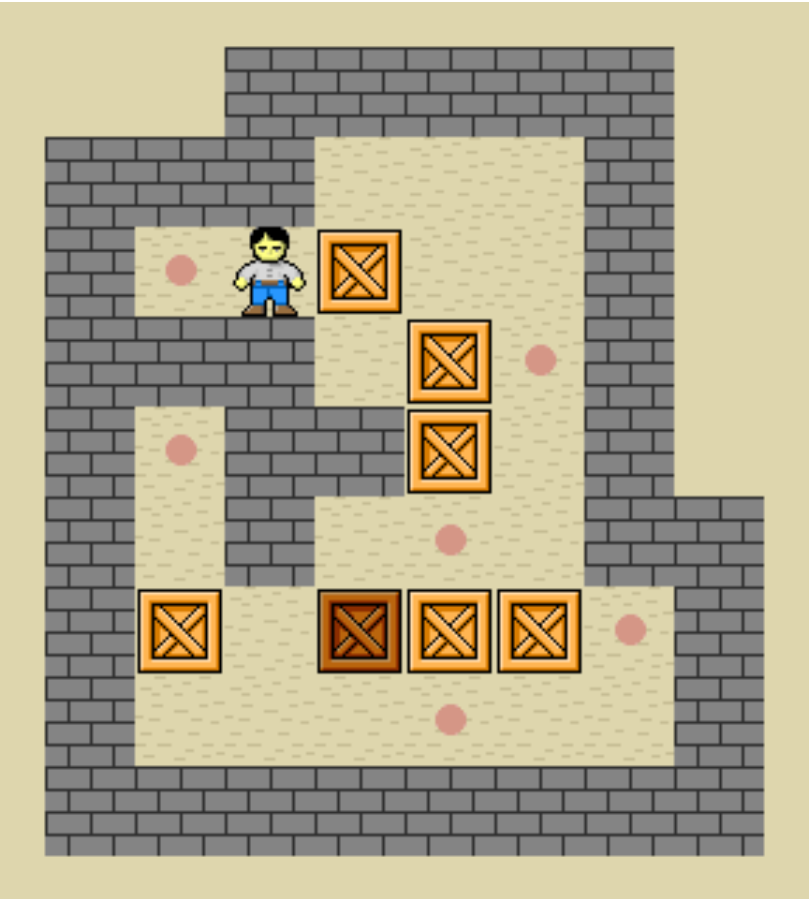
ECE
PARIS ÉCOLE D'INGÉNIEURS

Projet d'informatique

Théorie des Graphes



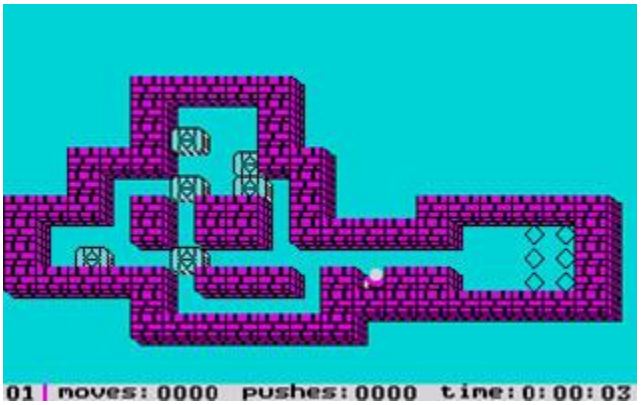
Julien Kuntz – Fabien Roussel

ING 2 – TD10

Table des matières

I. Présentation

Notre projet s'inspire d'un célèbre jeu vidéo de puzzle inventé au Japon : Sokoban. C'est un jeu de type « pousse-bloc » qui fait partie des problèmes de décision des plus complexes à résoudre pour un programme. Le jeu à développer est codé en C++ mais utilise la librairie allegro. Le jeu sera composé de 6 niveaux de difficulté croissante.



Ecran en jeu du jeu original Sokoban

NB : La partie programmation n'étant pas encore complètement terminée, nous ne parlerons que de ce qui a été réalisé et testé.

1) Objectif et règles du jeu

Les premiers niveaux du jeu sont d'une simplicité déconcertante cependant, pour un programme, la difficulté est bien plus forte. L'exemple donné dans notre cahier des charges le montre bien : « la taille de l'arbre de recherche pour une grille de 20x20 (taille modérée) est de l'ordre de 10^{98} » ce qui est considérable. L'objectif du jeu est de pousser toutes les caisses présentes sur le terrain sur leur point d'arrivé (un point vert par caisse). Mais le joueur devra faire face à des murs qui l'empêcheront de passer. Il ne pourra pas non plus traverser les caisses, en revanche il pourra les pousser sur le sol (pas sur un mur). Le joueur ne pourra pousser que les caisses et une seule à la fois. Le niveau se termine quand toutes les caisses sont sur un leur point d'arrivé (goal). Une case qui se trouve sur un goal est présentée en rouge. Une case blanche (sol) est une case sur laquelle on peut pousser une caisse ou que le joueur peut se déplacer. De même, les goals peuvent être traversés dans les mêmes conditions que les cases blanches.

Les touches pour que l'utilisateur déplace Mario seront :

- Flèche haut, bas, gauche, droite

Il n'y aura donc que quatre directions possibles (pas de diagonal).

Notre objectif est de créer un programme qui résoudra automatiquement les niveaux. Pour cela nous avons programmé le jeu à l'aide du code fourni par M. Diedler.

2) Ressources utilisées

Ce projet a été développé en langage C++ via l'application CodeBlocks.

Dans l'optique de partager facilement notre code ainsi que de conserver une sauvegarde de notre projet, nous avons utilisé un outil ainsi qu'une application permettant le versionning : Source Tree et BitBucket.

Le planning prévisionnel a été composé via Gantt Project.

Comme nous l'avons indiqué plus haut, nous avons utilisé le code de M. Diedler. Nous nous sommes aussi inspirés des algorithmes du cours mais nous avons préféré les recoder entièrement par nous-même.

Nous avons aussi utilisé la fonction gotoligcol de M. Diedler. Cette fonction se trouve dans le fichier console.h notre code.

FONCTION OU ALGORITHME	UTILITE	LIEN VERS LA SOURCE
gotoligcol	Aller à une certaine ligne et colonne	http://campus.ece.fr/course/view.php?id=174
New_sokoban_ solver	Jeu ou casse-tête utilisant allegro	http://campus.ece.fr/course/view.php?id=174

De nombreuses fonctions ont été trouvées grâce au site http://www.cplusplus.com/reference/

II. Phase Conception

1) Organisation de l'équipe

Afin de réaliser le projet efficacement, nous nous sommes rapidement répartis les tâches. Le binôme a été construit selon notre niveau, Julien Kuntz, ING2 nouveau et Fabien Roussel, ex-ING1.

Cependant, nous avions les mêmes bases en C++ et nous avions assisté aux mêmes cours tout au long de l'année. Nous sommes donc tous les deux tout aussi polyvalent, ce qui nous a permis de travailler séparément au début et de rapidement mettre en commun ce que nous avions fait. Le projet n'est pas encore fini puisque nous devions rendre ce rapport avant notre projet. Nous ne pouvons donc vous dire précisément comment nous allons organiser la fin de notre projet mais le planning prévisionnel est actuellement respecté ce qui peut donner une idée

de ce qu'il reste à finir. Nous avons donné le maximum de nous-même pour rendre ce projet le plus performant possible.

Avant de continuer la programmation, après avoir codé le déplacement du joueur, nous avons réalisé le diagramme de classe de notre projet (joint plus loin dans le rapport) puis nous nous sommes répartis les tâches.

