoqué précédemment, qui reste aujourd’hui largement utilisé en planification.

4 Bibliographie

Nous avons mis dans la bibliographie et sitographie non seulement les ouvrages et sites que nous avons mentionné lors de notre travail, mais aussi ceux dont nous nous sommes le plus servis.

Baki, *Planification et Ordonnancement Probabilistes sous Contraintes Temporelles*, sous la direction de Abdel-Illah Mouaddib, Université de Caen, 2006

Bonet et Geffner, *A Concise Introduction to Models and Methods for Automated Planning*, Morgan and Claypool, 2013

Ghallab, Nau, Traverso. *Automated planning : Theory and practice*. Morgan Kaufmann Publishers, 2004

Hayes et McCarthy, “Some philosophical problems from the standpoint of artificial intelligence”, *Machine Intelligence*, 4:463–502, 1969

Lavalle, *Planning Algorithms*, Morgan Kaufmann Publishers, 2006

Lipovetzky, *Structure and Inference in Classical Planning*, AI Access, 2014

Nau, Smith et Throop, “ Computer Bridge : A Big Win for AI Planning”, *AI Magazine*, 19 : 93-106, 1998

Nilsson, *The Quest for Artificial Intelligence*, Cambridge University Press, 2009

Nilsson, *Principles of Artificial Intelligence*, Springer Verlag, 1980

Wilkins, *Practical planning : Extending the Classical AI Paradigm*, Morgan Kaufmann Publishers, 1988

5 Sitographie

Bouzy, *Intelligence Artificielle Planification*, consulté sur http://helios.mi.parisdescartes.fr/~bouzy/Doc/IAL3/09_IA_planif_BB.pdf?fbclid=IwAR2nbbU0ZBG397Onqa1sC8YqLPEw7OaU684Fywx8JNynw3jeeSjtfjf8pN4 (12 mai 2020)

Cazenave, *Planification*, consulté sur https://www.lamsade.dauphine.fr/~cazenave/enseignement/Planification/Planification.pdf?fbclid=IwAR3i3mCxc63XexVaJeIHzn1TeOEnT6fAExH_GVkPFLJZBEcdH1SPR_54zhM (11 mai 2020)

Helmert, *An Introduction to PDDL*, consulté sur https://www.cs.toronto.edu/~sheila/2542/s14/A1/introtopddl2.pdf (15 mai 2020)

Lakemeyer, *The Situation Calculus: A Case for Modal Logic* p. 1-7 consulté sur http://ai.stanford.edu/~epacuit/socsit/lakemeyer.pdf (8 mai 2020)

Lakemeyer, *The Situation Calculus and Golog,* consulté sur https://www.hybrid-reasoning.org/media/filer/2013/05/24/hybris-2013-05-sitcalc-slides.pdf (8 mai 2020)

McCarthy, *Actions and other events in situation calculus,* p. 7-13, consulté sur http://www-formal.stanford.edu/jmc/sitcalc.pdf (9 mai 2020)

Norvig, Russel, *Artificial Intelligence A Modern Approach*,consulté sur http://aima.cs.berkeley.edu (12 mai 2020)

Picard, *Classical Planning : Planning and Acting* consulté sur https://emse.fr/~picard/cours/ai/chapter11-alt.pdf (2 mai 2020)

Vassos, *Introduction To Ai Strips Planning,* consulté sur http://www.dis.uniroma1.it/degiacom/didattica/dottorato-stavros-vassos/AI&Games-Lecture2-part1.pdf (6 mai 2020)

Walling, *Context: Plan-Space Planning*, consulté sur https://www.cs.uni.edu/~wallingf/teaching/161/sessions/session24.pdf (1 mai 2020)

A brief overview of AI planning, consulté sur https://users.aalto.fi/~rintanj1/jussi/planning.html (14 mai 2020)

Shakey, consulté sur http://www.ai.sri.com/shakey/ (2 mai 2020)