



**UNIVERSITÉ
CAEN
NORMANDIE**

Rapport de projet

Sokoban

Legot Quentin 06.13.45.06.86,
Page Arthur 07.68.88.61.20,
Besevic Ivan 06.37.72.82.73,
Couture Thorkil

10 mai 2020

Université de Caen Normandie

Table des matières

1	Introduction	3
1.1	Explication du projet	3
1.2	Le Sokoban, qu'est-ce que c'est ?	3
2	Manuel du jeu	4
2.1	Préambule	4
2.2	Lancement du jeu	4
2.3	Navigation dans les menus	4
2.4	Choix du niveau	5
2.5	Comment jouer ?	5
2.6	" <i>Generate levels</i> "	6
3	Conception du jeu	7
3.1	Organisation du projet	7
3.2	Fonctionnalités	7
3.2.1	Les états du programme	7
3.2.2	Le rendu du jeu	9
3.2.3	Mouvement du personnage dans la grille	11
3.2.4	Solveur	12
3.2.5	Générateur	14
4	Éléments techniques	16
4.1	Structure de données	16
4.2	Exécution	17
4.3	Bibliothèques utilisées	17
5	Expérimentations	18
5.1	Mesures de performances	18
5.2	Difficultés rencontrés	18
6	Conclusion	19
6.1	Récapitulatif	19
6.2	Améliorations possibles	19

1 Introduction

1.1 Explication du projet

Le but de notre projet était de créer un jeu de type « Sokoban » à la fois fonctionnel et riche. Une fois le jeu de base terminé, nous avons décidé d'ajouter diverses fonctionnalités telles que le jeu contre un ordinateur, la génération aléatoire de niveaux et le comptage des points. Nous avons par ailleurs mis en place des niveaux de difficulté pour rendre le jeu accessible à tous et ainsi éviter d'avoir un contenu trop monotone. Nous avons partagé le travail à quatre pour que chaque membre puisse ajouter sa touche personnelle au jeu. Nous allons donc ici vous parler du jeu en lui-même (ses principes de base et ses mécaniques), des étapes de la conception et des différentes fonctionnalités que nous avons ajouté au jeu de base.

1.2 Le Sokoban, qu'est-ce que c'est ?

Créé au Japon en 1982, le Sokoban est un jeu vidéo de type puzzle à un joueur dans lequel on incarne un ouvrier devant ranger des caisses dans un entrepôt. Le joueur peut se déplacer dans toutes les directions et peut seulement pousser une caisse à la fois. Les déplacements en diagonale ne sont pas possibles et le joueur ne peut pas tirer de caisses. Dans l'entrepôt, outre les caisses et les murs, se trouvent aussi des points. Pour gagner et finir le niveau, il faut que tous les points soient recouverts par des caisses. Suivant la spécificité du niveau le joueur peut se retrouver bloqué en poussant une caisse dans un coin. Le but du jeu est donc de finir le niveau en effectuant le moins de mouvements possibles.

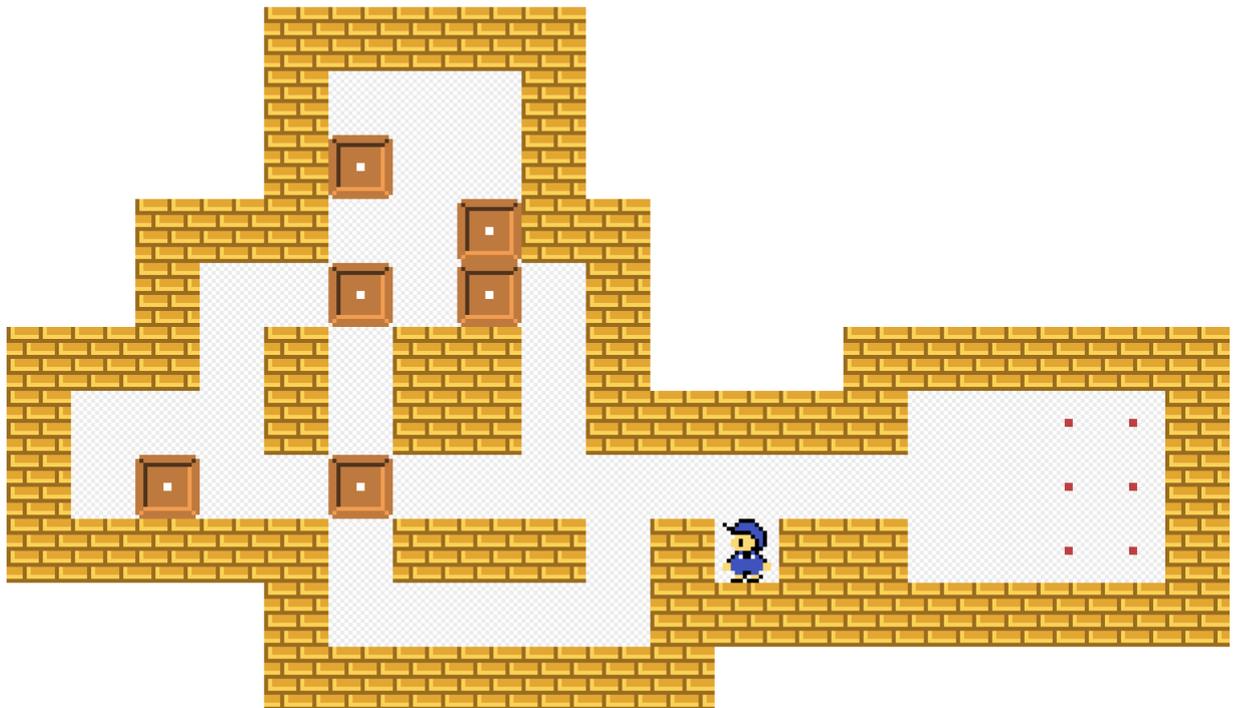


FIGURE 1 – Premier niveau du jeu original de 1982

2 Manuel du jeu

2.1 Préambule

Notre jeu a été développé pour les versions de Python supérieures à la 3.6, aucun test n'a été fait sur les précédentes versions.

Le jeu fonctionne sur Linux avec xorg et Windows 10, aucun test n'a été effectué sur Mac OS X (il peut y avoir des problèmes de dispositions clavier, appuyez sur W au lieu de Z par exemple).

2.2 Lancement du jeu

Tout d'abord, pour lancer le jeu, sachez qu'il vous faut avoir installé Python ainsi que la bibliothèque Pygame. Une fois ceci fait, voici les étapes à suivre pour lancer le jeu :

- Placez vous dans le répertoire racine du jeu, où se trouve le fichier *main.py*.
- Ouvrez un terminal et tapez "*python3 main.py*".
- Le jeu se lancera.

Le jeu étant lancé, une fenêtre s'ouvrira, vous menant directement au menu principal du jeu.

