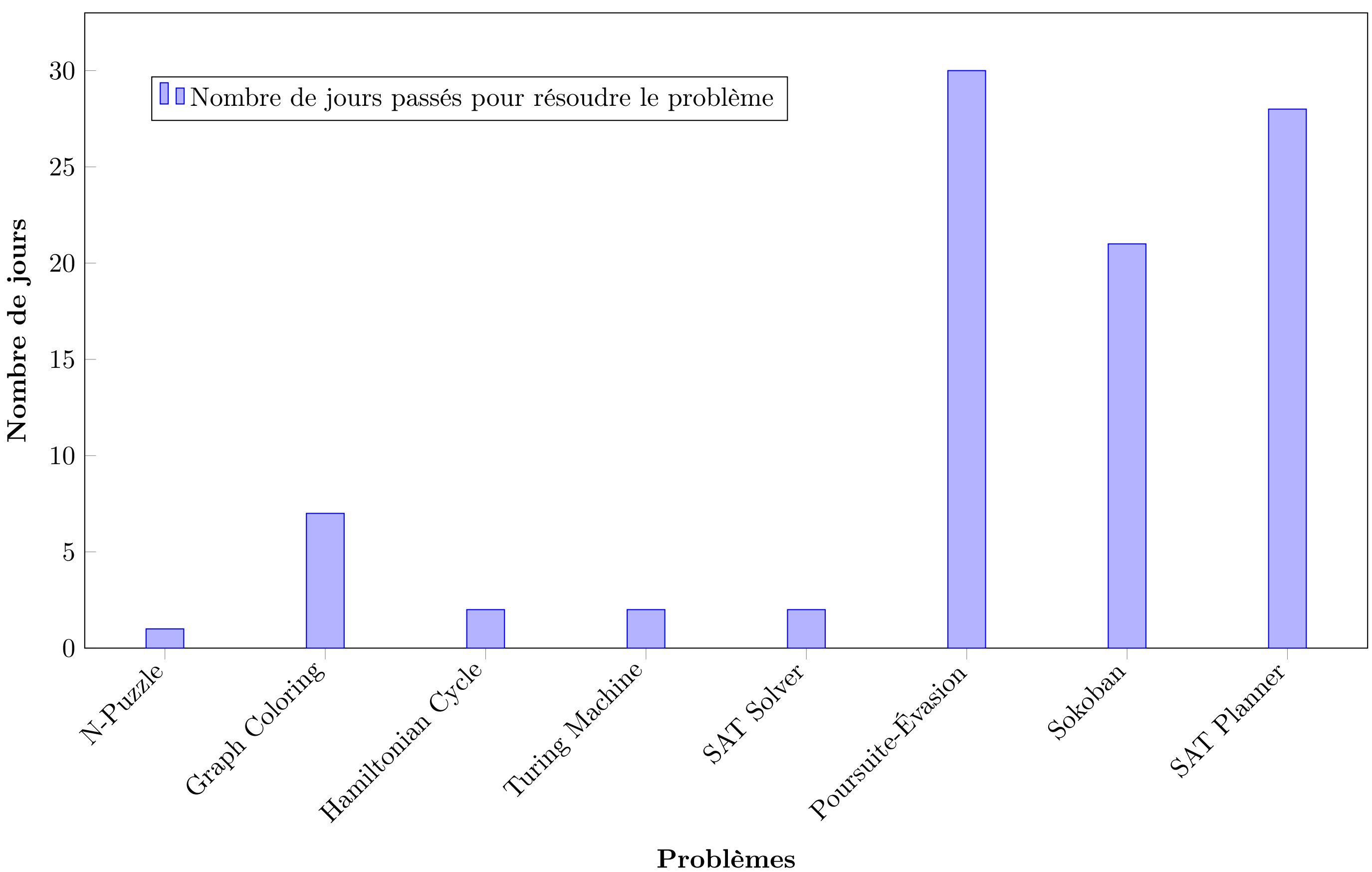
Compte Rendu

PATIA - PDDL4J Exercises's

Table des matières

Histogramme



GitHub
ALVES Romain *BOSSY Romain*
COMBE Gaétan *FAURIE Alban*

1 N-Puzzle

1.1 Description :

Le jeu N-Puzzle *(ou Taquin)* est un jeu de glissement comportant 9 tuiles carrées numérotées de 1 à 8 dans un cadre qui mesure 3 tuiles de haut et 3 tuiles de large, laissant ainsi une position de tuile vide. Les tuiles dans la même rangée ou colonne que la position vide peuvent être déplacées en les faisant glisser horizontalement ou verticalement, respectivement. Le but du puzzle est de placer les tuiles dans l'ordre numérique :

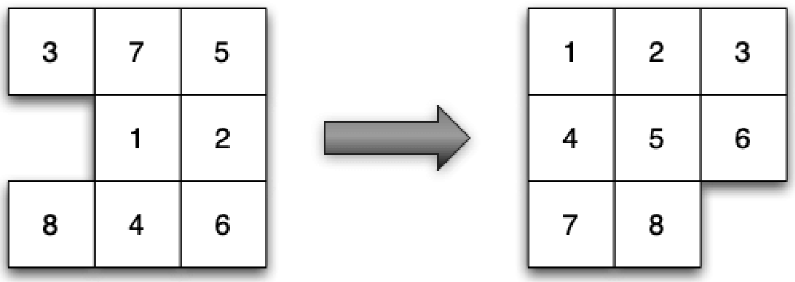


FIGURE 1 – Exemple de N-Puzzle avec 8 tuiles

1.2 Domaine :

1.2.1 Objets :

- **Case :** Les cases du jeu sur lesquelles sont présents les *nombres.*
- **Nombres :** Pièces du jeu à déplacer.
- **Direction :** Direction dans lesquelles sont déplaçables les pièces.

1.2.2 Actions :

- **Déplacer :**
  - **Paramètres :** `(?a ?b - case ?d - direction ?n - nombre)` :
    La pièce numérotée n est sur la case b et veut être déplacée sur la case a qui est dans la direction d par rapport à la case b.
  - **Préconditions :** `(and (vide ?a) (sur ?n ?b) (adjacente ?a ?b ?d))` :
    La case a est vide, la pièce numérotée n est présente sur la case b, et les cases a et b sont adjacentes dans la direction d.
  - **Effets :** La pièce n n'est plus sur la case b, qui devient alors vide, et est désormais sur la case a, qui n'est quant à elle plus vide.

1.2.3 Prédicats :

- **vide :** `(vide ?c - case)` : Vrai lorsque la case c est vide.
- **sur :** `(sur ?n - nombre ?c - case)` : Vrai si la pièce numérotée n est sur la case c.
- **adjacente :** `(adjacente ?a ?b - case ?d - direction)` : Vrai si la case a est adjacente à la case b dans la direction d *(a vers b).*

1.3 Problème :

Le problème est tel qu'il est présenté sur la Figure 1.

1.3.1 Objets :

- 9 cases : *a b c d e f g h i*, représentant les cases du taquin.
- 8 nombres : *n1 n2 n3 n4 n5 n6 n7 n8*, représentant les pièces du taquin.
- 4 directions : *haut bas gauche droite*, représentant les directions dans lesquelles ont peut pousser les pièces du taquin.

1.3.2 Etat inital :

- On associe chaque nombre à une case grâce au prédicat **(sur nombre case)** par exemple les prédicats (sur n3 a) (sur n7 b) associe le nombre 3 à la case a et le nombre 7 a la case b. On fait cela pour huit cases pour un terrain 3*3.
- On associe la case restante non associée à un nombre à vide grâce au prédicat **(vide nombre)**. Par exemple (vide d) associe la case d à vide.
- On donne la position de chaque case entre elles. Pour cela, on utilise le prédicat (adjacente nombre1 nombre2). Dans notre problème, on dit que la case a est à gauche de la case b. La case b est aussi a droite de a : ainsi, on a les prédicats (adjacente a b gauche) (adjacente b a droite).

1.3.3 Etat final :

- On donne la liste des associations des nombres aux cases, pour un terrain 3*3 on aura 8 prédicats. Par exemple, on veut que la case a soit associé à la valeur 1, la case b soit associé à la valeur 2, on a donc les prédicats (sur n1 a) (sur n2 b).
- la dernière case non associé doit avoir le prédicat (vide case). Par exemple la case i est vide, on a donc le prédicat (vide i).

