Planification avec préférences basée sur la théorie MAUT couplée à une intégrale de Choquet

Ce chapitre s'intéresse à la planification avec préférences dans le cas où les préférences sont formalisées à l'aide d'un modèle MAUT et d'une intégrale de Choquet. Par ailleurs, il introduit également le planificateur CHOPLAN.

La section 3.1 présente le formalisme PDDL3/MAUT qui permet de représenter les préférences PDDL3 à l'aide de critères MAUT et d'utiliser une intégrale de Choquet comme fonction objectif. Lorsque ce formalisme est utilisé, le décideur dispose d'un plus grand pouvoir expressif quant à l'expression de ses préférences dans les problèmes de planification. La section 3.2 s'intéresse quant à elle à la résolution des problèmes de planification avec préférences représentés selon le formalisme PDDL3/MAUT. Un algorithme générique de recherche guidée par une heuristique est proposé. Celui-ci doit être instancié à l'aide d'une règle de sélection et d'un ensemble (éventuellement vide) de règles de coupe. Six règles de sélection candidates et une règle de coupe sont suggérées. Ces dernières sont utilisées par le planificateur CHOPLAN dont l'implémentation est décrite dans la section 3.3. Pour finir, les performances de CHOPLAN sont évaluées par rapport aux planificateurs de l'art en utilisant un ensemble de problèmes issus des compétitions internationales de planification.

3.1 Extension du pouvoir expressif du langage PDDL3

Cette section présente un formalisme original pour la planification avec préférences. Déjà mentionné dans la section 1.2.3 sous l'appellation PDDL3/MAUT, celui-ci généralise la notion de préférence utilisée en PDDL3 par l'introduction de préférences floues. L'extension proposée élargit le périmètre des problèmes de planification avec préférences qu'il est possible de modéliser et de résoudre. Pour réaliser cette généralisation, les préférences PDDL sont encodées en tant que critères MAUT (cf. section 3.1.1) et la fonction objectif PDDL est définie à l'aide d'une intégrale de Choquet (cf. section 3.1.2). Les modifications à apporter au langage PDDL pour prendre en compte ces changements sont présentés dans la section 3.1.3.

3.1.1 Préférences PDDL3 et critères MAUT

Trois types de préférences ont été introduites dans la section 1.2.1 à savoir les préférences numériques, finales et de trajectoire. De plus, il a été expliqué dans la section 2.2 qu'un critère MAUT est modélisé par un attribut auquel est associée une fonction d'utilité partielle. Cette dernière représente les préférences du décideur par rapport aux valeurs de l'attribut. Cette section explique comment des critères MAUT peuvent être utilisés pour représenter les préférences d'un problème de planification.

Le cas des préférences numériques est le plus simple puisque ces dernières définissent intrinsèquement l'attribut à considérer. Il ne reste donc qu'à définir la fonction d'utilité partielle de l'attribut. Celle-ci peut être construite manuellement ou à l'aide de la méthode présentée dans la section 2.4.2. Par exemple, dans le problème *Rovers* [42], le décideur peut avoir envie de raisonner sur la quantité d'énergie consommée par le robot $N1$ comme mentionné dans la section 1.1.2. La variable numérique e_1 associée à la consommation d'énergie du robot $N1$ constitue naturellement l'attribut de cette préférence. Le décideur peut être complètement satisfait (respectivement totalement insatisfait) si 40 unités d'énergie (respectivement 100 unités) sont consommées lors de la mission. Par ailleurs, sa satisfaction n'évolue pas nécessairement linéairement et il peut fortement préférer une consommation de 80 unités par rapport à une de 100 unités mais ne préférer que modérément une consommation de 40 unités par rapport à une de 60 unités. Ces informations

préférentielles peuvent être utilisées pour construire une fonction d'utilité partielle $u_{EC} : [0, 120] \rightarrow [0, 1]$ représentant les choix du décideur comme illustré sur la figure 3.1. En conséquence, la préférence numérique du décideur quant à la consommation d'énergie du robot $N1$ peut être définie par l'attribut e_1 et la fonction d'utilité partielle u_{EC}. Ainsi, si dans un plan x, le robot $N1$ consomme 60 unités d'énergie au cours de sa mission alors l'utilité de la préférence dans ce plan est $u_{EC}(60) = 0.8$. Ceci s'interprète comme une grande satisfaction du décideur quant à la consommation d'énergie du robot $N1$ dans x.

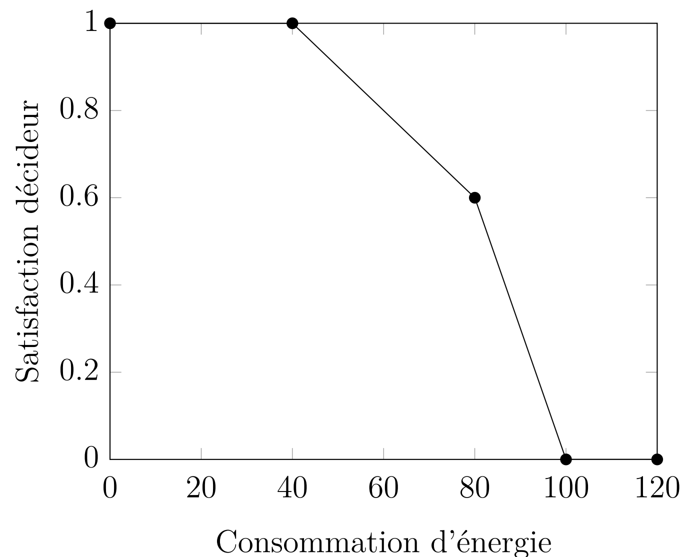

FIGURE 3.1 – Fonction d'utilité partielle u_{EC}

Le cas des préférences finales et des préférences de trajectoire est moins intuitif mais reste néanmoins simple à traiter. En effet, il suffit de considérer un attribut dont la valeur est définie par la sémantique de la préférence considérée (cf. définition 1.24). Ainsi, l'attribut vaut 1 si la préférence est satisfaite par rapport à la trajectoire considérée et 0 dans le cas contraire. Afin de respecter la nature binaire des préférences PDDL3, la fonction d'utilité partielle $u : \{0, 1\} \rightarrow [0, 1]$ associée à l'attribut est définie telle que $u(0) = 0$ et $u(1) = 1$. A titre d'illustration, dans le problème *Rovers*, la préférence `preference (always (at N1 L1))` signifie qu'il est préférable que le robot $N1$ soit toujours dans le lieu $L1$. Si dans un plan donné, le robot $N1$ se trouve dans une autre zone, alors la préférence est violée et son attribut prend pour valeur 0. En conséquence, la valeur d'utilité partielle associée à cette préférence dans le plan considéré est également nulle.

Le mécanisme considéré pour encapsuler les préférences finales et les préférences de trajectoire du PDDL3 permet de représenter ces dernières par un critère MAUT tout en conservant leur sémantique initiale. Cette méthode peut sembler relativement pauvre de prime abord puisqu'en respectant la sémantique du PDDL3, elle ne profite pas pleinement des avantages liés à la nature floue des critères MAUT ; néanmoins il n'en est rien. Tout d'abord, ce choix permet au formalisme PDDL3/MAUT d'être pleinement compatible avec la sémantique du PDDL3. Ce dernier s'inscrit donc dans la continuité des travaux réalisés autour du langage PDDL et ne constitue pas une alternative à la représentation de référence qu'est le PDDL3. Ainsi, tout planificateur capable d'utiliser le langage PDDL3 peut être étendu pour supporter l'extension PDDL3/MAUT. De plus, ce mécanisme d'encapsulation rend les préférences finales et les préférences de trajectoires commensurables avec les préférences numériques ce qui est particulièrement intéressant lors de la création de la fonction objectif comme expliqué dans la section 3.1.2. Pour finir, il convient de préciser que la nature floue de ces préférences peut néanmoins être exploitée pour élaborer des heuristiques comme présenté dans la section 3.2.

Les trois types de préférences considérées jusque là peuvent être vues comme des *préférences élémentaires* dans la mesure où elles ne s'intéressent qu'à un seul attribut des plans. Le langage PDDL3/MAUT permet également de représenter des *préférences non élémentaires* formées par agrégation d'un ensemble de préférences élémentaires. L'utilité d'une telle préférence est définie à l'aide d'une fonction d'agrégation évaluée avec les valeurs d'utilités des préférences élémentaires considérées. A titre d'illustration, dans un problème *Rovers* avec trois robots $N1$, $N2$ et $N3$, le décideur pourrait être intéressé par la création d'une préférence non élémentaire qui caractérise la consommation d'énergie engendrée par la mission sur la base des consommation d'énergie de chacun des robots. Soit x un plan ; p la préférence non élémentaire représentant la consommation d'énergie de la mission ; p_1, p_2, p_3 les trois préférences élémentaires agrégées par p qui représentent les consommations d'énergie individuelles des robots $N1$, $N2$ et $N3$; $y = z(x)$ le vecteur représentant le plan x dans l'espace des attributs $\Omega = \Omega_1 \times \Omega_2 \times \Omega_3$ c'est à dire le vecteur des consommations des robots ; ν une capacité définie sur $\mathfrak{P}(P)$ où P est l'ensemble des attributs relatifs à p_1, p_2 et p_3 ; et C_ν une intégrale de Choquet qui agrège les consommations individuelles des robots pour obtenir la consommation d'énergie de

la mission. L'utilité associée à la préférence non élémentaire p dans le plan x est définie par $C_\nu(u_1(y_1), u_2(y_2), u_3(y_3))$.

Pour finir, il convient de préciser qu'il est possible d'ajouter d'autres types de préférences élémentaires dans le formalisme PDDL3/MAUT. En effet, ce dernier possède par nature un grand pouvoir expressif puisque les critères MAUT peuvent représenter n'importe quelle variable. Par exemple, un critère pourrait dénombrer les états d'un plan dans lesquels une formule ϕ n'est pas vérifiée ; ce dernier constituerait alors une interprétation floue des préférences PDDL3 de type `always`. L'ajout de nouveaux types de préférences élémentaires permet également de créer des mécanismes de préférences spécifiques à une classe de problème donnée. La description BNF du formalisme PDDL3/MAUT présentée dans la section 3.1.3 se limite aux préférences élémentaires de type finale, de trajectoire ou numérique.

3.1.2 Fonction objectif PDDL et intégrale de Choquet

Une fois que des préférences ont été spécifiées, il faut se munir d'un mécanisme permettant de calculer la qualité (appelée utilité dans la terminologie MAUT) des plans solutions. En PDDL3, un coût de violation est associé à chaque préférence puis une fonction objectif est définie à partir de ces derniers. Dans le cas de l'extension PDDL3/MAUT, la fonction objectif est construite à l'aide d'une fonction d'agrégation comme le suggère la définition 2.2.

L'agrégation des préférences est plus simple et plus intuitive dans le formalisme PDDL3/MAUT puisque le modèle MAUT impose la *commensurabilité* des différents critères considérés (c.-à-d. la possibilité de les comparer par le biais d'une unité adéquate). En effet, il est relativement facile de comparer plusieurs préférences entre elles puisque ces dernières sont toutes définies en utilisant une fonction d'utilité partielle à valeur sur l'échelle de satisfaction commune ξ. Les problèmes qui mettent en oeuvre beaucoup de préférences numériques peuvent donc être encodés sans difficulté avec le formalisme MAUT. Ce propos est illustré une fois de plus à l'aide du problème *Rovers*. Si les robots peuvent être endommagés lorsqu'ils prélèvent un échantillon, le décideur peut introduire un critère qui modélise le risque d'endommagement des équipements des robots. L'espace de définition associé à ce risque est l'ensemble discret $\Omega_R = \{$`Très faible`, `Faible`, `Modéré`, `Elevé`, `Très élevé`$\}$.

Sans commensurabilité, il est impossible de déterminer si une amélioration sur le critère de consommation d'énergie (par exemple consommer 50 unités plutôt que 70) est préférée à une amélioration sur le critère de risque (par exemple passer d'un risque `Elevé` à un risque `Modéré`). En conséquence, construire une fonction objectif qui agrège les préférences du décideur relatives à la consommation d'énergie du robot $N1$, le risque de casse de ses équipements et la préférence de trajectoire `preference (always (N1 L1))` est difficile en PDDL3. Ceci explique notamment pourquoi tous les problèmes de planification avec préférences proposés lors des compétitions IPC n'utilisent au plus qu'une seule préférence numérique. En revanche, lorsque les préférences sont représentées par des critères MAUT, il est plus facile de les agréger puisqu'elles sont toutes commensurables entre elles. En effet, la notion d'utilité modélise la même quantité quelque soit la préférence considérée à savoir la satisfaction du décideur. Ainsi, le formalisme PDDL3/MAUT permet de représenter des problèmes avec un grand nombre de préférences numériques.

Bien que la fonction objectif PDDL puisse être quelconque, les problèmes de référence utilisent généralement une somme pondérée. L'opérateur d'agrégation retenu dans le cadre de cette étude est l'intégrale de Choquet 2-additive. Celle-ci généralise la somme pondérée en permettant au décideur de représenter des interactions entre paires de critères. A titre d'illustration, un décideur peut être prêt à accepter une grande consommation d'énergie pour un robot si le risque d'endommagement de ses équipements est faible. Dans ce cas, les deux critères sont substituables et leur indice d'interaction est négatif (cf. section 2.3.3). Le langage PDDL3/MAUT permet donc de représenter plus finement la complexité intrinsèque des préférences du décideur.

L'extension PDDL3/MAUT améliore le pouvoir expressif du modèle de préférences considéré mais complexifie également la réalisation de ce dernier. Ceci est notamment dû au fait que les décideurs doivent définir une fonction de capacité. Toutefois, la construction de la capacité peut être automatisée et outillée comme expliqué dans la section 2.4.2. Ainsi, la réalisation d'un modèle de préférences PDDL3/MCDA peut être considérée plus simple que celle d'un modèle de préférences PDDL3. En effet, il est plus intuitif pour les décideurs de comparer des alternatives de solutions entre elles (PDDL3/MAUT) que de définir les paramètres mathématiques d'une somme pondérée (PDDL3).

3.1.3 Langage formel pour l'extension PDDL3/MAUT

Cette section présente les éléments syntaxiques et sémantiques à considérer pour étendre le langage pddl3 à l'aide du formalisme maut.

Syntaxe de l'extension PDDL3/MAUT

Le pouvoir expressif du pddl étant très vaste, peu de planificateurs implémentent l'ensemble des fonctionnalités supportées par ce dernier. Les *exigences* (nommées `requirement`) sont des sous-ensembles cohérents du langage pddl qui ont été introduits afin de préciser les fonctionnalités qu'un planificateur peut mettre en œuvre. Par exemple, pour raisonner sur les expressions numériques et les préférences présentées respectivement dans les sections 1.1.3 et 1.2.3, il faut être capable de prendre en charge les éléments des exigences `numeric-fluents` et `preferences`.