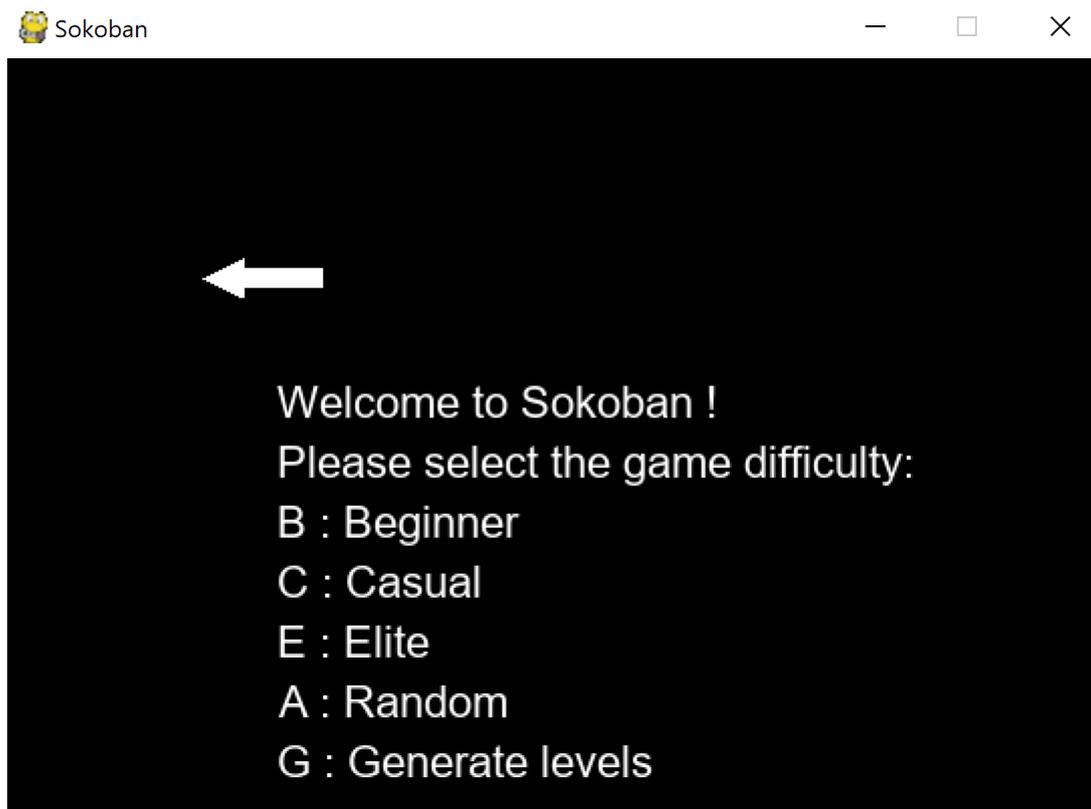


FIGURE 2 – Menu principal du jeu.

2.3 Navigation dans les menus

La navigation dans les menus se fait à la souris ou au clavier. Si vous utilisez la souris, cliquez simplement sur les lignes voulues ; pour le clavier, il faut appuyer sur les touches indiquées à

gauche des lignes. Pour les nombres, utilisez le pavé numérique ou faites *shift* + *NUM* si votre PC n'en est pas équipé.

Appuyez sur *ECHAP* ou sur la flèche qui va vers la gauche pour revenir en arrière ou quitter le jeu si vous êtes sur le premier écran. Vous pouvez appuyez sur la croix de la fenêtre pour quitter le jeu à tout moment.

2.4 Choix du niveau

Vous voulez jouer tout de suite ? Pas de problème, suivez simplement les étapes suivantes :

1. Choisissez le mode de jeu le plus adapté (*Beginner* , *Casual* , *Elite* ou *Random*).
2. Pour les trois premier choix, plusieurs niveaux s'offrent à vous, classés du plus simple au plus compliqué.
3. Pour le quatrième choix (*Random*), une grille sera crée selon vos préférences, vous n'avez qu'à entrer le nombres de caisses avec lesquelles vous souhaitez jouer (de 1 à 9).
4. Dernière étape enfin, vous devrez choisir si vous jouerez contre un ami (*Versus Player*), ou contre l'ordinateur (*Versus Computer*).

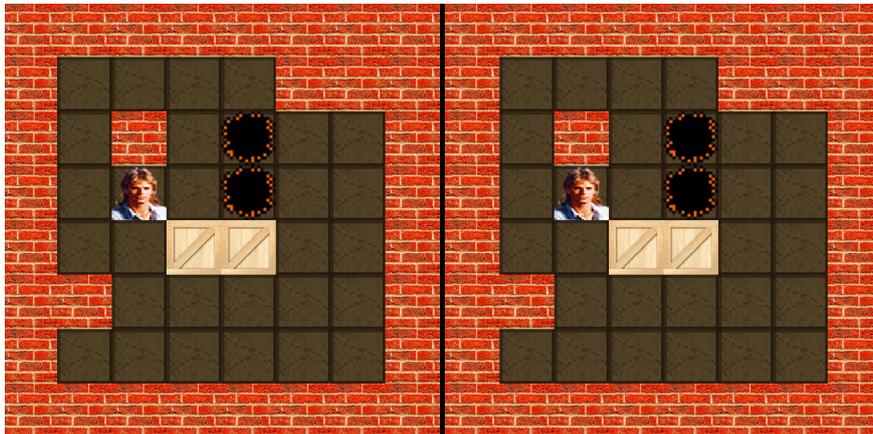


FIGURE 3 – Début de partie en mode deux joueurs. Difficulté : *Beginner*. Niveau : 1

2.5 Comment jouer ?

Si vous avez suivi les étapes précédentes, vous devriez voir apparaître la grille de jeu. Nous arrivons donc enfin à la partie la plus intéressante, comment jouer.

- Si vous jouer contre un ami, sachez que le joueur de gauche déplacera son personnage grâce au touches "*z,q,s,d*", respectivement "*haut, gauche,bas,droite*", et que le joueur de droite se servira des flèches du clavier. Le but, en mode deux joueurs est donc de placer toutes les caisses sur les trous en essayant de se déplacer au minimum. Ainsi le joueur ayant effectué le moins de coups pour placer ses caisses gagne la partie. Une fois la partie terminée, le nom du vainqueur est affiché et vous retournez automatiquement au menu principal.
- Si vous jouez contre l'ordinateur, vous devrez utiliser les touches "*z,q,s,d*" pour vous déplacer. L'ordinateur joue en même temps que vous (il se déplace quand vous vous déplacez) mais si jamais vous avez fini avant lui de placer toutes vos caisses, alors il finira sa partie automatiquement après vous.

- Sachez que si vous en avez assez de perdre ou tout simplement que le jeu vous ennueie, vous pouvez quitter à tout moment la partie grâce à la touche "*echap*" de votre clavier.

2.6 "Generate levels"

Vous avez sûrement dû vous rendre compte qu'une rubrique "*Generate levels*" est présente dans le menu du jeu. Vous pouvez y accéder en cliquant dessus ou en pressant la touche "*g*" de votre clavier.

Cette option permet de renouveler les niveaux du jeu entièrement. La méthode est la suivante :

- Tous les niveaux du jeu seront supprimés.
- Des niveaux seront créés grâce au générateur de niveaux (cf. Générateur 3.2.5).
- Ses niveaux seront ensuite testés par le solveur pour voir si il peut les résoudre (cf. Solveur 3.2.4).
- Si les niveaux sont résolubles par le solveur, alors ils prennent la place des anciens niveaux du jeu.

Tous les niveaux du jeux étant recrées, l'opération nécessite un certain temps. Nous vous conseillons donc de n'utiliser cette fonctions qu'en dernier lieu, lorsque vous vous serez lassé des niveaux actuels du jeux.

3 Conception du jeu

3.1 Organisation du projet

Voici la manière dont nous nous sommes réparties les tâches du projet. L'un de nos membres (COUTURE Torkil) nous ayant quitté en cours de projet, nous n'avons pas pu mentionner son nom dans ce tableau.

Fonctionnalités principales	Affichage	Solveur	Générateur
Ivan	Ivan	Quentin	Arthur

3.2 Fonctionnalités

3.2.1 Les états du programme

Pour que l'enchaînement des différentes mises en page du sokoban soient cohérentes, j'ai décidé de créer un fichier `logic/state.py` qui posséderait toutes les informations liées à l'état du programme avec des fonctions permettant de manipuler ces états.

