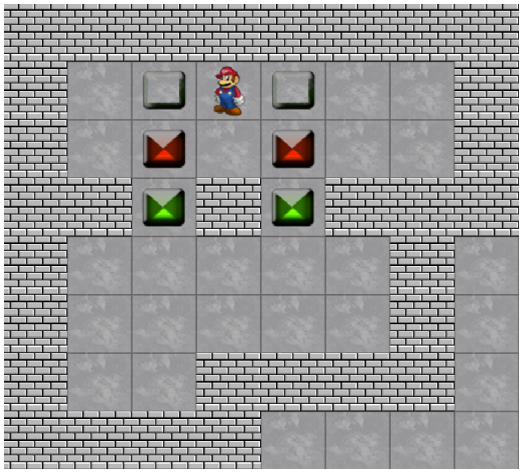
Projet - Sokoban : modélisation et résolution



Le sokoban est un petit jeu vidéo de puzzle inventé au Japon au début des années 1980. Il met en scène un gardien d'entrepôt (traduction littéral du nom du jeu) - le joueur - qui se déplace dans un entrepôt, représenté par une grille de cases, et dont le but est de ranger des caisses sur des cases cibles. Le plateau de jeu est constitué de deux types de cases, des murs que ni le joueur ni les caisses n'ont le droit de franchir et de cases vides. La structure de l'entrepôt et le nombre de caisses varient d'un niveau à l'autre. Les règles du jeu sont simples :

- il y a toujours autant de cases cibles que de caisses ;
- le joueur peut se déplacer dans les quatre directions cardinales ;
- le joueur a uniquement le droit de se déplacer vers une case inoccupée (et qui n'est pas un mur) ou de pousser une caisse et de se déplacer ainsi sur la case libérée. Il n'a pas le droit de tirer une caisse ni de passer par dessus ;
- pour pousser une caisse, la case adjacente de la caisse dans la direction de poussé doit être libre ! Le joueur ne peut pas pousser deux caisses en même temps ;
- le joueur gagne lorsque toutes les caisses sont sur les cases cibles, dans n'importe quel ordre. Il n'y a pas de caisses assignés à des points cibles particuliers ;
- le score du joueur est le nombre de poussées réalisées (indépendamment du nombre de déplacement du joueur).

L'objectif du projet est dans un premier temps de réaliser une modélisation du jeu, puis de réaliser un solveur. Des questions bonus permettent d'améliorer le solveur.

Exercice 1 – Modélisation du jeu - entre 4 et 6h

Note : Avant de commencer à coder, lisez attentivement TOUT le contenu de l'exercice

Le plateau de jeu se présente comme une grille en deux dimensions et on doit pouvoir représenter les éléments suivants : les murs qui sont statiques, les cases support des éléments mobiles qui se déplacent, les caisses, les cases cibles et le joueur. La modélisation proposée utilise deux classes différentes pour stocker les éléments selon leur nature dynamique ou statique :

- la classe `Niveau` contiendra les éléments statiques du jeu : les murs et les cases de déplacements, sous la forme d'un tableau à deux dimensions ; les cases cibles sous la forme d'une `ArrayList` ;
- la classe `Configuration` contiendra les éléments dynamiques du jeu : le joueur par une variable d'instance et les caisses sous la forme d'une `ArrayList`.

De cette manière on découple les informations ce qui permettra d'alléger la partie résolution, très gourmande en mémoire et en calcul.

Architecture générale

Tous les éléments du jeu héritent d'une même classe `Element` ; deux classes intermédiaires sont utilisées : `Immobile` et `Mobile`. La classe `Case` et la classe `Mur` héritent de `Immobile` ; la classe `Joueur` et la classe `Caisse` de `Mobile`.

Modélisation des déplacements avec `Position` et `Direction`

Pour les éléments immobiles, il n'est pas nécessaire d'enregistrer la position dans leur instance ; par contre, pour les éléments mobiles, vu que ceux-ci ne sont pas représentés dans la grille, il est nécessaire qu'ils stockent eux-même leur position, par le biais d'une classe `Position` qui représente un couple de coordonnées (x, y).
Afin de pouvoir faire évoluer les éléments mobiles, nous aurons également besoin d'une classe `Direction` qui permet de modéliser les actions possibles - les déplacement à droite, à gauche, en haut et en bas - sous la forme d'un couple de coordonnées (dx, dy) :

- $(0, -1)$ pour la direction `HAUT`
- $(0, 1)$ pour la direction `BAS`
- $(-1, 0)$ pour la direction `GAUCHE`
- $(1, 0)$ pour la direction `DROITE`