Julien Kuntz :

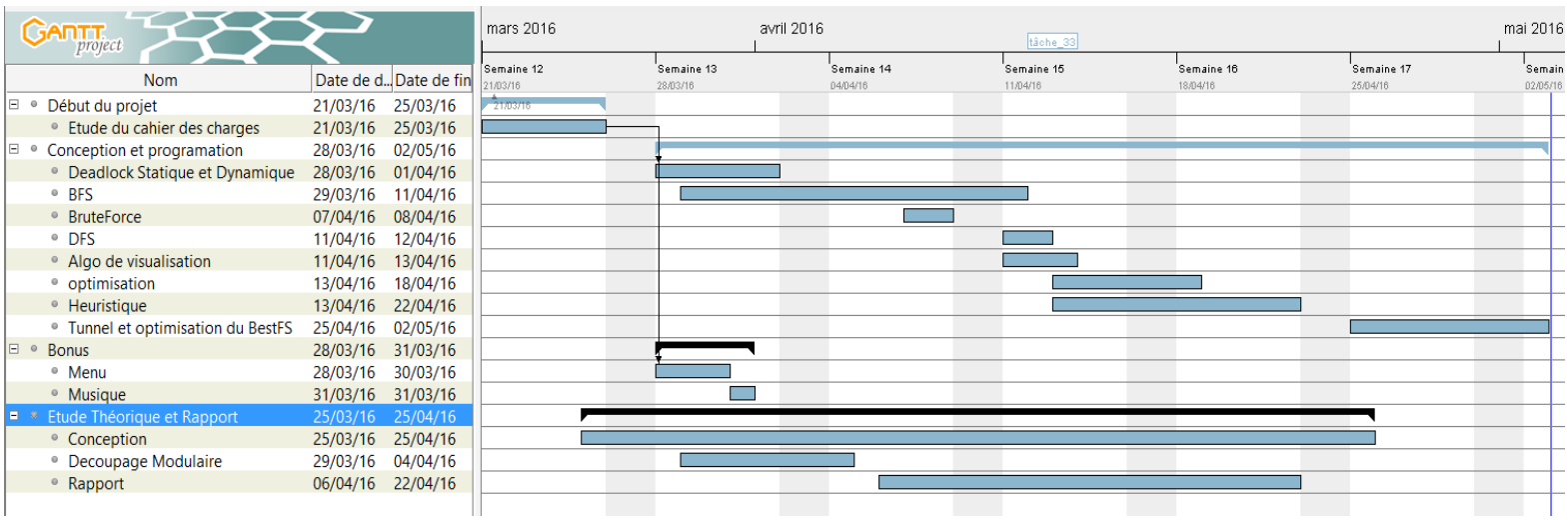
Je vais faire une partie de l'analyse heuristique et le niveau 3 avancé. Je coderai aussi un moteur 3D.

Fabien Roussel :

Quant à moi, j'ai d'abord travaillé sur le jeu de base. Je l'ai terminé (déplacement, du joueur, poussage des caisses etc.) et j'ai ensuite travaillé sur le menu afin de pouvoir accéder rapidement à un certain niveau lorsque nous testerons notre intelligence artificielle.

Avant de commencer tout code, j'ai pensé les algorithmes et écrit sur papier leur ACD. Ensuite je les ai codés. J'ai d'abord codé les deadlocks car c'était l'un des programmes qui allait me resservir tout au long du projet. Je l'ai codé de deux façons. Un qui ne s'actualise pas, et un qui, à chaque fois qu'une action est réalisée, actualise l'ensemble des deadLocks Ensuite j'ai réalisé le calcul de la zone Accessible du Joueur bruteForce, le BFS puis le DFS. J'ai ensuite dressé un premier point afin de commencer ce rapport. Puis j'ai commencé l'analyse heuristique. J'ai aussi fait l'intégralité du rapport

2) Planning prévisionnel et diagramme de Pert



Nom	Date de d...	Date de fin
Début du projet	21/03/16	25/03/16
Etude du cahier des charges	21/03/16	25/03/16
Conception et programation	28/03/16	02/05/16
Deadlock Statique et Dynamique	28/03/16	01/04/16
BFS	29/03/16	11/04/16
BruteForce	07/04/16	08/04/16
DFS	11/04/16	12/04/16
Algo de visualisation	11/04/16	13/04/16
optimisation	13/04/16	18/04/16
Heuristique	13/04/16	22/04/16
Tunnel et optimisation du BestFS	25/04/16	02/05/16
Bonus	28/03/16	31/03/16
Menu	28/03/16	30/03/16
Musique	31/03/16	31/03/16
Etude Théorique et Rapport	25/03/16	25/04/16
Conception	25/03/16	25/04/16
Decoupage Modulaire	29/03/16	04/04/16
Rapport	06/04/16	22/04/16

Pour réaliser la tâche	Durée (jours)	Tâches immédiates postérieures	Marge Totale
A Etude du cahier des charges	**3**	**/**	**0**
B Déplacement du personnage	**1**	**A**	**0**
C Conception et programmation des deadlocks statiques et dynamiques	**3**	**B**	**2**
D BFS	**5**	**B**	**0**
E Bruteforce	**2**	**B**	**3**
F DFS	**2**	**B**	**3**
G Algorithme de visualisation	**2**	**B**	**3**
H Menu	**3**	**B**	**27**
I Comparaison et optimisation	**7**	**C D E F G**	**0**
J Heuristique	**1**	**I**	**0**
K Astar	**3**	**J**	**0**
L Best First Search	**3**	**J**	**0**
M Optimisation	**3**	**K L**	**0**
N Analyse du Best FS avec les deadlock dynamique	**1**	**M**	**0**
O Tunnel et Macro-mouvement	**3**	**N**	**1**
P Zone Fermée	**2**	**N**	**0**
Q Optimisation du Best FS	**2**	**O P**	**0**
R Analyse des résultats et construction du Rapport	**5**	**Q**	**0**

