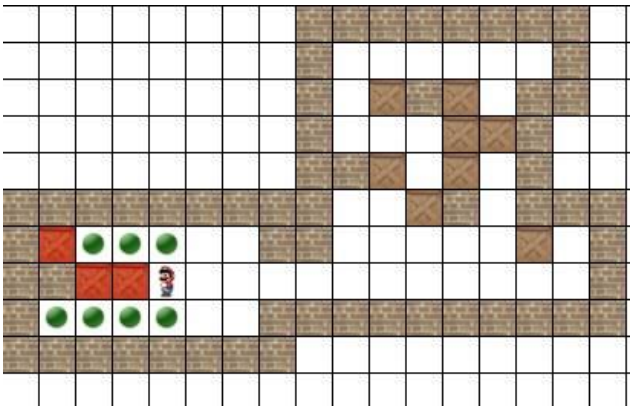
Avril 2016

Rapport projet SOKOBAN

Module Théorie des Graphes S4



ECE PARIS
ÉCOLE D'INGÉNIEURS

Romain MICHAU – Victor BIGAND – Paul RENOUIL
ING 2 TD03

Sommaire

Introduction

Dans le cadre de notre quatrième semestre à l'école d'ingénieurs ECE Paris, nous avons dû réaliser un jeu SOKOBAN. Ce projet s'inscrit dans le module de théorie des graphes et nous permet d'utiliser différents algorithmes que nous avons pu voir en cours.

Notre équipe est composée de Romain MICHAU, Victor BIGAND et Paul RENOUIL.

Ce projet diffère des trois précédents projets puisqu'une base de code a été fournie au préalable, c'est-à-dire que nous avons dû nous appuyer sur un code déjà écrit et donc nous imprégner d'un code inconnu.

Le jeu SOKOBAN est un jeu de réflexion populaire d'origine japonaise. Le but du jeu est très simple, il suffit de déplacer un certain nombre de caisses sur des emplacements nommés « goals », cependant la difficulté réside dans la carte du jeu, qui est un labyrinthe en 2D plus ou moins compliqué. Les différents chemins possibles pour le joueur peuvent être modélisés par des graphes d'état et le jeu peut donc être résolu à l'aide d'algorithmes de théorie des graphes, à savoir la recherche du plus court chemin par exemple. C'est pourquoi le développement d'un jeu SOKOBAN est bénéfique à l'apprentissage d'une partie de la théorie des graphes.

Les règles sont courtes : le joueur ne peut pas traverser les murs et ne peut que pousser une caisse (pas la tirer). Des cases nommées deadlocks (cases mortes) sont aussi activables, qui bloquent le jeu si une caisse se trouve dessus.

Nous devions modéliser chaque état du joueur et des caisses (leur position) par des nœuds, et déterminer le chemin de coût minimum (le moins de déplacement de caisses pour qu'elles soient toutes placées sur les goals) à l'aide de trois algorithmes différents :