2 Graph Coloring

2.1 Description :

En théorie des graphes, colorier un graphe signifie attribuer une couleur à chacun de ses sommets de telle sorte que deux sommets reliés par une arête soient de différentes couleurs. Le but est d'utiliser un nombre minimal de couleurs :

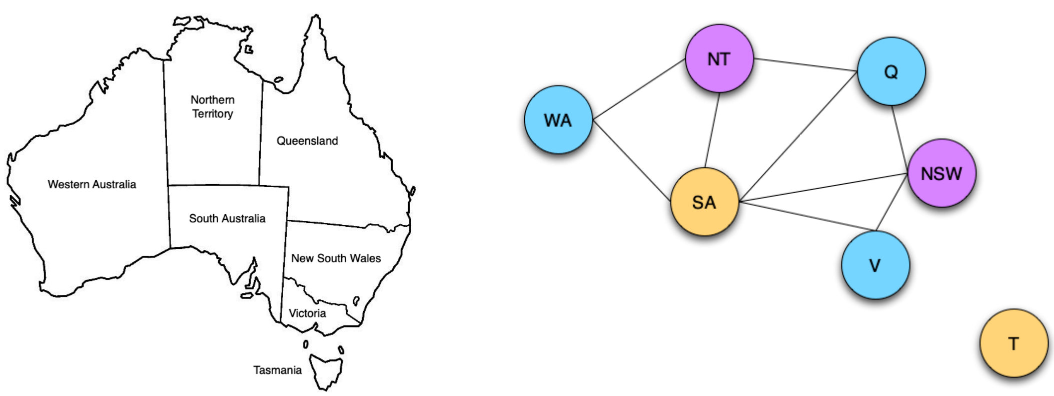


FIGURE 2 – Colorisation d'un graphe

La figure ci-dessus est la carte de l'Australie avec sa représentation sous forme de graphe. Dans ce cas, le nombre de couleurs est 3.

2.2 Domaine :

2.2.1 Objets :

- **Nœud :** Un nœud du graphe.
- **Arête :** Une arête liant deux nœuds du graphe.
- **Couleur :** Couleur du nœud.

2.2.2 Actions :

- **colore_2nœudss :**
  - **Paramètres :** (`?n1 ?n2 - nœud ?a - arrete ?c1 ?c2 - couleur`) :

ALVES Romain *BOSSY Romain*
COMBE Gaétan *FAURIE Alban*

- Les nœuds *n1* et *n2* sont reliés par l'arête *a* et seront coloriés par des couleurs différentes *c1* et *c2*.
  - **Préconditions :** `(and (vide ?n1) (vide ?n2) (sûr ?n1 ?n2 ?a) (sûr ?n2 ?n1 ?a) (diff ?c1 ?c2))` :
    - Les deux nœuds *n1* et *n2* ne sont pas coloriés et sont reliés par l'arête *a* et les couleurs *c1* et *c2* sont différentes.
  - **Effets :** `and (aCouleur ?n1 ?c1) (aCouleur ?n2 ?c2) (not (vide ?n1)) (not (vide ?n2)) (marquee ?a)` :
    - Le nœud *n1* est colorié avec la couleur *c1*, le nœud *n2* est colorié avec la couleur *c2* et ne sont donc plus vide, l'arête *a* reliant les deux nœuds est ainsi marquée.
- **colore_nœuds :**
  - **Paramètres :** `(?n1 ?n2 - noeuds ?a - arrete ?c1 ?c2 - couleur)` :
    - Les nœuds *n1* et *n2* sont reliés par l'arête *a*. le nœud *n1* est colorié avec la couleur *c1*, et la couleur *c2* devra colorier le nœud *n2*.
  - **Préconditions :** `(and (vide ?n1) (aCouleur ?n2 ?c2) (sur ?n1 ?n2 ?a) (sur ?n2 ?n1 ?a) (diff ?c1 ?c2))` :
    - le nœud *n1* n'est pas colorié et est relié par l'arête *a* au nœuds *n2*. Ce dernier est colorié par la couleur *c2*, qui est différente de la couleur *c1* qui sera utilisée pour colorier le nœud *n1*.
  - **Effets :** `and (aCouleur ?n1 ?c1) (not (vide ?n1)) (marquee ?a)` :
    - le nœud *n1* est colorié avec la couleur *c1*, ce qui permet de marquer l'arête *a*.

  e
- **marque_arrete :**
  - **Paramètres :** `(?n1 ?n2 - noeuds ?a - arrete ?c1 ?c2 - couleur)` :
    - L'arête *a* relie les nœuds *n1* et *n2* coloriés avec les couleurs *c1* et *c2*.
  - **Préconditions :** `(and (aCouleur ?n1 ?c1) (aCouleur ?n2 ?c2) (sur ?n1 ?n2 ?a) (sur ?n2 ?n1 ?a) (diff ?c1 ?c2))` :
    - Les nœuds *n1* et *n2* sont respectivement coloriés avec les couleurs différentes *c1* et *c2* et reliés par l'arête *a*.
  - **Effets :** `and (marquee ?a)` :
    - L'arête *a* est marquée.
- **colore_seul :**
  - **Paramètres :** `(?n - noeuds ?c - couleur))` :
    - le nœud *n* devra être colorié par la couleur *c*.
  - **Préconditions :** `(and (vide ?n))` :
    - le nœud *n* n'est pas colorié.
  - **Effets :** `and (not (vide ?n))(aCouleur ?n ?c)` :
    - le nœud *n* est colorié avec la couleur *c*.

2.2.3 Prédicats :

- **aCouleur :** `(aCouleur ?n - noeuds ?c - couleur)` : Vrai si le nœud *n* a la couleur *c*.
- **vide :** `(vide ?n - noeuds)` :; Vrai si le nœud *n* n'est pas encore colorié.
- **sur :** `(sur ?n1 ?n2 - noeuds ?a - arrete)` : Vrai si les nœudss *n1* et *n2* sont sur l'arête *a* (pas de symétrie !).
- **marquee :** `(marquee ?a - arrete)` : Vrai si l'arête *a* est marquée.
- **diff :** `(diff ?c1 ?c2 - couleur)` : Vrai si les deux couleurs *c1*et *c2* sont différentes.

2.3 Problème :

2.3.1 Objets :

- Ensemble de nœuds : *a b c*.
- Ensemble d'arêtes : *a1 a2 a3*.
- Ensemble de couleurs : *rouge bleu vert*.

ALVES Romain *BOSSY Romain*
COMBE Gaétan *FAURIE Alban*

2.3.2 Etat initial :

- Nous devons préciser que toutes les couleurs sont différentes, on rajoute ainsi les prédicats `(diff rouge bleu)` `(diff bleu vert)` `(diff vert rouge)`.
- Tous les nœuds sont sans couleur, donc on a les prédicats `(vide a)` `(vide b)` `(vide c)`.
- Il faut indiquer comment les nœuds sont liés entre eux, par exemple l'arête a1 est l'arc reliant les nœuds a et b est décrit par le prédicat `(sur a b a1)` `(sur b a a1)`.

2.3.3 Etat final :

- Il faut indiquer que toutes les arêtes sont marquées ce qui donne `(marquee a1)` `(marquee a2)` `(marquee a3)`.

3 Hamiltonian Cycle

3.1 Description :

En théorie des graphes, un graphe hamiltonien est un graphe possédant au moins un cycle traversant tous les sommets une et une seule fois. Un tel cycle élémentaire est alors appelé un cycle hamiltonien :

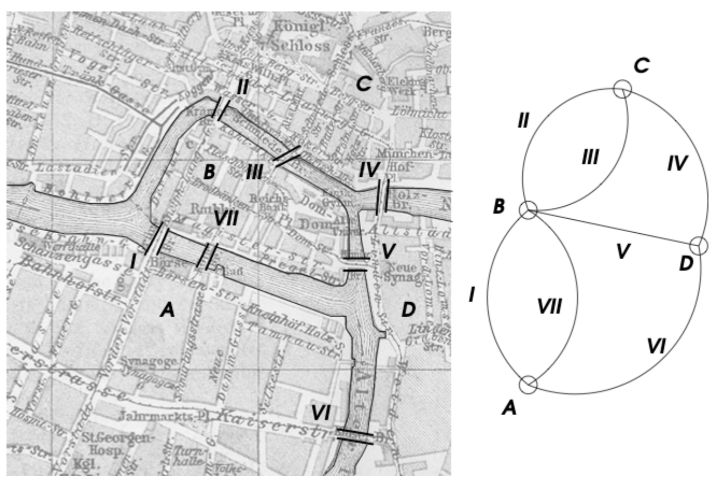


FIGURE 3 – Exemple de Cycle Hamiltonien

3.2 Domaine :

3.2.1 Objets :

- **Node :** Nœuds du graphe.
- **Edge :** Arêtes du graphe reliant les nœuds.

