

Rapport TPA phase 2

Groupe 3 Bleu : Sokoban

Basset Emilien , Goron Nathan, De La Rosa Louis-David, Demé Quentin



Contents

1	<u>Introduction</u>	2
2	<u>Cahier des Charges</u>	3
2.1	Résumé des objectifs , principe du jeu:	3
2.2	Pré-requis d'utilisation	3
2.3	Recensement des fonctionnalités	3
2.3.1	Fonctionnalités obligatoires	3
2.3.2	Fonctionnalités additionnelles	3
2.4	Découpage de l'application	4
3	<u>Explication du programme ; interface et moteur de jeu</u>	5
3.1	Moteur de jeu	5
3.1.1	Le menu	5
3.1.2	Découpage et fonctionnement du moteur de jeu	5
3.1.3	Communication jeu/utilisateur et utilisation du programme	7
3.1.4	Restriction de déplacement du personnage	8
3.1.5	Déplacement des caisses	8
4	<u>Algorithme de résolution A* et heuristique</u>	9
4.1	Présentation et objectif:	9
4.2	Explication détaillée avec exemples.	9
4.2.1	Fonctions g et h:	9
4.2.2	Etapes détaillées de l'algorithme:	10
4.3	Le cas du Sokoban:	10
4.3.1	Notion d'état	10
4.3.2	Fonctionnement	10
4.3.3	Implémentation:	13
5	<u>Schéma UML : diagramme de classe</u>	15
6	<u>Tests de fonctionnalité</u>	16
7	<u>Tests de performance de la résolution automatique</u>	17

1 Introduction

Parmi les projets proposés, nous avons fait le choix d'éviter les projets n'étant pas des jeux de peur qu'ils limitent notre créativité et notre motivation quant

a son développement. Le jeu de Sokoban nous paraissait être un projet ambitieux et intéressant de par la difficulté de l'implémentation de son IA de résolution automatique. Il s'agissait donc de développer un Jeu de Sokoban :

Un jeu de puzzle dans lequel un personnage déplace un certains nombres de caisses sur des emplacements dans un niveau fermé, le joueur gagne la partie si il parvient à boucher tous les emplacements avec les caisses.

Après avoir songé à l'utilisation du moteur de jeu Unreal Engine, nous avons décidé de développer ce projet en Python avec l'aide de la librairie PyGame qui offre une grande liberté dans le développement de l'interface graphique ainsi que de nombreuses méthodes facilitant les mécaniques de fonctionnement de base d'un jeu.

2 Cahier des Charges

2.1 Résumé des objectifs , principe du jeu:

Dans le cadre de L'UE "Travaux personnels encadrés" , il s'agira de développer un jeu de sokoban : Un joueur est introduit dans un niveau constitué de caisses et d'obstacles , il doit pousser toutes les caisses sans bloquer ces dernières afin de compléter le niveau. L'application sera agrémenté d'une fonctionnalité de résolution automatique dites anytime via un algorithme Astar ainsi que d'autres fonctionnalités additionnelles facultatives.

2.2 Pré-requis d'utilisation

Le jeu fonctionnera sous n'importe quelle machine sous linux/windows munie de Python(2.7 a 3.5) ainsi que des modules Pygame et PIL .Le jeu sera intuitif d'utilisation , ne nécessitera aucune connaissance informatique particulière et sera donc accessible à tout type de public .

2.3 Recensement des fonctionnalités

2.3.1 Fonctionnalités obligatoires

Le jeu devra obligatoirement présenter un moteur de jeu permettant au joueur de se déplacer dans le niveau en respectant les règles(pousser les caisses , collisions avec les blocs non traversables ...).Le jeu sera également muni d'une fonctionnalité de résolution automatique anytime.

2.3.2 Fonctionnalités additionnelles

L'application se verra ajouter les fonctionnalités suivantes :

- Une interface graphique intuitive et simple d'utilisation , dotée d'un menu principal pour guider le joueur dans le choix du niveau et les autres options disponibles .