Ce programme possède quatre situations possibles :

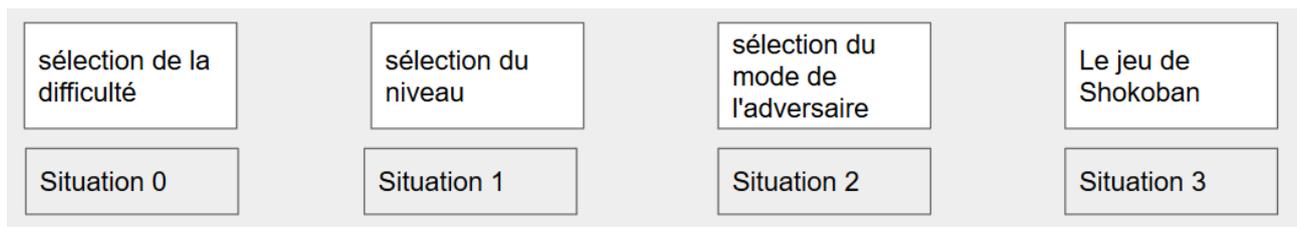


FIGURE 4 – Principales situations.

Ces situations se suivent de manière logique :

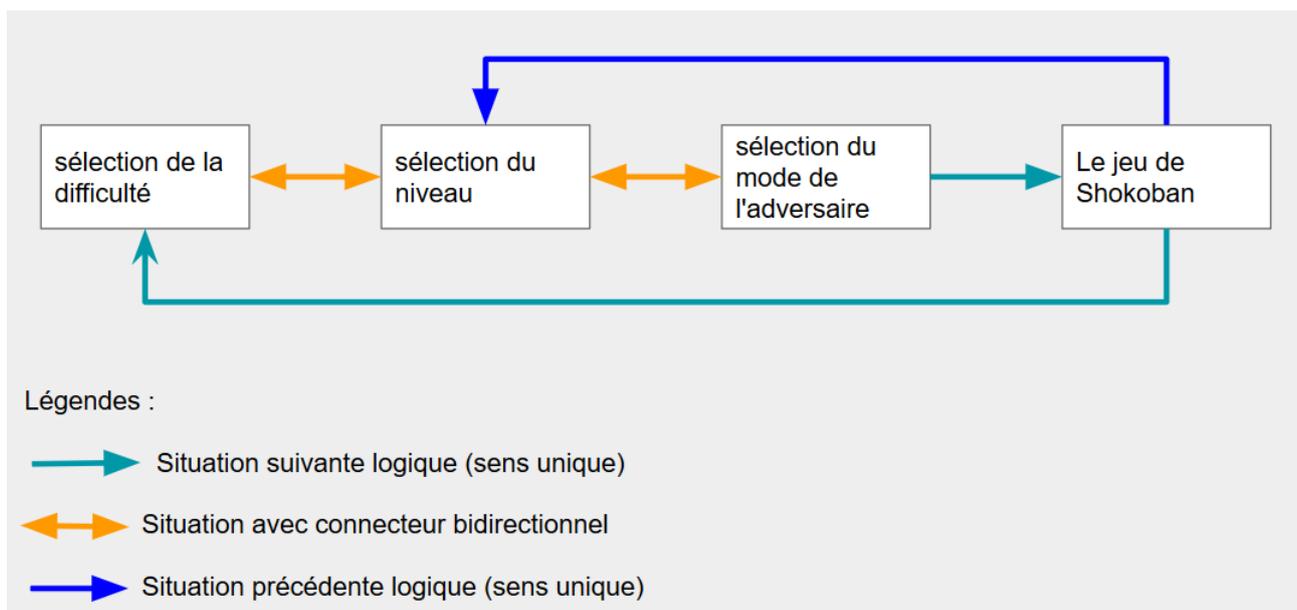


FIGURE 5 – Suivi des situations.

C'est la classe de state.py qui analyse les événements Pygame afin de prendre les bonnes décisions pour modifier l'état du jeu.

Ces événements Pygame vont donc modifier des variables python représentant l'état du jeu en fonction de la situation.

Celles-ci sont représentées par le schéma suivant :

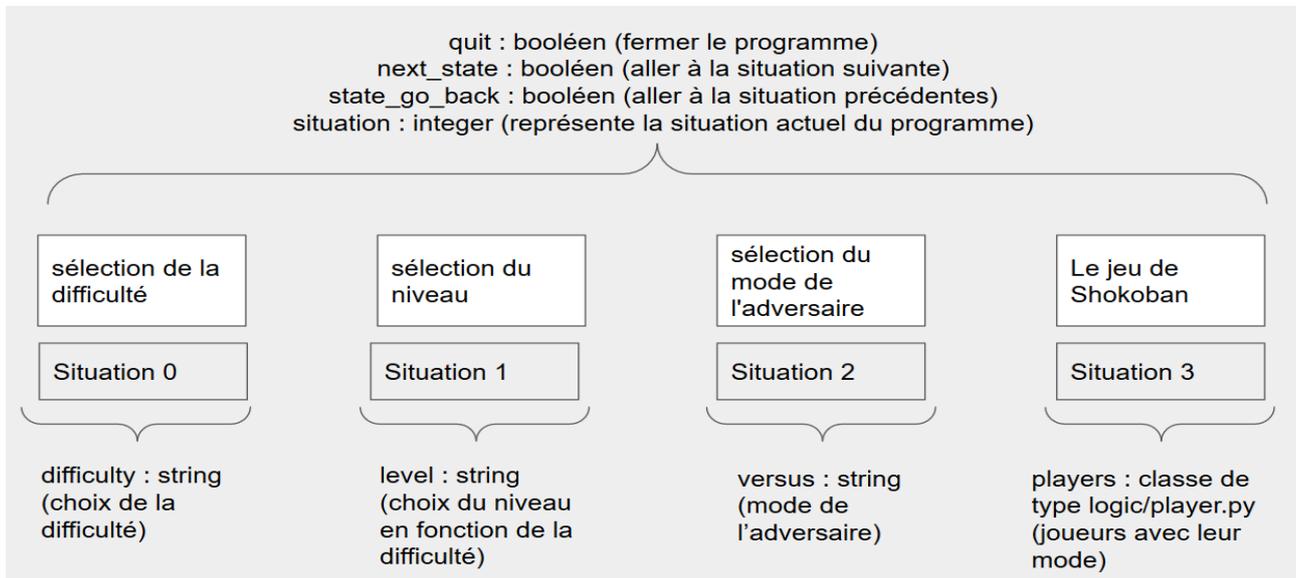


FIGURE 6 – Variables python modifiées selon la situation.

Chaque situation écoute des événements Pygame qui leur sont propres.

Dans notre jeu, nous écoutons uniquement les touches du clavier et de la souris (en prenant en compte la position pour la souris).

Les événements écoutés sont représentés par le schéma suivant :