3.2.2 Actions :

- **move :**
  - **Paramètres :** `(?from ?to - node ?e - edge)` :
    On veut aller du nœud *from* pour aller vers le nœud *to* en passant par l'arête *e*.
  - **Préconditions :** `(and (connected ?from ?to ?e) (on ?from) (leftToVisit ?to))` :
    Les noeuds *from* et *to* sont connectés via l'arête *e*, on est actuellement sur le noeud *from* et le noeud *to* n'a pas encore été visité.
  - **Effets :** `(and (not (on ?from)) (on ?to) (not (leftToVisit ?to))(visited ?to)`
    On est sur le noeud *to*, qui est désormais visité, et plus sur le noeud *from*.
- **move_to_start :**
  - **Paramètres :** `(?from ?to - node ?e - edge)` :
    On est sur le noeud *from* et on veut retourner au noeud de départ *to* via l'arête *e*.

- **Préconditions :** `(and (connected ?from ?to ?e) (on ?from) (leftToVisit ?to) (start ?to))` : Les noeuds *from* et *to* sont connectés via l'arête *e*, on est actuellement sur le noeud *from* et le noeud *to* est bien le noeud de départ.
- **Effets :** `(and (not (on ?from)) (on ?to) (not (leftToVisit ?to)) (visited ?to))` On est sur le noeud *to*, qui est le noeud de départ, et plus sur le noeud *from*.

3.2.3 Prédicats :

- **start :** (`start ?n - node`) : Vrai si le noeud n est le noeud de départ.
- **on :** (`on ?n - node`) : Vrai si on se trouve sur le noeud n.
- **visited :** (`visited ?n - node`) : Vrai si le noeud n a été visité.
- **leftToVisit :** (`leftToVisit ?n - node`) : Vrai si le noeud n n'a pas encore été visité.
- **connected :** (`connected ?n1 ?n2 - node ?e - edge`) : Vrai si les noeuds *n1* et *n2* sont reliés par l'arête *e*.

3.3 Problème :

3.3.1 Objets :

- Ensemble de nœuds : `a b c d`.
- Ensemble d'arêtes *(Edge)* : `I II III IV V VI VII`.

3.3.2 État initial :

- Premièrement, on ajoute le nœud de départ qui indique où le cycle commence. Par exemple, pour indiquer que le noeud de départ est a, on ajoute le prédicat `(start a)`.
- On ajoute la position courante, à l'état initial, le nœud du prédicat `(on a)` est le même noeud que du prédicat start.
- Il faut préciser tous les nœuds restants à visiter, c'est-à-dire tous les noeuds, on a donc les prédicats `(leftToVisit a) (leftToVisit b) (leftToVisit c) (leftToVisit d)`.
- Il faut rajouter deux prédicats par arête indiquant a quels noeuds i et j cette dernière est connectée, afin que la symétrie soit appliquée : `(connected i j I) (connected j i I)`

3.3.3 État final :

- On finit sur le même noeud de départ, donc avec le prédicat `(on a)`.
- Tous les nœuds doivent avoir été visités pour former un cycle, ainsi on rajoute les prédicats `(visited a) (visited b) (visited c) (visited d)`.

4 Turing Machine

4.1 Description :

Une Machine de Turing (MT) est un modèle mathématique de calcul décrivant une machine abstraite qui manipule des symboles sur une bande de papier selon une table de règles. Malgré la simplicité du modèle, il est capable de mettre en œuvre n'importe quel algorithme informatique.
La machine fonctionne sur une bande mémoire infinie divisée en cellules discrètes, chacune pouvant contenir un seul symbole tiré d'un ensemble fini de symboles appelé l'alphabet de la machine. Elle possède une "tête" qui, à tout moment de l'opération de la machine, est positionnée sur l'une de ces cellules, et un "état" sélectionné dans un ensemble fini d'états. À chaque étape de son fonctionnement, la tête lit le symbole dans sa cellule. Ensuite, en fonction du symbole et de l'état actuel de la machine, celle-ci écrit un symbole dans la même cellule et déplace la tête d'une étape vers la gauche ou la droite, ou arrête le calcul. Le choix du symbole de remplacement à écrire et de la direction du mouvement est basé sur une table finie qui spécifie quoi faire pour chaque combinaison de l'état actuel et du symbole lu.

4.2 Domaine :

4.2.1 Objets :

- **Cell :** Case de la machine de Turing.
- **State :** État de la machine de Turing.
- **Symbol :** Symboles lisibles par la mahcine de Turing.

ALVES Romain *BOSSY Romain*
COMBE Gaétan *FAURIE Alban*

4.2.2 Actions :

- **step_left :**
  - **Paramètres :** (`?zc ?zd - state ?sr ?sw - symbol ?c1 ?c2 - cell`) :
    - On est dans l'état *zc* sur la case *c1* où est inscrit le symbole *sr*, et on veut atteindre l'état *zd*, qui est à gauche de la case *c2*, qui possède quant à elle le symbole *sw*.
  - **Préconditions :** (`and (inState ?zc) (headOn ?c1) (hasSymbol ?sr ?c1) (atRight ?c1 ?c2) (transition_left ?zc ?zd ?sr ?sw)`) :
    - On est dans l'état *zc* avec la tête de lecture sur la case *c1* et le symbole *sr*. La case *c2* est à gauche de la case *c2* et la transition vers la gauche est possible.
  - **Effets :** `(and (not (inState ?zc)) (inState ?zd) (not (hasSymbol ?sr ?c1))(hasSymbol ?sw ?c1) (not (headOn ?c1)) (headOn ?c2)` :
    - On n'est plus dans l'état *zc* mais dans l'état *zd*. La case n'a plus le symbole *sr* mais le symbole *sw*. La tête de lecture n'est plus sur la case *c1* mais sur la case *c2*.
- **step_right :**
  - **Paramètres :** (`?zc ?zd - state ?sr ?sw - symbol ?c1 ?c2 - cell`) :
    - On est dans l'état *zc* sur la case *c1* où est inscrit le symbole *sr*, et on veut atteindre l'état *zd*, qui est à droite de la case *c2*, qui possède quant à elle le symbole *sw*.
  - **Préconditions :** (`(and (inState ?zc) (headOn ?c1) (hasSymbol ?sr ?c1) (atLeft ?c1 ?c2) (transition_right ?zc ?zd ?sr ?sw)`) :
    - On est dans l'état *zc* avec la tête de lecture sur la case *c1* et le symbole *sr*. La case *c2* est à gauche de la case *c2* et la transition vers la gauche est possible.
  - **Effets :** `(and (not (inState ?zc)) (inState ?zd)(not (hasSymbol ?sr ?c1)) (hasSymbol ?sw ?c1) (not (headOn ?c1)) (headOn ?c2)` :
    - On n'est plus dans l'état *zc* mais dans l'état *zd*. La case n'a plus le symbole *sr* mais le symbole *sw*. La tête de lecture n'est plus sur la case *c1* mais sur la case *c2*.
- **step_idle :**
  - **Paramètres :** (`?zc ?zd - state ?sr ?sw - symbol ?c1 - cell`) :
    - On est dans l'état *zc* et on veut passer dans l'état *zd*. On lit le symbole *sr* et on veut écrire le symbole *sw*. On est et on reste dans la case *c1*.
  - **Préconditions :** (`(and (inState ?zc) (headOn ?c1) (hasSymbol ?sr ?c1) (transition_idle ?zc ?zd ?sr ?sw))`) :
    - On est dans l'état *zc* avec la tête de lecture sur la case *c1* et le symbole *sr*. La case *c2* est à gauche de la case *c2* et la transition vers le nouvel état est possible.
  - **Effets :** `(and (not (inState ?zc)) (inState ?zd) (not (hasSymbol ?sr ?c1)) (hasSymbol ?sw ?c1)` :
    - On n'est plus dans l'état *zc* mais dans l'état *zd*. Sur la case *c1*, il n'y a plus le symbole *sr* mais le symbole *sw*.

