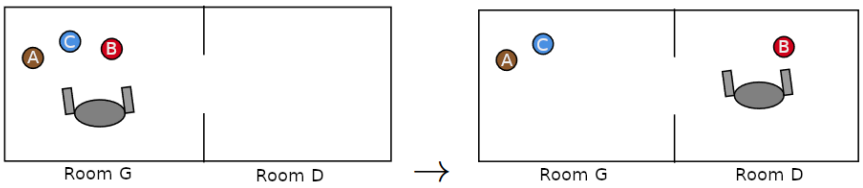
Planification et langage PDDL

Ces travaux pratiques sont organisés ainsi : la Partie 1 introduit les bases de la planification avec le langage PDDL au travers du problème Gripper. La Partie 2 se base sur les concepts introduits dans la Partie 1. Enfin, la Partie 3 montre comment on peut prendre en compte des valeurs numériques pour optimiser le coût d'un plan-solution en langage PDDL. Vous pourrez tester vos solutions en résolvant les problèmes de planification posés grâce au planificateur en ligne sur http ://editor.planning.domains.

1 Le problème Gripper

Le problème Gripper est un problème très simple permettant d'introduire le format du langage PDDL. Dans cette version du domaine, on considère un robot, Robby, qui dispose de deux bras avec pince lui permettant d'attraper et de poser des balles. Robby peut également se déplacer entre deux pièces, la pièce Droite et la pièce Gauche.



La formalisation dans le langage STRIPS est lourde. Par exemple, dans le cas où l'on a trois balles et deux pièces, on retrouve $3 \cdot 2 = 6$ propositions (encore appelées fluents) de la forme at_ball_X_R, et autant d'opérateurs instanciés de la forme pick_X_R ou drop_X_R. Pourtant, les balles et les pièces ont des comportements identiques les unes et les autres. Comparativement à un encodage STRIPS, le langage PDDL permet de conserver une certaine concision dans l'encodage qui permet de travailler avec des problèmes sensiblement plus complexes sans pour autant avoir à écrire un grand nombre de lignes pour décrire le domaine considéré.

Le but de cette séance est de prendre en main le langage PDDL tel qu'il a été présenté en cours. Ce dernier permet de factoriser le formalisme STRIPS et d'obtenir une représentation plus concise du même problème.

1.1 Domaine de planification

Il existe plusieurs instances de problèmes dans le domaine Gripper : à deux balles, trois balles, deux pièces, trois pièces, etc. Dans un premier temps, on cherche à capturer le dénominateur commun à toutes les instances Gripper, et à le formaliser dans un fichier domain.pddl. Un tel fichier s'organise comme le montre le bloc suivant.

```
(define (domain gripper)
    (:requirements :typing)

; Corps

)
```

Le corps du domaine de planification est ensuite composé, dans cet ordre, des éléments suivants :

Types : Chaque instance de planification est constituée de différents objets, qui ont des rôles déterminés par leurs types. Ici, on a le type ball et le type room. En PDDL, cela s'écrit comme suit :

```
(:types ball room)
```

Prédicats : À partir des différents objets de l'instance, on peut obtenir des fluents. Ces fluents sont des propositions (logique propositionnelle) construites en partant de prédicats, que l'on clôt en substituant les variables par des objets concrets. Par exemple, à partir du prédicat at_ball(?b − ball, ?r − room), on peut obtenir le fluent at_ball(ballA, roomL), en substituant ?b par un objet de type ball et ?r par un objet de type room. Notez que certains prédicats peuvent avoir une arité nulle : c'est le cas par exemple du prédicat hand_free()

```
(:predicates
    (at-ball ?b - ball ?r - room)
    (at-robby ?r - room)
    (carry ?b - ball)
    (hand-free )
)
```

Actions : Comme pour les prédicats, on définit des modèles d'actions et l'on obtient des actions instanciées en substituant les différents paramètres par les objets adéquats. Le code ci-dessous décrit les modèles d'actions permettant à Robby de se déplacer, qui prennent en paramètres deux variables (?start et ?dest) de type room.

```
(:action move
    :parameters (?start ?dest - room)
    :precondition (and (at-robby ?start))
    :effect (and (at-robby ?end)
                 (not (at-robby ?start)))
)
```

Questions

1. Complétez le fichier domain.pddl avec l'action pick.
2. Complétez le fichier domain.pddl avec l'action drop.

1.2 Instance

Le fichier domain.pddl ne modélise que le domaine général : Dans notre cas, nous avons simplement décrit le fait qu'un robot pouvait se déplacer d'une pièce à l'autre et manipuler des balles. Dans cette sous-partie, nous allons maintenant compléter le fichier instance.pddl qui permet de décrire les détails spécifiques au problème posé dans le domaine : état initial avec le nombre exact de balles et leur emplacement, les pièces, l'emplacement et l'état de Robby, le but du problème...

```
(define  (problem gripper3)
    (:domain gripper)

; Corps

)
```