Les modifications liées à l'extension pddl3/maut sont accessibles par l'intermédiaire de l'exigence `maut-preferences` qui elle-même s'appuie sur les exigences `numeric-fluents` et `preferences`. La description bnf [4] qui spécifie l'exigence `maut-preferences` est présentée sur la figure 3.2. Cette dernière s'appuie sur la description bnf du pddl qui est proposée dans [63]. Sans surprise, les deux éléments principaux de l'exigence `maut-preferences` sont les concepts de critères maut (`<maut-criterion>`) et d'intégrale de Choquet (`<choquet-integral>`). Les critères maut peuvent décrire des préférences finales ou des préférences de trajectoires (`<trajectory-criterion>`), des préférences numériques (`<numeric-criterion>`) ou encore des préférences non élémentaires (`<aggregation-criterion>`). Il convient de remarquer que d'un point de vue sémantique, les préférences finales s'interprètent sur une trajectoire (cf. définitions 1.23 et 1.24) ce qui justifie qu'elles soient représentées par un critère de type `<trajectory-criterion>`. L'intégrale de Choquet est quant à elle spécifiée à l'aide d'une capacité représentée par sa transformation de Möbius (`<mobius-capacity-list>`). La description proposée permet d'utiliser une intégrale de Choquet quelconque mais cette étude se limite au cas des intégrales de Choquet 2-additives.

```
<problem>                    ::= (...
                                  [<constraints>] :constraints
                                  [<maut-preferences>] :maut-preferences
                                  [<metric-spec>] :numeric-fluents
                                ...)



<maut-preferences>           ::=:maut-preferences (:maut-preferences <maut-spec>*)
<maut-spec>                  ::=:maut-preferences (:maut-criterion <maut-criterion>)
<maut-spec>                  ::=:maut-preferences (:choquet-integral <choquet-integral>)



<maut-criterion>             ::=:maut-preferences <numeric-criterion>
<maut-criterion>             ::=:maut-preferences <trajectory-criterion>
<maut-criterion>             ::=:maut-preferences <aggregation-criterion>

<numeric-criterion>          ::=:maut-preferences (:numeric-criterion <maut-criterion-symbol>
                                :attribute <basic-function-term>
                                :utility-function (<function-value>*))

<trajectory-criterion>       ::=:maut-preferences (:trajectory-criterion <maut-criterion-symbol>
                                :preference (<pref-name>))

<aggregation-criterion>      ::=:maut-preferences (:aggregation-criterion <maut-criterion-symbol>
                                :criteria (<maut-criterion-list>*)
                                :choquet-integral (<choquet-integral-symbol>))

<function-value>             ::=:maut-preferences (<number>, <number>)
<maut-criterion-list>        ::=:maut-preferences (<maut-criterion-symbol>)
<maut-criterion-symbol>      ::=:maut-preferences <name>



<choquet-integral>           ::=::maut-preferences (:choquet-integral <choquet-integral-symbol>
                                  :mobius (<mobius-capacity-list>*))

<mobius-capacity-list>       ::=:maut-preferences (<maut-criterion-symbol>
                                  [maut-criterion-symbol]* <number>)

<choquet-integral-symbol>    ::=:maut-preferences <name>



<metric-f-exp>               ::=:maut-preferences <choquet-integral-symbol>
```

Figure 3.2 – Description bnf de l'exigence pddl `maut-preferences`

Les figures 3.3, 3.4 et 3.5 illustrent la syntaxe de l'extension PDDL3/MAUT à l'aide d'un problème *Rovers* construit à partir des exemple présentés dans les sections 1.1.2 et 1.2.2 (cf. annexe B.3 pour visualiser la totalité de l'exemple). Ce problème met en œuvre une préférence numérique (`c-e1`) relative à la consommation d'énergie du robot $N1$, une préférence finale (`c-f1`) relative à l'acquisition d'un échantillon de sol du lieu $L6$ et deux préférences de trajectoires (`c-s1` et `c-a1`) relatives à la position du robot $N1$ au cours de la mission.

```
(:functions
  (energy ?x - robot) - number
)
```

FIGURE 3.3 – Syntaxe d'expression numérique primitive en PDDL

```
(:objects
  N1 - robot   ...
)

(:goal (and ...
  (preference f1 (possede_echantillon_sol N1 L6))
  ))

(:constraints (and ...
  (preference s1 (sometime (position N1 L1)))
  (preference a1 (at-most-once (position N1 L6)))
))
```

FIGURE 3.4 – Syntaxe de préférences finales et de trajectoires en PDDL

```
(:maut-preferences
  (:numeric-criterion c-e1
    :attribute (energy N1)
    :utility-function (
      (40, 1)
      (80, 0.6)
      (100, 0)
    ))

  (:trajectory-criterion c-f1
    :preference (f1))

  (:trajectory-criterion c-s1
    :preference (s1))

  (:trajectory-criterion c-a1
    :preference (a1))

  (:choquet-integral choqInt
    :mobius (
      (c-e1 0.35)
      (c-f1 0.25)
      (c-s1 0.2)
      (c-a1 0.2)
      (c-e1 c-f1 -0.1)
      (c-s1 c-a1 0.1)
    )
  )
)

(:metric maximize choqInt)
```

Figure 3.5 – Syntaxe de l'extension pddl3/maut

Sémantique de l'extension PDDL3/MAUT

Cette section présente la sémantique associée aux préférences lorsque ces dernières sont encodées dans des critères MAUT. Les définitions proposées s'appuient sur les éléments introduits dans les sections 1.1.3 et 1.2.3.

Définition 3.1 - Instance d'un problème de planification
Une instance d'un problème de planification avec préférences I_M est redéfinie par extension de la définition 1.14 en ajoutant la formule G', l'ensemble MC, la fonction objectif C_μ ainsi que la relation de préférences $\succsim$ à l'objet problème $Prob = (O,\, s_0,\, G',\, MC,\, C_\mu, \succsim)$ avec :

- G' une formule qui représente un ensemble d'objectifs étendus
- MC un ensemble de critères MAUT qui représentent des préférences
- C_μ une intégrale de Choquet 2-additive
- $\succsim$ une relation d'ordre construite à partir de MC et C_μ

Définition 3.2 - Utilité d'une préférence numérique
Soit t la trajectoire $\langle s_0, \cdots, s_n\rangle$ dont l'état final s_n est décrit par la paire (Atm, v), f une expression numérique primitive et $k \in MC$ le critère MAUT représentant la préférence numérique ayant pour attribut f et pour fonction d'utilité partielle $u_k : \Omega_f \rightarrow [0, 1]$. L'utilité associée à k sur la trajectoire t s'interprète par rapport à s_n et vaut $u_k^t = u_k(v_{Ind(f)})$.

Le problème *Rovers* présenté dans la section 1.2.2 (et dont la description exacte est donnée dans l'annexe B.4) est utilisé pour illustrer le calcul des utilités partielles associées à des préférences MAUT. L'une des solutions de ce problème est le plan 2 ; lequel est caractérisé par la trajectoire $t_2 = \langle s_0, \cdots, s_{10}\rangle$. L'exécution de ce plan conduit à un état final dans lequel la quantité d'énergie consommé par le robot $N1$ est de 50 unités (5 déplacements consommant chacun 10 unités d'énergie). Pour rappel, la fonction Ind permet d'associer à une expression numérique primitive f sa valeur dans les différents états du problème (cf. définition 1.16). Ainsi, la valeur associée au critère k relatif à la consommation d'énergie de $N1$ (nommé `c-e1` sur la figure 3.5) est celle de l'expression numérique primitive `(energy N1)` dans le vecteur v_{10} de s_{10}. L'utilité de ce critère est donc $u_{\texttt{c-e1}}^{t_2} = u_{EC}(50) = 0,9$ ce qui traduit une grande satisfaction du décideur quant à cette préférence.

Définition 3.3 - Utilité d'une préférence de trajectoire
Soit t la trajectoire $\langle s_0, \cdots, s_n \rangle$, Φ un objectif de trajectoire et $k \in MC$ le critère MAUT représentant la préférence de trajectoire associée à Φ ayant pour fonction d'utilité partielle $u_k : \{0, 1\} \rightarrow [0, 1]$. L'utilité associée à k sur la trajectoire t s'interprète par rapport à la valeur de vérité de Φ et vaut :

$$u_k^t = \begin{cases} 0 & \text{si} \quad \langle s_0, \cdots, s_n \rangle \models \neg\, \Phi \\ 1 & \text{si} \quad \langle s_0, \cdots, s_n \rangle \models \Phi \end{cases}$$

Les préférences de trajectoires `f1` et `a1` présentées sur la figure 3.4 sont vérifiées dans le plan 2. Les utilités des critères MAUT associés à ces préférences (nommés `c-f1` et `c-a1` sur la figure 3.5) valent donc donc $u^{t_2}_{\texttt{c-f1}} = u^{t_2}_{\texttt{c-a1}} = 1$. En revanche, l'utilité $u^{t_2}_{\texttt{c-s1}}$ du critère `c-s1` associé à la préférence `s1` est nulle puisque cette dernière est violée dans le plan 2 (le robot $N1$ ne se trouve jamais dans le lieu $L1$).

Définition 3.4 - Utilité d'une préférence non élémentaire
Soit t la trajectoire $\langle s_0, \cdots, s_n \rangle$; $k_1, \cdots, k_q$ des critères MAUT dont les valeurs d'utilité partielles sur t sont $u_1^t, \cdots, u_q^t$; ν une capacité ; et k un critère MAUT qui agrège les critères $k_1, \cdots, k_q$ à l'aide de l'intégrale de Choquet C_ν. L'utilité associée à k sur la trajectoire t vaut $u_k^t = C_\nu(u_1^t, \cdots, u_q^t)$.

Dans le cas de l'exemple de la section 1.2.2, le décideur pourrait souhaiter regrouper l'ensemble de ses préférences concernant la position du robot $N1$ (c.-à-d. `c-a1` et `c-s1`) en un unique critère `c-p`. Si la capacité ν est choisie de sorte à décrire une moyenne arithmétique, alors la valeur d'utilité du critère `c-p` dans le plan 2 est $u^{t_2}_{\texttt{c-p}} = C_\nu(u^{t_2}_{\texttt{c-a1}}, u^{t_2}_{\texttt{c-s1}}) = (u^{t_2}_{\texttt{c-a1}} + u^{t_2}_{\texttt{c-s1}})/2 = 0,5$ ce qui s'interprète comme une satisfaction moyenne du décideur.

Définition 3.5 - Utilité d'un plan
Soit I_M une instance d'un problème de planification, x un plan de trajectoire t et $u_1^t, \cdots, u_p^t$ les valeurs d'utilités partielles des critères MC de I_M sur t.

L'utilité U du plan x est définie telle que $U_x = C_\mu(u_1^t, \cdots, u_p^t)$. Le plan x_1 est dit au moins aussi préféré que le plan x_2 noté $x_1 \succsim x_2$ si et seulement si $U_{x_1} \geq U_{x_2}$.

Afin d'illustrer cette définition, l'utilité du plan 2 est calculée par rapport au modèle de préférences présenté sur la figure 3.5. Les utilités des différents critères du modèle ont été déterminées précédemment et valent respectivement $u^{t_2}_{\texttt{c-e1}} = 0.9$; $u^{t_2}_{\texttt{c-f1}} = 1$; $u^{t_2}_{\texttt{c-a1}} = 1$ et $u^{t_2}_{\texttt{c-s1}} = 0$. La capacité 2-additive μ utilisée est décrite sur la figure 3.5 par l'intermédiaire de ces coefficients de Möbius. Ainsi, l'utilité du plan 2 vaut $U = C_\mu(u^{t_2}_{\texttt{c-e1}}, u^{t_2}_{\texttt{c-f1}}, u^{t_2}_{\texttt{c-a1}}, u^{t_2}_{\texttt{c-s1}}) = 0.1 + 0.9 \times 0.25 + 0.15 + 0.2 = 0,675$.

L'extension du langage PDDL3 via l'exigence `maut-preferences` constitue l'une des contributions de cette étude. En effet, cette dernière enrichit le pouvoir expressif du langage PDDL3 en permettant d'utiliser facilement un nombre quelconque de préférences numériques, d'agréger des préférences élémentaires entre elles et de considérer d'éventuelles interactions entre les préférences du problème. De plus, une méthode outillée peut être utilisée pour construire efficacement le modèle de préférences correspondant. En conséquence, l'extension PDDL3/MAUT permet de représenter avec plus de précision les préférences des décideurs lors de la résolution des problèmes de planification.

Domaine de la gestion de crise

L'approche retenue pour faire cohabiter les préférences PDDL3 *avec le formalisme* MAUT *consiste à encoder les préférences* PDDL3 *au sein de critères préférentiels* MAUT. *Ceci est particulièrement intéressant pour le domaine de la gestion de crise puisque les décideurs peuvent ainsi utiliser à la fois des préférences* PDDL3 *et des préférences* MAUT *pour modéliser leurs attentes quant à la situation de crise qu'ils traitent. En plus des exemples mentionnés dans la section 1.2, ils peuvent ainsi représenter des préférences floues, des préférences non élémentaires ou encore prendre en compte les interactions entre leurs préférences. Une préférence non élémentaire pourrait par exemple être utilisée dans une situation où il serait souhaitable qu'une partie de la population soit la plus sédentaire possible (pour minimiser ses déplacements à la suite d'une inondation par exemple). Dans ce cas, des préférences relatives à la distribution de denrées alimentaires, à l'approvisionnement en électricité*

et à l'accès à l'information pourraient être combinées pour former un critère « d'incitation à la sédentarisation de la population » qu'il conviendrait de maximiser. Par ailleurs, dans le cas d'une inondation urbaine évaluée par rapport à deux critères relatifs à la surface de la ville inondée et à la hauteur d'eau maximale constatée, le modèle de préférences des décideurs pourrait inclure une interaction entre ces deux critères. Ainsi, un plan qui conduirait à une situation dans laquelle la surface inondée serait faible tandis que la hauteur d'eau serait grande pourrait être préféré à un plan qui conduirait à une situation dans laquelle la surface inondée et la hauteur d'eau seraient toutes deux moyennes.

3.2 Résolution du problème de planification avec préférences MAUT

3.2.1 Planification par recherche guidée par une heuristique

Les principales méthodes de l'art utilisées pour résoudre les problèmes de planification sont des techniques de *satisfaisabilité booléenne* [86, 87], de *satisfaction de contraintes* [41, 92, 108] ou de *recherche guidée par des heuristiques* [28, 80]. Les travaux présentés dans cette section s'appuient sur des mécanismes de recherche à l'aide d'heuristiques. Ces techniques sont les plus utilisées dans le domaine de la planification avec préférences notamment en raison de leur efficacité (cf. par exemple HPLAN-P [6] et LPRPG-P [39]).

Un graphe orienté $G = (V, E)$ est défini par la donnée d'un ensemble V de *nœuds* (ou *sommets*) et d'un ensemble E d'*arcs*. Les arcs sont des couples de sommets $e = (v_1, v_2)$ tels que $v_1, v_2 \in V$. Par ailleurs dans un graphe orienté, les arcs ont un sens ainsi (v_1, v_2) et (v_2, v_1) sont deux arcs différents. Un problème de planification peut être représenté par un graphe $G_P = (S, GA_\gamma)$ avec S l'ensemble des états dans lequel le système Σ étudié peut se trouver et GA_γ l'ensemble des transitions d'états représentées sous forme de paires de sommets. Par exemple, si $\gamma(s, a) = s'$ avec $a \in GA$ et $s, s' \in S$ alors l'arc $(s, s') \in GA_\gamma$. La résolution d'un problème de planification consiste donc à identifier un chemin de G_P qui relie l'état initial s_0 à

un état final $s \in S_{G'}$ avec $S_{G'} \subseteq S$ l'ensemble des états dans lesquels les objectifs G' sont vérifiés. Comme mentionné dans la section 1.1.2, les problèmes de planification sont généralement soumis à une telle combinatoire qu'il n'est pas envisageable de construire le graphe G_P entièrement. En conséquence, la construction de G_P doit être réalisée dynamiquement lors de la résolution du problème.

Plusieurs méthodes de recherche peuvent être employées pour résoudre un problème de planification. La *recherche en avant* part de l'état initial et essaye d'identifier un chemin qui mène à un état dans lequel les objectifs sont atteints. La *recherche en arrière* lui est diamétralement opposée puisqu'elle consiste à trouver un chemin des objectifs vers la situation initiale.

Les techniques de résolution proposées dans ce chapitre utilisent une méthode de recherche en avant à l'image de l'algorithme 3.1 présenté ci-après. La première étape de la recherche en avant consiste à construire l'ensemble des nœuds voisins du nœud initial. Le parcours du graphe se poursuit en sélectionnant l'un de ces nœuds qui est alors étendu à son tour de la même manière. Ainsi lors de la recherche, le graphe est naturellement divisé en deux parties : l'ensemble des *nœuds ayant été explorés* et l'ensemble des *nœuds non encore explorés* (cf. figure 3.6). La *frontière* est l'ensemble des nœuds non explorés qui peuvent être sélectionnés pour être étendus lors de la prochaine étape de l'algorithme. La notion de frontière est très importante puisque la façon dont cette dernière est étendue lors du parcours du graphe définit la *stratégie de recherche* employée.