- Un set de plusieurs styles pour le personnage et le niveau
- Une option de sauvegarde / chargement de partie
- Une fonctionnalité de retour arrière suite à un mouvement involontaire ou à une erreur dans un niveau

2.4 Découpage de l'application

Le programme sera composé des modules suivants :

- Le module sokoban qui servira de module "Init" au programme.
- Le module classes qui comprendra toutes les classes et méthodes nécessaire au fonctionnement du jeu.
- Le module constantes stockant les instanciations d'objets ,les importations d'images pour les personnages et les cases ainsi que les variables nécessaire pour les variations de tailles de niveaux à l'écran.
- Le module IA qui gérera l'exécution de l'algorithme Astar et de l'heuristique. Ce découpage pourra être modifié en fonction des choix de fonctionnalités additionnelles.
- Le module GUI qui assurera la communication entre le joueur et l'interface.
- Le module sauvegarde qui permettra la sauvegarde et chargement d'une partie dans le dossier "/sauvegardes"
- Le module imagery qui permettra le redimensionnement des niveau dans la fenêtre en fonction de la taille de ceux-ci via la librairie PIL
- Le module l10n contenant le dictionnaire des traductions pour assurer l'internationalité du programme

3 Explication du programme ; interface et moteur de jeu

3.1 Moteur de jeu

3.1.1 Le menu

L'une des principales modifications de l'interface lors de cette seconde phase n'est autre que l'ajout d'un menu interactif regroupant plusieurs options pour le joueur. Parmi ses options, on peut citer:

- Le bouton **Nouvelle Partie**, pour démarrer une partie.
- Le bouton **Charger Partie**, pour reprendre une partie sauvegardée.
- Le bouton **Option**, pour configurer certaines propriétés du jeu.
- Un bouton **Quitter**, pour quitter le jeu.

De plus, le menu se charge de jouer une musique d'introduction pour le joueur. Tout cela est commandé par la fonction **collectionMenu()**.

Dans un premier temps la fonction se charge de placer les boutons et l'interface globale. Les boutons sont simplement des images que l'on vient coller sur l'interface. Pour finir, la fonction s'engage dans une boucle infinie en attendant un événement de la part de l'utilisateur. On obtient ainsi la fenêtre suivante:

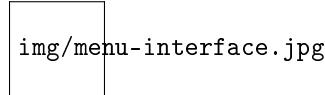


Figure 1: Menu du jeu Sokoban

3.1.2 Découpage et fonctionnement du moteur de jeu

Notre moteur de jeu repose sur 5 classes:

- **Sprite**
- **Personnage**
- **Caisse**
- **Niveau**
- **LevelCollection**

Les classes **Sprite**, **Personnage** et **Caisse** servent à représenter visuellement les objets suivants:

- **Personnage**: Classe représentant le personnage contrôlé par le joueur.
- **Caisse**: Classe représentant les entités manipulable par le personnage.
- *Les murs*: Moteur de difficulté du jeu; Ils bloquent le déplacement des caisses et du personnage.
- *Les stèles*: représentent les emplacements sur lesquels le joueur doit placer les caisses.
- *Les cases vides*: Cases sur lesquelles le personnage et les caisses peuvent être déplacées.

Pour construire notre plan de jeu, nous avons choisi de décomposer notre niveau en 2 grilles nommées *GameP* et *GameO* contenant à elles deux les objets cités plus haut. La grille *gameP*(game Plan) contient les cibles et les espaces vides tandis que la grille *gameO*(game Obstacle) contient les murs, les caisses et le personnage. L'affichage du niveau consiste donc en la superposition des deux plans et permet ainsi une vérification de victoire plus facile à la fin du jeu :



Figure 2: Aperçu des deux grilles superposées



Figure 3: Aperçu de la grille GameO

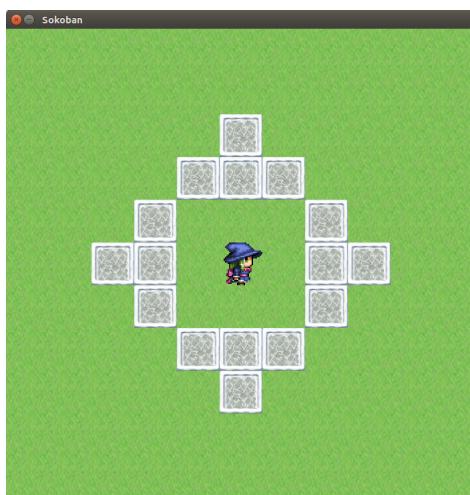


Figure 4: Aperçu de la grille GameP

3.1.3 Communication jeu/utilisateur et utilisation du programme

Pour les déplacements du personnage nous utilisons la méthode `event.key` de PyGame qui permet de gérer des événements liés à l'utilisation du touches du clavier , 6 touches événement sont actuellement gérés : touche directionnelle droite(`K-RIGHT` : déplacement du personnage vers la droite) ,touche directionnelle gauche(`K-LEFT` : déplacement du personnage vers la gauche) ,touche directionnelle haut(`K-UP` : déplacement du personnage vers le haut) ,touche directionnelle bas(`K-DOWN`: déplacement du personnage vers le bas) , pavé