La classe est munie également d'une méthode `ArrayList<Direction> getDirections()` qui permet de renvoyer l'ensemble des directions. La classe `Position` a un opérateur `Position add(Direction)` (respectivement `Position sub(Direction)`) qui permet d'obtenir la position dans le sens de la `Direction` passée en paramètre en additionnant la direction à la position courante (respectivement dans le sens inverse en soustrayant la direction).
À noter que pour des raisons de sécurité, les objets `Position` doivent être immutables : une instance ne pourra pas changer ses valeurs de coordonnées, elles sont fixées une fois pour toute lors de la création de l'objet. Ainsi, les opérations sur les positions renverront toujours une nouvelle instance `Position`.
Il sera également nécessaire de tester l'égalité entre deux instances de `Position` et donc de coder une méthode `boolean equals(Object)`.

Positionnement et déplacement des `Elements`

Un élément `Immobile` n'a pas besoin d'avoir une méthode de positionnement puisqu'il n'a pas l'information correspondante (pas d'attribut `Position`). Un élément `Mobile` a une méthode `boolean setPosition(Position)` afin de pouvoir fixer ses coordonnées lors de l'*initialisation* ou d'un *déplacement* et qui renvoie `true` si le placement est possible et `false` sinon.
De plus, la classe `Element` est munie d'une méthode `boolean bougerVers(Direction)` qui tente de faire bouger l'élément dans la direction souhaitée (renvoie `true` en cas de succès, `false` en cas d'échec), et dont le comportement diffère en fonction de l'élément :

- un `Mur` n'est jamais déplaçable ;
- une `Case` qui ne contient pas d'autres éléments est toujours déplaçable (elle est vide - cela n'a pas d'intérêt de la déplacer et il ne se passe rien quand elle est déplacée, la méthode renvoie juste `true`, mais cela facilitera le codage) ;
- une `Caisse` ne peut être déplacée que si la case de destination est vide ;
- un `Joueur` ne peut bouger que si la case de destination est vide, ou contient une caisse qui peut être poussée dans la direction souhaitée (dans ce cas, la `Caisse` est poussée puis le `Joueur` déplacé).

Attention : la méthode `bougerVers(Direction)` n'a pas à changer directement les coordonnées de l'objet, il faudra qu'elle appelle la méthode `setPosition(Position)` de l'instance.

Méthodes supplémentaires pour les `Elements`

Afin de connaître facilement la nature d'un objet, la classe `Element` est munie d'une variable d'instance `Type type` qui peut prendre quatre valeurs : `MUR`, `CASE`, `CAISSE`, `JOUEUR`. La classe `Type` ne contient rien d'autre que cette *énumération* des types possibles.
De plus, la classe `Mobile` contient une référence vers sa `Configuration` afin d'interagir avec les autres `Element`s du jeu.
La classe `Joueur` sera munie d'une variable `ArrayList<Direction> histo` pour garder l'historique des coups joués par le joueur.

Les classes `Mur`, `Case` et `Caisse` n'ont pas d'attributs ou de méthodes supplémentaires.

Classe `Niveau`

La classe `Niveau` est munie d'un tableau `grille` à deux dimensions composé d'objets `Immobile`, et d'une variable `ArrayList<Position> cibles`. La `grille` représente le niveau, en considérant qu'il y a des murs sur tous les bords du niveau. Il n'y aura donc pas à gérer explicitement les tests pour savoir si un déplacement hors du plateau est tenté. La liste `cibles` enregistre les positions des cibles du niveau (là où les caisses doivent être déplacées). On ne fera pas d'objet en particulier pour les cibles, connaître leur position suffit.
Un `Niveau` comporte les méthodes suivantes :

- `int getX()` et `int getY()` permettent de connaître la taille du niveau ;
- `boolean addMur(Position)` ajoute un mur et renvoie `true` en cas de succès (pas de mur déjà posé) ;
- `boolean addCible(Position)` rajoute une cible (succès si la case n'est pas un mur ou ne contient pas déjà une cible) ;
- `boolean estCible(Position)` teste si une cible se trouve à la `Position` passée en paramètre ;
- `boolean estVide(Position)` teste si la case ne contient pas de mur ;
- `Immobile get(Position)` renvoie l'objet stocké à la position en paramètre dans le tableau `grille`.

Composition de la classe `Configuration`

Attributs :

- une `ArrayList` de `Caisse` qui dénote toutes les caisses du niveau ;
- un `Joueur` ;
- un `Niveau`.