La stratégie de recherche détermine le chemin à emprunter à partir de la frontière dans le but de progresser jusqu'à un nœud satisfaisant les objectifs. Il convient de distinguer les stratégies de recherche dites *aveugles* à l'image des stratégies de recherche en profondeur [55] ou en largeur [96] des stratégies dites *informées* à l'image de la recherche par le *meilleur en premier* (ou encore *Best-First-Search* ou *BFS*) [113]. Dans le cas des stratégies de recherche aveugles, aucune information spécifique au problème à résoudre n'est utilisée pour choisir les nœuds de la frontière qui sont étendus. Par exemple, avec une stratégie de recherche en profondeur (respectivement en largeur), le nœud de la frontière sélectionné pour être étendu est celui dont la profondeur (nombre d'arcs qui le sépare du nœud initial) est la plus grande (respectivement la plus petite). Ainsi, en considérant la frontière présentée sur la figure 3.6, le nœud A (respectivement C) serait étendu par une

stratégie de recherche de type en profondeur (respectivement en largeur). En revanche, les stratégies informées sont caractérisées par le fait qu'elles s'appuient sur des informations spécifiques au problème à résoudre. Par exemple, dans le cadre d'une stratégie BFS, les nœuds de la frontière sont triées en fonction de leur potentiel ; lequel est évalué à l'aide d'une heuristique. Le meilleur nœud au sens de l'heuristique considérée est alors retenu pour poursuivre la recherche. Si la fonction heuristique f doit être minimisée, le nœud E de la frontière de la figure 3.6 serait alors sélectionné par la stratégie BFS.

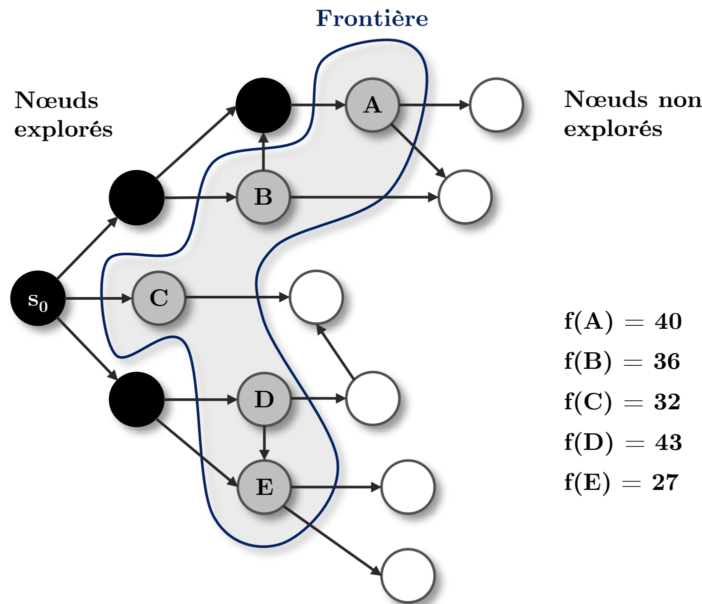

FIGURE 3.6 – Principe de la recherche en avant dans un graphe

L'algorithme 3.1 est un algorithme générique de résolution de problèmes de planification par recherche en avant dans un graphe. Il s'agit d'un algorithme itératif qui cherche des solutions de qualité croissante et ce jusqu'à ce que les ressources à disposition (temps alloué au calcul notamment) soient épuisées ce qui est caractérisé par la fonction `AreResourcesExhausted()`. Pour instancier l'algorithme 3.1, il faut préciser une *règle de sélection* ainsi qu'un ensemble (éventuellement vide) de *règles de coupe*. La règle de sélection implémente la stratégie de recherche et les mécanismes itératifs de l'algorithme. Les règles de coupe sont quant à elles utilisées pour identifier et supprimer les nœuds qu'il n'est pas nécessaire de visiter lors de la suite de la recherche.

L'algorithme consiste à sélectionner (`selectionRule.selectNode()`) puis traiter des nœuds de la frontière. Si le nœud sélectionné satisfait les objectifs à atteindre, une solution a alors été identifiée et l'algorithme débute une nouvelle itération (`selectionRule.nextIteration()`). De plus, si la qualité de cette solution est supérieure à la meilleure solution connue, cette dernière est ajoutée à la liste des solutions à retourner lorsque l'algorithme se termine. En revanche, si le nœud sélectionné ne satisfait pas les objectifs à atteindre, l'algorithme se poursuit en construisant son voisinage à partir de l'ensemble des actions du problème. Si une action est applicable (`IsApplicable()` $= true$) dans le nœud considéré et qu'elle est admissible par rapport aux règles de coupe (`IsPrunable()` $= false$), le nœud est alors étendu (`ExpandNode()`) conformément à la sémantique présentée dans les sections 1.1.3, 1.2.3 et 3.1.3. Dans ce cas, la frontière de la recherche est alors mise à jour (`selectionRule.updateFrontier()`) en conséquence.

Des règles de sélection pour la planification avec préférences sont proposées dans la section 3.2.2. Par ailleurs, les règles d'applicabilité (qui définissent le comportement de la fonction `IsApplicable()`) sont présentées avec les règles de coupe dans la section 3.2.3 puisque leurs implémentations respectives sont similaires. Les règles d'applicabilité ne diffèrent des règles de coupe que par le fait que leur présence est obligatoire dans l'algorithme pour garantir la validité des plans solutions retournés. En effet, pour un état s et une action a, ces règles autorisent l'extension de s par a uniquement si les préconditions de a sont vérifiées dans s et si l'exécution de a dans s ne viole aucun des objectifs de trajectoire du problème.

Input : param paramètres qui précisent les ressources disponibles
s_0 état initial du problème
G formule des objectifs à atteindre
GA ensemble des actions closes du problème

Output : solution liste de plans solutions

Data : selectionRule la règle de sélection utilisée
prunningRules ensemble des règles de coupe utilisées
current nœud courant de l'algorithme
child nœud obtenu par extension du nœud courant

```
Algorithm SEARCH(param, s_0, G, GA)
    selectionRule.initFrontier(s_0)
    while AreResourcesExhausted(param) = false and
    selectionRule.isFrontierEmpty() = false do
        current ← selectionRule.selectNode()
        if current.evaluateFormula(G) = true then
            if current.getUtility() > solution.getBestUtility() then
                solution.add(current)
            end
            selectionRule.nextIteration(s_0)
        else
            for all action ∈ GA do
                if IsApplicable(current, action) = true and
                 IsPrunable(prunningRules, current, action) = false then
                    child ← ExpandNode(current, action)
                    selectionRule.updateFrontier(child)
                end
            end
        end
    end
    return solution

Function IsPrunable(prunningRules, current, action)
    for all prunningRule ∈ prunningRules do
        if prunningRule.IsActionPrunable(current, action) = true then
            return true
        end
    end
    return false
```

Algorithme 3.1 : Résolution itérative de problèmes de planification par recherche dans un graphe. Les objets `selectionRule` et `pruningRules` doivent être précisés pour décrire complètement la méthode de résolution.

3.2.2 Règles de sélection pour la planification avec préférences

La règle de sélection est l'élément clef de l'algorithme 3.1 puisqu'elle détermine les performances de ce dernier. Pour définir une règle de sélection, il faut préciser une *stratégie de recherche* et une *heuristique*. La stratégie de recherche caractérise le comportement de la règle de sélection lorsqu'une solution est identifiée (via la fonction `nextIteration()`) et définit les mécanismes d'extension de la frontière lors de la recherche (cf. fonctions `initFrontier()`, `isFrontierEmpty()`, `selectNode()` et `updateFrontier()`). Ces mécanismes s'appuient sur une heuristique $h(I_M, s)$ (fonction `evaluateHeuristic()`) qui estime le potentiel d'un nœud s par rapport aux objectifs et aux préférences de l'instance de planification I_M considérée. Dans le cas de la planification avec préférences, les heuristiques peuvent être relativement complexes à écrire puisqu'elles doivent réaliser un compromis entre objectifs et préférences. En effet, si la recherche est focalisée sur les objectifs, il est alors facile d'identifier des solutions mais ces dernières ont de fortes chances d'être médiocres au regard du modèle de préférences considéré. En revanche, si la recherche est seulement focalisée sur les préférences, il se peut qu'aucune solution ne soit jamais identifiée en raison de la forte combinatoire des problèmes de planification. La solution retenue dans cette étude consiste à construire les heuristiques $h(I_m, s)$ à partir d'estimations $\Delta(I_M, s)$ et $\Lambda(I_M, s)$ qui jugent respectivement le nœud s seulement par rapport aux objectifs du problème et seulement par rapport au modèle de préférences considéré.

Ces travaux proposent six règles de sélection dont les performances sont évaluées dans la section 3.3. Ces dernières respectent toutes la structure générique des règles de sélection qui est précisée par l'algorithme 3.2. Afin de faciliter la compréhension du lecteur, leur construction est exposée selon une approche bottom-up. Pour commencer, trois estimations Δ_1, Λ_1 et Λ_2 sont décrites. Trois heuristiques h_1, h_2 et h_3 sont ensuite élaborées par agrégation de ces estimations. Finalement, deux stratégies de recherche SR_1 et SR_2 sont présentées. SR_1 et SR_2 peuvent chacune être utilisées conjointement avec h_1, h_2 ou h_3.

```
Object SelectionRule
    Function nextIteration()
    Function initFrontier(s0)
    Function isFrontierEmpty()
    Function selectNode()
    Function updateFrontier(node)
    Function evaluateHeuristic(node)
```

Algorithme 3.2 : Structure d'une règle de sélection

Distance d'un nœud aux objectifs

Cette section s'intéresse à $\Delta_1(I_M, s)$ qui constitue une estimation du nombre minimal d'actions qu'il faut exécuter pour atteindre les objectifs G' de I_M à partir du nœud s. Un nœud est d'autant plus intéressant que sa distance aux objectifs est faible. En effet, privilégier de tels nœuds lors de la recherche permet généralement d'identifier rapidement une solution au problème. Le calcul de Δ_1 repose sur l'heuristique employée par le planificateur Fast-Forward [80,81]. Pour que le calcul puisse être réalisé efficacement (en temps polynomial par rapport aux données d'entrées du problème), cette heuristique utilise une version simplifiée I'_M du problème à résoudre. Ainsi, le problème est *relaxé* en ignorant les effets négatifs des actions (méthode déjà employée auparavant dans [27,28]) et en ignorant les objectifs étendus du problème (la notion d'objectifs étendus étant en réalité postérieure au planificateur Fast-Forward). L'heuristique proposée par Fast-Forward consiste à appliquer la procédure de résolution par *graphe de planification* proposée dans Graphplan [25, 26] avec le problème de planification relaxé.

Afin d'expliquer le calcul de $\Delta_1(I_M, s)$, la notion de *graphe de planification relaxé* est précisée. Il s'agit d'une structure composée de plusieurs ensembles de prédicats P_i et de plusieurs ensembles d'actions A_i qui se succèdent les uns aux autres (voir tableau 3.1 et figure 3.7 pour un exemple). Le premier ensemble de prédicats P_0 contient l'ensemble des atomes de s. Le premier ensemble d'action A_1 est quant à lui obtenu en sélectionnant l'ensemble des actions du problème applicables dans

P_0. Le deuxième ensemble de prédicats P_1 contient tous les prédicats de P_0 (ce qui s'interprète comme l'utilisation d'un opérateur fictif `No-Op` sur chacun de ces prédicats) ainsi que l'ensemble des effets *positifs* des actions de A_1. La construction du graphe de planification relaxé se poursuit de la sorte jusqu'à ce qu'un ensemble de prédicats P_m contenant tous les objectifs à atteindre soit construit.

Une fois que le graphe de planification relaxé a été élaboré, il est possible d'en extraire une solution (voir [79] pour l'algorithme exact). La solution est une séquence d'ensembles d'actions $\langle O_1, \cdots, O_m \rangle$ qui est obtenue par recherche en arrière dans le graphe de planification relaxé. Ainsi, pour construire l'ensemble O_m, il faut commencer par identifier les prédicats p_j^m de P_m qui correspondent aux objectifs à atteindre. Si un prédicat p_j^m ne peut pas être obtenu par l'opérateur `No-Op` et si l'action a qui permet de l'obtenir n'appartient pas à un ensemble d'actions A_i tel que $i < m$ (les actions ne vérifiant pas cette condition n'ont pas été représentées sur la figure 3.7 afin de simplifier l'illustration), alors l'action a est ajoutée à l'ensemble d'actions O_m. Les prédicats p_j^m correspondant aux objectifs à atteindre pouvant être obtenus par l'opérateur `No-Op` et les préconditions des actions de O_m deviennent alors les prédicats p_j^{m-1} à considérer pour construire l'ensemble d'actions O_{m-1}. L'algorithme d'extraction de la solution se termine après que l'ensemble O_1 ait été construit. Dans l'exemple proposé par le tableau 3.1 et la figure 3.7, les prédicats p_j^3 de P_3 à considérer pour construire O_3 sont p_3 et p_5. Le prédicat p_3 peut être obtenu par l'opérateur `No-Op` (représenté par une flèche en pointillés sur la figure 3.7) tandis que le prédicat p_5 peut être obtenu par l'action a_4 qui constitue donc l'unique action de O_3. La solution extraite du graphe de planification relaxé de cet exemple est $\langle \{a_1\}, \{a_2 ; a_3\}, \{a_4\} \rangle$.

Action	**Préconditions**	**Effets Positifs**	**Effets Négatifs**
1	p_1	p_2	$\neg p_3$
2	p_2	p_3	-
3	p_2	p_4	$\neg p_1$
4	p_1, p_4	p_5	$\neg p_4$
5	p_4	p_3	-

TABLEAU 3.1 – Actions pour illustrer la notion de graphe de planification relaxé

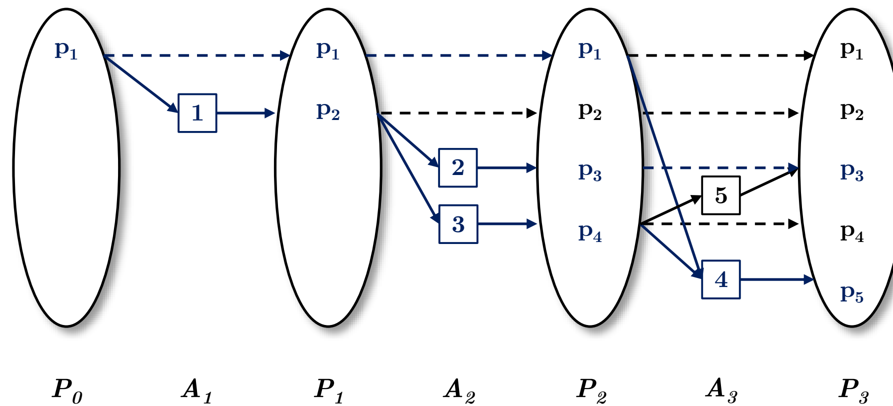

FIGURE 3.7 – Graphe de planification relaxé de $s = \{p_1\}$ à $G = \{p_3, p_5\}$.
En bleu, l'illustration de la méthode de construction de la solution.

La valeur de Δ_1 est identique à celle de l'heuristique proposée par FAST-FORWARD et correspond à la taille de la solution $\langle O_1, \cdots, O_m \rangle$ extraite du graphe de planification relaxé (cf. définition 3.6). Ainsi dans l'exemple proposé, la distance du nœud s aux objectifs G vaut $\Delta_1(I_M, s) = 4$.