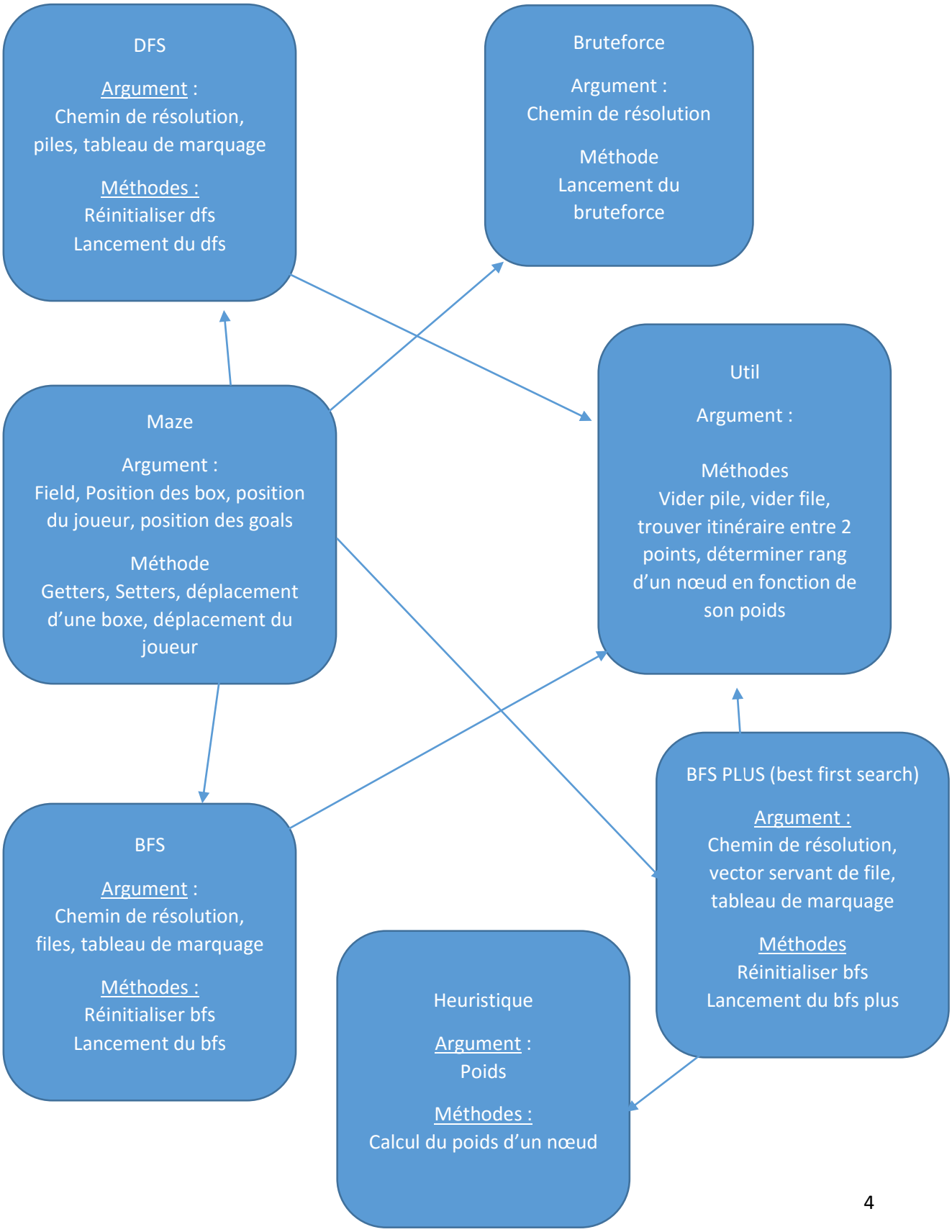
- L'algorithme de force brute, qui teste toutes les solutions possibles jusqu'à trouver la bonne,
- L'algorithme BFS (Breadth First Search) de parcours en largeur,
- L'algorithme DFS (Depth First Search) de parcours en profondeur.

Nous devions aussi visualiser la vitesse d'exécution de ces algorithmes, la charge mémoire, et le nombre de nœuds explorés.

Dans un second temps (un deuxième niveau), le but est de travailler sur une heuristique, à savoir une fonction permettant de savoir à tel moment vers quel nœud aller, et donc d'améliorer les performances des algorithmes cités précédemment.

Le troisième et dernier niveau nous propose d'améliorer les performances des algorithmes selon certains critères.

Diagramme de classes



Utilisation du jeu

- Déplacement du personnage : touches fléchées
- Activation des deadlocks : « 9 »
- Activation du bruteforce : « 1 »
- Activation du BFS : « 2 »
- Activation du DFS : « 3 »
- Activation du BestFirstSearch sans deadlocks dynamique : « 4 »
- Activation du BestFirstSearch avec deadlocks dynamique : « 5 »

Les choix de programmation

Pour une modularité optimale, nous avons décidé de créer une classe pour chaque type de résolveur. C'est-à-dire une classe bruteforce, une classe BFS, une classe DFS et une classe Best First Search. Nous avons également créé une classe util qui contient de nombreuses fonctions.

Bruteforce

Le bruteforce nécessite peu de conteneurs. Nous lui avons uniquement attribué un std :: vector qui renvoye le chemin permettant la réalisation du niveau.

BFS

Nous avons utilisé les conteneurs standards pour les BFS c'est-à-dire :

- Une file permettant d'appliquer le principe FIFO nécessaire au fonctionnement du BFS. Pour la file nous avons utilisé le conteneur qui nous paraissait le plus approprié c'est-à-dire une « std ::queue<Maze> »
- Un tableau pour le marquage des états, permettant de marquer les états déjà visités, nous avons utilisé pour cela un « std ::vector<Maze>»
- Un tableau contenant tous les coups que doit effectuer le personnage afin de terminer le niveau.

DFS

Nous avons utilisé les conteneurs standard pour un DFS c'est-à-dire :

- Une pile permettant d'appliquer le principe LIFO nécessaire au fonctionnement du DFS.Pour la file nous avons utilisé le conteneur qui nous paraissait le plus approprié c'est-à-dire une « std :: stack<Maze> »
- Un tableau pour le marquage des états, permettant de marquer les états déjà visités, nous avons utilisé pour cela un « std ::vector<Maze>»
- Un tableau contenant tous les coups que doit effectuer le personnage afin de terminer le niveau.