4.2.3 Prédicats :

- **inState :** (`inState ?z - state`) : Vrai si la machine est dans l'état *z*.
- **headOn :** (`headOn ?c - cell`) : Vrai si la tête de lecture est sur la case *c*.
- **hasSymbol :** (`hasSymbol ?s - symbol ?c - cell`) : Vrai si le symbole *s* est écrit sur la case *c*.
- **atLeft :** (`atLeft ?a ?b -cell`) : Vrai si la case *a* se trouve à gauche de la case *b*.
- **atRight :** (`atRight ?a ?b - cell`) : Vrai si la case *a* se trouve à droite de la case *b*.
- **transition_left :** (`transition_left ?zc ?zd - state ?sr ?sw - symbol`) : Vrai si la transition vers la gauche est possible, depuis l'état courant *zc* vers l'état *zd*, avec le symbole *sr* lu sur la bande et le symbole *sw* écrit sur la bande.
- **transition_right :** (`transition_right ?zc ?zd - state ?sr ?sw - symbol`) : Vrai si la transition vers la droite est possible, depuis l'état courant *zc* vers l'état *zd*, avec le symbole *sr* lu sur la bande et le symbole *sw* écrit sur la bande.
- **transition_idle :** (`transition_idle ?zc ?zd - state ?sr ?sw - symbol`) : Vrai si la transition sans déplacement est possible, depuis l'état courant *zc* vers l'état *zd*, avec le symbole *sr* lu sur la bande et le symbole *sw* écrit sur la bande.

4.3 Problème :

4.3.1 Objets :

- Ensemble d'états : `z0 z1 halt`.
- Ensemble de symboles : `blank zero one`.
- Ensemble de cellules sur la bande : `c0 c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 c11 c12 c13 c14 c15`.

4.3.2 État initial

- On indique la position de la tête de lecture grâce au prédicat `(headOn c8)`.
- On précise dans quel état la machine de Turing commence, dans notre cas on peut le décrire par le prédicat `(inState z0)`
- Pour chaque cellule de la bande, il faut préciser le symbole écrit dessus. Par exemple sur la cellule c1, on a le symbole blank. Ainsi on ajoute le prédicat `(hasSymbol blank c1)`
- On doit rajouter un prédicat par transition de la machine de Turing. Par exemple le quintuplet `(<-, z0, z1, blank, blank)` donnera le prédicat `(transition_left z0 z1 blank blank)`. Le déplacement de la machine de Turing est dans le nom du prédicat, z0 est l'état de départ, z1 l'état d'arrivée, le premier blank est le symbole lu, et le dernier symbole blank est le symbole écrit. Pour pouvoir aller sur la droite, on utilise le prédicat `(transition_right z0 z0 zero zero)`.
- Il faut indiquer quelles cellules sont à gauche et à droite. Par exemple la cellule c1 est à gauche de la cellule c2 et à droite de la cellule c0 ce qui donne les prédicats `(atLeft c1 c2) (atRight c1 c0)`. Il faut le faire pour chaque cellule de notre bande.

4.3.3 État final

- Un seul prédicat `(inState halt)` est présent pour l'état final, qui indique que l'état de la machine de Turing est sur l'état arrêt.

5 SAT Solver

5.1 Description :

Étant donné une formule en FNC telle que $(x_1 \vee x_3 \vee \neg x_4) \wedge (x_4) \wedge (x_2 \vee \neg x_3)$. Un problème SAT consiste à déterminer s'il existe une attribution de valeurs aux variables propositionnelles de sorte que la formule évalue à vrai. Ici, une solution est : $x_1 = x_2 = x_3 = true$.

Il convient de noter que, si nous attribuons la valeur vrai à une variable, toutes les clauses contenant cette variable peuvent être supprimées de la formule SAT à résoudre, et peut être supprimée des clauses le contenant. De même, si nous attribuons la valeur faux à une variable, toutes les clauses contenant cette variable peuvent être supprimées de la formule SAT à résoudre, et peut être supprimée des clauses le contenant.

5.2 Domaine :

5.2.1 Objets :

- **Variable :** Variable d'un problème SAT.
- **Clause :** Clause d'un problème SAT.

5.2.2 Actions :

- **set_var_true :** La variable v prend pour valeur *True* et ne pourra pas prendre une autre valeur.
  - **Paramètres :** `(?v - variable)` :
    - On a la variable v que l'on souhaite passer à *True*.
  - **Préconditions :** `(and (hasNoValue ?v))` :
    - La variable v n'a préalablement pas de valeur assignée.
  - **Effets :** `(and (not (varIsFalse ?v)) (varIsTrue ?v) (not (hasNoValue ?v)))` :
    - La variable v est instanciée à *True*, donc elle n'est pas instanciée à *False* et a désormais une valeur.

ALVES Romain *BOSSY Romain*
COMBE Gaétan *FAURIE Alban*

- **set_var_false :** La variable v prend pour valeur *False* et ne pourra pas prendre une autre valeur.
  - **Paramètres :** (`?v - variable`) :
    - On a la variable v que l'on souhaite passer à *False*.
  - **Préconditions :** (`and (hasNoValue ?v)`) :
    - La variable v n'a préalablement pas de valeur assignée.
  - **Effets :** `(and (not (varIsTrue ?v)) (varIsFalse ?v) (not (hasNoValue ?v)))` :
    - La variable v est instanciée à *False*, donc elle n'est pas instanciée à *True* et a désormais une valeur.
- **set_clause_true :** La clause c devient est vraie car au moins une de ses variables devant être vraie à pour valeur *True*.
  - **Paramètres :** (`?c clause ?v - variable`) :
    - On a la clause c avec la variable v que l'on souhaite passer à *True*.
  - **Préconditions :** (`and (varTrueIn ?v ?c) (varIsTrue ?v)`) :
    - La variable v est à *True* dans la clause c, et la variable v est à *True*.
  - **Effets :** `(and (clauseTrue ?c))` :
    - La clause c est instanciée à *True*.
- **set_clause_true_var_neg :** La clause c devient est vraie car au moins une de ses variables devant être fausse à pour valeur *False*
  - **Paramètres :** (`?c clause ?v - variable`) :
    - On a la clause c avec la variable v que l'on souhaite passer à *True*.
  - **Préconditions :** (`and (varFalseIn ?v ?c) (varIsFalse ?v)`) :
    - La variable v est à *False* dans la clause c, et la variable v est à *False*.
  - **Effets :** `(and (clauseTrue ?c))` :
    - La clause c est instanciée à *True*.

5.2.3 Prédicats :

- **varIsTrue :** (`varIsTrue ?v - variable`) : La variable v a la valeur *True*.
- **varIsFalse :** (`varIsFalse ?v - variable`) :La variable v a la valeur *False*.
- **varTrueIn :** (`varTrueIn ?v - variable ?c - clause`) : La variable v doit être à *True* dans la clause c.
- **varFalseIn :** (`varFalseIn ?v - variable ?c - clause`) : La variable v doit être à *False* dans la clause c.
- **clauseTrue :** (`clauseTrue ?c - clause`) : La clause c est évaluée à *True*.
- **hasNoValue :** (`hasNoValue ?v - variable`) : La variable v n'a encore de valeur.

5.3 Problème :

5.3.1 Objets :

- Ensemble de variables booléennes : x_1 x_2 x_3 x_4.
- Ensemble des clauses composées de variables.

5.3.2 État initial :

- Chaque variable a un prédicat pour lui indiquer qu'il n'a pas de valeur booléenne attribuée. On a donc les clauses (`hasNoValue` x_1)(`hasNoValue` x_2)(`hasNoValue` x_3)(`hasNoValue` x_4).
- Il faut indiquer pour chaque littéral de chaque clause s'il y a la négation "not" présente ou non. Par exemple, dans la clause C1 de formule $(x_1 \wedge x_3 \wedge \overline{x}_4)$, on rajoute les prédicats (varTrueIn x1 c1) (varTrueIn x3 c1) (varFalseIn x4 c1).

5.3.3 Etat final :

- Toutes les clauses doivent être vraies, on rajoute un prédicat par clause. C'est-à-dire `(clauseTrue c1) (clauseTrue c2) (clauseTrue c3)`.