Constructeurs :

- un constructeur qui prend en paramètre le niveau et la position initiale du joueur ;
- un constructeur par copie.

Méthodes :

- `int getX()` et `int getY()` similaires à celles de `Niveau` ;
- `boolean addCaisse(Position)` pour placer une nouvelle caisse ;
- `boolean estCible(Position)` pour savoir si la position est une cible ;
- `boolean estVide(Position)` similaire à celle de `Niveau` ;
- `Element get(Position)` qui permet de renvoyer l'`Element` à la position en paramètre : le `Joueur`, une `Caisse`, un `Mur` ou une `Case`. C'est cette méthode qui sera principalement utilisée ! Celle de `Niveau` renvoie uniquement des informations statiques. Celle-ci peut renvoyer toute l'information présente dans la configuration, y compris les élément mobiles ;
- `boolean bougerJoueurVers(Direction)` permet de déplacer le joueur (renvoie `true` en cas de succès) ;
- `boolean victoire()` permet de savoir si le joueur a gagné ou pas ;

Munissez également cette classe d'une méthode `String toString()` pour pouvoir afficher une configuration en suivant les règles classiques de représentation du Sokoban :

- Mur : #
- Cible : .
- Case : " "
- Joueur : @, Joueur sur une cible : +
- Caisse : $, Caisse sur une cible : ∗

Un exemple de représentation :