Définition 3.6 - Estimation Δ_1
Soit I_M une instance d'un problème de planification et s un état. Soit $\texttt{FF}(I_M, s, G)$ l'heuristique employée par FAST-FORWARD et $\langle O_1, \cdots, O_m \rangle$ la solution extraite du graphe de planification relaxé de s vers les objectifs G de I'_M. L'estimation $\Delta_1(I_M, s)$ est définie par :

$$\Delta_1(I_M, s) = \texttt{FF}(I_M, s, G) = \textstyle\sum_{i=1}^{m} |O_i|$$

Qualité d'un nœud par rapport aux préférences

Cette section s'intéresse aux estimations $\Lambda_1(I_M, s)$ et $\Lambda_2(I_M, s)$ qui jugent le potentiel d'un nœud s au regard des préférences du problème. Les méthodes à mettre en œuvre pour calculer ces valeurs diffèrent nécessairement de celles employées dans le cas de Δ_1 en raison de la différence sémantique qu'il existe entre objectifs et préférences. En effet, puisque tous les objectifs d'un problème doivent être vérifiés dans un plan pour qu'il soit valide, ces derniers sont tous aussi importants les uns que les autres. En revanche, les préférences n'impactent généralement pas

le calcul de l'utilité du plan de façon homogène (sauf dans le cas de modèles de préférences particuliers). Il en résulte naturellement que deux plans satisfaisant des sous-ensembles de préférences différents ne sont pas évalués de manière identique. Cette distinction sémantique entre objectifs et préférences, aussi évidente soit elle, met en exergue la nécessité de considérer l'*importance* individuelle de chaque préférence du modèle pour élaborer les estimations $\Lambda_1(I_M, s)$ et $\Lambda_2(I_M, s)$.

La grandeur $\Lambda_1(I_M, s)$ constitue une estimation de l'utilité finale de tout plan qui serait construit par extension de la trajectoire $\langle s_0, \cdots, s \rangle$. Elle consiste simplement à évaluer la fonction objectif dans le nœud s (voir définition 3.7). Un tel calcul est rapide à réaliser et tient naturellement compte de l'importance relative des différentes préférences du problème. En s'appuyant sur la valeur d'utilité *locale* observée dans le nœud s, cette estimation est optimiste vis à vis des préférences satisfaites dans s et pessimiste vis à vis des préférences violées dans s.

Définition 3.7 - Estimation Λ_1
Soit I_M une instance d'un problème de planification et t la trajectoire de l'état initial s_0 à l'état s. Si $u_1^t, \cdots, u_p^t$ sont les valeurs d'utilités partielles des critères MC de I_M par rapport à la trajectoire $\langle s_0, \cdots, s \rangle$, alors :

$$\Lambda_1(I_M, s) \;=\; C_\mu\,(u_1^t, \cdots, u_p^t)$$

L'estimation Λ_1 tient compte de l'importance relative des différentes préférences du problème mais ne considère aucune information quant à l'évolution future de la satisfaisabilité de ces dernières. Il est possible de proposer une estimation qui combine pour chaque préférence la *difficulté* inhérente à sa satisfaction avec son *importance*. A titre d'exemple, l'une des heuristiques proposées dans [6] consiste à évaluer la fonction objectif du problème dans les différents nœuds P_i d'un graphe de planification relaxé puis à pondérer ces valeurs en fonction de la profondeur i des nœuds P_i considérés. Par ailleurs, l'heuristique utilisée dans LPRPG-P [39] peut être interprétée comme une modification de la structure de graphe de planification relaxé qui permet de tenir compte des préférences du problème et de leurs poids respectifs. L'approche retenue dans cette étude utilise également le concept de graphe de planification relaxé mais repose sur la nature floue des critères MAUT utilisés pour représenter les préférences du problème.

La valeur des préférences finales et des préférences de trajectoire est par nature binaire puisque ces dernières ne peuvent se trouver que dans deux états (satisfaites ou violées). La grandeur $\Lambda_2(I_M, s)$ est construite en remplaçant l'interprétation usuelle de ces préférences par une estimation de l'effort à produire pour les satisfaire à partir du nœud s. Il convient de remarquer que cette interprétation est compatible avec la sémantique initiale de ces préférences puisque les deux cas limites d'un effort nul ou d'un effort infini s'interprètent respectivement comme le fait que la préférence soit satisfaite ou violée dans s.

Le calcul de $\Lambda_2(I_M, s)$ est analogue à celui de $\Lambda_1(I_M, s)$ puisqu'il consiste à évaluer la fonction objectif du problème par rapport à la trajectoire $\langle s_0, \cdots, s \rangle$. Néanmoins, dans le cas de $\Lambda_2(I_M, s)$, le calcul des valeurs d'utilités partielles des préférences diffère. En effet, en raison de la modification précédemment mentionnée, les critères MAUT associées aux préférences sont redéfinis. Dans le cas d'une préférence finale p représentée par un critère k, la valeur y_k de ce critère est déterminée à partir de l'heuristique `FF`. En effet, la grandeur $\texttt{FF}(I_M, s, p)$ constitue bien une estimation de l'effort à fournir pour satisfaire p depuis le nœud s. Néanmoins, il est impossible d'associer à cette estimation une fonction d'utilité partielle qui serait valable pour tout problème de planification. En effet, il se peut que $\texttt{FF}(I_M^1, s_1, p_1) = \texttt{FF}(I_M^2, s_2, p_2) = 15$ en considérant que s_1 est un nœud à fort potentiel au vue de la complexité de l'instance I_M^1 mais que s_2 est un nœud à faible potentiel au vue de la complexité de l'instance I_M^2. Afin de résoudre ce problème, la valeur y_k est définie comme le rapport entre $\texttt{FF}(I_M, s, p)$ et $\texttt{FF}(I_M, s_0, p)$. La valeur du critère s'interprète donc comme l'effort à fournir pour satisfaire la préférence p depuis le nœud s par rapport à l'effort à fournir pour la satisfaire depuis le nœud représentant l'état initial. Il devient alors possible de caractériser entièrement le critère k associé à une préférence finale quelconque p et ce quelque soit le problème de planification considéré. L'espace de définition de k est $\Omega_k = \mathbb{R}^+$ avec $\mathbf{1}_k = 0$ et $\mathbf{0}_k = 5$ les éléments parfaitement satisfaisant et totalement insatisfaisant de k. La fonction d'utilité partielle u_h du critère k est représentée sur la figure 3.8. Lorsque le rapport y_k vaut 1, la satisfaction associée à la préférence est moyenne (l'effort à fournir pour atteindre p depuis s est le même que celui pour l'atteindre depuis s_0). Cette dernière augmente lorsque la valeur de y_k diminue (l'effort à fournir pour atteindre p depuis s est plus faible que celui pour l'atteindre depuis s_0). En revanche, la satisfaction associée à la préférence diminue lorsque la valeur

y_k augmente (l'effort à fournir pour atteindre p depuis s est plus grand que celui pour l'atteindre depuis s_0). Pour finir, la définition 3.8 explicite formellement le calcul de la valeur d'utilité partielle associée à une préférence finale dans le cadre de l'estimation Λ_2.

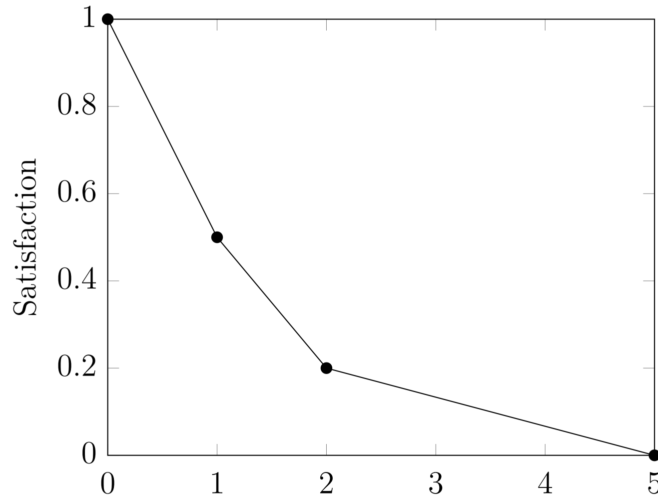

FIGURE 3.8 – Fonction d'utilité partielle u_h

Définition 3.8 - Utilité d'une préférence finale dans Λ_2
Soit I_M une instance d'un problème de planification, p la préférence relative à l'objectif final ϕ représentée par le critère k, s un état et t la trajectoire $\langle s_0, \cdots, s\rangle$. Soit $u_h : [0,5] \rightarrow [0,1]$ la fonction d'utilité partielle linéaire par morceaux définie telle que $u_h(0) = 1$; $u_h(1) = 0.5$, $u_h(2) = 0.2$ et $u_h(5) = 0$. La valeur d'utilité partielle du critère associé à la préférence finale ϕ est définie par :

$$u_k^t = \begin{cases} u_h\left(\dfrac{\texttt{FF}(I_M, s, \phi)}{\texttt{FF}(I_M, s_0, \phi)}\right) \text{ si } \texttt{FF}(I_M, s_0, \phi) \neq 0\,; \\ 1 \text{ si } \texttt{FF}(I_M, s_0, \phi) = 0 \text{ et } \texttt{FF}(I_M, s, \phi) = 0\,; \\ 0 \text{ si } \texttt{FF}(I_M, s_0, \phi) = 0 \text{ et } \texttt{FF}(I_M, s, \phi) \neq 0\,; \end{cases}$$

Les deux dernières conditions de la définition 3.8 permettent de s'assurer que la valeur u_k^t est correctement définie même lorsque $\texttt{FF}(I_M, s_0, \phi) = 0$ (c.-à-d. quand la préférence p est satisfaite dans l'état initial s_0 du problème). Dans ce cas, la valeur d'utilité u_k^t vaut 1 si p est également satisfaite dans s et 0 si elle ne l'est pas.

Les préférences de trajectoires sont plus complexes à traiter puisque leur sémantique est relativement riche. Cette étude ne s'intéresse qu'aux préférences de trajectoires non temporelles du PDDL3 à savoir `always`, `sometime`, `at-most-once` et `sometime-before` (voir définition 1.23). Le traitement de ces quatre types de préférences dans le cadre de l'estimation Λ_2 est à présent précisé. Le cas des préférences `always` et `at-most-once` est relativement simple puisqu'aucune modification n'est apportée au calcul de la définition 3.3. En effet, leur sémantique se prête peu à l'introduction d'une notion de difficulté ou d'effort à fournir pour satisfaire la préférence depuis un nœud du problème. Par exemple, considérer l'effort à produire pour satisfaire le prédicat ϕ d'une préférence `always` ϕ depuis un nœud s n'a aucun sens puisque si ϕ n'est pas vraie dans s alors la préférence est violée. De même, pour une préférence `at-most-once` ϕ, chercher à satisfaire ϕ ou $\neg\phi$ peut avoir des effets contre-productifs au vue de la sémantique de cette préférence. En revanche, la sémantique associée aux préférences de trajectoire `sometime` et `sometime-before` peut être modifiée avantageusement dans le cadre du calcul de Λ_2. Pour expliciter ces modifications, les automates de Büchi [33, 133] de ces préférences sont représentés sur la figure 3.9 [47, 65]. Ces derniers précisent l'évolution de la valeur de vérité de ces préférences lorsqu'une transition d'états survient. Par exemple, la préférence `sometime` ϕ est initialement dans l'état S_0 ; lequel est un état de la préférence et ne doit pas être confondu avec s_0 qui lui est un état du problème. Si une action ayant pour effet positif ϕ (respectivement effet négatif $\neg\phi$) est exécutée dans s_0, le problème évolue vers un état s dans lequel l'état de la préférence est S_2 (respectivement S_1).

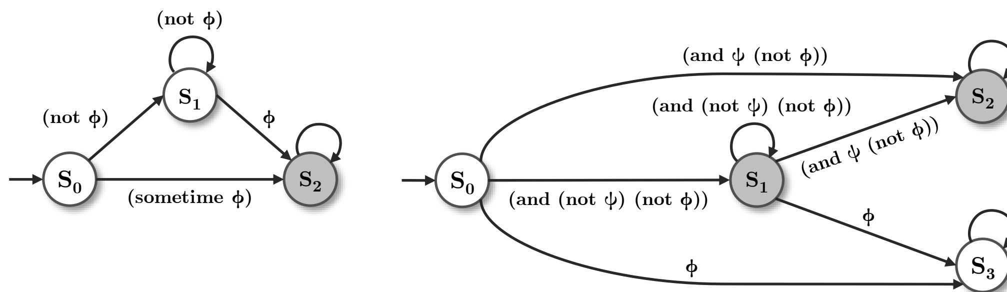

FIGURE 3.9 – Automates de Büchi des préférences `sometime` ϕ et `sometime-before` ϕ ψ ; d'après [65]

L'automate de Büchi de `sometime` ϕ ne possède qu'un seul état dans lequel la préférence est satisfaite (cercle gris sur la figure 3.9). Cet état S_2 n'est accessible que lorsque ϕ devient vraie ce qui conduit naturellement à caractériser l'effort à fournir pour satisfaire cette préférence depuis un nœud s en fonction du prédicat ϕ (cf. définition 3.9). Au vu des mécanismes utilisés, le calcul de l'utilité d'une préférence `sometime` pour l'estimation Λ_2 est analogue à celui présenté dans la définition 3.8.

Définition 3.9 - Utilité d'une préférence `sometime` dans Λ_2
Soit I_M une instance d'un problème de planification, p une préférence de trajectoire de type `sometime` ϕ représentée par le critère k, s un état et t la trajectoire $\langle s_0, \cdots, s\rangle$. Soit $u_h : [0,5] \rightarrow [0,1]$ la fonction d'utilité partielle linéaire par morceaux définie telle que $u_h(0) = 1$; $u_h(1) = 0.5$, $u_h(2) = 0.2$ et $u_h(5) = 0$. La valeur d'utilité partielle du critère k est définie par :

$$u_k^t = \begin{cases} u_h\left(\dfrac{\texttt{FF}(I_M, s, \phi)}{\texttt{FF}(I_M, s_0, \phi)}\right) & \text{si } \texttt{FF}(I_M, s_0, \phi) \neq 0\,; \\ 1 \;\text{ si }\; \texttt{FF}(I_M, s_0, \phi) = 0 \;\text{ et }\; \texttt{FF}(I_M, s, \phi) = 0\,; \\ 0 \;\text{ si }\; \texttt{FF}(I_M, s_0, \phi) = 0 \;\text{ et }\; \texttt{FF}(I_M, s, \phi) \neq 0\,; \end{cases}$$

L'automate de Büchi de `sometime-before` $\phi\ \psi$ est plus complexe notamment parce qu'il possède deux états (S_1 et S_2) dans lesquels la préférence est satisfaite. L'état S_2 est le plus intéressant puisque la préférence ne peut plus jamais être violée lorsqu'il est atteint. Pour qu'une transition d'états conduise à S_2, cette dernière doit générer un état dans lequel ψ et $\neg\,\phi$ sont simultanément vrais. La condition $\neg\,\phi$ est problématique puisque l'heuristique `FF` s'appuie sur le concept de graphe de planification relaxé qui par définition ne considère pas les effets négatifs du problème. En conséquence, le calcul de la valeur d'utilité partielle associée à une préférence de type `sometime-before` $\phi\ \psi$ est réalisé en fonction du prédicat ψ. Ce dernier tient néanmoins compte de la valeur de vérité de `sometime-before` $\phi\ \psi$ (via la condition $t \models p$ dans la définition 3.10) afin de respecter la sémantique initiale de la préférence.

Définition 3.10 - Utilité d'une préférence `sometime-before` dans Λ_2
Soit I_M une instance d'un problème de planification, p une préférence de trajectoire de type `sometime-before` ϕ ψ représentée par le critère k, s un état et t la trajectoire $\langle s_0, \cdots, s\rangle$. Soit $u_h : [0,5] \rightarrow [0,1]$ la fonction d'utilité partielle linéaire par morceaux définie telle que $u_h(0) = 1$; $u_h(1) = 0.5$, $u_h(2) = 0.2$ et $u_h(5) = 0$. La valeur d'utilité partielle de k est définie par :

$$u_k^t = \begin{cases} u_h\left(\dfrac{\texttt{FF}(I_M, s, \phi)}{\texttt{FF}(I_M, s_0, \phi)}\right) \text{ si } t \models p \text{ et } \texttt{FF}(I_M, s_0, \phi) \neq 0\,; \\ 1 \text{ si } t \models p \text{ et } \texttt{FF}(I_M, s_0, \phi) = 0 \text{ et } \texttt{FF}(I_M, s, \phi) = 0\,; \\ 0 \text{ si } t \models p \text{ et } \texttt{FF}(I_M, s_0, \phi) = 0 \text{ et } \texttt{FF}(I_M, s, \phi) \neq 0\,; \\ 0 \text{ si } t \models \neg p\,; \end{cases}$$

Les préférences numériques ne sont pas modifiées dans le cadre du calcul de Λ_2 puisque leur nature floue est par définition pleinement exploitée. Une extension envisageable consisterait à tenir compte de l'évolution potentielle des préférences numériques entre le nœud s et les nœuds objectifs. Pour conclure, le calcul de l'estimation Λ_2 est précisé par la définition 3.11.