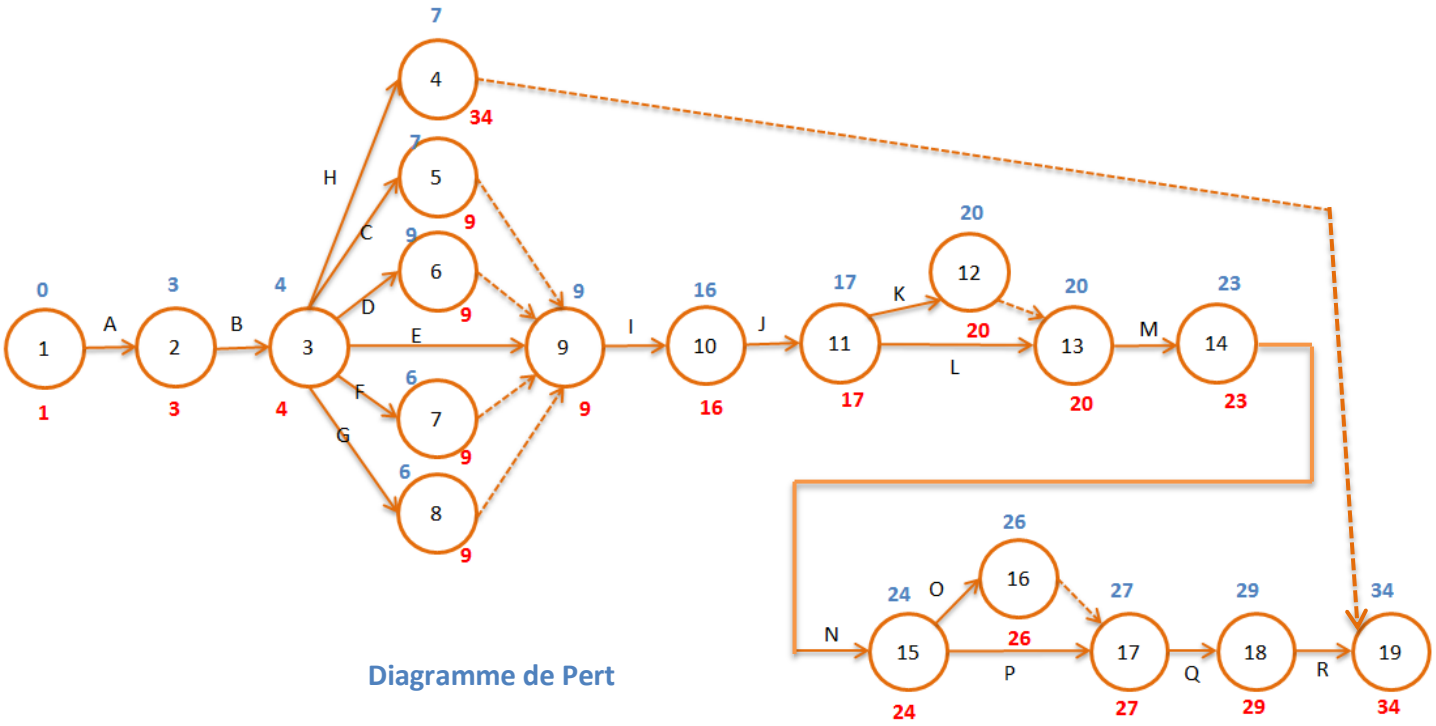


Diagramme de Pert

Le chemin critique est : **A** -> **B** -> **D** -> **I** -> **J** -> **K, L** -> **M** -> **N** -> **P** -> **Q** -> **R**

3) Diagramme de classe

Pour ce projet, nous avons décidé d'articulé notre jeu autour de la classe Maze car c'est à cet endroit que nous appliquerons nos algorithmes solveurs.

Voici, ci-dessous, une copie de notre diagramme de classe qui a été réalisée assez tôt dans le développement de notre projet.

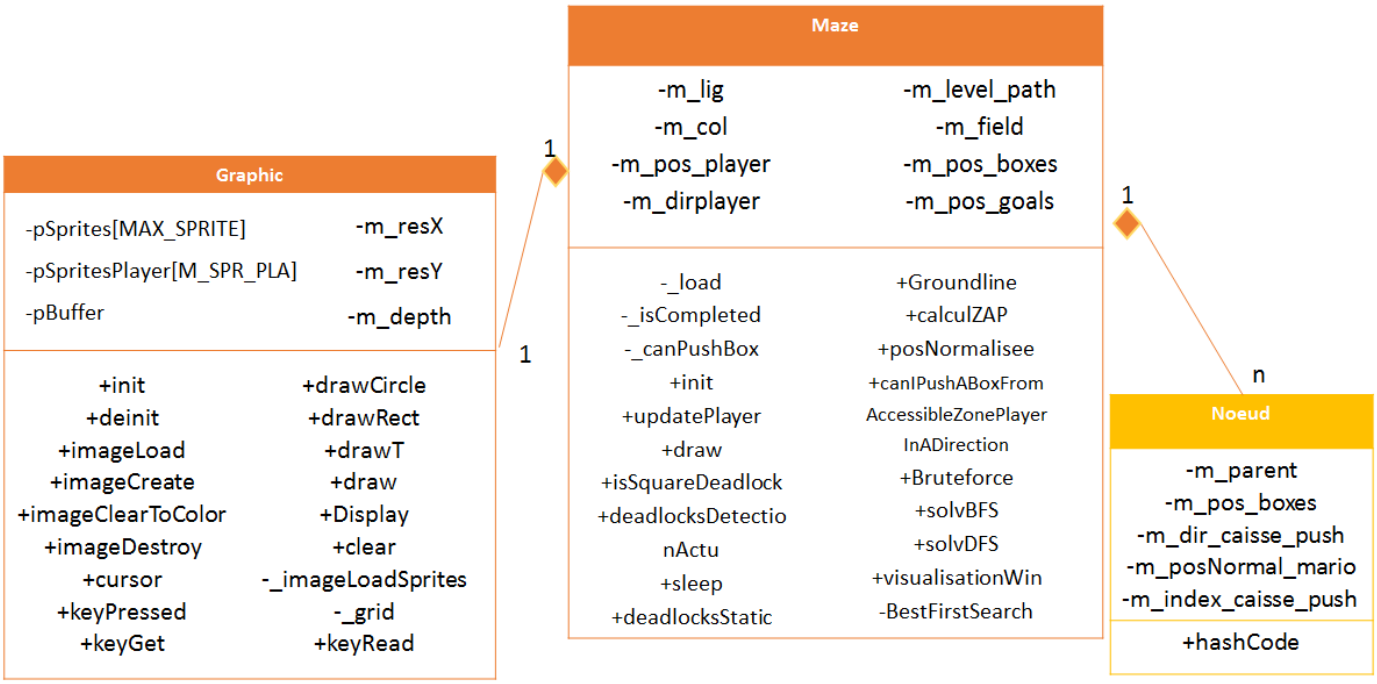


Diagramme de classe du Projet Sokoban

4) Découpage modulaire du projet

Fichiers .h	Sous-programmes	Utilité
console.h	Void gotoligcol	positionner le curseur à l'écran aux coordonnées transmises en paramètres
	void setColor	Permet de mettre de la couleur au texte affiché à l'écran
	bool isKeyboardPressed	Utile pour savoir si le joueur a tapé sur une touche
	void gotoCol	positionner le curseur à l'écran à la coordonnée transmise en paramètres
	int getInputKey	Permet de récupérer la touche saisie par l'utilisateur

Menu.h	Void musique	Permet de jouer de la musique en boucle
	Bool inGame	Boucle de jeu dans laquelle on appelle
	Void menu	Affiche le menu à l'écran
	void chargement_niveau	Afin de charger le niveau auquel veut jouer l'utilisateur
Maze.h	std::vector<char> Bruteforce	Solveur Bruteforce qui permet de résoudre certains des niveaux tests
	void solvBFS	Solveur BFS qui permet de résoudre tous les niveaux easy
	void solvDFS	Solveur DFS qui permet de résoudre tous les niveaux easy
	void visualisationWin	Permet de visualiser à l'écran la solution du niveau après appel d'un solveur
	std::unordered_set<short unsigned> calculZAP	Permet de calculer la zone accessible par le joueur
	unsigned short posNormalisee	Détermine la position normalisée de Mario à partir de sa Zone Accessible
	Bool canIPushABoxFromAccessibleZonePlayerInADirection	Booléen permettant de savoir si une caisse peut être poussé à partir d'une zone accessible de Mario
	void sleep	Sleep (petit délai entre deux actions) recoder avec clock_t
	bool isSquareDeadlock	Permet de savoir si la case visée par une caisse est une deadlock
	void deadlocksStatic	Identifie toutes les cases mortes en début de niveau
	Bool updatePlayer	Actualise la position du joueur en fonction de la saisie utilisateur et de son ancienne position
Graphic.h	void drawCircle	Permet de dessiner un cercle avec une certaine couleur, un rayon, une position
	void clear	L'écran redevient blanc (efface l'écran)
	void display	Afficher
	void drawRect	Permet de dessiner un carré avec une certaine couleur, un rayon, une position
	int keyPressed	Détecte si une touche est appuyée
State.h	Size_t hashCode	Permet de concaténer toutes les données du nœud dans un size_t