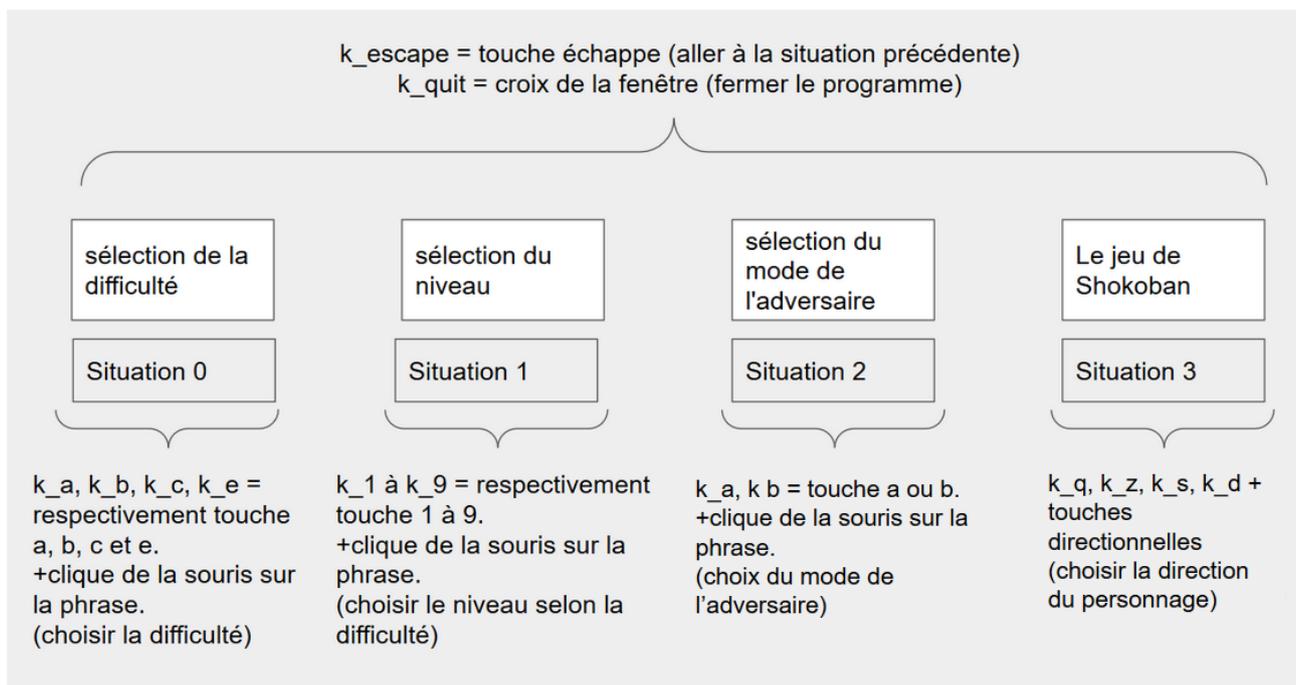


FIGURE 7 – Les événements écoutés selon la situation.

3.2.2 Le rendu du jeu

Une fois que l'utilisateur arrive dans la situation de jeu, la classe de `logic/state.py` va demander la génération des classes `players` de `logic/player.py`.

Cette classe `player` va à son tour demander la création d'une grille spécifique aux joueurs. Dans le programme on a donc une grille indépendante par joueur.

Tout d'abord une grille de lettres va être créée à partir d'une génération automatique (cf. La partie d'Arthur Page 3.2.5) ou d'un fichier `.sok`.

Dans ma partie, je vais vous parler de la création de la grille à partir des fichiers `.sok`.

Voici le contenu d'un fichier `.sok` :

```

1 #####
2 #   ###
3 # # . #
4 # @ . #
5 # $$ #
6 ##  #
7 #   #
8 #####

```

FIGURE 8 – Contenu d'un fichier `.sok`.

Celui-ci va être converti en grille de lettres :

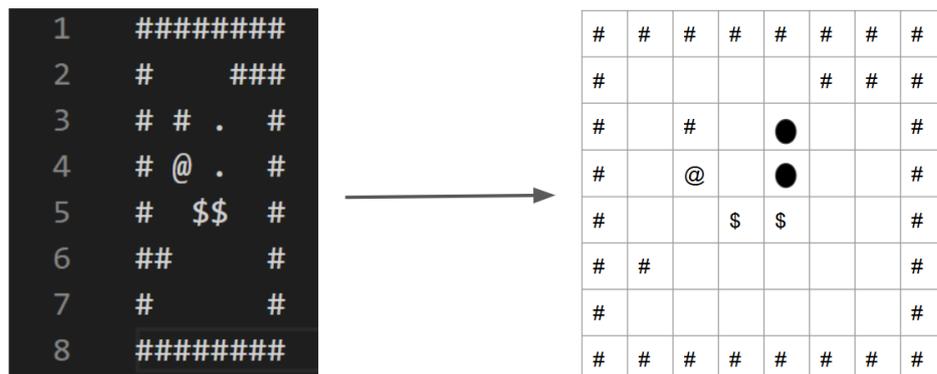


FIGURE 9 – Conversion fichier `.sok` en grille python.

Ensuite chaque lettre de la grille va être convertie en un sprite Pygame qui lui est associé.

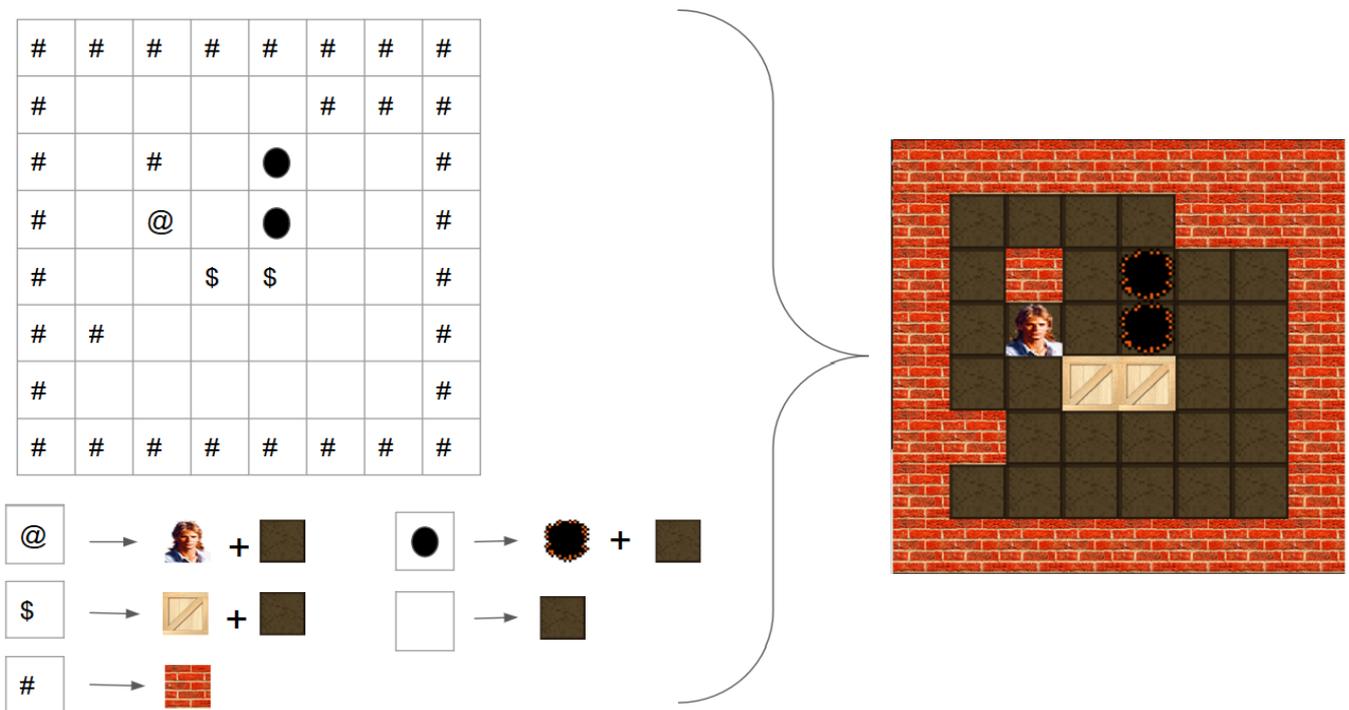


FIGURE 10 – Conversion des lettres de la grille en sprites.

On a deux types de sprite Pygame dans le programme :

- Les sprites superposables [joueur(@), boites(\$), trous(.), air()]
- Les sprites non superposables [mur(#)]

Pour positionner les sprites sur la fenêtre Pygame, nous avons dû effectuer des mesures sur la/les grilles.