numérique(KP-MULTIPLY: lance la résolution automatique du niveau avec As-tar) et la touche échap(KP-ESCAPE : fermeture du jeu)

3.1.4 Restriction de déplacement du personnage

Lorsque le joueur presse une des touche événement de déplacement citées plus ci-dessus , la méthode spéciale `déplace()` de la classe Personnage retournant un booléen est appelée pour vérifier si le déplacement est autorisé de la manière suivante:



Figure 5: Le joueur rencontre un mur, il ne peut pas se déplacer

3.1.5 Déplacement des caisses

Le déplacement des caisses est régie selon les règles suivantes :

- Le joueur ne peux pas tirer les caisses, il perd donc d'office la partie s'il pousse une caisse sur une case de coin qui n'est pas une stèle.
- Le joueur ne peux pousser qu'une seule caisse à la fois.

Dans le même ordre idée que pour la classe **Personnage**, la classe **Caisse** est dotée d'une méthode spéciale `déplace()` vérifiant si le déplacement d'une caisse est autorisé, à partir de la position du joueur et de la direction du déplacement demandé. La fonction vérifie d'abord si la prochaine case est bien une caisse et ensuite si la case suivante est une case vide ou une case stèle. Si cette condition est vérifiée le déplacement de la caisse s'effectue comme exposé ci dessous:



Figure 6: Exemple de déplacement autorisé



Figure 7: Exemple de déplacement impossible

4 Algorithme de résolution A* et heuristique

4.1 Présentation et objectif:

L'algorithme A* est un algorithme souvent utilisé pour trouver des chemins dans la résolution de jeux. Il est dans notre cas implémenté sur une grille de Sokoban. En dehors du sokoban où cela nous importe peu, il peut être interrompu pendant son utilisation et relancé plus tard. C'est un algorithme de type "anytime". Le but de A* est de trouver le chemin *le plus court* pour aller d'un point de départ à un point d'arrivée. Pour cela l'algorithme garde une liste de toutes les étapes futures que l'on appelle "liste ouverte". Cette liste est établie par la fonction $g(n)$. Ensuite, l'algorithme choisit parmi cette liste une étape future qui soit "*à priori*" la meilleure pour nous mener à l'objectif en un temps qui soit le plus court possible. Pour que cela puisse se faire, nous avons besoin d'une heuristique qui puisse nous aider à déterminer qu'elle est "*à priori*" la meilleure option. Une fois que cette option est choisie, l'algorithme passe à la "liste fermée".

4.2 Explication détaillée avec exemples.

4.2.1 Fonctions g et h :

A chaque étape de l'algorithme, les fonctions $g(n)$ et $h(n)$ sont exécutées pour n étant un point d'arrivée. Cela nous donne les informations suivantes.

ler du point de départ à l'arrivée n .

up pour aller de n à l'objectif final.

un d'étapes si on choisit le point n.

4.2.2 Etapes détaillées de l'algorithme:

Voici les étapes de l'algorithme en langage algorithmique:
Soient: OPEN la liste ouvert

Ajouter DEPART à OPEN

Tant que OPEN n'est pas vide:

 Choisir le N ayant le plus petit f(n)

 Si N est l'objectif alors PASS

 Déplacer N dans CLOSED

 Pour chaque N' = PeuBouger(N, direction):

 g(N')=g(N)+1

 h(N')

 Si N' dans OPEN et nouveau N' n'est pas mieux, CONTINUE

 Si N' dans CLOSED and nouveau N' n'est pas mieux, CONTINUE

 supprimer tout N' étant dans OPEN et CLOSED (en même temps)

 Ajouter N comme parent de N'

 Ajouter N' à OPEN

 FIN POUR

 FIN TANT QUE

Si on arrive ici, alors il n'y a pas de solutions.