Objets : Il s'agit ici de définir les différents objets que l'on trouve dans le problème de planification décrit. Chaque objet correspond à un type, ainsi :

```
(:objects
    ballA ballB ballC - ball
    roomL roomR - room
)
```

État initial : L'état initial est un état complètement spécifié. Il s'agit ici d'énumérer les fluents vérifiés initialement. Ceux qui ne sont pas présents ne sont pas vrais dans l'état initial. Dans ce problème Gripper3, on aurait donc :

```
(:init
    (at-ball ballA roomL)
    (at-ball ballB roomL)
    (at-ball ballC roomL)
    (at-robby roomL)
    (hand-free )
)
```

But : Contrairement à l'état initial, le but est généralement un état partiel représenté par une formule logique que l'état final doit vérifier. Ici, on se restreint aux buts conjonctifs où l'on exige qu'un ensemble de fluents soient finalement vrais. Si l'on cherche à avoir toutes les balles dans l'autre pièce, on écrira alors :

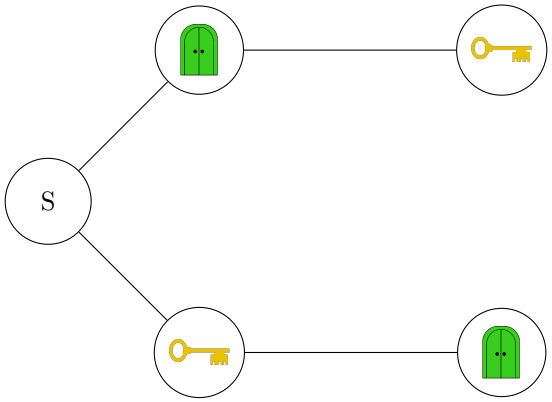
```
(:goal
    (and
    (at-ball ballA roomR)
    (at-ball ballB roomR)
    (at-ball ballC roomR)
    )
)
```

Questions

1. Complétez les fichiers fournis, et testez votre écriture en résolvant le problème de planification grâce au planificateur en ligne sur http ://editor.planning.domains.

2 Explorer

Dans ce problème, un robot est chargé de récupérer des clés afin d'ouvrir les coffres posés dans certaines pièces d'un labyrinthe. Pour ce faire, il peut se déplacer d'une pièce à l'autre, et il peut tenir une clé. Un coffre ne peut être ouvert que si le robot tient une clé, et toutes les clés peuvent ouvrir tous les coffres. L'image suivante présente l'exemple que l'on va chercher à modéliser et à résoudre.



Par la suite, on modélise d'abord le domaine, puis l'instance (le problème spécifique) auquel on s'intéresse.

2.1 Modélisation du domaine

On modélise dans un premier temps la topologie du labyrinthe, ainsi que les actions permettant au robot de se déplacer. Le labyrinthe est naturellement vu comme un graphe. Ainsi, dans un fichier domain.pddl :

Questions
Dans un fichier domain.pddl :

1. Définissez les trois types nécessaires dans le problème.
2. Définissez un prédicat at−robot permettant de localiser le robot. Quelle est son arité ?
3. Définissez un prédicat connected permettant de représenter les arêtes du graphe.
4. Ajoutez au domaine une action move permettant au robot de se déplacer d'une pièce à l'autre, pour peu qu'elles soient connectées.

On modélise maintenant les autres éléments du domaine, à savoir les clés, les coffres, ainsi que les actions associées. On utilisera deux prédicats d'arité 0 : has−key et empty−hand, qui modélisent respectivement le fait que le robot a une clé dans la main, et qu'il a la mains vide.

Questions

1. Définissez des prédicats at−key et at−door symbolisant la présence d'une clé ou d'un coffre dans une pièce.
2. Définissez les prédicats has−key et empty−hand, ainsi qu'un prédicat open symbolisant le fait qu'un coffre est ouvert.
3. Ajouter une action pick−key permettant au robot de ramasser une clé dans la pièce où il se trouve.
4. Ajouter une action open−door permettant au robot d'ouvrir un coffre, pour peu que les conditions le lui permettent.

2.2 Modélisation de l'instance

Il s'agit maintenant de modéliser le problème présenté sur le schéma.