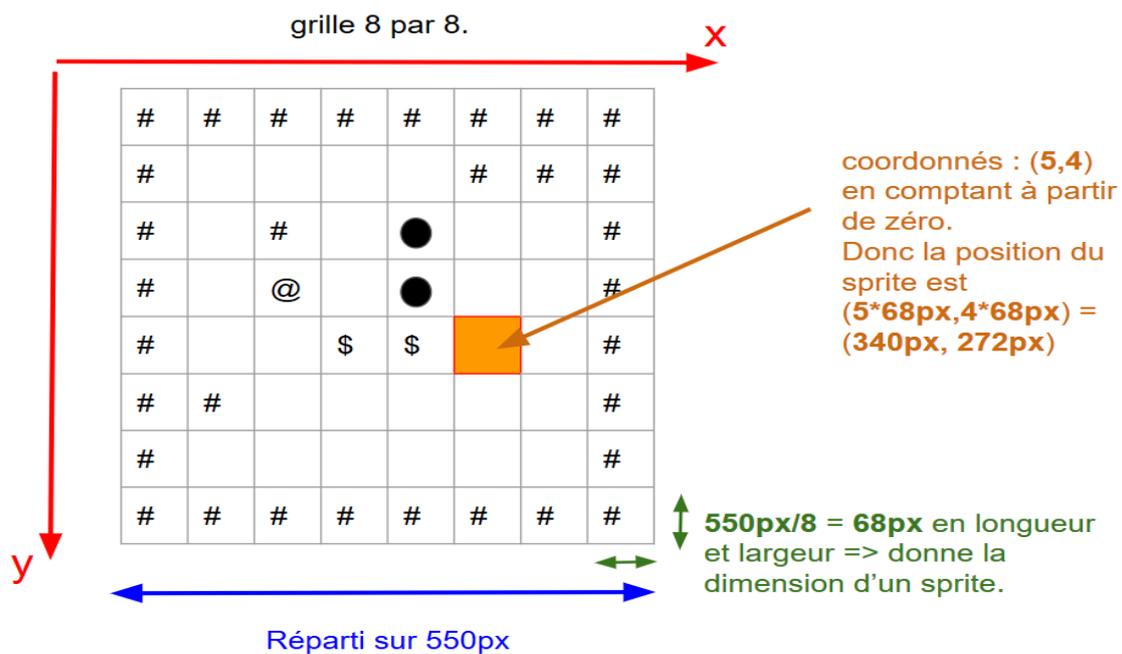


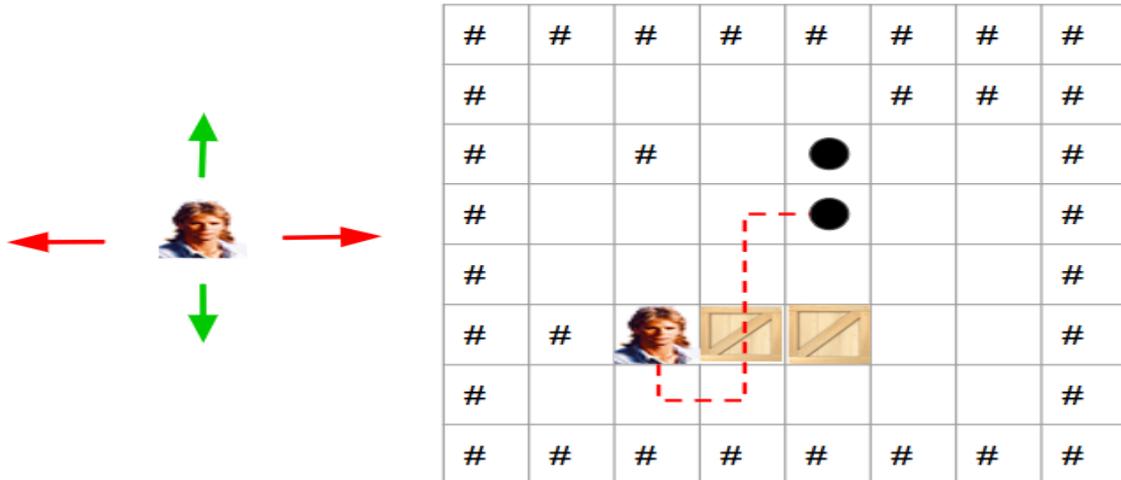
FIGURE 11 – Mesures effectuées pour chaque sprites de la grille.

Une fois que tout est en place le programme situé dans interface/screen.py peut afficher le rendu des sprites à partir de leur positions calculés.

3.2.3 Mouvement du personnage dans la grille

Le personnage se déplace uniquement de une case, en verticalement ou horizontalement. Les mouvements du personnage sont restreints par des règles.

Voici un contexte, représenté par le schéma suivant :



Légendes :

- Mouvement impossible
- Le mouvement de une case est possible
- Chemin possible pour mettre une boîte dans un trou

FIGURE 12 – Les mouvements possibles du personnage dans la grille.

La logique liée au mouvement respecte le pseudo-code suivant :

Fonction Move_Player(player, direction)

```

si la nouvelle coordonnée du personnage est dans la grille
  sprite_target = sprite visé par le déplacement du personnage
  si sprite_target est un sprite superposable
    on essaye le mouvement de la boîte se trouvant sur sprite_target avec la fonction move_box(sprite_target, direction)
    si la boîte se trouvant sur sprite_target ne peut pas se déplacer
      retourner False
    si il y a pas de boîte se trouvant sur sprite_target ou si la boîte a pu se déplacer correctement
      On met à jour les informations du personnage
      Puis on met à jour la position du sprite du personnage
      retourner True
  
```

Fonction Move_Box(sprite_target, direction)

```

  boite = sprite_target
  si la nouvelle coordonnée de boîte est dans la grille
    sprite_target = sprite visé par le déplacement de la boîte
    si sprite_target est un sprite superposable et si le sprite_target n'est pas une autre boîte
      On met à jour les informations de la boîte
      Puis on met à jour la position du sprite de la boîte
      retourner True
  retourner False
  
```

FIGURE 13 – Pseudo-code du déplacement du personnage dans la grille.

3.2.4 Solveur

Le solveur se découpe en 2 parties : l'algorithme A* qui permet de trouver le chemin le plus court en partant d'un point A pour aller à un point B et la classe solveur en elle-même qui va essayer de résoudre le niveau.

Dans un premier temps, le solveur essaie de chercher la caisse qui se trouve la plus proche du joueur et va essayer de trouver un chemin vers le point de validation (traits gris) puis on calcule le chemin de la caisse vers le point grâce à l'algorithme A* pour vérifier que la caisse peut effectivement se placer dessus (traits verts).

S'il y arrive on va réutiliser l'algorithme A* pour déterminer le chemin que doit parcourir le personnage pour aller jusqu'à la caisse (traits jaunes). On calcule ensuite le chemin que doit parcourir le personnage pour pouvoir pousser la caisse jusqu'au point (traits rouges), et on ajoute l'emplacement de la caisse en mémoire pour que l'algorithme A* soit informé qu'il ne peut plus traverser cet emplacement.