Tableau listant les principaux fichiers .h associé aux prototypes des sous-programmes qu'ils contiennent, et leur utilité

Ce tableau présente les Entrées/Sorties des principaux sous-programmes du projet :

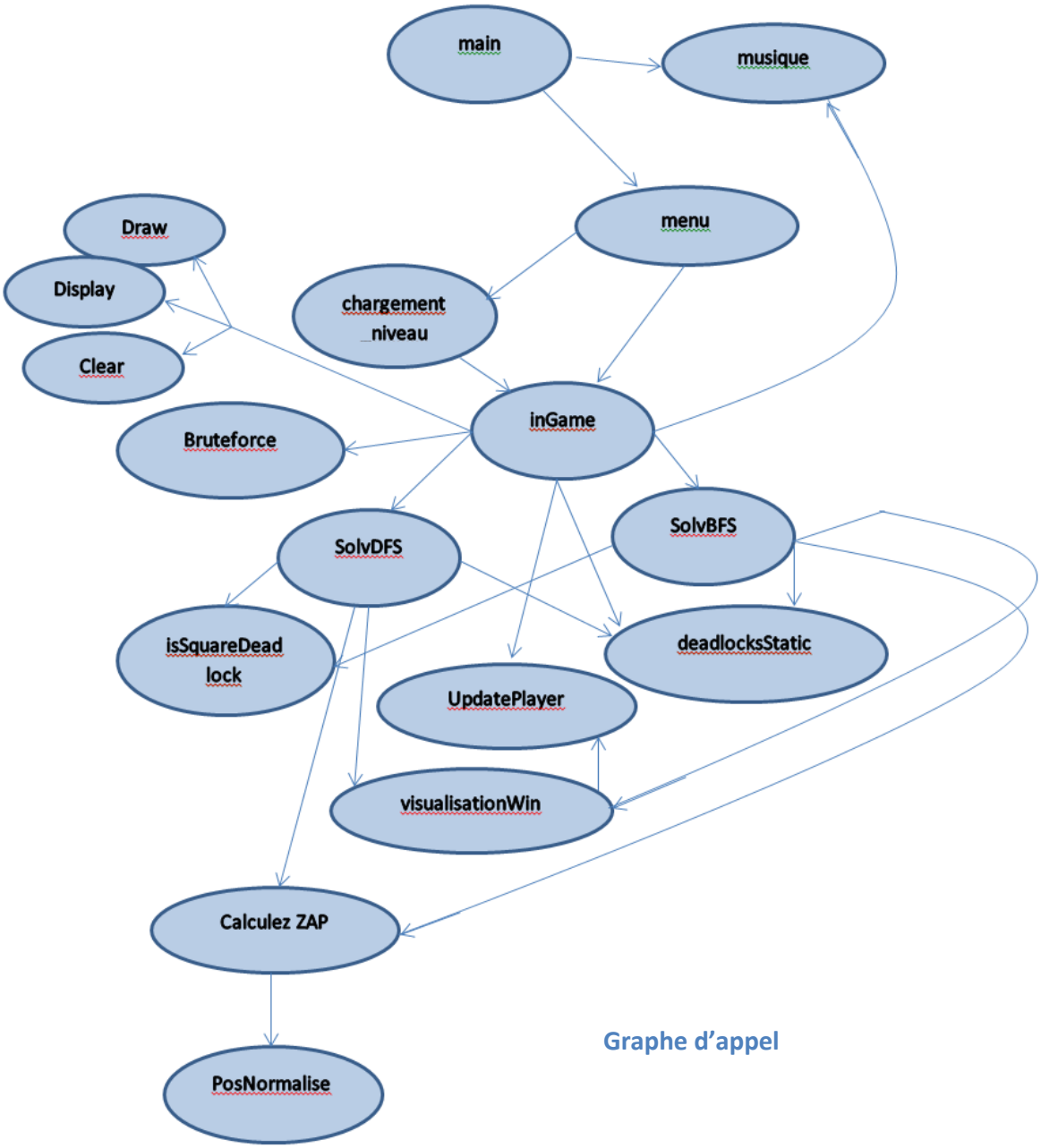
Sous-programmes	Entrées internes	Sorties internes	Entrées externes	Sorties externes
void gotoligcol	int lig, int col	/	/	affichage du curseur aux coordonnées ligne, colonne
std::vector<char> Bruteforce	bool win,std::vector<char> & direction	std::vector<char> directions	/	/
void solvBFS	/	/	/	/
void solvDFS	/	/	/	/
void visualisationWin	Nœud* lastNoeud	/	/	Affiche à l'écran la solution d'un solveur
bool isSquareDeadlock	short unsigned posBox, short unsigned dir	Bool isSquareDL	/	/
void deadlocksStatic	/	/	/	Affiche à l'écran les cases mortes (en rouge)
std::unordered_set< short unsigned> calculZAP	short unsigned posPlayer, const std::vector<unsigned short>& PosBoxes	std::unordered_set < short unsigned> ZAP	/	/
bool canIPushABoxFromAccessibleZonePlayerInADirection	unsigned short int boxPos, std::unordered_set<unsigned short>& ZAP, char dir, const std::vector<unsigned short>& pos_boxes	Bool isPushable	/	/
unsigned short posNormalisee	std::unordered_set<short unsigned>& ZAP	unsigned short posNorm	/	/
void musique	/	/	/	Une musique est jouée en fond
bool inGame	std::string path, Graphic g	Bool win		Affiche le labyrinthe
bool updatePlayer	char dir	Bool win	/	Modifie la position du joueur

III. Phase Réalisation

1) Choix de programmation

Pour déplacer Mario l'utilisateur peut se servir des flèches.

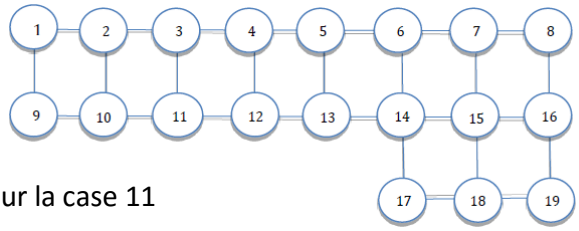
Afin de résumer les interactions entre les différents sous-programmes, nous avons créé un avec un graphe d'appel simplifié :



Graphe d'appel

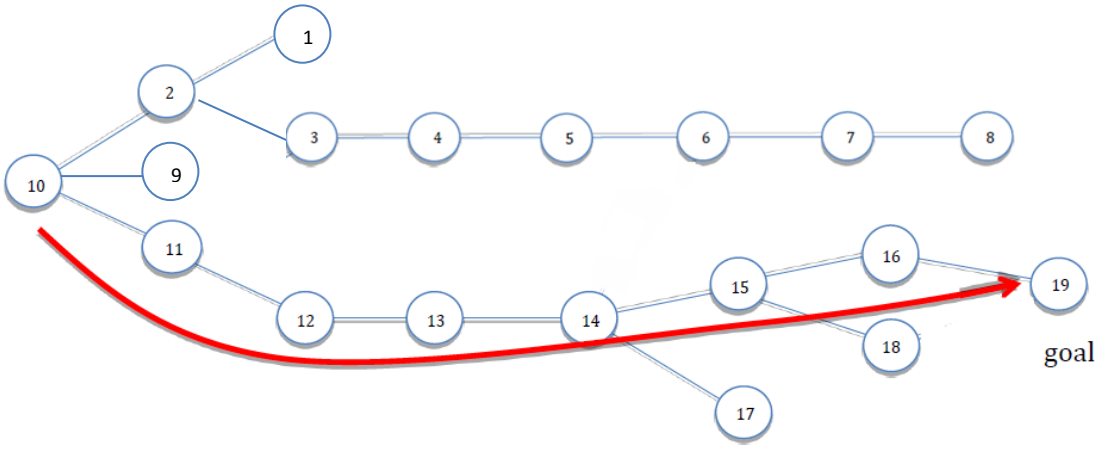
En ce qui concerne les conteneurs de la STL, nous avons choisi d'utiliser :