Questions
Dans un fichier instance.pddl :

1. Définissez les différents objets intervenant dans l'instance
2. Définissez l'état initial de l'instance :
   (a) Encodez le graphe dans l'état initial à l'aide des prédicats connected. *Remarque : (connected roomA roomB) et (connected roomB roomA) sont deux fluents indépendants.*
   (b) Ajoutez les différents objets, ainsi que l'état initial du robot.
3. Définissez le but, qui est d'ouvrir les deux coffres.
4. Complétez les fichiers fournis, et testez votre écriture en résolvant le problème de planification grâce au planificateur en ligne sur http ://editor.planning.domains.

3 Distance de rotation

La planification peut aussi permettre de calculer des distances entre deux entités lorsqu'on travaille avec des notions de distances à base d'actions.

Par exemple, au sein du génome humain, on peut définir une distance entre deux déclinaisons d'un même gêne en comptant le nombre *minimum* d'opérations de type addition, destruction, substitution... de bases nucléotidiques permettant d'obtenir un gène à partir d'un autre.

Dans cette partie, on s'inspire de cette problématique pour s'intéresser à la distance entre des anagrammes de taille n, que l'on appelle *distance de rotation*. Pour deux mots $v = v_1v_2 \cdots v_n$ et $w = w_1w_2 \cdots w_n$, on note $d(v, w)$ le nombre minimal d'opérations permettant de transformer v en w. On considère les deux opérations suivantes :

— **Petite rotation :** Pour $i \in \{1, \ldots, n-1\}$, $w_1 \cdots w_i w_{i+1} \cdots w_n \longrightarrow w_1 \cdots w_{i+1} w_i \cdots w_n$. Il s'agit essentiellement d'échanger les positions de deux lettres contiguës.
Exemple : une rotation de abba à l'indice 1 donne baba.

— **Grande rotation :** Pour $i \in \{2, \ldots, n-1\}$, $w_1 \cdots w_{i-1} w_i w_{i+1} \cdots w_n \longrightarrow w_1 \cdots w_{i+1} w_i w_{i-1} \cdots w_n$. Il s'agit de faire une rotation autour d'un pivot w_i, c'est-à-dire d'échanger les lettres de part et d'autre de w_i.
Exemple : une grande rotation de python autour de la lettre y en position 2 donne typhon.

L'enjeu consiste à trouver *la plus petite séquence* d'opérations atteignant notre but. Il nous faudra pour cela ajouter quelques directives supplémentaires au programme.

Questions

1. Créez un fichier domain.pddl dans lequel vous définirez les deux types et les deux prédicats utiles au problème *Indication : un mot peut être représenté par ses lettres, chacune à une position donnée.*

Afin de trouver des plans optimaux, il va nous falloir associer un coût (sous forme numérique) à chaque action. En PDDL il s'agit d'ajouter dans le domaine :

— La mention :**action−costs** dans les :**requirements** ;
— La définition (:**functions** (total−cost) − number) dans le corps du problème, traditionnellement entre :**predicates** et les actions.
— Un coût à chaque action. En pratique, on ajoute la ligne (**increase** (total−cost) 1) dans les effets, pour attribuer un coût de 1 à l'action.

Questions

1. Définissez une action small−rotation qui prend en paramètres deux positions contiguës, ainsi que leurs valeurs associées, et qui effectue une petite rotation sur ces valeurs. On donnera à cette action un coût de 1.
2. Définissez une action big−rotation qui prend en paramètres trois positions contiguës, ainsi que les valeurs associées aux positions extrêmes, et qui effectue une grande rotation sur ces valeurs. On donnera à cette action un coût de 1.

Une fois le domaine défini, l'instance peut être définie comme précédemment, à ceci près que :

— Il faut initialiser le coût total à 0. Cela se fait en rajoutant (= (total−cost) 0) dans l'état initial.
— Il faut indiquer au planificateur que la métrique à minimiser est total−cost. Il suffit de rajouter, *après* la déclaration du but, la mention (:**metric** minimize (total−cost))

Questions

1. Définissez les instances associées aux problèmes suivants :
   — Harpe ⟶ Phare
   — Chien ⟶ Niche
   — Rabachai ⟶ Charabia
   — Guerison ⟶ Soigneur
2. Augmentez le coût de l'action big−rotation à 2, et réévaluez le troisième exemple. Que constatez-vous ?
3. Même question, mais en augmentant son coût à 4.
4. Essayez de résoudre le problème Cumulostratus ⟶ Stratocumulus
5. Réflexion : comment procéder pour remplacer les coffres par des portes de communication fermées à clef et qui doivent être ouvertes pour avancer dans le labyrinthe ?