Définition 3.11 - Estimation Λ_2
Soit I_M une instance d'un problème de planification, s un état et t la trajectoire $\langle s_0, \cdots, s\rangle$. Soit $u_1^t, \cdots, u_p^t$ les valeurs d'utilités partielles des critères de I_M calculés par rapport à t en utilisant les définitions 3.8, 3.9 et 3.10 en priorité plutôt que la définition 3.3. L'estimation Λ_2 est définie par :

$$\Lambda_2(I_M, s) = C_\mu\,(u_1^t, \cdots, u_p^t)$$

Heuristiques pour la planification avec préférences

Les trois estimations Δ_1, Λ_1 et Λ_2 ayant été présentées, les heuristiques h_1, h_2 et h_3 peuvent à présent être introduites. Le fonctionnement de ces trois heuristiques est relativement similaire. En effet, elles privilégient toutes les nœuds ayant un fort potentiel vis à vis des objectifs à atteindre (caractérisé par Δ_1) tout en tenant

compte du potentiel de ces derniers par rapport aux préférences du problème (caractérisé par Λ_1 et Λ_2). Une approche similaire est utilisée par le planificateur HPLAN-P dont l'heuristique est élaborée à partir de plusieurs estimations du potentiel d'un nœud ; lesquelles sont agrégées via un ordre lexicographique [6]. En effet, les nœuds sont dans un premier temps triés en fonction de leur distance par rapport aux objectifs puis les éventuelles égalités sont départagées à l'aide d'une fonction qui estime le potentiel des nœuds par rapport aux préférences du problème. L'approche retenue dans cette étude consiste à construire les heuristiques h_1, h_2 et h_3 à partir de trois agrégations différentes des grandeurs Δ_1, Λ_1 et Λ_2.

Pour réaliser ces agrégations, il convient dans un premier temps de rendre les grandeurs Δ_1, Λ_1 et Λ_2 commensurables. En tant que score calculé sur la base d'un modèle de préférences MAUT, Λ_1 et Λ_2 sont toutes deux à valeur dans $\xi = [0, 1]$ et représentent une satisfaction quant à la qualité potentielle d'un nœud s. Ce n'est en revanche pas le cas de l'estimation Δ_1. Construite à partir de Δ_1 en utilisant les mécanismes précédemment décrits et la fonction d'utilité partielle u_h (cf. figure 3.8), la grandeur Δ_1' est quant à elle commensurable avec Λ_1 et Λ_2.

Définition 3.12 - Estimation Δ_1'
Soit I_M une instance d'un problème de planification, s un état et $u_h : [0, 5] \to [0, 1]$ la fonction d'utilité partielle linéaire par morceaux définie telle que $u_h(0) = 1$; $u_h(1) = 0.5$; $u_h(2) = 0.2$ et $u_h(5) = 0$. $\Delta_1'(I_m, s)$ est définie par :

$$\Delta_1'(I_M, s) \; = \; u_h\left(\frac{\Delta_1(s, G)}{\mathtt{FF}(I_M, s_0, G)} \right)$$

L'opérateur d'agrégation retenu pour construire les heuristiques h_1, h_2 et h_3 est une intégrale de Choquet 2-additive de capacité ρ. Il convient de ne pas confondre la fonction de capacité ρ qui spécifie l'agrégation réalisée par les heuristiques h_1, h_2 et h_3 avec la fonction de capacité μ qui est utilisée pour agréger les différentes préférences du problème à résoudre. L'expression de ρ est explicitée dans la section suivante lorsque les aspects itératifs de l'algorithme sont précisés.

Définition 3.13 - Heuristique h_1
Soit I_M une instance d'un problème de planification, s un état et C_ρ une intégrale de Choquet 2-additive de capacité ρ. La valeur de l'heuristique h_1 dans le nœud s est définie par :

$$h_1(I_M, s) \;=\; C_\rho\Big(\,\Delta'_1(I_M, s)\,,\,\Lambda_1(I_M, s)\,\Big)$$

Définition 3.14 - Heuristique h_2
Soit I_M une instance d'un problème de planification, s un état et C_ρ une intégrale de Choquet 2-additive de capacité ρ. La valeur de l'heuristique h_2 dans le nœud s est définie par :

$$h_2(I_M, s) \;=\; C_\rho\Big(\,\Delta'_1(I_M, s)\,,\,\Lambda_2(I_M, s)\,\Big)$$

Définition 3.15 - Heuristique h_3
Soit I_M une instance d'un problème de planification, s un état et C_ρ une intégrale de Choquet 2-additive de capacité ρ. Soit $\Lambda_3(I_M, s) = 0,6 * \Lambda_2(I_M, s) + 0,4 * \Lambda_1(I_M, s)$. La valeur de l'heuristique h_3 dans le nœud s est définie par :

$$h_3(I_M, s) \;=\; C_\rho\Big(\,\Delta'_1(I_M, s)\,,\,\Lambda_3(I_M, s)\,\Big)$$

En ce qui concerne la définition 3.15, Λ_3 a été construit à partir d'une pondération entre Λ_1 et Λ_2 afin que h_3 soit similaire à h_1 et h_2. Ceci permet notamment aux trois heuristiques d'utiliser la même fonction de capacité ρ. Néanmoins, l'heuristique h_3 aurait également pu être définie par $h_3(I_M, s) = C_{\rho'}(\Delta'_1(I_M, s), \Lambda_1(I_M, s), \Lambda_2(I_M, s))$; auquel cas une seconde fonction de capacité ρ' aurait du être spécifiée.

Pour généraliser, il convient de préciser qu'un nombre quelconque d'estimations pourraient être utilisées pour construire ce type d'heuristiques. Intuitivement, plus le nombre de critères d'estimation utilisé augmente et plus la probabilité de faire face à des plateaux (un grand nombre de solutions ayant le même score) au cours de la recherche diminue. En revanche, il devient alors plus difficile de construire la fonction de capacité ρ.

Stratégies de recherche

Trois heuristiques h_1, h_2 et h_3 ont été proposées pour implémenter la fonction `EvaluateHeuristic()` de la règle de sélection. Cette section s'intéresse quant à elle à la définition des fonctions `initFrontier()`, `isFrontierEmpty()`, `selectNode()`, `updateFrontier()` et `nextIteration()` qui caractérisent la stratégie de recherche de la règle de sélection (cf. algorithmes 3.1 et 3.2).

Les deux stratégies de recherche SR_1 et SR_2 considérées dans cette étude utilisent une stratégie de recherche de type le meilleur nœud en premier (BFS). Celle-ci est implémentée à l'aide de deux listes afin d'éviter d'éventuels problèmes de boucles infinies lors de la recherche (cf. algorithme 3.3). La liste `closeList` contient l'ensemble des nœuds déjà explorés lors du parcours du graphe. L'objet `openList` est quant à lui une *liste ordonnée* qui représente la frontière de la recherche. Ainsi, la position d'un nœud dans `openList` est déterminée par sa valeur de coût (`cost`) ; laquelle est calculée via l'heuristique (`EVALUATEHEURISTIC()`) de la règle de sélection. La position d'un nœud dans `openList` est déterminée automatiquement lors de l'ajout du nœud à la liste ou lors de l'appel à la fonction `sort()`.

Input : s_0 état initial du problème
child nœud de l'espace de recherche

Data : closeList liste des nœuds explorés
openList liste ordonnée des nœuds à explorer

Object <u>SELECTIONRULE</u>

Function `initFrontier(`s_0`)`
openList ← `newSortedList()`
closeList ← `newList()`
openList`.add(`s_0`)`

```
Function isFrontierEmpty()
    return ( openList.size() = 0 )

Function selectNode()
    node ← openList.getFirstElement()
    openList.remove(node)
    closeList.add(node)
    return node

Function updateFrontier(child)
    cost ← EvaluateHeuristic(child)
    if openList.contains(child) = false and
    closeList.contains(child) = false then
        child.setCost(cost)
        openList.add(child)
    else if child.getCost() > cost then
        child.setCost(cost)
        if openList.contains(child) = false then
            openList.add(child)
        else
            openList.sort()
        end
    end
```

Algorithme 3.3 : Implémentation d'une règle de sélection de type BFS. La fonction `EvaluateHeuristic()` détermine la valeur d'un nœud au regard de l'heuristique considérée. Les fonctions `newList()` et `newSortedList()` retournent respectivement une structure de données de type liste et de type liste ordonnée. Les fonctions `add()`, `getFirstElement()`, `size()` et `contains()` s'interprètent comme les opérateurs usuels d'une liste.

L'algorithme 3.1 est un algorithme itératif puisqu'il retourne une succession de plans de qualité croissante au cours de la recherche (voir définition 1.29, section 1.2.4). Cette propriété est très appréciable dans le cadre de la planification avec préférences puisque la complexité combinatoire des problèmes rend généralement impossible l'identification d'une solution optimale. Les modifications apportées à la stratégie de recherche lorsqu'une nouvelle solution est identifiée sont définies dans la fonction `nextIteration()` de la règle de sélection.

L'approche retenue dans cette étude consiste à redémarrer la recherche avec une nouvelle fonction de capacité ρ lors de chaque itération. En utilisant la représentation de Möbius, cela peut être réalisé à partir de la capacité ρ_1^m (cf. tableau 3.2) en diminuant la valeur de Δ de α tout en augmentant la valeur de $\Delta\ \Lambda$ de α à chaque fois qu'une nouvelle solution est identifiée et ce jusqu'à ce que la capacité obtenue soit ρ_2^m. De façon analogue, le passage de ρ_2^m à ρ_3^m est réalisé en diminuant la valeur de $\Delta\ \Lambda$ de α tout en augmentant la valeur de Λ de α à chaque fois qu'une nouvelle solution est identifiée. En procédant ainsi, chaque solution est généralement plus difficile à trouver que la précédente mais a de grandes chances d'être meilleure au regard du modèle de préférences considéré.

	Δ	$\Delta\ \Lambda$	Λ
ρ_1^m	1	0	0
ρ_2^m	0	1	0
ρ_3^m	0	0	1

TABLEAU 3.2 – Illustration de l'évolution de la transformation de Möbius de la capacité ρ. Δ et Λ sont les critères associés aux objectifs et aux préférences.

Il convient de remarquer que cette approche exploite pleinement le pouvoir expressif de l'opérateur d'agrégation qu'est l'intégrale de Choquet. En effet, la procédure proposée pour construire la représentation de Möbius de ρ considère une interaction positive $I_{\Delta,\Lambda} > 0$ entre les critères d'objectifs et de préférences modélisant ainsi une complémentarité entre ces derniers. Ainsi, toutes choses égales par ailleurs, les solutions équilibrées qui sont performantes à la fois sur les critères d'objectifs et de préférences obtiennent un meilleur score que les solutions non équilibrées. De plus,

l'approche retenue permet de construire dynamiquement la fonction de capacité ρ ce qui constitue un avantage certain. En effet, il est relativement difficile de déterminer l'équilibre entre objectifs et préférences permettant de conduire la recherche le plus efficacement possible puisque ce dernier peut varier en fonction du problème considéré. Il reste néanmoins à préciser la valeur du paramètre α. Intuitivement, plus α est grand et plus la difficulté du problème augmente rapidement lors de chaque itération. Il convient donc de choisir une valeur relativement faible tout en s'assurant que le gain observé entre deux solutions consécutives n'en devienne pas pour autant négligeable. Sur la base des expérimentations et tests réalisés, la valeur empirique $\alpha = 0.1$ a été retenue. La fonction `nextIteration()` de la stratégie de recherche SR_1 implémente l'approche qui vient d'être décrite.

```
Input :  s0 état initial du problème
Data :   closeList liste des nœuds explorés
         openList liste ordonnée des nœuds à explorer

Object SelectionRule

Function nextIteration(s0)
    UpdateHeuristicCapacity()
    openList ← newSortedList()
    closeList ← newList()
    openList.add(s0)
```

Algorithme 3.4 : Implémentation de `nextIteration()` pour la stratégie de recherche SR_1. `UpdateHeuristicCapacity()` met à jour la capacité ρ de l'heuristique conformément à la procédure précédemment mentionnée.

La stratégie de recherche SR_2 est une variante de SR_1 qui exploite les différences structurelles entre la fonction objectif C_μ qui sert à évaluer les solutions et les heuristiques h_1, h_2 et h_3 qui servent à guider la recherche. En effet, il convient de mentionner que l'ordre des solutions retournées par les trois heuristiques proposées ne préjugent pas nécessairement de l'ordre de ces solutions par rapport à la relation

de préférences $\succsim$ du problème. Ceci s'explique notamment par le fait que les heuristiques h_1, h_2 et h_3 utilisent des éléments extérieurs à la fonction C_μ (à savoir les considérations relatives aux objectifs à atteindre) et ne considèrent que la qualité locale des nœuds dans lesquels elles sont évaluées. En conséquence, si x_1, x_2 et x_3 sont les trois premières solutions retournées par une recherche de type BFS basée sur l'une des heuristiques proposées, le classement de ces solutions par rapport à la relation de préférences du problème pourrait être $x_3 \succsim x_1 \succsim x_2$. Il en résulte que redémarrer la recherche avec une nouvelle fonction de capacité ρ à chaque fois qu'une nouvelle solution est trouvée peut empêcher l'algorithme d'identifier des solutions potentiellement intéressantes (x_3 dans l'exemple précédent). La stratégie de recherche SR_2 est donc implémentée de sorte à ne redémarrer la recherche avec une nouvelle fonction de capacité qu'après que β solutions aient été identifiées à partir de la capacité ρ actuellement utilisée. La valeur $\beta = 10$ a été retenue sur la base d'observations empiriques. En effet, cette valeur semble être un bon compromis entre amélioration de la qualité des solutions et augmentation du temps de calcul.

```
Input :  s0 état initial du problème
Data :   closeList liste des nœuds explorés
         openList liste ordonnée des nœuds à explorer

Object SelectionRule

Function nextIteration(s0)
    solutionNumber ← solutionNumber + 1
    if solutionNumber mod 10 = 0 then
        UpdateHeuristicCapacity()
        openList ← newSortedList()
        closeList ← newList()
        openList.add(s0)
    end
```

Algorithme 3.5 : Implémentation de `nextIteration()` pour la stratégie de recherche SR_2. `UpdateHeuristicCapacity()` met à jour la capacité ρ de l'heuristique conformément à la procédure précédemment mentionnée.

Les règles de sélection de l'algorithme 3.1 sont construites à partir d'une heuristique et d'une stratégie de recherche. Trois heuristiques h_1, h_2 et h_3 ainsi que deux stratégies de recherche SR_1 et SR_2 ont été proposées. Les six configurations correspondantes SR_1-h_1, SR_1-h_2, SR_1-h_3, SR_2-h_1, SR_2-h_2 et SR_2-h_3 sont évaluées dans la section 3.3.

3.2.3 Règles de coupe pour la planification avec préférences

Etant donné s un nœud du graphe (un état du système Σ) et a un arc du graphe (une action close du problème de planification), une règle de coupe a pour objectif de déterminer si l'état s' obtenu par exécution de a dans s doit être considéré dans la suite de la recherche (c.-à-d. ajouté à la frontière) ou non. Cette section présente les deux *règles d'applicabilité* de l'algorithme 3.1 ainsi que la règle de coupe dite de *Pareto-dominance*. Conceptuellement, les règles d'applicabilité peuvent être interprétées comme des règles de coupe dont la présence est obligatoire pour que les solutions retournées par l'algorithme 3.1 soient des plans valides.