- Les vectors en tant que simple tableau car ils sont bien pratiques que des int* ou autres et possèdent de nombreuses fonctions toutes faites. Dans notre code, ils sont utilisés pour stocker des directions, des positions de caisses, etc.
- Les unordered_map, ou table de hachage, ont été mises à profit dans notre BFS et notre DFS car elles permettent de garder un temps d'accès constant quelque soit le nombre d'entrées de la table. Elles sont surtout utile pour éviter les doublons notamment grâce à une fonction de hachage (dans la classe Nœud*)
- Les unordered_set sont très utiles pour stocker seulement des données comme des tableaux de marquage. Par exemple, pour notre zone accessible du joueur (ZAP) nous avons utilisé ce conteneur pour marquer les positions auxquels Mario peut accéder car il est peu couteux en mémoire est simple d'utilisation.
- Nous utilisons aussi des piles et des files pour respectivement le DFS et BFS. Une pile sera aussi implémentée pour stocker les directions au lieu d'un vector d'ici la fin du projet.



Voici l'utilisation d'une file (queue) pour une BFS pour le niveau 3 des tests. Le goal se trouve sur la case 19 et la caisse sur la case 11

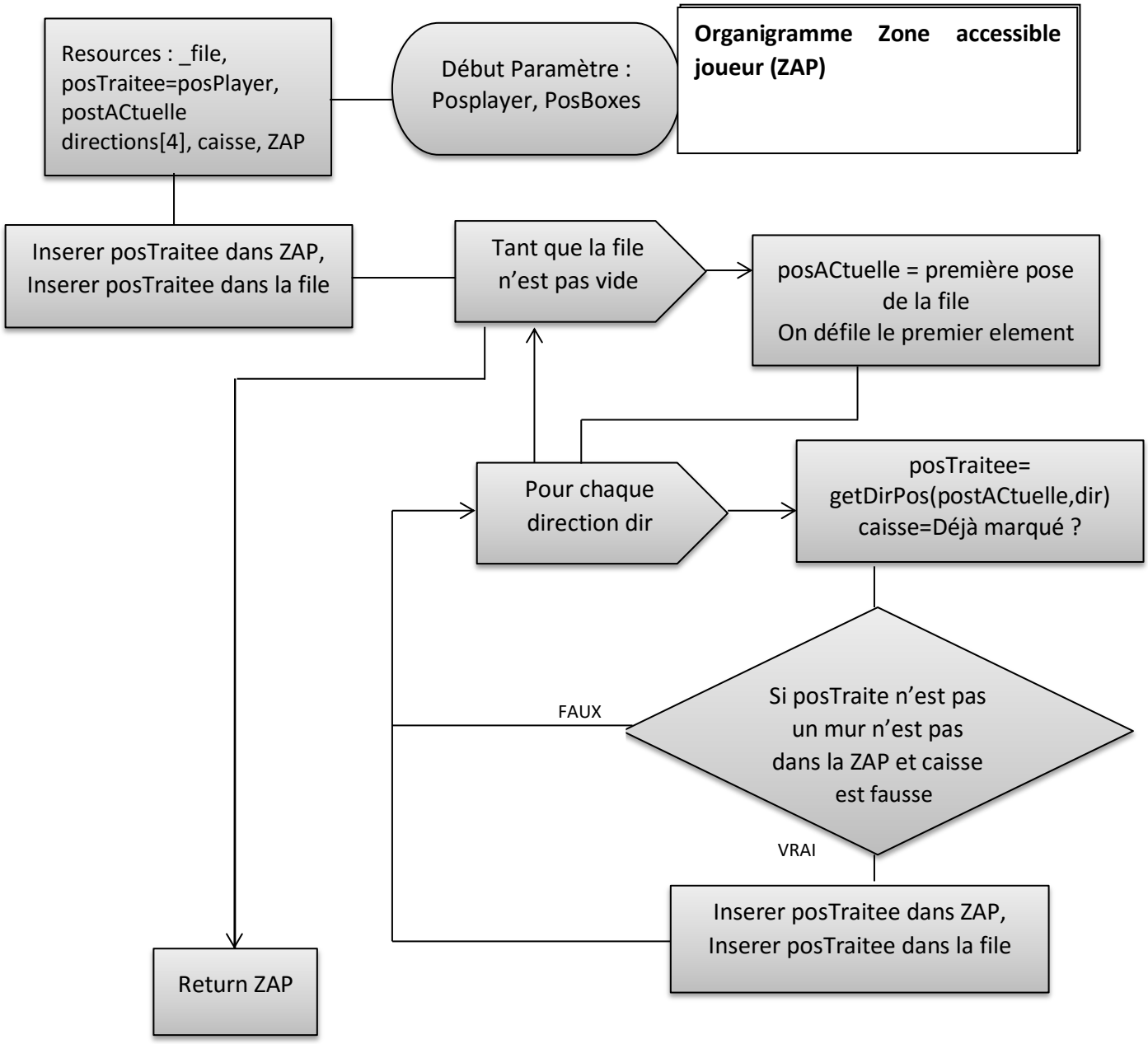
Fonctionnement du BFS en fonction de la poussée de la boite :



2) Algorithme de Théorie des Graphes et leurs scénarii

Afin de représenter différents sous-programme, nous avons opté pour des organigrammes.

Nous avons décidé de représenter celui du bruteforce et celui du calcul de la zone accessible pour le joueur. Le bruteforce n'étant pas le plus pertinent nous avons décidé de le mettre en Annexe.



Mais quelle est l'utilité de la ZAP et à quoi sert-elle ?

C'est assez simple, elle permet de récupérer toute la zone ou le joueur peut avancer sans pousser de caisse. Cela permet ainsi de réduire le nombre de caisses poussables possibles quand

on réalise l'un de nos solveurs. En effet, le solveur ne s'intéressera qu'aux caisses qui sont accessibles à partir de la ZAP ce qui aura pour effet d'augmenter la vitesse d'exécution.