ALVES Romain *BOSSY Romain*
COMBE Gaétan *FAURIE Alban*

6 Poursuite-Évasion

6.1 Description :

Le problème de la "Poursuite-Évasion" est un jeu joué sur un graphe dans lequel un intrus et des poursuivants se déplacent. L'objectif des poursuivants est de capturer l'intrus. Cela est réalisé lorsqu'un poursuivant et l'intrus se trouvent sur le même nœud.
L'intrus se déplace à une vitesse infinie tandis que les poursuivants explorent chaque nœud du graphe. L'intrus peut donc se placer sur des nœuds déjà explorés par ses poursuivants. Une stratégie gagnante pour les poursuivants consiste à trouver un plan de déplacement tel que, quel que soit le mouvement de l'intrus, il finira par être capturé. Il est facile de voir que toutes les configurations de jeu (graphe et nombre de poursuivants) n'ont pas une stratégie gagnante. On suppose que les deux agents sont sur le même noeud au début.

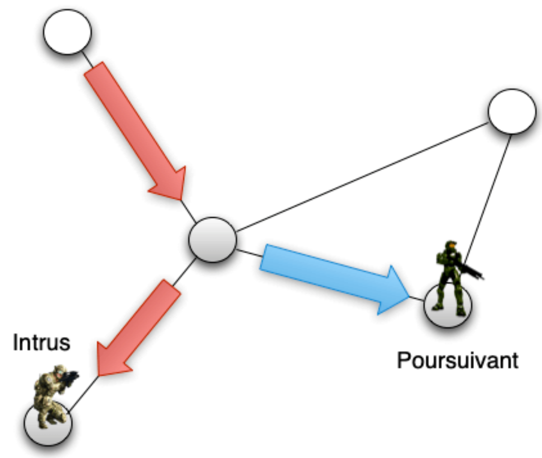


FIGURE 4 – Exemple de Poursuite-Evasion

Dans notre cas, nous avons mis en place des piles associées à chaque nœud, afin de savoir quelles sont les arêtes reliées à ce dernier. Si une arête est décontaminée, elle est enlevée de la pile des deux nœuds qu'elle relie. Les piles sont composées à l'origine de toutes les arêtes adjacentes au nœud dont dépend la pile, ainsi que deux arêtes factices *bottom*, placée au fond de la pile, et *top*, placée au sommet de la pile.

6.2 Domaine :

6.2.1 Objets :

- **Nœud :** Nœuds du graphe.
- **Agent :** Agents présents sur le graphe.
- **Arete :** Arêtes du graphe reliant les nœuds.

6.2.2 Actions :

- **move_reste1noeud :**
  - **Paramètres :** `(?ag1 - agent ?n1 - noeud ?n2 - noeud ?arlink - arete ?ardessuspile1 - arete ?ardessuspile2 - arete ?ardessouspile1 -arete ?ardessouspile2 - arete)` :
    - On veut déplacer l'agent *ag1* du nœud *n1* vers le nœud *n2*, reliés par l'arête *arlink*. On prend également en paramètre les arêtes qui sont au-dessus et en dessous de l'arête *arlink* dans les piles des nœuds *n1* et *n2*.
  - **Préconditions :** `(and (agentSur ?ag1 ?n1) (link ?n1 ?n2 ?arlink) (link ?n2 ?n1 ?arlink) (on ?arlink ?ardessouspile1 ?n1) (on ?ardessuspile1 ?arlink ?n1) (estbottom ?ardessouspile1) (esttop ?ardessuspile1) (on ?arlink ?ardessouspile2 ?n2) (on ?ardessuspile2 ?arlink ?n2)` :
    - L'agent *ag1* est sur le nœud *n1*, qui est relié au nœud *n2* via l'arête *arlink*. L'arête *arlink* est présente dans les piles des nœuds *n1* et *n2*.
  - **Effets :** `(and (agentSur ?ag1 ?n2) (not (agentSur ?ag1 ?n1)) (estAreteMarquee ?arlink) (not (on ?arlink ?ardessouspile1 ?n1))`

ALVES Romain *BOSSY Romain*
COMBE Gaétan *FAURIE Alban*

`(not (on ?ardessuspile1 ?arlink ?n1)) (not (on ?arlink ?ardessouspile2 ?n2))`
`(not (on ?ardessuspile2 ?arlink ?n2)) (on ?ardessuspile1 ?ardessouspile1 ?n1)`
`(on ?ardessuspile2 ?ardessouspile2 ?n2))` :

  - L'agent est désormais sur le noeud *n2* et non plus sur le noeud *n1*. L'arête traversée *arlink* est donc marquée, et est retirée des piles des noeuds *n1* et *n2*.

- **move_2_agent_meme_noeuds_au_debut :**
  - **Paramètres :** `(?ag1 - agent ?ag2 - agent ?n1 - noeud ?n2 - noeud ?arlink - arete ?ardessuspile1 - arete ?ardessuspile2 - arete ?ardessouspile1 -arete ?ardessouspile2 - arete)` :
    - Les deux agents *ag1* et *ag2* sont sur le même noeud *n1*. Le but est de déplacer l'agent *ag1* vers le noeud *n2*. L'arête *arlink* qui relie les noeuds *n1* et *n2* est présente dans les piles des deux noeuds.
  - **Préconditions :** `(and (agentSur ?ag1 ?n1) (agentSur ?ag2 ?n1) (agentDiff ?ag1 ?ag2) (link ?n1 ?n2 ?arlink) (link ?n2 ?n1 ?arlink)(on ?ardessuspile1 ?arlink ?n1) (on ?arlink ?ardessouspile1 ?n1) (on ?ardessuspile2 ?arlink ?n2) (on ?arlink ?ardessouspile2 ?n2))` :
    - Les agents *ag1* et *ag2* sont sur le noeud *n1*, lié au noeud *n2* via l'arête *arlink*, présente quant à elle dans les piles des deux noeuds.
  - **Effets :** `(and (agentSur ?ag1 ?n2) (not (agentSur ?ag1 ?n1)) (agentSur ?ag2 ?n1) (estAreteMarquee ?arlink) (not (on ?arlink ?ardessouspile1 ?n1)) (not (on ?ardessuspile1 ?arlink ?n1)) (not (on ?arlink ?ardessouspile2 ?n2)) (not (on ?ardessuspile2 ?arlink ?n2)) (on ?ardessuspile1 ?ardessouspile1 ?n1) (on ?ardessuspile2 ?ardessouspile2 ?n2)))` :
    - L'agent *ag1* est désormais sur le noeud *n2* et non plus sur le noeud *n1*, tandis que l'agent *n2* reste sur le noeud *n1*. L'arête traversée *arlink* est donc marquée, et est retirée des piles des noeuds *n1* et *n2*.
- **rejoindre_quand_marque :**
  - **Paramètres :** `(?ag1 - agent ?n1 - noeud ?n2 - noeud ?arlink - arete ?bottom - arete ?top - arete)` :
    - L'agent *n1* est sur le noeud *n1* et cherche à se déplacer vers le noeud *n2*, reliés par l'arête *arlink*. Cette action ne marque pas d'arête et ne modifie pas les piles des noeuds.
  - **Préconditions :** `(and (agentSur ?ag1 ?n1) (link ?n1 ?n2 ?arlink) (link ?n2 ?n1 ?arlink) (on ?top ?bottom ?n1)(estbottom ?bottom) (esttop ?top)` :
    - L'agent *ag1* est sur le noeud *n1*, relié au noeud *n2* par l'arête *arlink*. On vérifie si l'arête est marquée en regardant si elle est contenue dans la pile du noeud *n1*.
  - **Effets :** `(and (agentSur ?ag1 ?n2) (not (agentSur ?ag1 ?n1)))` :
    - L'agent *ag1* est désormais sur le noeud *n2* et non plus sur le noeud *n1*.

6.2.3 Prédicats :

- **link :** `(link ?n1 - noeud ?n2 - noeud ?ar - arete)` : Vrai si l'arête *arlink* relie les noeuds *n1*e t *n2*.
- **agentSur :** `(agentSur ?ag - agent ?n - noeud)` : Vrai si l'agent *ag* est sur le noeud *noeud*.
- **estAreteMarquee :** `(estAreteMarquee ?ar - arete)` : Vrai si l'arête *ar* est marquée.
- **estbottom :** `(estbottom ?ar - arete)` : Vrai lorsque l'arête factice *bottom* est au fond de la pile.
- **esttop :** `(esttop ?ar - arete)` : Vrai lorsque l'arête factice *top* est au sommet de la pile.
- **on :** `(on ?ar1 - arete ?ar2 - arete ?p - noeud)` : Vrai si l'arête *ar1* est sur l'arête *ar2* dans la pile du noeud *p*.
- **agentDiff :** `(agentDiff ?ag1 - agent ?ag2 - agent)` : Vrai si les agents *ag1* et *ag2* sont différents.