Règles d'applicabilité

Par définition (cf. section 1.1.3), une action a est applicable dans un état s si $s \models Pre_a$ c'est à dire si les préconditions de a sont vérifiées dans l'état s. La première règle d'applicabilité consiste simplement à vérifier que cette condition soit respectée et à couper le nœud s' dans le cas contraire. Le nœud s' n'est donc pas ajouté à la frontière et ne peut plus être considéré au cours de la recherche.

La deuxième règle d'applicabilité s'assure que les objectifs de trajectoire ne soient pas violés au cours de la recherche. Ainsi la transition $\gamma(s, a) = s'$ n'est admissible que si pour tout objectif de trajectoire Φ du problème, $\langle s_0, \cdots, s' \rangle \models \Phi$. Si cette condition n'est pas respectée, le nœud s' n'est pas ajouté à la frontière de la recherche. L'algorithme 3.6 précise l'implémentation de ces deux règles d'applicabilité.

```
Input : node nœud courant de la recherche
        action action à appliquer dans node
Data :  preconditionTest résultat du test sur les préconditions
        trajectoryTest résultat du test sur les objectifs de trajectoire
        G_T ensemble des objectifs de trajectoire
        Φ objectif de trajectoire

Function IsApplicable(node, action)
    preconditionTest ← true
    trajectoryTest ← true
    if !( node.getState() |= action.getPreconditions() ) then
        preconditionTest ← false
    end
    for all Φ ∈ G_T do
        if !( node.getTrajectory() |= Φ ) then
            trajectoryTest ← false
        end
    end
    return preconditionTest and trajectoryTest
```

Algorithme 3.6 : Fonction `IsApplicable()` de l'algorithme 3.1

Règle de Pareto-dominance

Cette règle de coupe repose comme son nom le suggère sur le concept de *dominance au sens de Pareto* ; lequel est introduit par la définition 3.16.

Définition 3.16 - Dominance au sens de Pareto [52]
Soit P un ensemble d'attributs, $x^a, x^b \in X$ deux solutions et $y^a, y^b \in Y$ leurs vecteurs respectifs dans $Y = \{ z(x) \mid x \in X \}$.

- x^a domine fortement x^b ($x^a \succ_P x^b$) si et seulement si $\forall k \in P, x_k^a > x_k^b$;
- x^a domine faiblement x^b ($x^a \succ_p x^b$) si et seulement si $\forall k \in P, x_k^a \geq x_k^b$ et $\exists k \in P, x_k^a \neq x_k^b$.

La règle de Pareto-dominance consiste à couper le nœud s' obtenu par l'application de a dans s si ce dernier est dominé par un nœud de la frontière. Afin de s'assurer que l'évaluation de la règle de coupe puisse être réalisée efficacement, seuls les 25 premiers nœuds de la frontière sont considérés. En outre, avant de vérifier qu'un nœud de la frontière s_f domine le nœud s', il convient de s'assurer que sa structure soit similaire à celle de s'. Dans le cas contraire, il pourrait exister une action a' applicable dans s' mais pas dans s_f. Couper le nœud s' supprimerait alors une partie de l'espace de recherche pouvant contenir des solutions.

Deux états s_f et s' ont une structure similaire s'ils sont caractérisés par le même ensemble d'atomes ($Atm_f = Atm'$) et si les trajectoires $\langle s_0, \cdots, s_f \rangle$ et $\langle s_0, \cdots, s' \rangle$ sont indifférenciées au sens des objectifs et préférences de trajectoire du problème. Ce dernier point peut être vérifié à l'aide des automates de Büchi des objectifs et préférences de trajectoire pddl (cf section 3.2.2). En effet, l'état de ces automates dans s_f et s' tient, par construction, compte de l'ensemble des actions présentes dans les trajectoires $\langle s_0, \cdots, s_f \rangle$ et $\langle s_0, \cdots, s' \rangle$. Ainsi, il suffit de s'assurer que tous les états des automates de Büchi des objectifs et préférences de trajectoire du problème aient la même valeur dans s_f et s'.

```
Input : node nœud courant de la recherche
        action action à appliquer dans node
Data :  openList liste ordonnée des nœuds à explorer
        fnode nœud de la frontière

Function IsActionPrunable(node, action)
    for i ← 1 to Max(25, openList.size()) do
        fnode ← openList.getElement(i)
        if SameStructure(fnode, node) = true then
            if StrongParetoDominance(fnode, node) = true then
                return true
            end
        end
    end
    return false
```

Algorithme 3.7 : Fonction `IsActionPrunable` de la règle de Pareto-dominance

Domaine de la gestion de crise
Malgré son apparente complexité, la démarche présentée pour résoudre les problèmes de planification avec préférences est relativement intuitive. En effet, la méthode de recherche en avant (qui consiste à identifier une progression de l'état initial vers un état final vérifiant les objectifs) est analogue aux processus cognitifs couramment employés par les décideurs de la gestion de crise. Par ailleurs, la stratégie de recherche utilisée est relativement naturelle puisqu'elle consiste, dans un premier temps, à chercher des plans satisfaisant les objectifs puis à prendre en compte les préférences des décideurs de manière progressive afin d'identifier de meilleurs solutions. Finalement, les règles de coupe sont utilisées pour éliminer les pistes de recherche qui semblent les moins prometteuses.

3.3 Implémentation et résultats expérimentaux

Un planificateur nommé CHOPLAN a été implémenté afin d'évaluer les mécanismes et algorithmes mentionnés dans la section 3.2. Cette section précise le fonctionnement de CHOPLAN et expose la démarche retenue pour le comparer aux planificateurs de l'art (cf. section 3.3.1). Par ailleurs, les résultats expérimentaux obtenus sont présentés et analysés dans la section 3.3.2.

3.3.1 Implémentation et démarche expérimentale

CHOPLAN (« CHO » pour Choquet et « PLAN » pour planificateur) a été développé à l'aide du langage de programmation Java. Il repose sur trois modules (*lexeur*, *parseur* et *solveur*) qui sont invoqués successivement lors de la résolution d'un problème de planification. Le *lexeur* traite les fichiers textes du problème (`domain.pdll` et `problem.pddl`) afin de les convertir en séquences d'entités lexicales. Ces éléments syntaxiques sont alors analysés par le *parseur* qui leur associe une sémantique et crée les objets informatiques correspondants. Ces derniers sont naturellement adaptés à la méthode de résolution choisie (par exemple des nœuds caractérisés par des atomes logiques dans le cas de la recherche guidée par des heuristiques).

Le *solveur* s'appuie sur ces objets informatiques pour implémenter l'algorithme de résolution du problème. Dans le cas de CHOPLAN, il s'agit évidemment de l'algorithme 3.1 présenté dans la section 3.2.

CHOPLAN a été implémenté à l'aide de la librairie PDDL4J [115] qui fournit un lexeur capable de traiter les versions 1.2, 2.1 et 3 du langage PDDL. De plus cette librairie propose un parseur pour la version 1.2 du langage PDDL. Ce dernier est particulièrement intéressant puisqu'il s'appuie sur les mécanismes d'inertie présentés dans [90]. La notion d'inertie permet de simplifier les différentes formules du problème ce qui se traduit généralement par une diminution de l'espace de recherche à explorer. Dans le cadre de cette étude, tous les éléments (non fournis par la librairie PDDL4J) nécessaires à l'implémentation de l'algorithme 3.1 ont été développés comme indiqué sur la figure 3.10. Ceci inclut notamment la prise en charge de l'exigence `maut-preferences` présentée dans la section 3.1.

	PDDL 1.2	**PDDL 2.1**	**PDDL 3**	**PDDL3/MAUT**
Lexeur	PDDL4J			
Parseur				
Solveur				

FIGURE 3.10 – Planificateur CHOPLAN. Sur fond gris, les fonctionnalités issues de la librairie PDDL4J et sur fond blanc les fonctionnalités développées.

Les performances de CHOPLAN sont évaluées en utilisant des problèmes de référence issus des compétitions internationales de planifications (IPC). Seuls les domaines de planification avec préférences qui sont susceptibles d'utiliser à la fois des préférences finales et des préférences de trajectoires PDDL3 ont été considérés. Les domaines *Rovers*, *Openstacks*, *Pathways*, *PipesWorld*, *Storage*, *TPP* et *Trucks* (voir [65] pour une description précise) sont les seuls à respecter cette condition. Dans le cadre de cette étude, ce sont les domaines *Rovers* et *Openstacks* qui ont été retenus puisque les autres domaines utilisent l'exigence PDDL `existential-preconditions` qui n'est pour le moment pas supportée dans CHOPLAN. Pour chacun de ces domaines, il existe 20 problèmes contenant uniquement des préférences finales (dénotés SP pour « Simple Preferences ») et 20 problèmes contenant des préférences finales et des préférences de trajectoire (dénotés QP pour « Qualitative Preferences »).

Comme la majorité des problèmes de référence utilisés, ces problèmes utilisent une seule préférence numérique. Afin de tester les performances de CHOPLAN sur des problèmes impliquant plusieurs préférences numériques (entre 2 et 16), une variante du domaine *Rovers* a été élaborée. Ainsi 10 problèmes (dénotés MP pour « MAUT Preferences ») contenant des préférences finales, des préférences de trajectoire ainsi que des préférences numériques ont été proposés. En conséquence, l'évaluation de CHOPLAN s'appuie sur 90 problèmes de planification avec préférences.

Les problèmes considérés sont résolus avec les six configurations SR_1-h_1, SR_1-h_2, SR_1-h_3, SR_2-h_1, SR_2-h_2 et SR_2-h_3 de CHOPLAN. Les résultats ainsi obtenus sont comparés à ceux des planificateurs de l'art SGPlan5 [83] et LPRPG-P [39]. SGPlan5 a remporté en 2006 la $5^{\text{ème}}$ compétition internationale de planification dans la catégorie planification avec préférences. Il s'agit de la seule compétition au cours de laquelle la résolution des problèmes de planification avec préférences *Rovers* et *Openstacks* a été proposée. La méthode de résolution de SGPlan5 consiste à subdiviser le problème en sous-problèmes qui sont résolus indépendamment les uns des autres. Ces mécanismes de division sont optimisés pour chacun des domaines considérés. Ainsi, une variante nommée SGPlan-W est également retenue afin d'évaluer les capacités de résolution de SGPlan5 en tant que solveur générique. Cette dernière consiste simplement à invoquer SGPlan5 en lui fournissant des problèmes dont les noms initiaux ont été modifiés (voir [39] pour plus de détails). LPRPG-P est quant à lui le planificateur de l'art le plus récent. Il est capable de traiter l'ensemble des préférences PDDL3 et peut gérer des préférences numériques. Sa méthode de résolution couple des mécanismes de programmation linéaire avec une recherche heuristique basée sur un graphe de planification relaxé. Par ailleurs, l'ensemble des problèmes considérés a également été résolu avec une version de ChoPlan basée uniquement sur l'estimation Δ_1. Ce planificateur nommé *Contrôle* constitue une implémentation de l'heuristique FAST-FORWARD dans CHOPLAN. Ne cherchant pas à optimiser la fonction objectif, il fournit des valeurs de référence pour estimer le gain qualitatif apporté par les planificateurs CHOPLAN, SGPlan5, SGPlan-W et LPRPG-P.

Tous les tests ont été réalisés sur la même machine (3.2Ghz CPU, 8Go RAM). Pour chacun des 90 problèmes considérés, les planificateurs ont disposé de 10 minutes de temps de calcul. La qualité des plans solutions a été calculée à l'aide du validateur

de plan val [82]. Cet outil est utilisé lors des compétitions internationales de planification afin de vérifier la validité des solutions et d'évaluer leurs scores par rapport à la fonction objectif du problème considéré. Son utilisation permet de comparer ChoPlan aux planificateurs de l'art de manière cohérente et ce même si celui-ci résout une version pddl3/maut des problèmes pddl considérés.

3.3.2 Résultats expérimentaux

Les figures 3.11 à 3.15 représentent la qualité des solutions obtenus par Contrôle, lprpg-p, SGPlan5, SGPlan-W et ChoPlan pour l'ensemble des problèmes étudiés. Dans tous les problèmes considérés, la fonction objectif doit être *minimisée*. Ainsi, plus la valeur associée à la solution d'un planificateur est faible et plus ce dernier est performant. Dans le cas où un planificateur retourne plusieurs solutions, seule la meilleure est représentée. Une absence de valeur pour un problème donné signifie que le planificateur n'a pas réussi à résoudre le problème concerné. Dans le but de ne pas surcharger la présentation de ces résultats, une seule des six configurations de ChoPlan a été représentée sur chaque figure. La configuration retenue est celle ayant obtenue les meilleurs résultats sur le domaine considéré. Les performances des différentes configurations de ChoPlan peuvent néanmoins être analysées à l'aide des tableaux 3.3, 3.4 et 3.5 qui synthétisent l'ensemble des résultats obtenus.