Afin de donner un exemple d'utilisation de ce sous-programme, voici l'algorithme du BFS

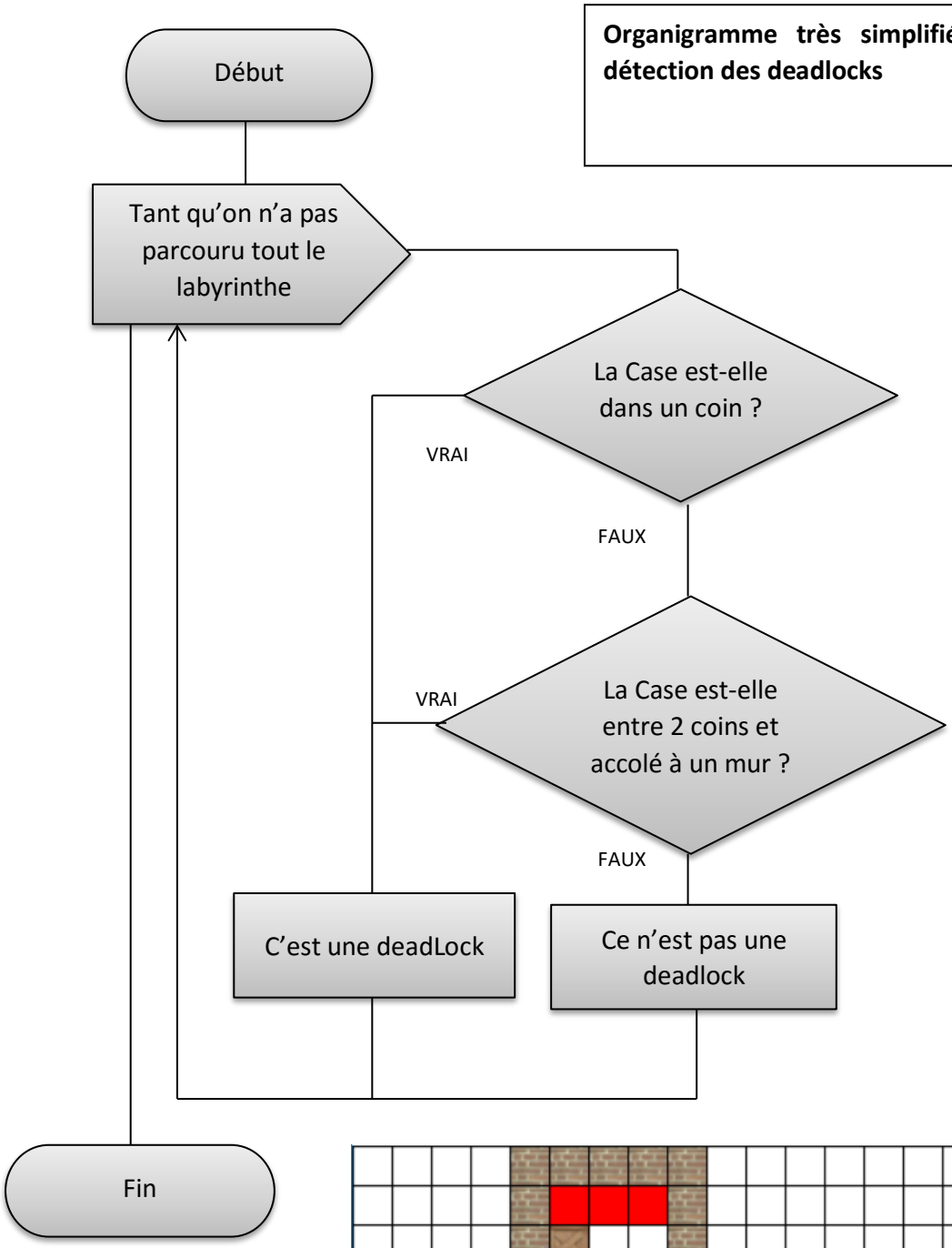
Algorithme du BFS

0. Déclaration des variables
1. Calculer la zone accessible du Joueur (ZAP) et récupérer la position normalisée
2. Créer un nœud Initial
3. Le Marquer dans la table de Hachage
4. Enfiler le nœud initial dans la file
5. Tant que la file n'est pas vide
   - 5.1 Défiler le nom courant et le stocker dans une variable : nCourant (Nœud*)
   - 5.2 Vérifier si l'état est solution, Est ce que toutes les caisses sont sur un goal ? Si oui on met le booléen win a vrai
   - 5.3 Calculer ZAP parent
   - 5.4 Pour chaque caisse (i) du Noeud courant : nCourant
     - 5.4.1 On stock la position de la box
     - 5.4.2 On associe aux caisses leurs positions parents avant d'en déplacer une
     - 5.4.3 Pour chaque direction (Haut bas gauche droite) (j)
       - 5.4.1.1 pour chaque caisse accessible grâce à la ZAP parent,
         Puis-je pousser la caisse de « i » dans la direction « j » ? Si oui
         - 5.4.1.1.1 On actualise la position de la caisse poussé dans le vector
         - 5.4.1.1.2 Recalculer la nouvelle ZAP enfant et la position normalisée
         - 5.4.1.1.3 Créer un nouvel état (Nœud*) : newStade avec les données ci-dessus
         - 5.4.1.1.4 On détermine sa valeur de hachage
         - 5.4.1.1.5 Si l'état n'est pas un doublon (table de Hachage)
           - 5.4.1.1.1.1 Enfiler newStade
           - 5.4.1.1.1.2 L'ajouter dans la table de Hachage
         - 5.4.1.1.1 Sinon on supprime newstade
     - 5.4.4 Réitération de 5.1 jusqu'ici
   - 5.5 Appeler le sous-programme d'affichage de la solution trouvé

Algorithme du DFS

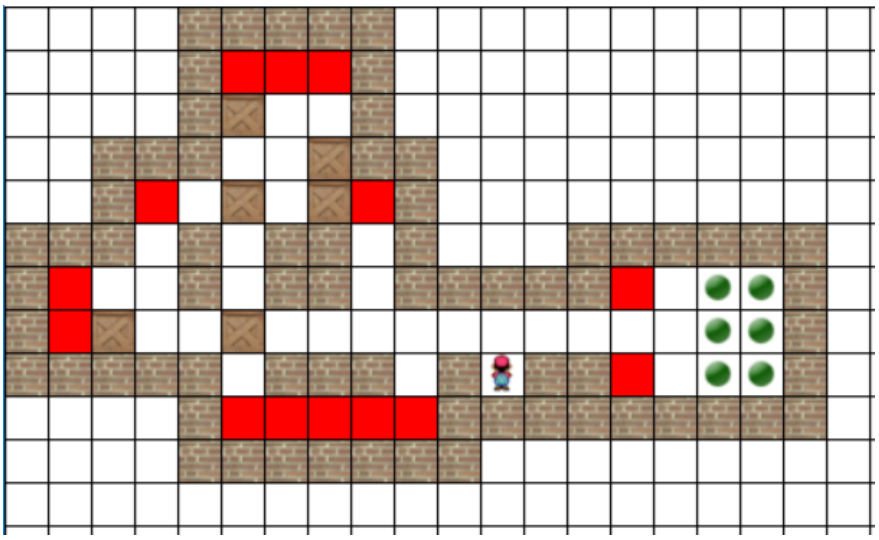
La logique est exactement la même avec pour seul exception qu'au lieu d'utiliser un file (queue) on utilise une pile (stack) et donc on dépilera au lieu de défiler.