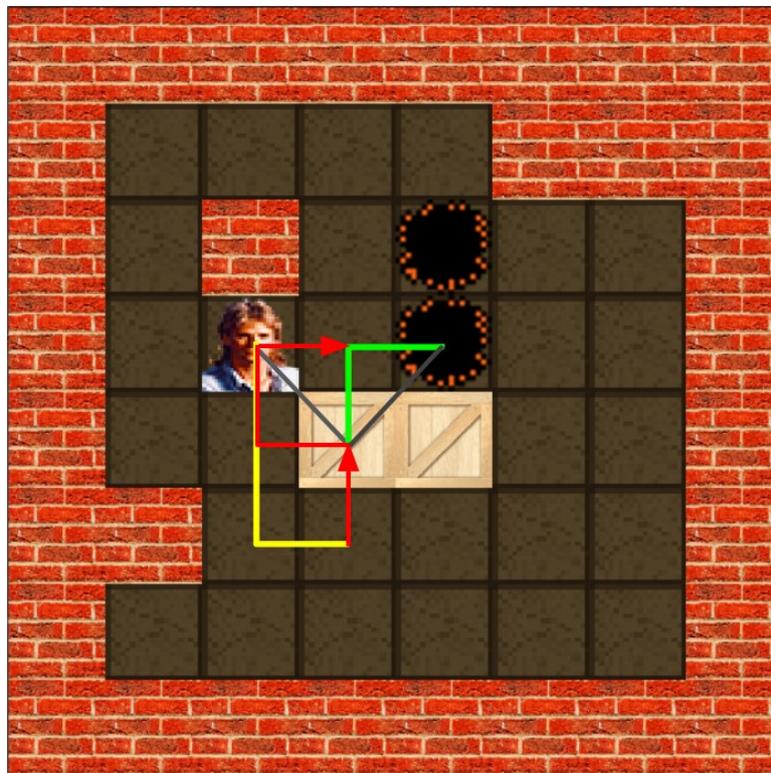


FIGURE 14 – Tout les chemins / calculs effectués par le solveur pour placer la première caisse

En effet, l'algorithme ne change pas la disposition du terrain au fur et à mesure de son exécution afin que ce changement se fasse en temps réel, quand le joueur joue et se déplace. Nous stockons donc le terrain de base ainsi que 2 listes de Tuples qui servent à dire à l'algorithme A* qu'il peut traverser une caisse (étant donné que nous ne changeons pas la disposition du terrain s'il bouge la caisse il peut à nouveau se poser sur cet emplacement) et l'autre qui sert au contraire à dire à l'algorithme A* qu'il ne peut plus traverser tel ou tel emplacement.

Dès que la caisse se trouve à l'emplacement voulu on cherche à nouveau la caisse la plus proche et le point de validation le plus proche de cette caisse (traits gris), on calcule le chemin de la caisse (traits verts). On approche ensuite le joueur de la caisse (trait jaune), on détermine le chemin à suivre grâce à l'algorithme (traits rouges) et on répète ses étapes jusqu'à ce qu'il ne reste plus un seul point de validation valide.

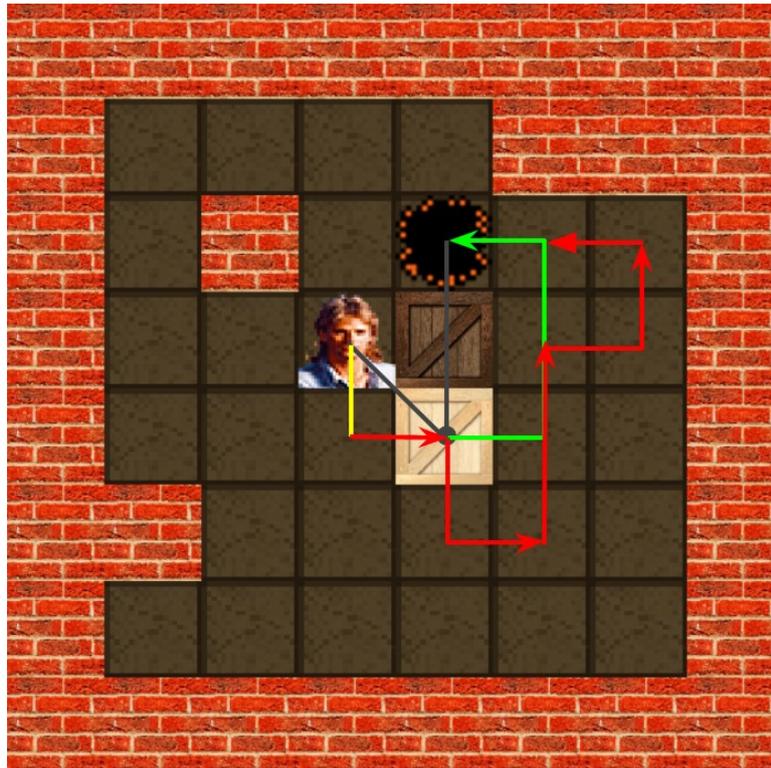


FIGURE 15 – Calculs effectués pour déplacer la seconde caisse

function SOLVEUR(grid, From :Node, Goal :Node)

```

    closedList = []
    openList = []
    came_from = {}
    openList.add(From)
    while openList is not empty do
        current_node = openList.popTheSmallestItem()
        if current_node['x'] == NoeudArrive['x'] then
            return constructPath()
        end if
        for each current_node neighbor ∈ grid do
            if not(neighbor ∈ closedList || (neighbor ∈ openList & neighbor.cost ≤
getNeighborFromOpenList().cost)) then
                neighbor.cost = current_node["cost"] + 1
                neighbor.heuristic = neighbor.cost * (|neighbor.x - Goal.x| + |neighbor.y -
Goal.y|)
                current_node.append(neighbor)
            end if
        end for
        closedList.append(current_node)
    end while
    Raise Exception
end function

```

▷ File prioritaire

FIGURE 16 – Algorithme A*

3.2.5 Générateur

Pour éviter de rendre le jeu lassant et augmenter sa durée de vie, il nous est venu comme idée d'implémenter un générateur de niveaux. Il sera expliqué dans cette section le fonctionnement du générateur ainsi que ses particularités.

Le générateur peut se configurer de plusieurs manières. On peut ainsi choisir un nombre de caisses maximum et un nombre de caisses minimum, ainsi que la taille de la grille et la difficulté. Nous allons donc voir étape par étape en quoi chacun de ses paramètres intervient dans la génération d'une grille.

On notera que pour des raisons de simplicité, toutes les grilles générées sont des grilles carrées. Voici comment le générateur construit une nouvelle grille :

1. Création de la grille de base.
2. Remplissage de la grille.
3. Vérifications.

Création de la grille de base

Le générateur créé tout d'abord une grille carrée de la taille demandée. Il fait en sorte que cette grille soit seulement constituée de murs en faisant le tour ; le reste de la grille étant vide. Il initialise aussi trois dictionnaires dans lesquels seront stockées les positions du joueurs, des caisses et des trous.

Remplissage

Une fois cette grille créée, le générateur doit la remplir. Il affecte à des variables le nombre de caisses et de trous qu'il doit placer. Une variable contenant le nombre de murs est aussi créée, elle contient un nombre correspondant au à la moitié de la taille de la grille multiplié par la difficulté. Plus un niveau est difficile, plus il contient donc de murs.

Le générateur remplit ensuite la grille en excluant le deux premières lignes (**resp.** colonnes) de chaque colonnes (**resp.** colonnes). L'exclusion de ses cases permet d'éviter que les caisses ne se retrouvent accolées aux murs extérieurs de la grille (ce qui rendrait la génération des trous plus contraignante).

Le générateur utilise la fonction "*randrange*" du module "*random*" de Python pour choisir au hasard où il doit placer une caisse un mur ou un trou. Si la case choisie contient déjà un objet, le générateur en choisit une autre, et continue de choisir au hasard des cases jusqu'à ce que tous les objets soient placés.

Le générateur place dans cet ordre, le joueur, puis les murs, suivis des caisses et en dernier les trous.

Une fois placé, un objet voit sa position ajoutée dans son dictionnaire correspondant pour faciliter l'étape de vérification. On obtient donc un algorithme de ce genre pour un objet :

```

while nombred'objet ≠ 0 do
  x ← randrange(2,tailledelagrille - 3)
  y ← randrange(2,tailledelagrille - 3)
  if grille[x][y] = empty then
    grille[x][y] ← objet
    nombred'objet ← nombred'objet - 1
    DicObjet[objet] ← [x,y]
  end if
end while

```

Vérification Une fois tous les objets placés sur la grille, le générateur procède à plusieurs vérifications pour voir si la grille est conforme à certains critères. Ainsi si une case vide est entourée de murs, elle sera transformée en mur.