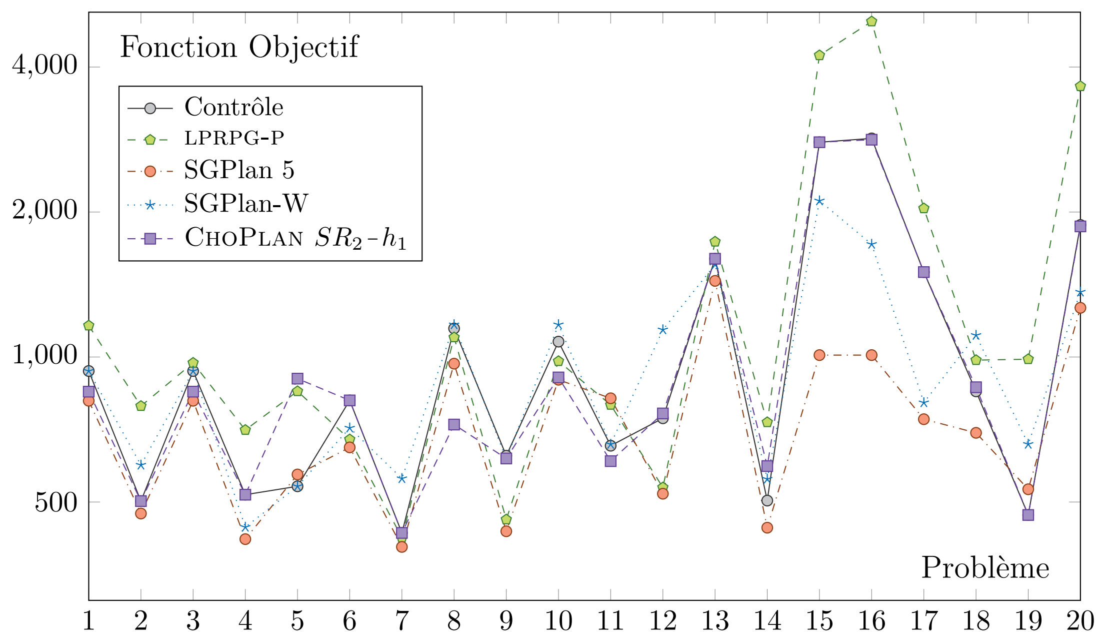

Figure 3.11 – Résultats pour *Rovers* - *Simple Preferences*

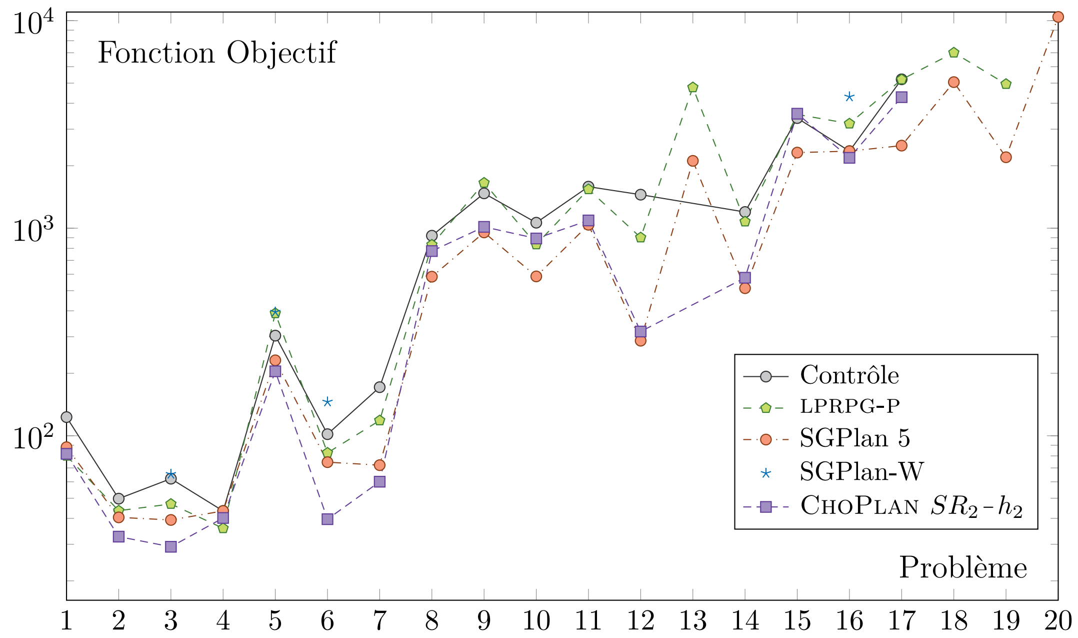

FIGURE 3.12 – Résultats pour *Rovers* - *Qualitative Preferences*

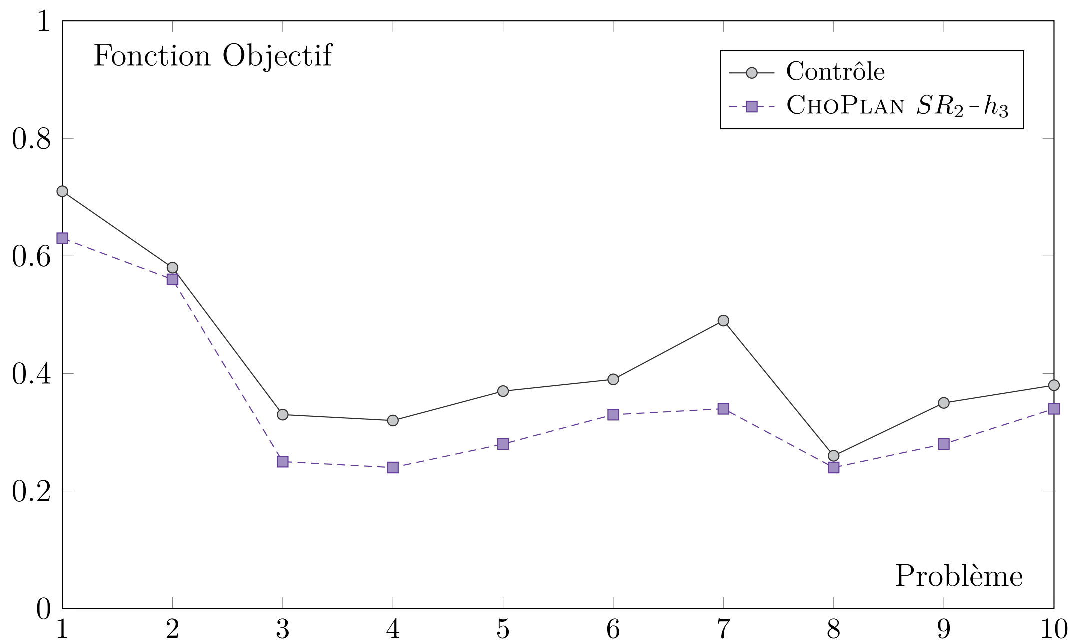

FIGURE 3.13 – Résultats pour *Rovers* - MAUT *Preferences*

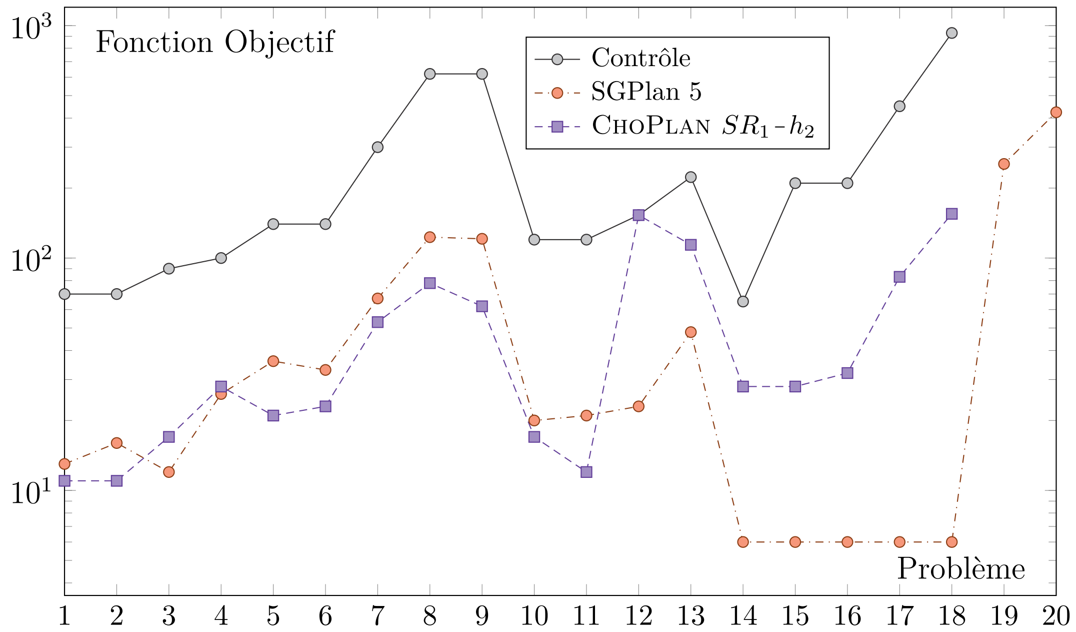

FIGURE 3.14 – Résultats pour *Openstacks* - *Simple Preferences*

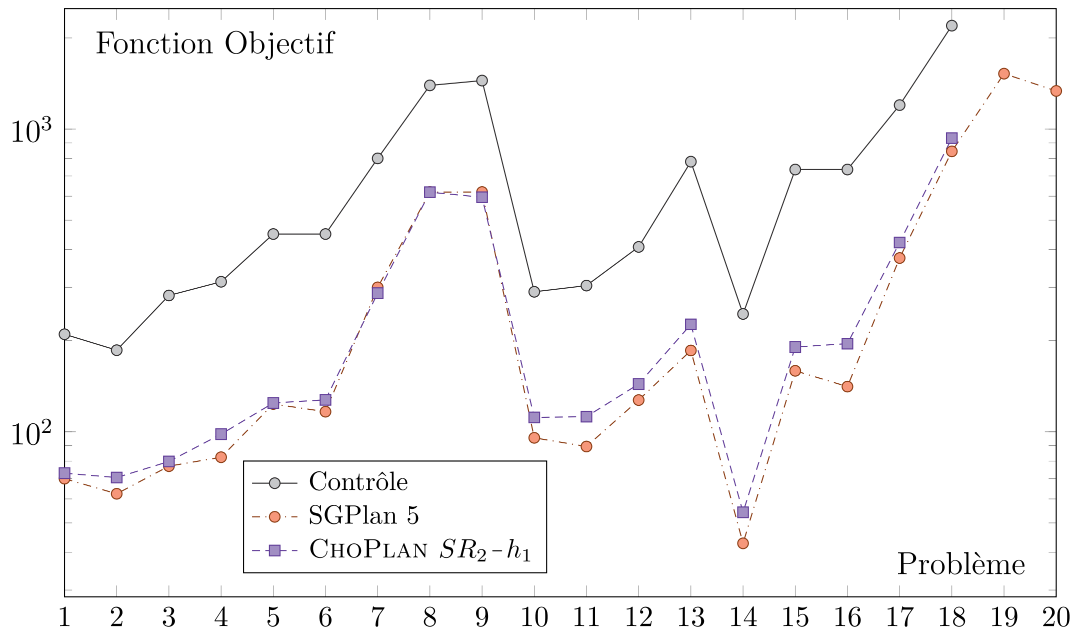

FIGURE 3.15 – Résultats pour *Openstacks* - *Qualitative Preferences*

Le tableau 3.3 présente le nombre de problèmes résolus par chacun des planificateurs considérés ainsi que l'efficacité de ces derniers. L'efficacité est définie comme le nombre de fois qu'un planificateur résout un problème en trouvant une solution dont la qualité est supérieure ou égale à celle de la solution retournée par le planificateur de Contrôle. Sans surprise, les problèmes de type *Rovers - MP* qui sont exprimés selon le formalisme PDDL3/MAUT n'ont été résolus que par les planificateurs dérivés de CHOPLAN.

Le planificateur SGPlan5 exhibe des performances remarquables puisqu'il est capable de résoudre tous les problèmes SP et QP (80/90) tout en obtenant un score d'efficacité élevé (77/90). Ces performances semblent néanmoins fortement liées aux optimisations réalisées spécifiquement pour le domaine considéré puisque SGPlan-W obtient quant à lui les moins bons résultats avec seulement 24 problèmes résolus.

Le nombre de solutions identifiées par le planificateur LPRPG-P est moyen (39/90) ce qui s'explique par l'incapacité de ce dernier à traiter les problèmes *Openstacks* (à cause d'une exigence PDDL non supportée). Toutefois, il convient de préciser que les tests réalisés ne reflètent pas pleinement le potentiel de LPRPG-P. En effet, ce dernier peut par ailleurs résoudre les problèmes *Pathways*, *Storage*, *TPP* et *Trucks* qui n'ont pas été retenus dans le cadre de cette étude.

CHOPLAN obtient quant à lui de bons résultats puisqu'il est capable de résoudre 82 des 90 problèmes considérés et ce avec une bonne efficacité (entre 82 et 86 selon la configuration employée). Les heuristiques utilisées par CHOPLAN peuvent être considérées comme des variantes de FAST-FORWARD qui consistent à faire dériver la recherche pour prendre en compte progressivement les préférences du problème. En conséquence, il est normal que CHOPLAN trouve exactement le même nombre de solutions que le planificateur Contrôle.

Ces premiers résultats rendent compte des performances propres des planificateurs (nombre de solutions trouvées et nombre de fois qu'un planificateur identifie une meilleure solution que celle retournée par Contrôle) sans pour autant comparer directement les planificateurs les uns par rapport aux autres.

Nombre Solutions & Efficacité	Contrôle	LPRPG-P	SGPlan 5	SGPlan-W	ChoPlan SR_1 - h_1	ChoPlan SR_1 - h_2	ChoPlan SR_1 - h_3	ChoPlan SR_2 - h_1	ChoPlan SR_2 - h_2	ChoPlan SR_2 - h_3
Rov. SP	20	20	20	20	20	20	20	20	20	20
Rov. QP	16	19	20	4	16	16	16	16	16	16
Rov. MP	10	-	-	-	10	10	10	10	10	10
OpS. SP	18	-	20	0	18	18	18	18	18	18
OpS. QP	18	-	20	0	18	18	18	18	18	18
Solutions	**82/90**	**39/90**	**80/90**	**24/90**	**82/90**	**82/90**	**82/90**	**82/90**	**82/90**	**82/90**
Rov. SP	-	6	17	11	15	14	13	16	14	13
Rov. QP	-	16	20	4	20	19	19	20	19	19
Rov. MP	-	0	0	0	10	10	10	10	10	10
OpS. SP	-	2	20	2	20	20	20	20	20	20
OpS. QP	-	2	20	2	20	20	20	20	20	20
Efficacité	**-**	**26/90**	**77/90**	**19/90**	**85/90**	**83/90**	**82/90**	**86/90**	**83/90**	**82/90**

TABLEAU 3.3 – Nombre de solutions identifiées et éfficacité des planificateurs

La notion de score IPC (qui comme son nom le suggère est utilisée lors des compétitions internationales de planification) permet d'analyser comparativement les performances d'un ensemble de planificateurs Π. Le score d'un planificateur π pour un problème p est défini par :

$$\textit{score}\,(\pi, p) \;=\; \textit{meilleure-qualité}\,(p) \;\div\; \textit{qualité}\,(\pi, p)$$

La fonction *meilleure-qualité* (p) retourne la qualité de la meilleure solution trouvée par l'ensemble des planificateurs de Π tandis que la fonction *qualité* (π, p) désigne la qualité de la solution identifiée par π. Si le planificateur π n'a pas résolu le problème p, alors $\textit{score}\,(\pi, p) = 0$. Le score IPC de π pour un ensemble P de problèmes vaut :

$$\textit{score-IPC}\,(\pi, P) \;=\; \sum_{p \in P} \textit{score}\,(\pi, p)$$

Le score IPC est un indicateur intéressant puisqu'il tient compte de la différence de qualité entre les solutions des planificateurs (en les comparant à la meilleure trouvée) tout en récompensant les planificateurs capables de résoudre un grand nombre de problèmes. Le tableau 3.4 présente les scores IPC des planificateurs étudiés. Les tendances observées avec le tableau 3.3 sont confirmées puisque SGPlan 5 obtient le meilleur score et SGPlan-W le moins bon tandis que les résultats de LPRPG-P sont moyens.

Avec des scores IPC compris entre 57,75 et 61,41 (soit un gain d'environ 50 % par rapport au planificateur de Contrôle), CHOPLAN exhibe des performances tout à fait satisfaisantes. Par ailleurs, les six configurations considérées ont des scores relativement similaires. Ceci semble indiquer que le mécanisme retenu (agrégation d'estimations à l'aide d'une intégrale de Choquet avec modification de la capacité lors de chaque itération) joue un rôle prépondérant dans les performances globales de CHOPLAN. Néanmoins, des différences notables entre configurations peuvent être observées pour certains domaines. Par exemple, SR_2 - h_2 est meilleure que SR_2 - h_1 sur *Openstacks* - *SP* mais moins efficace sur *Openstacks* - *QP* (voir tableau 3.5 pour une analyse détaillée).

Score IPC	Contrôle	LPRPG-P	SGPlan 5	SGPlan-W	CHOPLAN SR_1 - h_1	CHOPLAN SR_1 - h_2	CHOPLAN SR_1 - h_3	CHOPLAN SR_2 - h_1	CHOPLAN SR_2 - h_2	CHOPLAN SR_2 - h_3
Rov. SP	15,49	13,30	19,32	15,56	15,22	15,45	14,95	15,71	15,50	14,94
OpS. SP	2,09	0	17,00	0	4,24	11,21	7,54	4,31	10,93	7,66
Tot. SP	**17,58**	**13,30**	**36,31**	**15,56**	**19,46**	**26,66**	**22,49**	**20,02**	**26,43**	**22,60**
Rov. QP	8,94	11,53	18,06	1,54	13,34	13,52	13,31	13,49	13,74	13,46
OpS. QP	5,39	0	19,86	0	15,88	11,66	12,17	15,99	10,92	12,14
Tot. QP	**14,32**	**11,53**	**37,92**	**1,54**	**29,22**	**25,18**	**25,48**	**29,49**	**24,66**	**25,61**
Rov. MP	8,10	0	0	0	9,25	9,57	9,79	9,28	9,57	9,82
Total	**40,01**	**24,83**	**74,23**	**17,09**	**57,93**	**61,41**	**57,75**	**58,79**	**60,66**	**58,03**

TABLEAU 3.4 – Score IPC des planificateurs

Le tableau 3.5 synthétise les résultats obtenus par CHOPLAN de sorte à étudier l'impact des stratégies de recherche SR_1 et SR_2 ainsi que celui des heuristiques h_1, h_2 et h_3. Pour cela, tous les résultats impliquant SR_1 et SR_2 (respectivement h_1, h_2 et h_3) ont été additionnés et ce quelle que soit l'heuristique employée (respectivement la stratégie de recherche employée) puis triés en fonction du type de problème considéré (SP, QP ou MP).