4.3 Le cas du Sokoban:

4.3.1 Notion d'état

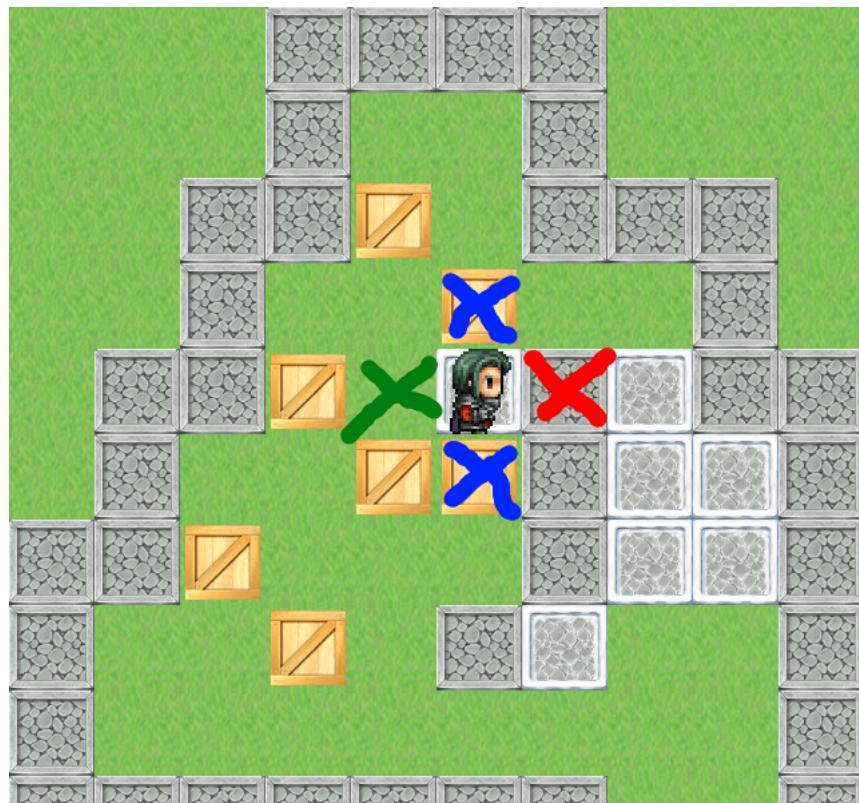
Supposons le cas où l'on souhaite exécuter l'algorithme dans un plan 2D et parcourir le chemin le plus court d'un point A à un point B. Un état serait alors défini par la position qui est actuellement lu. À partir de ceci, La position initial (donc l'état initial) serait les coordonnées du point A et l'état final serait les coordonnées du point B.

Dans notre cas, c'est différent car on ne souhaite pas parcourir une distance la plus courte possible, on souhaite dans un premier temps trouver une solution à une combinaison de positions pour des objets pouvant éventuellement rencontrer des obstacles. Un état est donc une combinaison de la grille du jeu, c'est à dire une liste contenant la position de toutes les caisses et de toutes les stèles. À partir de ceci, on peut définir l'état initial comme la grille au moment où on exécute A* et l'état final la combinaison de position pour laquelle sur chaque stelle, une caisse y correspond.