D'autres test sont réhibitoires. Si ils ne sont pas valides, le générateur recommence à générer une grille depuis le début (retour à la création de la grille de base).

On distingue trois de ces tests :

- Si une caisse est entourée par plus de deux murs.
- Si il n'y a pas autant de caisses que de trous.
- Si il n'y a pas exactement un joueur.
- Si il y a moins de boîtes que demandé en appel de la classe.

Pour rendre l'exécution des vérifications plus rapide, les dictionnaires contenant les positions des objets de la grille sont utilisés, pour éviter de devoir parcourir la grille afin de les trouver. Si tous ces tests renvoient la valeur "*True*", alors la grille est considérée comme valide et la classe *Generate* la retourne.

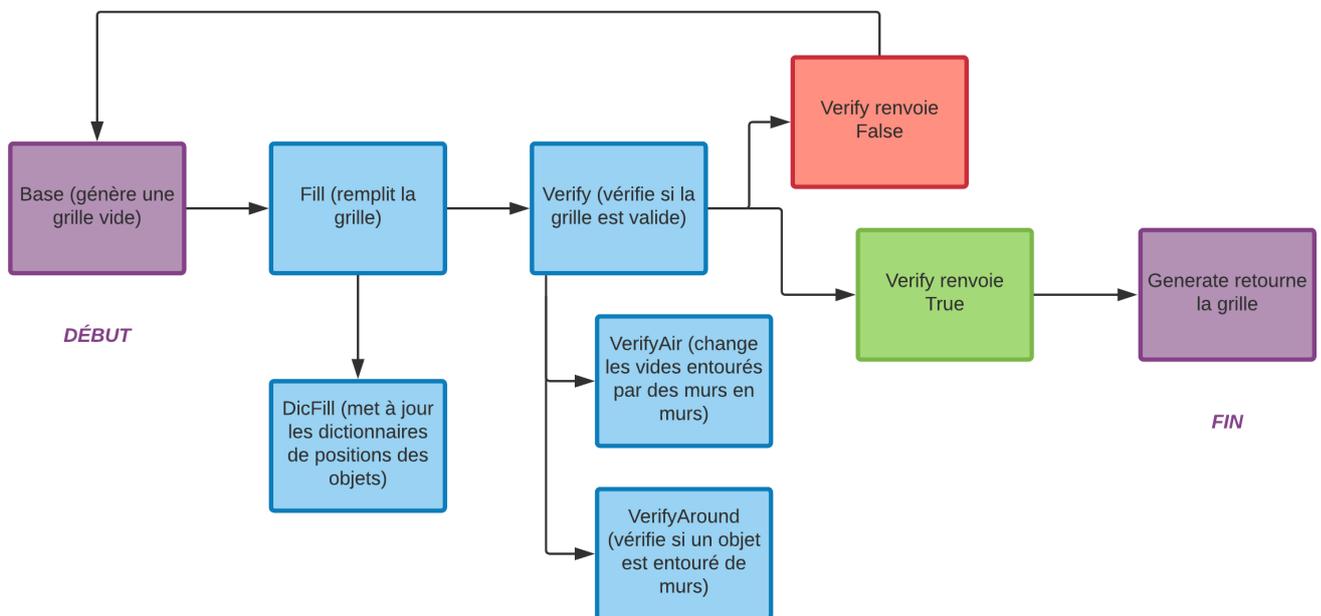


FIGURE 17 – Fonctionnement de la classe Generate.

4 Éléments techniques

4.1 Structure de données

Explication de l'organisation des dossiers :

- Config
 - Contient les constantes utiles au fonctionnement du jeu.
 - Le chemin absolu du répertoire du dossier parent afin de pouvoir récupérer facilement les fichiers .sok qui contiennent les schémas des labyrinthes.
 - Le titre et les dimensions de la fenêtre.
 - L'identifiant de notre Event pygame personnalisé.
- Images
 - Contient les sprites affichés à l'écran.
- interface
 - Contient la logique de la répartition des sprites ou des textes affichés à l'écran.
 - La logique du menu de sélection des niveaux.
- levels
 - Contient les fichiers .sok qui possèdent les schémas des labyrinthes du jeu.
- logic
 - Contient la logique de démarrage du programme.
 - Contient la logique du bon enchaînement des processus entre le menu et le jeu en lui-même, en tenant compte du type de l'adversaire dans le jeu.
 - Contient la logique du jeu en lui-même. Les déplacements possibles et la réussite du niveau.
- map
 - Contient les différents sprites (sous-classes de Pygame) qui sont affichées dans le jeu. Il y a une distinction entre les sprites qui peuvent être en mouvement (les boîtes) et les sprites statiques (les murs).
 - Contient la logique de la création de la grille de jeu.
- Solver
 - Contient la logique de l'algorithme A*.
 - Contient la logique de l'intelligence artificielle du jeu shokoban.
- Generator
 - Contient le générateur de niveaux aléatoire.
 - Contient le vérificateur permettant de valider ou non une grille générée.