Organigramme très simplifié de détection des deadlocks

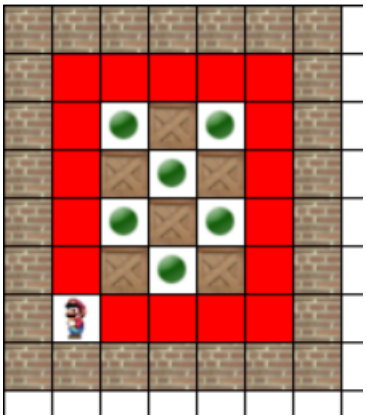


Afin de montrer l'emplacement des deadlocks, le joueur peut appuyer sur la touche K

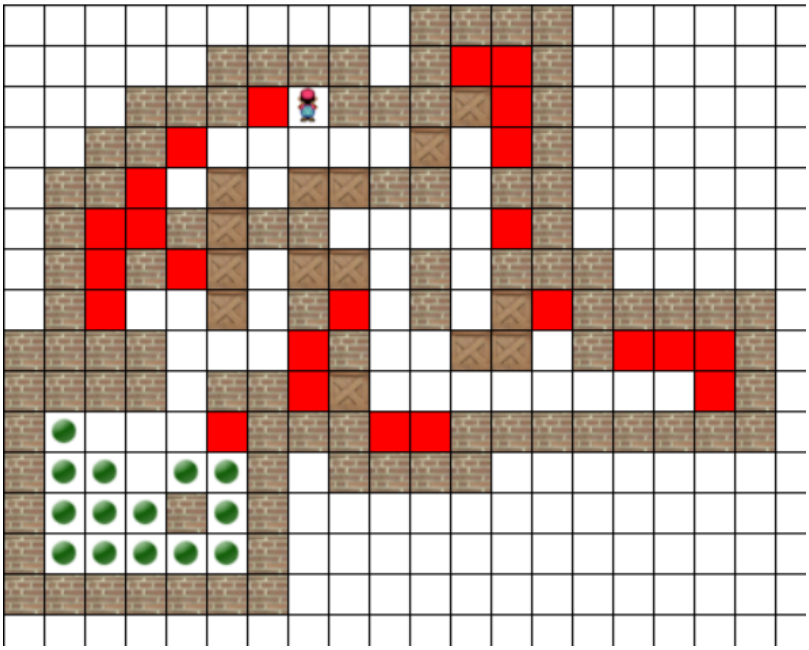
Voici plusieurs où on détecte les deadlocks. D'un rapide coup d'œil on voit que toutes les deadlocks sont détectées.



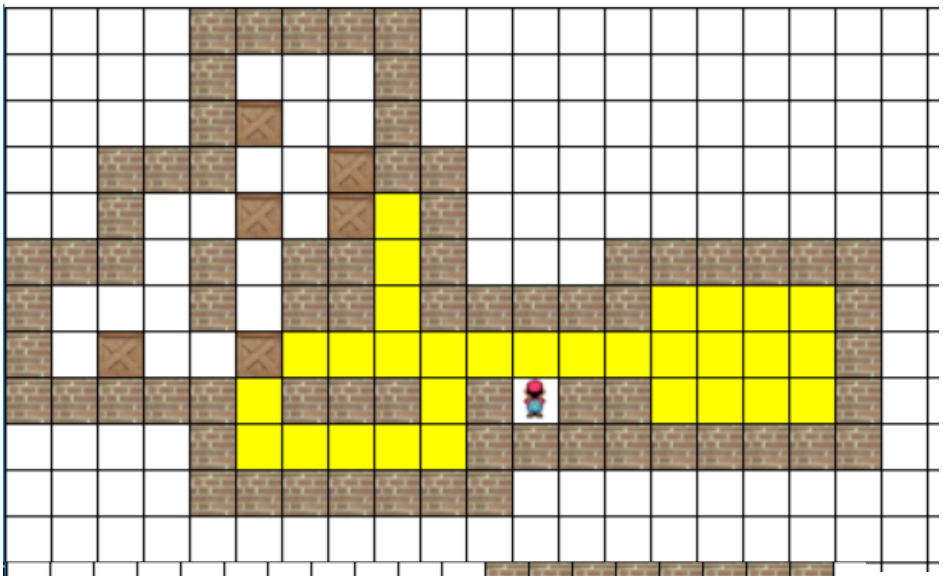
Niveau Médium 1



Niveau Esay 7

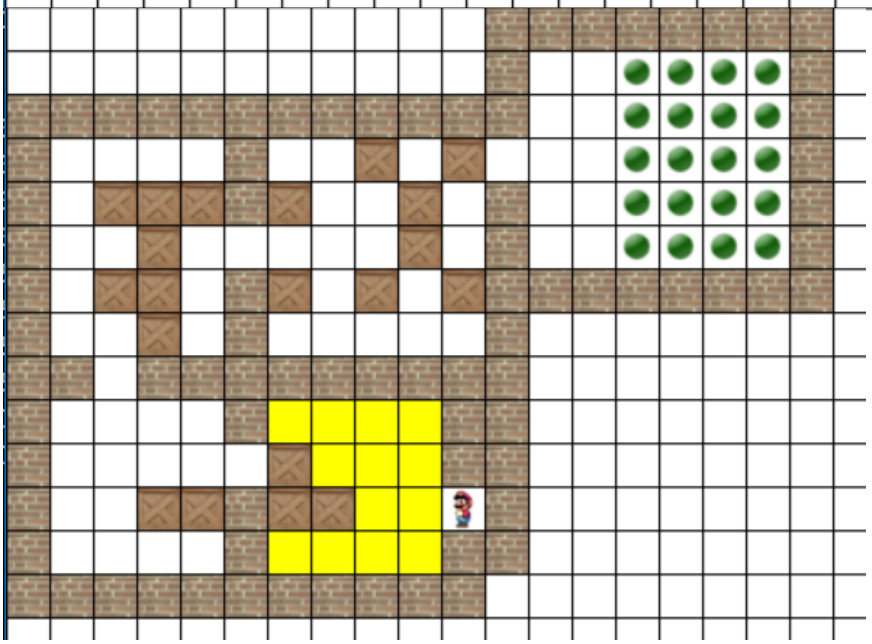


Niveau Hard 2



Niveau M 1

En ce qui concerne la zone accessible du joueur, on peut la visualiser en jaune ci-dessous. Les goals sont aussi colorier en jaune car ils sont accessibles



Niveau H 1

Après avoir réalisé ces algorithmes, il faut pouvoir visualiser la solution.

Voici la méthode :

A chaque tour de la boucle de jeu l'algorithme fait :

- Si la pile de directions à suivre par Mario n'est pas vide alors :

➔ On stock la première direction de la pile dans une variable
➔ Ensuite on dépile
➔ On actualise la position du joueur dans la direction récupérer
➔ On appelle la fonction Sleep pour faire un rapide pause afin de bien percevoir le déplacement du joueur

3) Nos quelques bonus

Pour les bonus, vu qu'ils ne seront probablement pas sujets à des points supplémentaires, nous n'en n'avons fait que deux :

Musique : La musique est activable et désactivable **en jeu** uniquement grâce à la **touche L.**

Menu de présentation : Un menu est disponible afin d'accéder aux différents niveaux. Pas obligatoire, mais sacrément plus pratique.

4) Problèmes rencontrés et solutions apporté

De nombreuses difficultés ont été rencontrées durant l'avancé de ce projet. En effet, le jeu avait d'abord été codé par Fabien avant d'être posté par M. Diedler sur campus. Cependant, le jeu de M. Diedler était beaucoup plus fini. Nous avons donc décidé de l'utiliser pour continuer le jeu.

De plus Julien a eu beaucoup de mal à installer allegro et source tree car il code sur Mac. Il a donc pris un retard très conséquent dans l'élaboration du projet et n'a pas encore été en mesure de le rattraper en raison d'une accumulation du travail à côté.

Au niveau du code, nous avons eu deux majeures difficultés :

- La première pour le code du bruteforce. En effet nous voulions faire un bruteforce « optimisé » mais nous n'avons pas réussi. L'erreur n'ayant pas pu être identifiée, nous nous sommes recentrés sur un bruteforce classique. Les conséquences sont qu'il ne marche que pour les niveaux test.
- Pour le BFS (breadth first search), nous avions remarqué que le programme ne trouvait pas toujours la solution et que parfois il s'arrêtait de fonctionner pour chaque fois une raison différente. Nous avons réussi à identifier la première erreur car nous nous sommes aperçus que lors du calcul de la ZAP nous envoyions les

coordonnées du nouvel emplacement de la caisse et non des anciennes. La seconde erreur a été notifiée par M. Diedler. En effet un delete d'un Nœud* contenait encore une référence sur le nœud précédent ce qui engendrait une erreur.