4.3.2 Fonctionnement

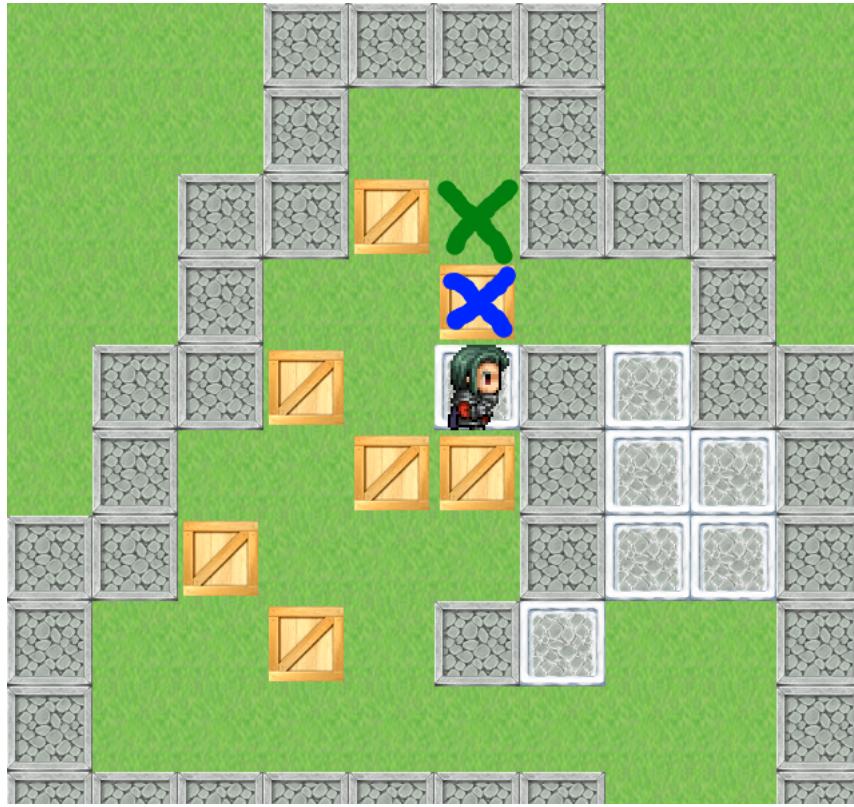
L'algorithme fonctionne dans une boucle qui a pour condition d'arrêt la correspondance entre caisses et stèles expliquée précédemment.

Pour commencer, l'algorithme lance une boucle finie sur les 4 positions autour du joueur.



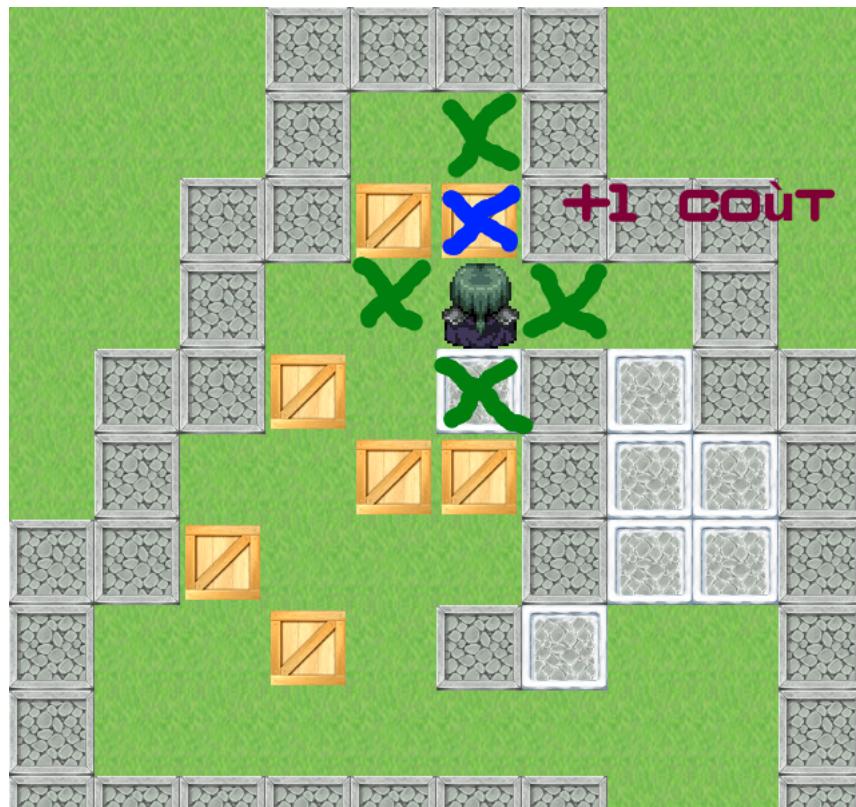
*Les 4 directions scannées par A**

Sur chaque position, il va regarder si il est possible de s'y déplacer (croix verte), de s'y déplacer à condition de déplacer une caisse (croix bleu) où si c'est impossible (croix rouge).



Cas de rencontre d'une caisse

A partir de là, s'il s'agit d'une caisse qu'il faut déplacer, il va mesurer la distance avec la stelle la plus proche. Ensuite il mesure cette distance une fois le déplacement effectué et vérifie si le déplacement est intéressant. Ce dernier vérifie si l'état actuel correspond à l'état final.