6.3 Problème :

6.3.1 Objet :

- Ensemble de noeuds : `a b c d`.
- Ensemble d'arêtes : `ar1 ar2 ar3 ar4`, auxquelles on rajoute toujours des arêtes non présentes dans le graphe : `top` et `bottom`. Ces arêtes permettent de gérer la pile.
- Ensemble d'agents : `ag1 ag2`.

ALVES Romain *BOSSY Romain*
COMBE Gaétan *FAURIE Alban*

6.3.2 État initial :

- Il faut rajouter tous les arcs liants deux noeuds : `(link i j arete) (link j i arete)`, afin de respecter la symétrie.
- Il faut indiquer où se trouvent les agents au départ. Dans nos problèmes, les agents doivent absolument commencer sur le même noeud de départ. Par exemple, nos deux agents vont commencer sur le noeud a, on rajoute alors les prédicats `(agentSur ag1 a) (agentSur ag2 a)`.
- L'arête `top` doit être considérée comme `top` et l'arête `bottom` doit être considérée comme le `bottom`. On rajoute dans tous nos problèmes les deux prédicats `(esttop top) (estbottom bottom)`. Ils permettent de préciser que ce sont les arêtes présentes au sommet et au fond de chaque pile afin de gérer le cas où les piles sont vides.
- On a une pile par noeud indiquant les arêtes restantes à visiter. Par exemple, la pile du noeud est liée à 2 arêtes `ar1 ar3` non visités. On a donc les prédicats `(on top ar1 a) (on ar1 ar3 a) (ar3 bottom a)`.

6.3.3 État final :

- L'ensemble des arêtes doivent être marquées, on a donc les prédicats `(estAreteMarquee ar1) (estAreteMarquee a2) (estAreteMarquee ar3) (estAreteMarquee ar4)`.

7 Sokoban

7.1 Description :

Le Sokoban est un jeu où gardien d'entrepôt (divisé en cases carrées), le joueur doit disposer des caisses sur des cases cibles. Il peut se déplacer dans les quatre directions et pousser (mais pas tirer) une caisse à la fois. Une fois que toutes les caisses ont été rangées (parfois un vrai casse-tête), le niveau est terminé et le joueur passe au niveau suivant, qui est généralement plus difficile. L'idéal est de réussir avec le moins de mouvements possibles (mouvements et poussées). Il a été démontré que résoudre les niveaux de Sokoban est un problème NP-difficile, dans le sens où le jeu appartient à la classe plus générale des problèmes de planification de déplacement, où le joueur est autorisé à pousser ou tirer un ou plusieurs objets à la fois. Le jeu intéresse également les chercheurs en intelligence artificielle, car résoudre les niveaux pose des problèmes difficiles, pour lesquels il n'existe actuellement aucun algorithme de résolution rapide.

7.2 Domaine :

7.2.1 Objets :

- **Caisse :** Caisse à bouger par l'agent.
- **But :** But sur lequel une caisse doit se retrouver en position finale.
- **Agent :** Position de l'agent.
- **Case :** Une case du jeu, où peuvent se trouver les différents objets tels que l'agent, une caisse ou un but.
- **Direction :** Direction dans laquelle une caisse va pouvoir être poussée.

7.2.2 Actions :

- **deplacer_agent :**
  - **Paramètres :** `(?d - direction ?a - agent ?c1 ?c2 - case)` :
    - On veut déplacer l'agent *a* qui est sur la case *c1* vers la case *c2* qui est dans la direction *d* par rapport à l'agent.
  - **Préconditions :** `(and (agentSur ?a ?c1) (empty ?c2) (adjacente ?c2 ?c1 ?d))` :
    - L'agent est sur la case *c1*, qui est adjacente à la case *c2*, qui est vide, dans la direction *d*.
  - **Effets :** `(and (not (agentSur ?a ?c1)) (empty ?c1) (agentSur ?a ?c2) (not (empty ?c2)` :
    - L'agent *a* n'est plus sur la case *c1*, qui devient vide par conséquent, et est désormais sur la case *c2*, qui n'est désormais plus vide.
- **deplacer_caisse :**
  - **Paramètres :** `(?d - direction ?a - agent ?c1 ?c2 ?c3 - case ?caisse - caisse)` :
    - On veut déplacer la caisse *caisse* dans la direction *d*. On doit donc vérifier que les cases dans cette direction sont libres.

ALVES Romain *BOSSY Romain*
COMBE Gaétan *FAURIE Alban*

  - **Préconditions :** `((and (agentSur ?a ?c1) (caisseSur ?caisse ?c2) (caisseLibre ?caisse) (empty ?c3) (adjacente ?c2 ?c1 ?d) (adjacente ?c3 ?c2 ?d)))` :
    - L'agent *a* est sur la case *c1*, la caisse *caisse* est sur la case *c2*, qui ne possède pas de but, et qui est entourée par les cases *c1* et *c3* dans la direction *d*.
  - **Effets :** `(and (not (agentSur ?a ?c1)) (empty ?c1) (agentSur ?a ?c2) (not (empty ?c2)) (not (caisseSur ?caisse ?c2)) (caisseSur ?caisse ?c3) (not (empty ?c3))` :
    - L'agent *a* a été déplacé de la case *c1*, qui est devenue vide, vers la case *c2*, qui n'est plus vide. La caisse a également été déplacée de la case *c2* vers la case *c3*, qui n'est désormais plus vide.
- **deplacer_caisse_sur_but :**
  - **Paramètres :** `(?d - direction ?a - agent ?c1 ?c2 ?c3 - case ?caisse - caisse ?b - but)` :
    - On veut déplacer la caisse *caisse* sur la case qui possède le but *b*.
  - **Préconditions :** `((and (agentSur ?a ?c1) (caisseSur ?caisse ?c2) (caisseLibre ?caisse) (empty ?c3) (butSurCase ?b ?c3) (butVide ?b) (adjacente ?c2 ?c1 ?d) (adjacente ?c3 ?c2 ?d)))` :
    - L'agent *a* est sur la case *c1*. La caisse *caisse* est sur la case *c2*. Les cases *c1*, *c2* et *c3* sont adjacentes dans la direction *d*, et cette dernière possède le but *b* qui est actuellement vide.
  - **Effets :** `(and (not (agentSur ?a ?c1)) (empty ?c1) (agentSur ?a ?c2)(not (empty ?c2)) (not (caisseSur ?caisse ?c2)) (caisseSur ?caisse ?c3) (not (empty ?c3)) (not (butVide ?b)) (caisseSurBut ?caisse) (not (caisseLibre ?caisse))` :
    - L'agent *a* n'est désormais plus sur la case *c1*, qui devient vide, mais sur la case *c2*. La caisse *caisse* n'est plus sur la case *c2* mais désormais sur la case *c3*, qui n'est donc plus vide. La caisse devient donc une caisse sur le but *b* et n'est donc plus libre.
- **deplacer_caisse_hors_but :**
  - **Paramètres :** `(?d - direction ?a - agent ?c1 ?c2 ?c3 - case ?caisse - caisse ?b - but)` :
    - On veut déplacer la caisse *caisse* vers la case qui ne possède pas le but *b*.
  - **Préconditions :** `((and (agentSur ?a ?c1) (caisseSur ?caisse ?c2) (caisseSurBut ?caisse) (empty ?c3) (butSurCase ?b ?c2) (adjacente ?c2 ?c1 ?d) (adjacente ?c3 ?c2 ?d)))` :
    - L'agent *a* est sur la case *c1*. La caisse *caisse* est sur la case *c2*. Les cases *c1*, *c2* et *c3* sont adjacentes dans la direction *d*. La case *c2* possède le but *b*, ce qui n'est pas le cas de la case *c3*.
  - **Effets :** `(and (not (agentSur ?a ?c1)) (empty ?c1) (agentSur ?a ?c2)(not (empty ?c2)) (not (caisseSur ?caisse ?c2)) (butVide ?b) (caisseSur ?caisse ?c3) (not (empty ?c3)) (not (caisseSurBut ?caisse)) (caisseLibre ?caisse)` :
    - L'agent *a* n'est plus sur la case *c1*, qui est désormais vide, mais sur la case *c2*. La caisse n'est plus sur la case *c2* mais désormais sur la case *c3*, qui n'est désormais plus vide. La caisse *caisse* n'est plus sur le but et est libre.
- **deplacer_caisse_de_but_sur_but :**
  - **Paramètres :** `(?d - direction ?a - agent ?c1 ?c2 ?c3 - case ?caisse - caisse ?b1 ?b2 - but)` :
    - On veut déplacer la caisse *caisse* depuis la case *c2* qui contient le but *b1* vers la case *c3* qui possède le but *b2*, en la poussant avec l'agent *a* dans la direction *d*.
  - **Préconditions :** `((and (agentSur ?a ?c1) (caisseSur ?caisse ?c2) (empty ?c3) (butSurCase ?b1 ?c2) (butSurCase ?b2 ?c3) (butVide ?b2) (caisseSurBut ?caisse) (adjacente ?c2 ?c1 ?d) (adjacente ?c3 ?c2 ?d)))` :
    - L'agent *a* est sur la case *c1*. La caisse *caisse* est sur la case *c2*. La case *c3* est actuellement vide. Les cases *c2* et *c3* possèdent respectivement les buts *b1* et *b2*. Le but *b2* doit être vide, tandis que la caisse *caisse* doit être sur le but *b1*. Les cases *c1*, *c2* et *c3* sont adjacentes dans la direction *d*.
  - **Effets :** `(and (not (agentSur ?a ?c1)) (empty ?c1) (agentSur ?a ?c2)(not (empty ?c2)) (not (caisseSur ?caisse ?c2)) (butVide ?b1) (caisseSur ?caisse ?c3) (not (empty ?c3)) (not (butVide ?b2)) (caisseSurBut ?caisse)` :
    - L'agent *a* n'est plus sur la case *c1*, qui est désormais vide, mais sur la case *c2*. La caisse n'est plus sur la case *c2* mais désormais sur la case *c3*, qui n'est désormais plus vide. Le but *b1* est désormais compté comme vide car il n'a plus de caisse sur lui, au contraire du but *b2* qui n'est quant à lui plus vide, car la caisse *caisse* est désormais sur lui.