BEST FIRST SEARCH

Le best first Search a un fonctionnement très proche de celui du BFS, les conteneurs utilisés sont donc semblables à une différence près.
Le std ::queue<Maze> du BFS classique ne peut pas être utilisé dans ce contexte. En effet, les états doivent y être insérés en fonction de leurs poids, il faut donc être capable d'y insérer des états en plein milieu. Nous avons donc fait le choix d'utiliser un std ::vector<Maze>

Algorithmes de théorie des graphes

Brute force :

Nous avons décidé de coder notre bruteforce en itératif.
Voilà son fonctionnement :

Tant que le niveau n'est pas résolu :

Essayer tous les chemins possibles en 1 coup, puis tous les chemins possibles en 2 coups, puis en 3 coups….

Renvoyer le chemin menant à la victoire.

BFS :
Le BFS reçoit le Maze m en paramètre, qui lui est envoyé par le main.
Créer une file pour la zone accessible et créer une file pour les Field.
Créer un vector pour marquer les zones accessibles et un vector pour marquer les Fields.
Créer un booléen Win et l'initiateur à false.

Déterminer la zone accessible au personnage (c'est-à-dire sans avoir à pousser des boxes).
Enfiler cette zone et enfiler le Field de l'état actuel dans les file_zone et file_field.
Ajouter la zone accessible dans le vector marque_zone et le field dans le vector marque _field.

Tant que le win = false

Changer l'état du jeu (Maze) en lui attribuant pour field le front de la file_field et donner pour position au personnage une des positons accessibles dans le front de la file_zone.

Regarder pour tous les boxes s'il est possible de les bouger vers le haut, et sans provoquer de deadlocks.

- Si c'est possible, simuler une pousse de la boxe vers le haut, puis marquer le field obtenu ainsi que la zone accessible. Enfiler le field et la zone accessible obtenue. Vérifier également que l'on n'est pas dans un état gagnant. Si c'est le cas win = true.
- Sinon ne rien faire.

Regarder pour tous les boxes s'il est possible de les bouger vers le bas, et sans provoquer de deadlocks.

- Si c'est possible, simuler une pousse de la boxe vers le haut, puis marquer le field obtenu ainsi que la zone accessible. Enfiler le field et la zone accessible obtenue.
- Sinon ne rien faire.

Regarder pour tous les boxes s'il est possible de les bouger vers la gauche, et sans provoquer de deadlocks.

- Si c'est possible, simuler une pousse de la boxe vers le haut, puis marquer le field obtenu ainsi que la zone accessible. Enfiler le field et la zone accessible obtenue.
- Sinon ne rien faire.

Regarder pour tous les boxes s'il est possible de les bouger vers la droite, et sans provoquer de deadlocks.

- Si c'est possible, simuler une pousse de la boxe vers le haut, puis marquer le field obtenu ainsi que la zone accessible. Enfiler le field et la zone accessible obtenue.
- Enregistrer la position.
- Sinon ne rien faire.

Défiler les files file_field et file_zone.

Quand win== true
Backtrack des positions où doit passer le joueur pour gagner.
BFS pour trouver le coup à faire pour relier les positions enregistrées.

Retourner le chemin pour gagner au main.

DFS :

Le DFS reçoit le Maze m en paramètre qui lui est envoyé par le main.
Créer une pile pour les zones accessibles et créer une pile pour les fields.
Créer un vector pour marquer les zones accessibles et un vector pour marquer les fields.
Créer un booléen win et l'initialiser à false.

Déterminer la zone accessible au personnage (c'est-à-dire sans avoir à pousser des boxes).
Empiler cette zone et empiler le field de l'état actuel dans les pile_zone et pile_field.
Ajouter la zone accessible dans le vector marque_zone et le field dans le vector marque _field.

Tant que le win = false

Changer l'état du jeu (Maze) en lui attribuant pour field le front de la pile_field et donner pour position au personnage une des positons accessibles dans le front de la pile_zone.

Regarder pour toutes les boxes s'il est possible de les bouger vers le haut, et sans provoquer de deadlocks.

- Si c'est possible, et que l'état n'est pas marqué, simuler une pousse de la boxe vers le haut, puis marque le field obtenu ainsi que la zone accessible. Empiler le field et la zone accessible obtenue. Vérifier également que l'on est pas dans un état gagnant. Si c'est le cas win = true.

- Sinon ne rien faire.