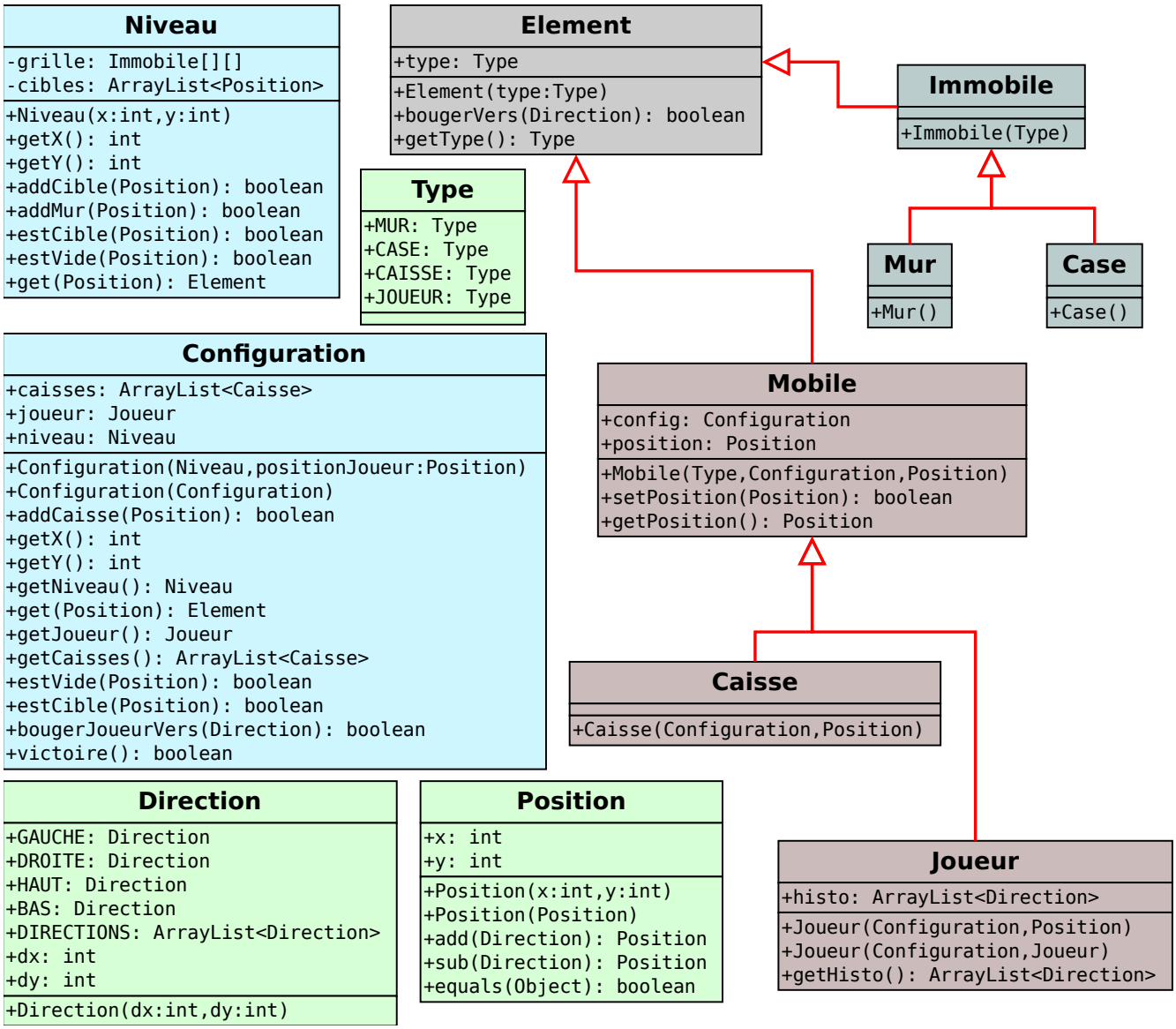
```
######
#@$. #
######
```

Important : Règles à suivre

La figure suivante présente le diagramme UML du jeu, sans aucune information sur `static`, `final`, `abstract`, `public`, `protected`, `private`, ... Votre travail consiste à implémenter le jeu à partir des éléments fournis et en faisant attention :

- de respecter absolument le nommage des variables et des classes et le squelette de la hiérarchie fournie ;
- de coder de telle manière que votre jeu soit sécurisé ! Il ne doit pas être possible pour un utilisateur de déplacer des caisses sans une action du joueur, de téléporter le joueur, ...
- de réaliser un code clair, élégant et si possible utilisant quand c'est préférable le type `Enum` (voir par exemple `https://openclassrooms.com/courses/apprenez-a-programmer-en-java/les-enumerations-1`). Faîtes également l'économie des instances créées quand c'est possible !

Vous avez le droit d'ajouter des méthodes ou des variables si vous le pensez nécessaire. Ajoutez au moins des méthodes `toString()` pour votre débugage. L'évaluation portera sur tous les points suscités. Une classe pour tester votre code et réaliser l'affichage graphique du jeu sont fournies sur `http://www-connex.lip6.fr/~baskiotisn/index.php/2017/09/20/2i002-introduction-a-la-programmation-objet-20` lire le `README.txt`.



Exercice 2 – Résolution naïve du sokoban - 30min

Les solveurs proposés dans la suite suivent tous la même logique :

1. une configuration est chargée dans le solveur
2. le solveur joue un coup (ce qu'on appellera également un *pas*)
3. le solveur teste si la configuration est une victoire
4. on boucle tant qu'une solution n'est pas trouvée

L'interface `SolverInterface` permet de représenter ses opération, elle contient les méthodes suivantes :

- `void set(Configuration)` : permet de définir la configuration que l'on souhaite résoudre ;
- `public Configuration getConfiguration()` : permet de récupérer la configuration courante du solveur ;
- `public boolean stop()` : renvoie `true` si le solveur a fini (trouvé une solution ou est bloqué) ;
- `public int getTotalSteps()` : renvoie le nombre de configuration testées ;
- `public void step()` : progresse d'un pas le solveur ;

Coder un solveur naïf qui ne fait que jouer chaque coup au hasard.

Exercice 3 – Résolution aléatoire "intelligente" - difficile (1 à 2h)

Le problème évident du solveur précédent est que votre joueur passe son temps à faire des actions inutiles : seules les actions qui consistent à pousser des caisses ont un intérêt et font progresser la résolution. Ainsi, une première amélioration consiste à identifier les **coups** d'une configuration - les actions possibles sur les caisses - et jouer au hasard parmi celles-ci. Un coup correspond ainsi à un couple `(Position p,Direction d)`, avec `p` la position de la caisse et `d` la direction de poussée.
Pour identifier tous les coups possibles d'une configuration, nous avons besoin de connaître toutes les cases atteignables par le joueur. Pour cela, on propose d'utiliser l'algorithme de Dijkstra de calcul de plus court chemin dans un graphe ou ici dans un labyrinthe. Cet algorithme utilise une grille d'entiers pour représenter les cases. Á la fin de l'algorithme, chaque case aura pour valeur la distance minimale (en termes de déplacements) qui la sépare de la position du joueur. Les cases sans distances sont les cases que le joueur ne peut pas atteindre. Cet algorithme donne également les déplacements nécessaires pour atteindre une case donnée : il suffit de considérer la case cible, de trouver la case voisine qui est de distance la plus faible, de se positionner à cette case et d'itérer successivement jusqu'à ce retrouver sur la case du joueur. En prenant dans l'ordre inverse les déplacements effectués, nous obtenons la liste des déplacements pour atteindre la case.
L'algorithme de Djikstra utilise une liste pour stocker temporairement les cases en cours de traitement. Le principe est le suivant :