Au vu de la très faible différence entre les scores de SR_1 - h_* et SR_2 - h_*, il apparait clairement que les performances de CHOPLAN sont identiques quelque soit la stratégie de recherche employée. Pour le temps de calcul considéré (10 minutes par problème), la variante SR_2 n'apporte donc pas de gain significatif quant à la qualité des solutions identifiées. Toutefois, il convient de préciser que le nombre de solutions intermédiaires identifiées par la stratégie SR_2 (non visible dans les résultats présentés qui ne retiennent que la meilleure solution) est généralement supérieur à celui de la stratégie SR_1. En conséquence, bien que l'apport de SR_2 par rapport à SR_1 soit négligeable dans le cas général, ce dernier pourrait être plus significatif pour des temps de calcul faible (de l'ordre de la seconde).

En ce qui concerne les heuristiques h_1, h_2 et h_3, les résultats sont un peu plus contrastés. L'heuristique h_2 qui associe une interprétation floue aux préférences PDDL3 est celle qui obtient les meilleures performances. Elle surpasse grandement h_1 sur les problèmes ne contenant que des préférences finales ce qui s'explique intuitivement par le fait que h_2 est une heuristique plus informée que h_1. De plus, elle obtient également de meilleurs résultats sur les problèmes de type MP bien que la différence observée soit plus faible. En revanche, l'heuristique h_1 est plus performante dans le cas des problèmes QP qui, pour rappel, contiennent des préférences finales et des préférences de trajectoires. Ce résultat peut sembler surprenant et son interprétation n'est a priori pas aisée. Il s'explique peut être par le traitement différent que subissent, dans l'heuristique h_2, les préférences `always` et `at-most-once` d'une part et les préférences `sometime` et `sometime-before` d'autre part. En effet, ce dernier pousse l'heuristique h_2 à être plus optimiste ou pessimiste au regard des préférences `always` et `at-most-once` (qui sont jugées soient vraies soient fausses) qu'elle ne l'est pour les préférences `sometime` et `sometime-before`

(qui sont jugées en fonction de l'effort à fournir pour les satisfaire). Ceci entraine implicitement une modification du calcul de l'utilité dont le résultat n'est alors plus entièrement représentatif des préférences du problème diminuant ainsi l'efficacité de l'heuristique. Une solution pourrait être de remplacer Λ_2 par deux estimations Λ_2^1 (responsable des préférences `always` et `at-most-once`) et Λ_2^2 (responsable des préférences `sometime`, `sometime-before` ainsi que des préférences finales). En agrégeant ces deux grandeurs correctement, il serait alors possible de créer une estimation Λ_2' qui ne souffrirait pas du défaut précédemment mentionné.

Le cas de l'heuristique h_3 est également intéressant. Bien qu'elle obtiennent globalement les plus mauvais résultats, il s'agit de l'heuristique la plus efficace pour les problèmes de type MP. Néanmoins, cette catégorie ne contenant que 10 problèmes, il convient de rester prudent quant à une éventuelle généralisation de ce résultat. Pour rappel, l'estimation Λ_3 (utilisée par h_3) a été construite via une somme pondérée de Λ_1 (utilisée par h_1) et de Λ_2 (utilisée par h_2). Ce comportement transparait dans le résultat final puisque les valeurs obtenues par h_3 sont comprises entre celles de h_1 et de h_2. Ainsi, il pourrait être intéressant de construire une estimation Λ_4 à partir d'une intégrale de Choquet qui représenterait une substituabilité parfaite entre Λ_1 et Λ_2. Une heuristique basée sur Δ_1 et Λ_4 combinerait peut être les avantages de h_1 et h_2 et pourrait ainsi être très performante sur l'ensemble des problèmes considérés.

Score IPC ChoPlan	$SR_1 - h_*$	$SR_2 - h_*$	$SR_* - h_1$	$SR_* - h_2$	$SR_* - h_3$
Tot. SP	68,60	69,06	39,49	53,09	45,09
Tot. QP	79,88	79,75	58,71	49,84	51,08
Tot. MP	28,61	28,67	18,53	19,15	19,60
Total	**177,09**	**177,48**	**116,72**	**122,08**	**115,78**

TABLEAU 3.5 – Synthèse des scores IPC des configurations de CHOPLAN

Les figures 3.16 à 3.18 comparent le planificateur CHOPLAN et les planificateurs Contrôle, LPRPG-P, SGPlan 5 et SGPlan-W. Ces graphiques ont pour abscisse la qualité des solutions identifiées par CHOPLAN et pour ordonnée la qualité des solutions identifiées par le planificateur auquel CHOPLAN est comparé. Seuls les problèmes ayant été résolus par les deux planificateurs sont représentés. De plus, la fonction identité est également représentée afin de faciliter l'analyse. Lorsqu'un point est au dessus de la courbe de la fonction identité, cela signifie que la solution retournée par CHOPLAN pour le problème en question est meilleure que celle du planificateur auquel il est comparé. La différence de qualité entre les solutions identifiées par les planificateurs est d'autant plus grande que le point est éloigné de la courbe de la fonction identité. Afin d'apprécier correctement cette différence de qualité, il convient de remarquer que l'échelle utilisée sur ces figures est logarithmique. En outre, la configuration de CHOPLAN retenue pour cette analyse est celle ayant conduit à l'identification des meilleurs résultats à savoir SR_1 - h_2.

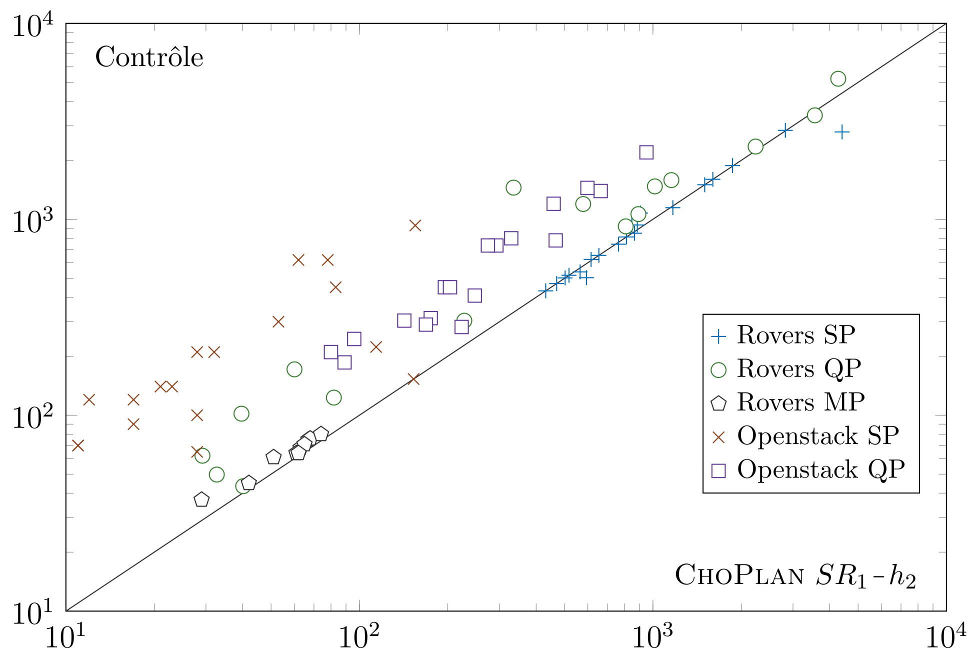

FIGURE 3.16 – Comparaison de CHOPLAN SR_1 - h_2 et de Contrôle

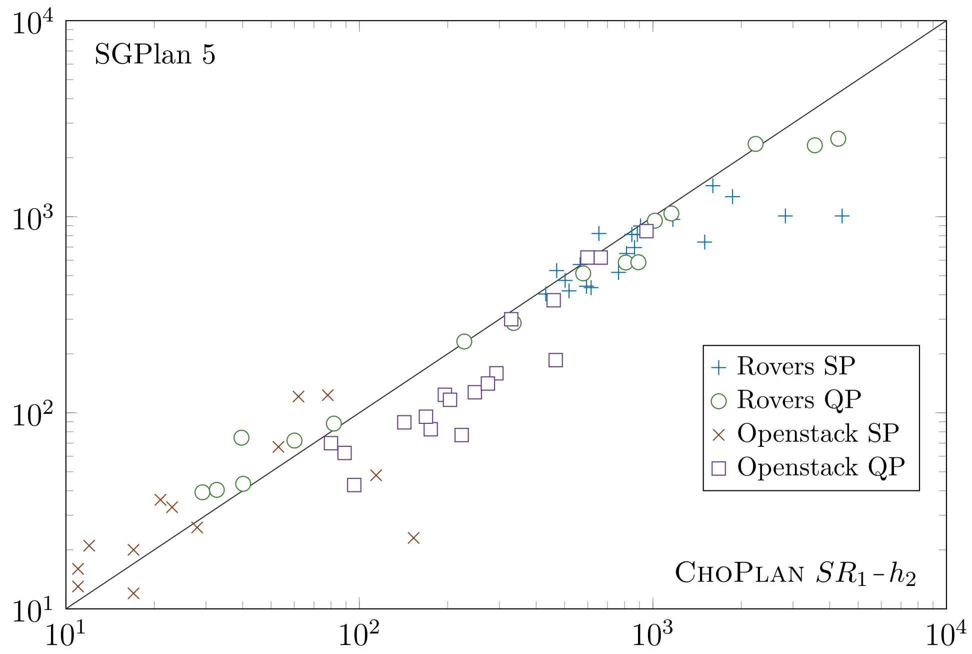

FIGURE 3.17 – Comparaison de CHOPLAN SR_1 - h_2 et de SGPlan 5

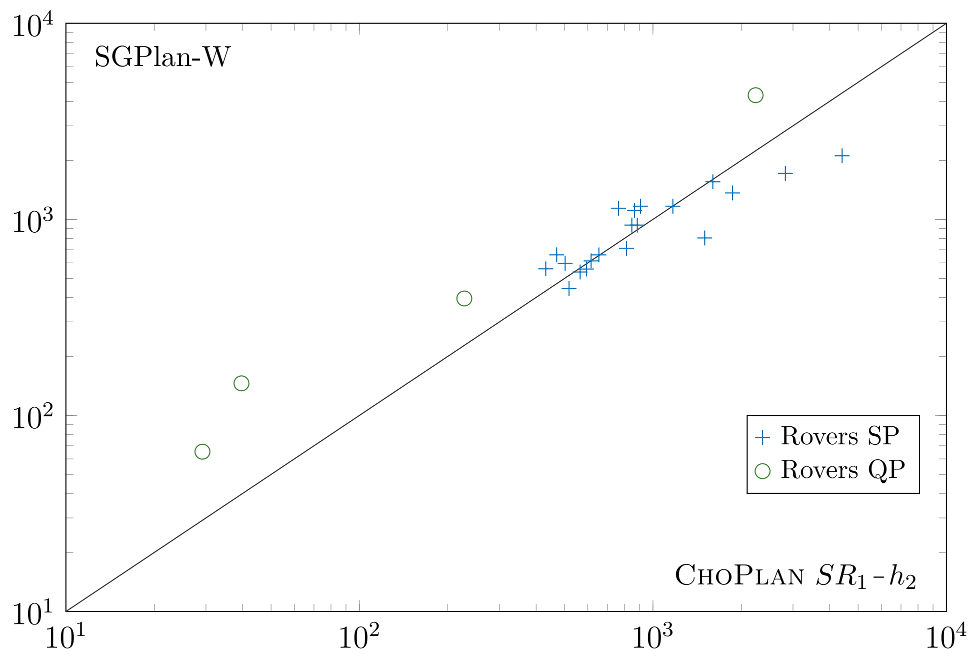

FIGURE 3.18 – Comparaison de CHOPLAN SR_1 - h_2 et de SGPlan-W

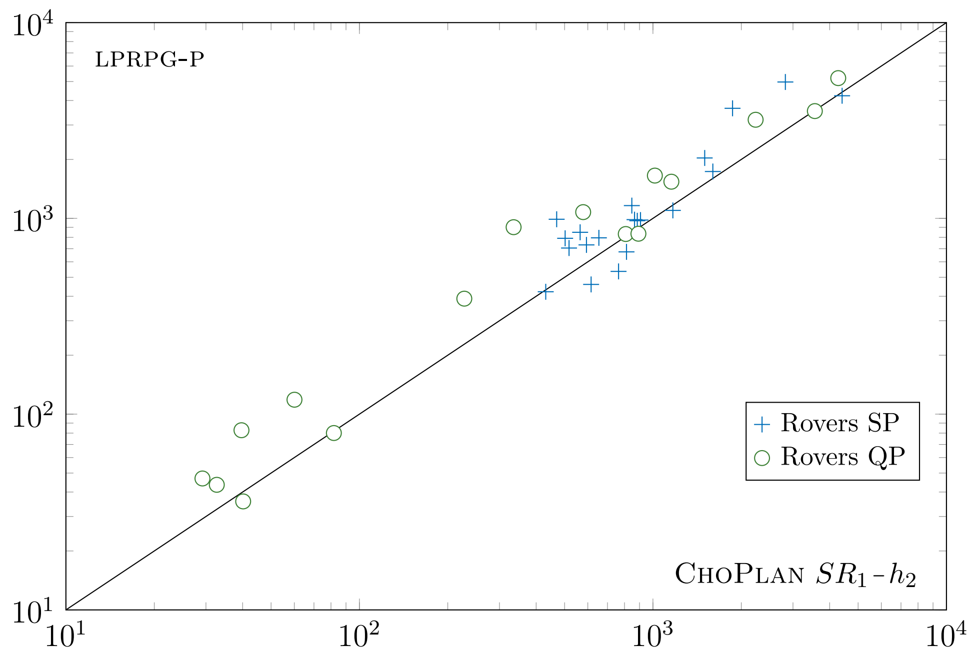

FIGURE 3.19 – Comparaison de CHOPLAN SR_1 - h_2 et de LPRPG-P

Pour finir, il reste à étudier l'impact de la règle de coupe mentionnée dans la section 3.2.3. Le tableau 3.6 présente la différence de temps de calcul observée lorsque CHOPLAN SR_1 - h_2 est utilisé avec et sans la règle de Pareto-dominance forte. Le gain observé gain varie fortement en fonction des problèmes considérés ce qui s'explique intuitivement par le fait que certains domaines sont naturellement plus prompts à engendrer des nœuds Pareto-dominés que d'autres. C'est par exemple le cas du domaine *Rovers* dans lequel un robot peut effectuer des allers-retours entre deux positions. Bien qu'une telle action puisse être inutile du point de vue de l'atteinte des objectifs, elle peut néanmoins dégrader un critère de coût de déplacement.

Pareto-F	**Gain Temps Calcul**
Rovers SP	42,65 %
Rovers QP	34,23 %
Openstack SP	10,73 %
Openstack QP	12,84 %

TABLEAU 3.6 – Impact de la règle de Pareto-dominance forte

Conclusion

Ce chapitre a débuté par la description du formalisme PDDL3/MAUT qui généralise le langage PDDL3 par l'introduction de préférences floues. Ce dernier enrichit le pouvoir expressif du PDDL3 puisqu'il permet notamment de représenter facilement plusieurs préférences numériques, de construire des préférences par agrégation d'autres préférences ou encore de tenir compte des interactions entre les différentes préférences du problème. Un algorithme itératif de résolution de problèmes de planification avec préférences a ensuite été présenté. Celui-ci s'appuie sur des heuristiques qui évaluent le potentiel d'un nœud à l'aide de deux estimations respectivement relatives aux objectifs à atteindre et aux préférences du problème. L'originalité de ces heuristiques consiste à agréger ces deux estimations à l'aide d'une intégrale de Choquet. De plus, une règle de coupe permettant d'identifier et de supprimer les nœuds qu'il n'est pas nécessaire de visiter lors de la recherche a été proposée. Le planificateur CHOPLAN repose sur ces mécanismes et a été implémenté dans le cadre de cette étude. Ses performances ont été comparées à celles de plusieurs planificateurs de l'art. Les résultats obtenus suggèrent que l'approche proposée est efficace puisque CHOPLAN identifie de meilleures solutions que celles proposées par le planificateur de Contrôle, SGPlan-W et LPRPG-P. Sur l'ensemble des problèmes considérés, seuls les résultats du planificateur SGPlan 5 (qui pour rappel utilise des mécanismes d'optimisation différents pour chaque domaine) surpassent ceux de CHOPLAN.