7.2.3 Prédicats :

- **empty :** (`empty ?c - case`) : La case *c* n'est pas un mur, et n'a pas d'agent ou de caisse sur elle.
- **caisseSur :** (`caisseSur ?c - caisse ?case - case`) : La caisse *c* est sur la case *case*.
- **agentSur :** (`agentSur ?a - agent ?case - case`) : L'agent *a* est sur la case *case*.
- **butSurCase :** (`butSurCase ?b - but ?case - case`) : Le but *b* est sur la case *case*.
- **butVide :** (`butVide ?b - but`) : Le but *b* n'a pas de caisse sur lui.
- **adjacente :** (`adjacente ?c1 ?c2 - case ?d - direction`) : La case *c1* est adjacente à la case *c2* dans la direction *d*.
- **caisseLibre :** (`caisseLibre ?c - caisse`) : La caisse *c* n'est pas sur un but.
- **caisseSurBut :** (`caisseSurBut ?c - caisse`) : La caisse *c* ets sur un but.

8 SAT Planner

8.1 Description :

Un problème SAT est un problème consistant à déterminer s'il existe une assignation de valeurs de vérité (vrai ou faux) aux variables d'une formule booléenne qui rend cette formule vraie, c'est-à-dire satisfiable. En d'autres termes, le problème SAT cherche à savoir s'il existe une combinaison d'assignations de vérité qui satisfait toutes les clauses de la formule.

Un exemple de formule booléenne est une conjonction de disjonctions de littéraux, où un littéral est une variable propositionnelle ou sa négation.
Une clause est une disjonction de variables propositionnelles : $(x_1 \vee x_3 \vee \overline{x}_4)$.
Une formule sous Forme Normale Conjonctive (CNF) est sous la forme : $(x_1 \vee x_3 \vee \overline{x}_4) \wedge (x_4) \wedge (x_2 \vee \overline{x}_3)$.
Un SAT Problème consiste donc à évaluer des variables à Vrai ou Faux.

Le but ici est donc de transformer une action en une conjonction de clauses :
$\neg MOVE(R, l1, l2, 0) \vee \neg MOVE(R, l2, l1, 0)$ devient $\neg a \vee \neg b$,
$\neg at(R, l1, 0) \wedge at(R, l1, 1) \longrightarrow MOVE(R, l2, l1, 0)$ devient $\neg c \wedge d \longrightarrow b$,
etc...

8.2 Domaine :

8.2.1 Constructeur :

- **SATEncoding(Problem problem, int steps)** : Traduit le problème instancié par le *parser* PPDL en problème SAT en version CNF.
  - Paramètres :
    - *Problem problem* : Problème à résoudre.
    - *int steps* : Nombre d'étapes pour résoudre le problème.

8.2.2 Méthodes :

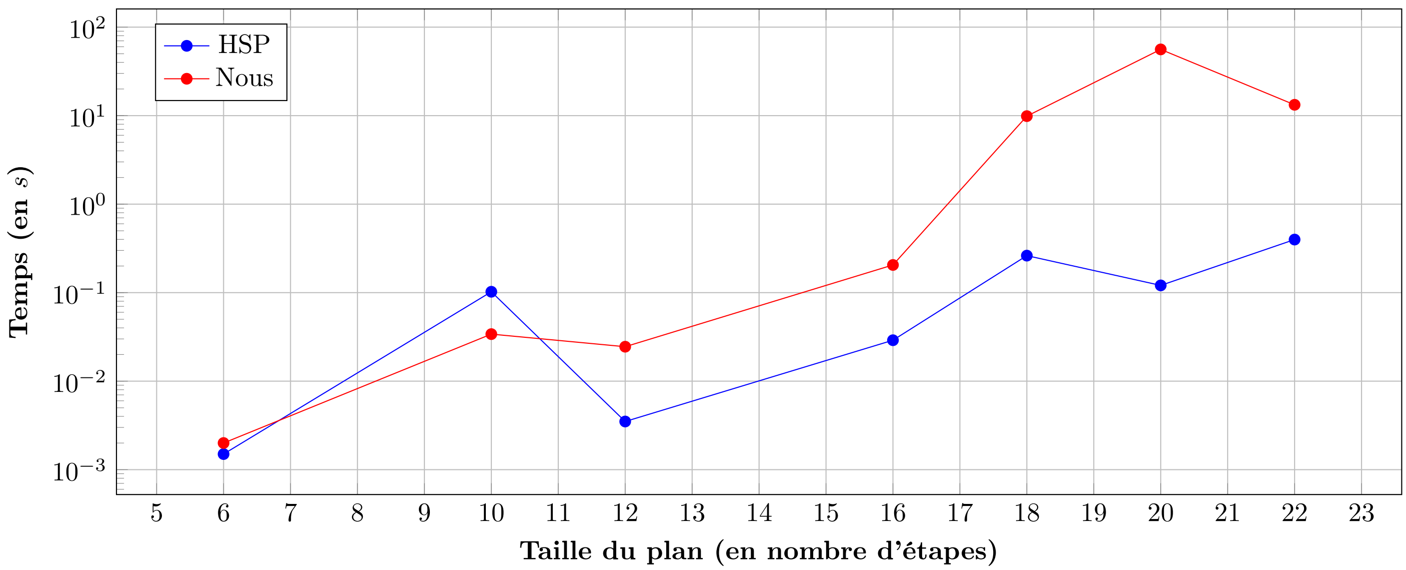
- **void convertInitGoal(boolean goal) :** Convertit l'état initial et l'état objectif en SAT.
  - Paramètre : *boolean goal* : Vrai si on convertit l'état objectif, Faux si on convertit l'état initial.
- **void convertAction(Action action, int step) :** Convertit l'action passée en paramètre.
  - Paramètres :
    - *Action action* : Action à convertir.
    - *int step* : Numéro de l'étape à résoudre.
- **void convertActionDisjunction(int step) :** Convertit une action en une disjonction CNF.
  - Paramètre : *int step* : Numéro de l'étape à convertir.
- **void convertStateTransitions(int step) :** Convertit toutes les transitions d'état pour e numéro d'étape passé en paramètre.
  - Paramètre : *int step* : Numéro de l'étape à convertir.
- **void addClause(int[] clause) :** Ajoute une clause au tableau passé en paramètre.