Regarder pour toutes les boxes s'il est possible de les bouger vers le bas, et sans provoquer de deadlocks.

- Si c'est possible, et que l'état n'est pas marqué, simuler une pousse de la boxe vers le haut, puis marque le field obtenu ainsi que la zone accessible. Empiler le field et la zone accessible obtenue.
- Sinon ne rien faire.

Regarder pour toutes les boxes s'il est possible de les bouger vers la gauche, et sans provoquer de deadlocks.

- Si c'est possible, et que l'état n'est pas marqué, simuler une pousse de la boxe vers le haut, puis marque le field obtenu ainsi que la zone accessible. Empiler le field et la zone accessible obtenue.
- Sinon ne rien faire.

Regarder pour toutes les boxes s'il est possible de les bouger vers la droite, et sans provoquer de deadlocks.

- Si c'est possible, et que l'état n'est pas marqué, simuler une pousse de la boxe vers le haut, puis marquer le field obtenu ainsi que la zone accessible. Empiler le field et la zone accessible obtenue.
- Enregistrer la position.
- Sinon ne rien faire.

Quand win== true
Backtrack des positions où doit passer le joueur pour gagner.
BFS pour trouver les coups à faire pour relier les positions.

Retourner le chemin pour gagner au main.

Best First Search

Le BFS reçoit le Maze m en paramètre qui lui est envoyeé par le main.
Créer un vector pour les zones accessibles et créer une file pour les fields.
Créer un vector pour marquer les zones accessibles et un vector pour marquer les fields.
Créer un booléen win et l'initialiser à false.

Déterminer la zone accessible au personnage (c'est-à-dire sans avoir à pousser des boxes).
Enfiler cette zone et enfiler le field de l'état actuel dans les vector_zone et vector_field.
Ajouter la zone accessible dans le vector marque_zone et le field dans le vector marque _field.

Tant que le win = false

Changer l'état du jeu (Maze) en lui attribuant pour field le front de la liste_field et donner pour position au personage une des positions accessibles dans le front du vector_zone.

Regarder pour toutes les boxes s'il est possible de les bouger vers le haut, et sans provoquer de deadlocks.

- Si c'est possible, simuler une pousse de la boxe vers la droite, puis marquer le field obtenu ainsi que la zone accessible.
- Demander à la fonction heuristique le poids du nœud.
- En fonction du poids, insérer la zone et le field dans les vectors_ et vector_field.
- Enregistrer la position de la boxe.
- Sinon ne rien faire.

Regarder pour toutes les boxes s'il est possible de les bouger vers le bas, et sans provoquer de deadlocks.

- Si c'est possible, simuler une pousse de la boxe vers la droite, puis marquer le field obtenu ainsi que la zone accessible.
- Demander à la fonction heuristique le poids du nœud.
- En fonction du poids, insérer la zone et le field dans les vectors_ et vector_field.
- Enregistrer la position de la boxe.
- Sinon ne rien faire.

Regarder pour toutes les boxes s'il est possible de les bouger vers la gauche, et sans provoquer de deadlocks.

- Si c'est possible, simuler une pousse de la boxe vers la droite, puis marquer le field obtenu ainsi que la zone accessible.
- Demander à la fonction heuristique le poids du nœud.

- En fonction du poids, insérer la zone et le field dans les vectors_ et vector_field.
- Enregistrer la position de la boxe.
- Sinon ne rien faire.

Regarder pour toutes les boxes s'il est possible de les bouger vers la droite, et sans provoquer de deadlocks.

- Si c'est possible, simuler une pousse de la boxe vers la droite, puis marquer le field obtenu ainsi que la zone accessible.
- Demander à la fonction heuristique le poids du nœud.
- En fonction du poids, inserer la zone et le field dans les vectors_ et vector_field.
- Enregistrer la position de la boxe.
- Sinon ne rien faire.

Enlever la dernière valeur des vector_field et vector_zone.

Quand win== true
Backtrack des positions où doit passer le joueur pour gagner.
BFS pour trouver les coups à faire pour relier les positions.

Retourner le chemin pour gagner au main.