5) Protocoles de tests et réponses aux questions

Afin de résumer les performances de nos programmes, nous avons dressé un tableau :

	Algorithmes		BFS		DFS	
Niveau	Longueur minimale en déplacements du Mario	Longueur minimale en poussées de caisses	Sans deadlocks	Avec deadlocks	Sans deadlocks	Avec deadlocks
Facile 1	33	8	0,001 s	0,0003 s	0,001 s	0,0005 s
Facile 2	16	3	0,001 s	0,0002 s	0,11 s	0,04 s
Facile 3	41	13	0,007 s	0,004 s	0,005 s	0,002 s
Facile 4	23	7	0,4 s	0,1 s	0,15 s	0,06 s
Facile 5	29	6	2,3 s	1,8 s	6 s	2,2 s
Facile 6	110	28	1.9 s	0,7 s	0,2 s	0,06 s
Easy 7	40	6	14,1 s	12,6 s	>7 min	2 min
Facile 8	97	32	0,005 s	0,005 s	0,01 s	0,0008 s
Facile 9	30	10	0,001 s	0,00015 s	0,005 s	0,0005 s
Facile 10	108	21	0,03 s	0,01 s	0,01 s	0,008 s

Ces valeurs sont très approximatives.

Le brute force n'est pas représenté ici car il n'exécute que les niveaux test. Le temps d'exécution des niveaux faciles sont extrêmement long pour lui. En revanche un niveau facile réalisé à moitié est rapide a exécuté pour le bruteforce.

- Quel(s) algorithme(s) donne(nt) une solution optimale ?

De façon évidente le bruteforce ne donne pas la solution la plus optimale puisqu'il fait TOUS les déplacements possibles tant qu'il n'a pas trouvé la solution du niveau. Il est donc long et couteux en mémoire.

On pourrait croire que le DFS est moins intéressant que le BFS dans de nombreux cas mais au contraire il est parfois (et même souvent) meilleur que le BFS. Là où il est plus lent est quand se situe plusieurs caisses juxtaposées. Cependant, la solution trouvée n'est pas forcément la plus courte en termes de longueur minimale.

Enfin le BFS est peut-être le plus stable. Pas toujours le plus rapide en temps, il est néanmoins le plus efficace pour trouver le chemin le plus court. S'il existe plus d'une solution, on peut être sûr qu'il trouvera celle avec le chemin le plus court.

Pourquoi le DFS est plus rapide que le BFS ? Tout simplement car il va d'abord au bout de la pile avant de tester un autre nœud, alors que le BFS fait le tout en parallèle.

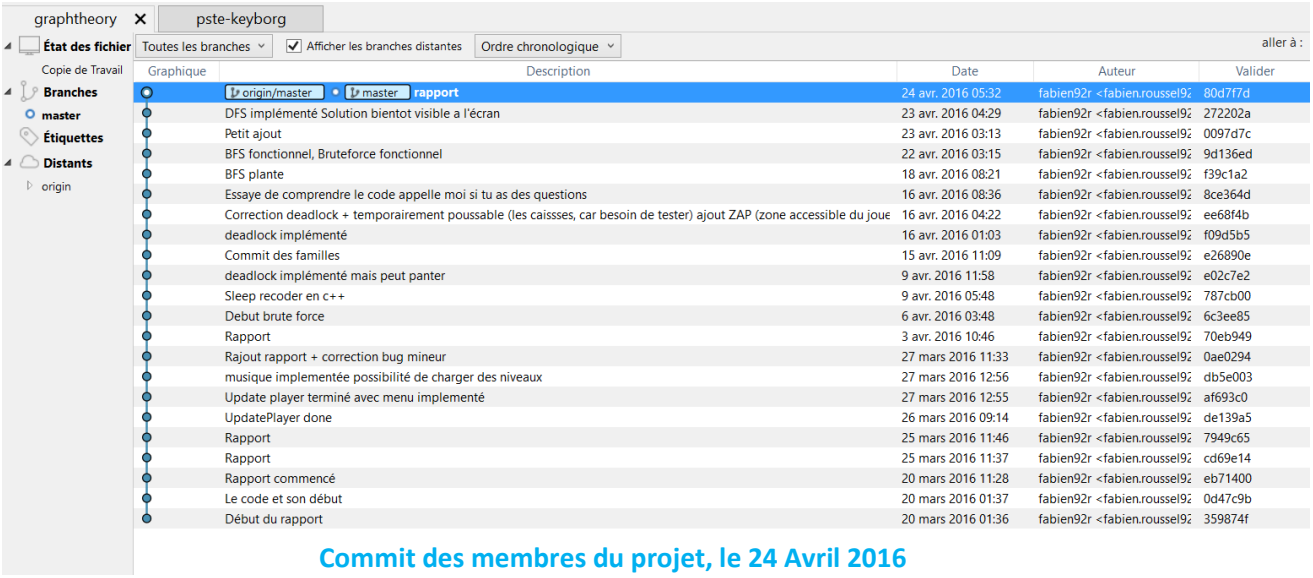
- Quel sont les impacts sur l'arbre de recherche quand on active l'algorithme de détection des deadlocks ?

L'arbre de recherche diminue très grandement. Cela permet de gagner en temps d'exécution car on évite de continuer à chercher une solution lorsqu'un nœud a une caisse sur une deadlock. La mémoire s'en voit aussi affecté car on a moins d'information à stocker.

6) Utilisation des outils SourceTree et BitBucket

Nous avons mis à profit le logiciel de versionning SourceTree à l'aide de BitBucket.

Cependant seulement un membre s'est avancé dans le projet. C'est pourquoi il n'y a qu'une seule personne qui a push lors du développement du projet.



Commit des membres du projet, le 24 Avril 2016

IV. Bilan

1) Conclusion collective

Ce projet fut très assez drôle car il a permis de mobiliser nos connaissances afin de créer une mini intelligence artificielle. On a pu voir à quel point c'est facile pour l'homme de résoudre ce genre de jeu mais que pour l'ordinateur, cela est beaucoup plus complexe. Le résultat actuel du jeu est globalement satisfaisant. Nous espérons aller au bout du niveau 3.

Cependant l'implication de chacun des membres du groupe n'étaient pas la même ce qui ne permet pas de fournir un rapport aussi complet que nous aurions voulu.

Nous avons eu la chance que M. Diedler nous donne de nombreuses aides, notamment pour le BFS, lors de la séance de suivi de projet.

2) Conclusions personnelles

-Julien Kuntz :

Ce membre du groupe n'a pas donné sa conclusion personnelle.

-Fabien Roussel :

J'ai trouvé ce projet particulièrement intéressant car il touche à un domaine qui m'intéresse : l'intelligence artificielle (c'est d'ailleurs l'une des raisons qui m'ont poussé à choisir l'écosse comme destination en départ pour l'ING 3). J'apprécie ce genre de code car je le trouve à la fois utile, intelligent et je prends plaisir à l'optimiser.

On voit bien l'enjeu de la conception ici car il est dur de créer le programme directement. Concevoir l'algorithme permet donc d'avoir une vision bien plus précise de qu'on devra faire.

J'ai dû faire face à peu de difficultés, mais des grandes. J'ai cependant réussi à en venir à bout au bout de plusieurs jours de recherches et de *cout*.

Annexe

Algorithme BruteForce