L'ensemble des fichiers et leurs relations : Voir la figure 18

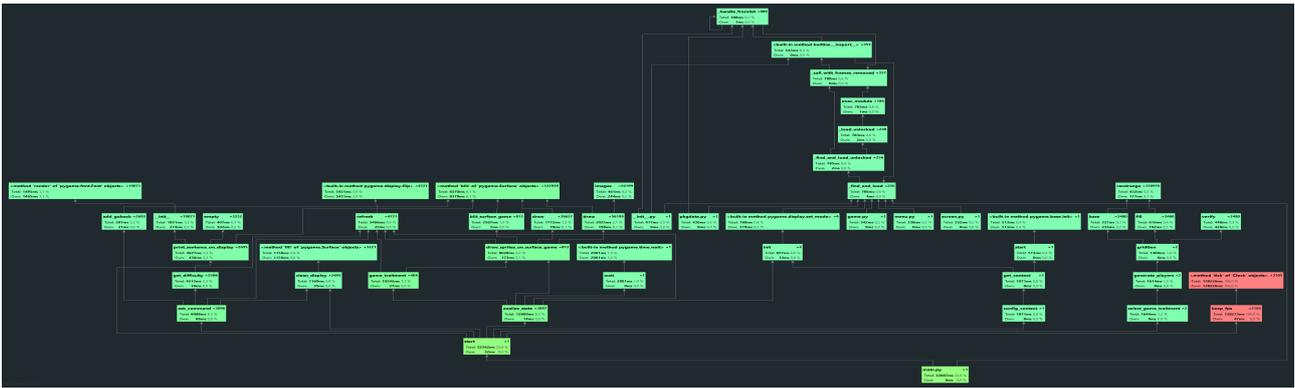


FIGURE 18 – Graphique de structure de données

4.2 Exécution

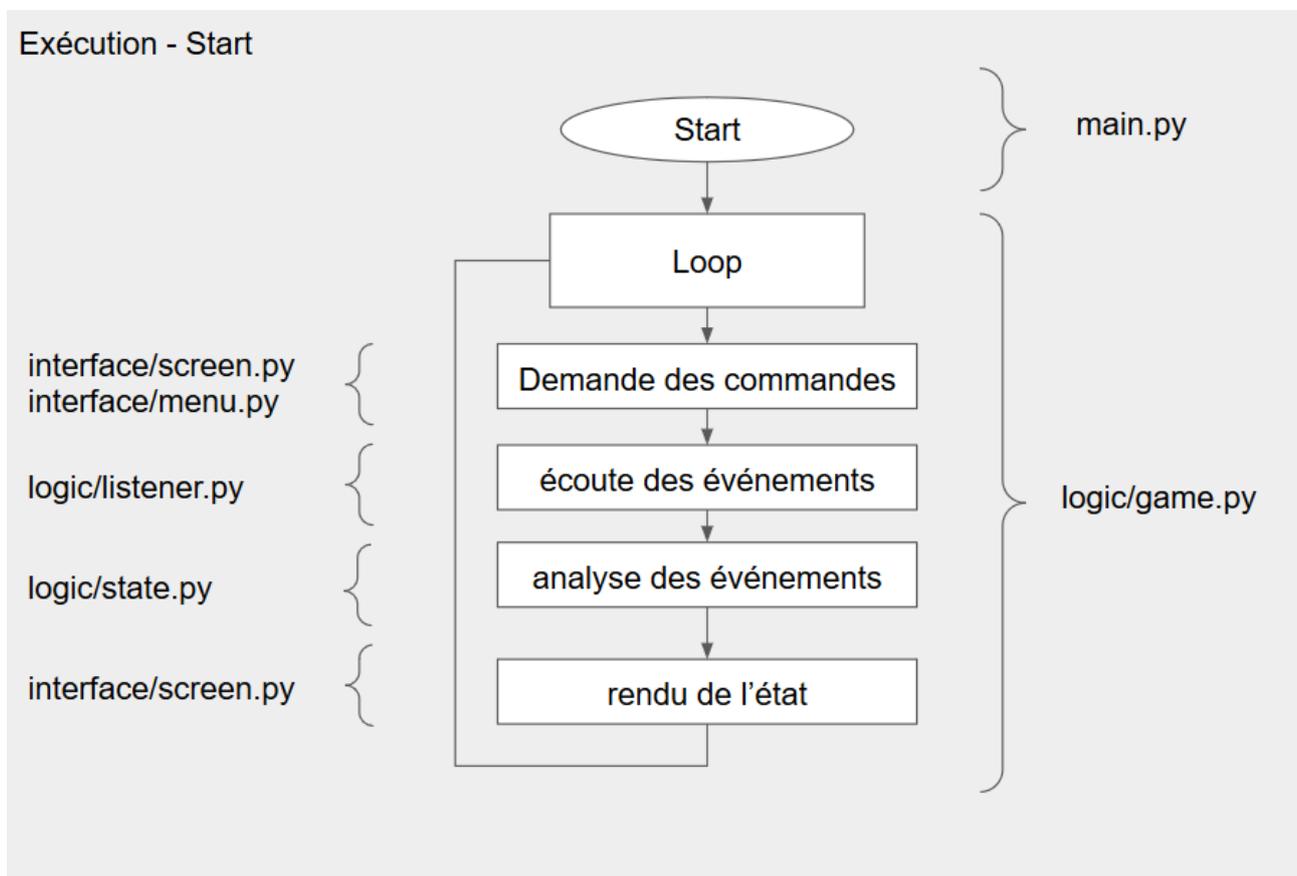


FIGURE 19 – Croquis de l'exécution du programme.

4.3 Bibliothèques utilisées

Voici la liste des bibliothèques externes utilisés :

- heapq
 - Permet de créer une file prioritaire pour l'algorithme A*
- Pygame
 - Moteur graphique utilisé dans le projet.

5 Expérimentations

5.1 Mesures de performances

Nous avons utilisé plusieurs outils pour mesurer les performances de notre programme, dans un premier temps, nous avons vérifié, quelles fonctions demandent le plus de temps de calcul grâce à un outil intégré à pyCharm.

Ce graphique (figure 18) nous permet de voir que la fonction `keep_fps` (`interface/screen.py`) nous prend beaucoup de temps de calcul. Nous avons décidé de la supprimer mais cela faisait monter énormément l'utilisation processeur par notre programme(voir figure 20).



FIGURE 20 – Utilisation du cœur après suppression de `keep_fps`

Les mesures de performances ont été effectuées en obligeant le programme à utiliser un cœur précis (le numéro 7 par exemple) du processeur afin d'être le plus précis possible.

Cela demandait 60% à 70% des ressources d'un cœur ayant une vitesse de base de 3.6 Ghz. Nous avons donc réintégré cette fonction à notre programme, et il est maintenant compliqué de constater une différence d'utilisation du processeur au moment où on lance le programme. En effet, le but de `keep_fps` était de bloquer de nombre d'images par secondes à 20, il est donc normal que son temps d'exécution soit le plus long du programme. Au niveau de l'utilisation de la mémoire vive, le programme ne dépasse pas les 35 Mo. Par conséquent, tout pc pouvant afficher un bureau est capable de lancer de manière fluide Sokoban.

5.2 Difficultés rencontrés

Le générateur de niveau et le solveur furent les parties les plus compliquées à réaliser.

Pour le générateur, il ne s'agit pas de placer des éléments de décor au hasard car il faut que chaque niveau soit réalisable que ce soit par le joueur que par le solveur ainsi que le personnage ou les caisses n'apparaissent pas dans un endroit où ils seraient bloqués. Pour plus d'informations sur son fonctionnement, reférez-vous à la partie dédiée : (3.2.5).

Concernant le solveur, il fallait trouver une solution complète sans changer la disposition du niveau, nous vous en parlons en détail dans la partie 3.2.4.

6 Conclusion

6.1 Récapitulatif

Le jeu terminé, nous étions plutôt contents de nous. Nous avons en effet réussi à atteindre l'objectif fixé, mis à part quelques petits inconvénients (cf ??). Nous avons en effet réussi à produire un jeu totalement fonctionnel. Il est non seulement possible de jouer à deux mais aussi de jouer contre son propre ordinateur. Certaines fonctionnalités telles que le comptage des points ou bien la sélection possible de plusieurs niveaux a aussi permis d'ajouter beaucoup de contenu au jeu. Bien que lent pour la génération de certains niveaux aléatoires, le jeu tourne très bien. Un ordinateur peu puissant est en mesure de l'exécuter sans problèmes de performances. L'objectif principal a donc été largement atteint puisque le jeu est parfaitement fonctionnel et que différentes fonctionnalités supplémentaires ont pu être ajoutées. Ce fut donc en somme une très bonne expérience qui nous a permis à tous de mettre nos connaissances en commun, et de progresser aussi bien individuellement que collectivement.

6.2 Améliorations possibles

Nous allons répertorier ici les améliorations auxquelles nous avons pensé mais qui n'ont pas pu se réaliser, notamment par manque de moyens.

- Un solveur plus efficace.

Le solveur que nous avons conçu prend en charge un certain nombre de niveaux mais ne peut pas tous les résoudre. Nous avons fait en sorte que tous les niveaux du jeu soient résolubles pour pallier à ce problème. Malgré tout, il aurait été appréciable que le solveur puisse être en mesure de résoudre n'importe quel niveau qui lui soit présenté. De plus il est possible sur certains niveaux (comme le niveau 2) que le joueur réussisse à faire moins de déplacement que le solveur, il y a donc quelques améliorations possibles.

- Un générateur plus rapide

Le générateur quand à lui peut générer n'importe quelle grille jouable en suivant certains paramètres. Le problème réside dans le temps qu'il lui faut pour arriver à créer une grille jouable. On pourrait éviter ce problème en générant une grille suivant des patterns (motif prédéfinis de grille). Ainsi une grille se générerait plus rapidement mais ne serait plus vraiment aléatoire puisque le fait d'ajouter des patterns sous-entend que ceux-ci ont déjà été défini par avance. De plus, on pourrait se retrouver en présence de grilles lassantes puisqu'un même pattern pourrait apparaître plusieurs fois. Une connaissance plus poussée de Python aurait donc sûrement pu permettre d'optimiser le générateur.