1. Toutes les cases sont initialisées à 0 (valeur représentant une case sans distance) sauf la case du joueur qui est initialisé à 1 ;
2. la liste des cases en cours de traitement contient une case, celle du joueur ;
3. Tant que la liste n'est pas vide :
   - retirer une case de la liste ; ce sera la case courante avec la distance courante ;
   - pour chaque case voisine (dans les 4 directions), si la case n'a jamais été visitée (distance de 0) ou si la distance de la case est plus grande que la distance courante plus 1, mettre distance + 1 dans la case et l'ajouter à la liste ; sinon ne rien faire.

Le but de l'exercice est d'implémenter l'algorithme de Djikstra, puis de coder le solveur correspondant. Quelques indications :

- Coder une classe `ListeCoups` qui se chargera de l'algorithme de Djikstra. Elle doit prendre une configuration, calculer la distance de toutes les cases au joueur avec l'algorithme de Djikstra, puis construire la liste de tous les coups possibles (un coup n'est possible que si le joueur peut

atteindre la case adjacente à la caisse dans la direction opposée à la direction de poussé) et enfin avoir une méthode qui permette de retourner la liste des déplacements pour effectuer le coup.
- Vous aurez également besoin d'une classe `Coup` pour stocker le couple représentant un coup.
- Pour l'algorithme de Djikstra, il est conseillé d'utiliser la classe `LinkedList` plutôt qu'un `ArrayList`.
- Pour le solveur, il est conseillé d'avoir une méthode `ArrayList<Direction> getNextCoup()` qui renvoie un coup aléatoirement parmi ceux possibles sous la forme d'une liste des directions à prendre pour le jouer. Cette liste sera stocké dans une variable d'instance ; la méthode `step()` dépile un déplacement et l'effectue tant que la liste n'est pas vide. Si celle-ci est vide, la méthode `getNextCoup()` est appelée pour récupérer le nouveau coup à jouer.
- Si jamais le solveur est bloqué (plus aucun coup possible), on réinitialise la configuration à la configuration initiale.

Exercice 4 – Solveur vraiment intelligent - Bonus (de 1h jusqu'au jour de l'an et plus si affinité)

Parmi les problèmes du solveur aléatoire, on peut au moins identifier les suivants :
- On ne détecte pas les configurations qui sont impossibles à résoudre et qui sont donc vouées à l'échec (par exemple une caisse dans un coin).
- On retombe souvent sur des configurations que l'on a déjà vu.
- il est aléatoire ! pas d'exploration systématique de toutes les solutions possibles à partir d'une configuration donnée.

Pour traiter le premier problème, il est possible de choisir une méthode heuristique, à base de règles, en spécifiant à la main les configurations que l'on sait impossible à résoudre (deux caisses qui se font faces dans un couloir que l'on ne pourra jamais pousser, une caisse dans un coin, ...). Une autre solution est d'utiliser une méthode algorithmique : si il existe une sous-configuration en enlevant des caisses de la configuration qui n'est pas solvable, alors la configuration ne peut pas être solvable. Il serait trop long de tester toutes les sous-configurations, par contre il est faisable facilement de tester toutes les sous-configurations à une caisse, voir à deux caisses.

Pour traiter le second problème, nous avons besoin d'une mémoire des configurations déjà testées (et en particulier des configurations qui ont menées à un échec). Pour cela, il est conseillé d'utiliser une `HashMap` afin d'économiser de la mémoire (structure de stockage qui permet de rechercher un élément en temps négligeable). Il est nécessaire d'avoir une représentation unique en chaîne de caractère pour utiliser une telle structure. On peut par ailleurs remarquer que beaucoup de configurations sont équivalentes : peu importe où le joueur est situé sur le plateau, ce qui compte ce sont les coups accessibles .... Pour coder toutes les configurations équivalentes de manière unique en chaîne de caractères, on peut utiliser le codage suivant : $(x_j, y_j)|(x_{c_1}, y_{c_1})|\cdots|(x_{c_n}, y_{c_n})$ où (x_{c_i}, y_{c_i}) représente les coordonnées de la caisse i et (x_j, y_j) les coordonnées du joueur. Attention, afin de coder de manière unique, il est nécessaire d'ordonner les caisses dans un ordre (par exemple dans l'ordre des x croissants puis des y croissants), et également que la position du joueur soit codé de manière unique, par exemple la première case atteignable dans le même ordre croissant de x et de y.