  - Paramètre : *int[] clause* : Tableau auquel on ajoute une clause.
- **void solve() :** Ajoute une clause au tableau passé en paramètre.
- **String getPlan() :** Récupère le plan sous forme textuelle.
  - Valeur de retour : Plan sous forme de chaîne de caractère.
- **String actionToString(Action a, int step) :** Renvoie l'action sous forme textuelle.
  - Paramètres :
    - *Action a* : Action qui sera renvoyée sous forme textuelle.
    - *int step* : Numéro de l'étape à résoudre.
  - Valeur de retour : Action sous forme de chaîne de caractère.

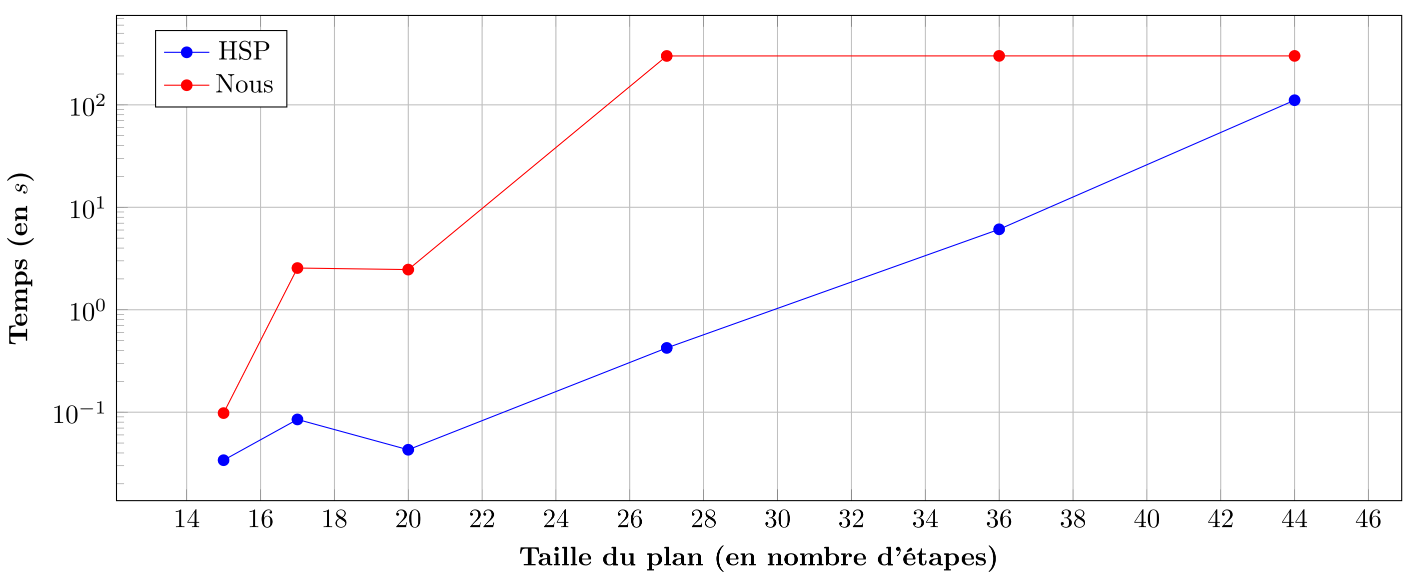
9 Benchmark

Tous les graphes ci-dessous comparent le temps **moyen** (en s) de traduction et résolution pour chaque taille de plan (en nombre d'étapes).

9.1 Blocksworld :

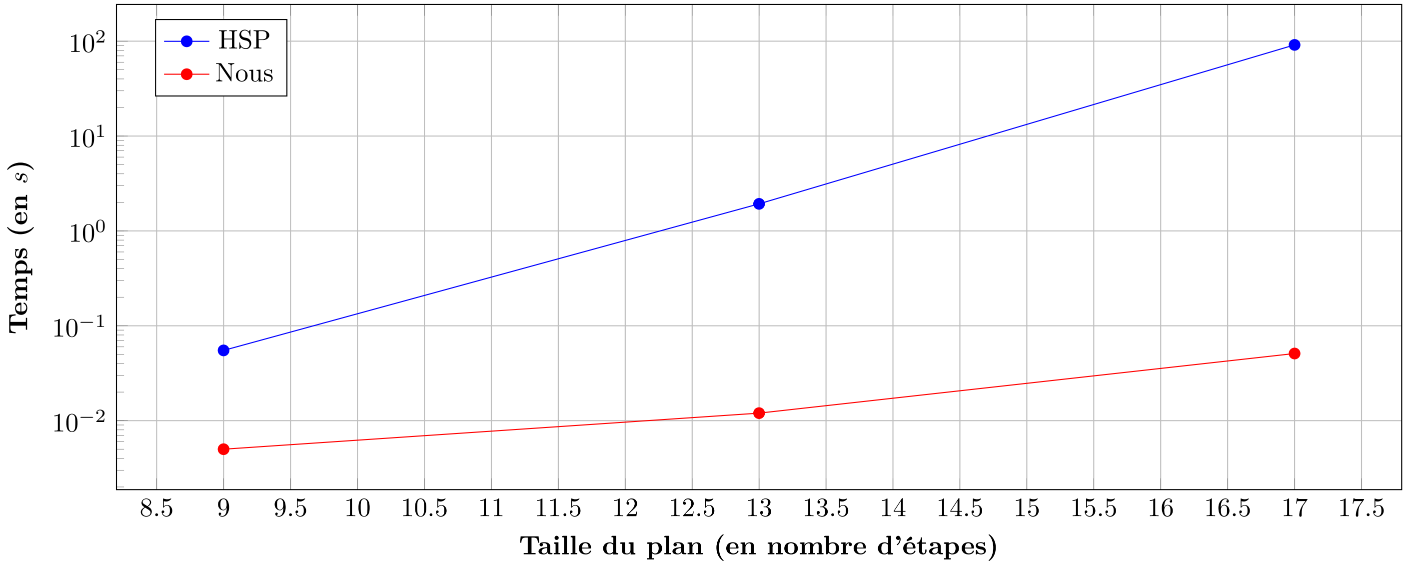


9.2 Logistics :



ALVES Romain *BOSSY Romain*
COMBE Gaétan *FAURIE Alban*

9.3 Gripper :



9.4 Dépôts :

On a systématiquement un dépassement mémoire peu importe la quantité mémoire allouée.

10 Conclusion

Nous avons quelques incohérences au niveau des benchmarks, notamment la stagnation à 300s dès que l'on dépasse 26 étapes dans la catégorie logistique. De notre point de vue, la cause de cette incohérence est dû à une mauvaise utilisation de la bibliothèque sat4j.

Dans la même catégorie, le fait d'être plus rapide que HSP pour Gripper. Notre hypothèse pour expliquer cela est que chaque version SAT d'un problème Gripper a un ratio variables/clauses assez éloigné de 1, ce qui fait que le solveur SAT peut plus rapidement trouver une solution que HSP.

De plus, le fait d'avoir un dépassement systématique de la mémoire pour le problème dépôt, malgré l'allocation d'une grande quantité de mémoire. Peut-être dû au nombre exponentiel de clauses et de variables pour ce type de problème.

Vous pouvez voir sur l'histogramme présent au début que nous avons eu un pic de difficulté sur poursuite évasion qui nous a fait perdre beaucoup de temps. Cela est dû à la difficulté de retranscription basée sur les arêtes.

Nous sommes heureux d'avoir complété tous les challenges, ce ne fut pas une mince affaire. Les problèmes étaient intéressants et diversifiés.

11 Feedback

Indiquer les hypothèses utilisées pour résoudre les problèmes, notamment pour le poursuite-évasion ou les agents doivent commencer sur le nœud. Indiquer aussi qu'il faut décontaminer pas seulement les nœuds, mais aussi les arêtes.

Aider un peu plus les étudiants sur la partie SAT Planner, qui peut être complexe à saisir. Aider également sur l'installation et l'utilisation des librairies nécessaires car cela peut faire perdre énormément de temps.

ALVES Romain *BOSSY Romain*
COMBE Gaétan *FAURIE Alban*