Nouvelle position Fin de la boucle, il relance à la nouvelle position du joueur.

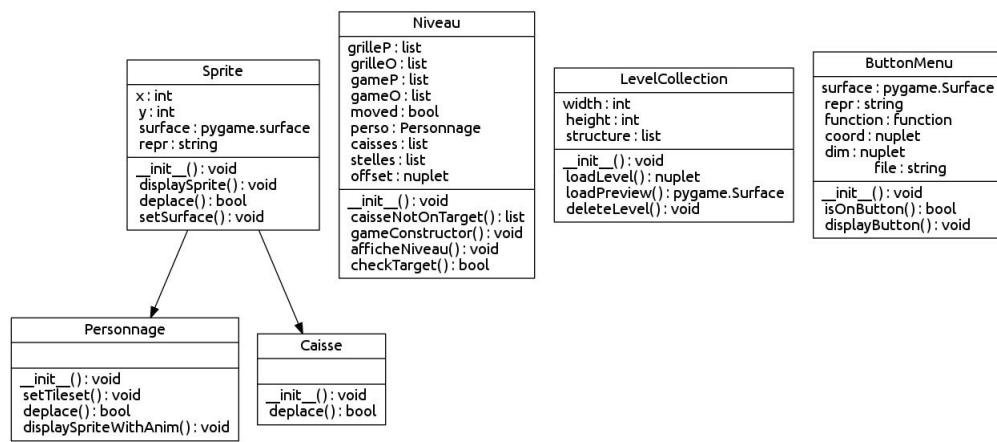
4.3.3 Implémentation:

Tout d'abord, nous avons besoin d'une structure pour les nœuds:

```
class Node implements Comparable
public Node parent;
public int move;
public Location where;
public int g;
public int h;
public int f() return g+h;
public Node(Location where, Node parent, int move);
public int compareTo(Object o);
```

Tout d'abord, il est plus pratique de garder la liste OPEN triée pour que l'on puisse avoir facilement accès à notre meilleur option. Il faut donc s'assurer que lorsqu'on ajoute un terme à la liste, celle-ci soit triée automatiquement. Ensuite il est efficace que nous puissions trouver des nœuds de la liste grâce à l'heure position sur la grille. Cela pour que nous puissions déterminer si 'n' est déjà dans OPEN ou dans CLOSE.

5 Schéma UML : diagramme de classe



6 Tests de fonctionnalité

Une batterie de tests ont été effectués via la fonction assert() pour vérifier la fonctionnalité du programme:

- tests sur classe LevelCollection
 - Aux extrêmes: grille de 99999999x99999999 -> erreur out of range
 - grille unitaire :grille 1x1 -> fonctionnel
- tests sur classe Niveau
 - test sur grille incompatible (blocs différents aux même coordonnées)
-> erreur lors des test de déplacement liés aux joueurs
 - test sur des grilles Plan et Obstacle de différentes taille -> erreur out of range si le personnage sort de la première grille
- tests sur le module Astar
 - lancement avec un nombre de caisse supérieur à celui des stelles -> arrêt lorsque plus de stelle disponible
 - test avec caisse inaccessible sur le niveau "niveautest1.slc" -> erreur
 - test sur niveau sans caisse -> retourne false (pas une erreur)
 - test sur niveau sans stelle -> retourne false (pas une erreur)

7 Tests de performance de la résolution automatique

L'heuristique utilisée effectuant tout les cas possibles , l'exécution de la résolution automatique s'avère très longue dans la plupart des niveaux proposées. Nous avons effectués des tests afin d'obtenir une ordre d'idée des capacités de l'algorithme sur plusieurs niveaux :

Nom du niveau	score de difficulté	temps de résolution de l'algorithme
Aruba Eri	125	9 secondes
AC Easy	148	30 minutes(environ)
AC DIAMOND	1170	INDETERMINÉ(>5 heures)
Amazing 0	6722	INDETERMINÉ (>5 heures)

Le score de difficulté est déterminé par ((le nombres de cases vides - le nombre d'obstacle) * le nombres de caisse a placer pour compléter le niveau). Les niveaux dont le temps de résolution est indéterminé on été soumis a l'exécution d'astar pendant plus de 5 heures au minimum , sans résultats .