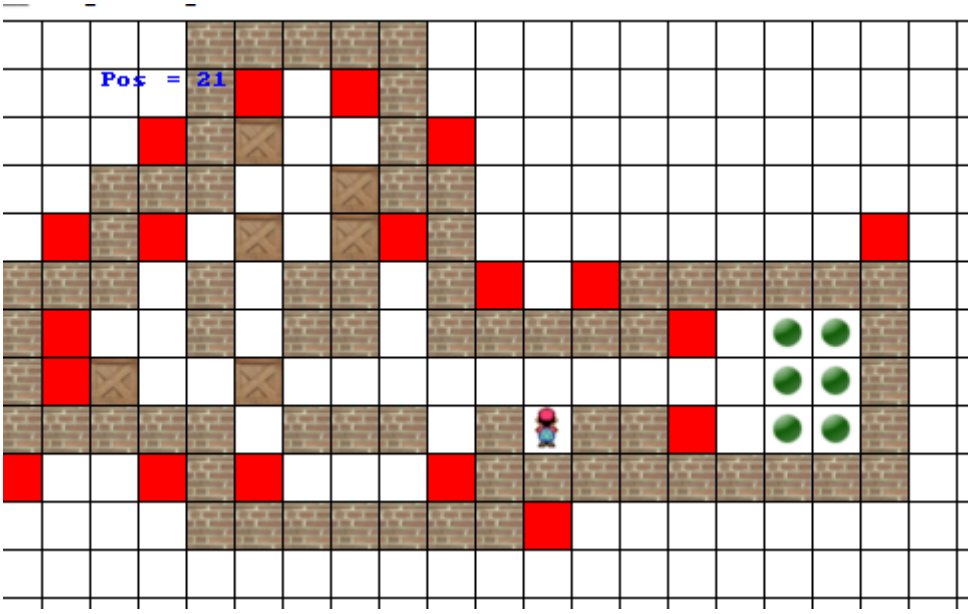
Réponses aux questions du sujet

Pour comparer l'efficacité des différents algorithmes, je donne les résultats obtenus sur le niveau easy 7 pour chaque type d'algorithme

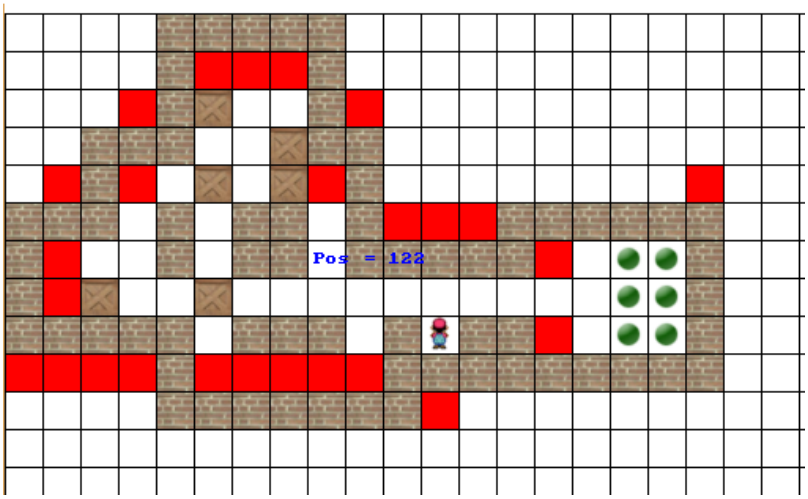
	Temps de résolution	Nœuds explorés	Charge mémoire	Qualité de la réponse obtenue
Brute force	Très long, la durée de résolution du bruteforce dépend du nombre de mouvements nécessaires pour résoudre le niveau. Le bruteforce est un algorithme naïf, il ne prend pas en compte les deadlocks ou les mur	Pas de notion de nœud pour un bruteforce		Réponse optimale
BFS sans détection des deadlocks	12 secondes	20 000 nœuds	25 Mo	Réponse optimale (40 mouvements nécessaires pour résoudre le niveau)
BFS avec détection des deadlocks	1 seconde	843 nœuds	0.1 Mo	Réponse optimale (40 mouvements nécessaires pour résoudre le niveau)
DFS sans détection des deadlocks	5 secondes	9000 nœuds	1.1Mo	Réponse non optimale (7500 mouvements nécessaires pour résoudre le niveau)
DFS avec détection des deadlocks	1 seconde	652 nœuds	80ko	Réponse non optimale (7500 mouvements nécessaires pour résoudre le niveau)
Best first search sans detections des deadlocks	<1 seconde	102 noeuds	12Ko	Réponse moyenne (49 mouvements nécessaires pour résoudre le niveau)
Best first search avec detections des deadlocks	<1 seconde	66 noeuds	8Ko	Réponse moyenne (40 mouvements nécessaires pour résoudre le niveau)

DeadLocks

Pour détecter les deadlocks, nous avons d'abord sélectionné tous les coins et les avons marqués.



Puis nous avons relié entre eux tous les coins par des deadlocks. A condition qu'il n'y ait aucun goal entre les coins.



Heuristique

Pour notre heuristique nous avons choisi la fonction suivante :

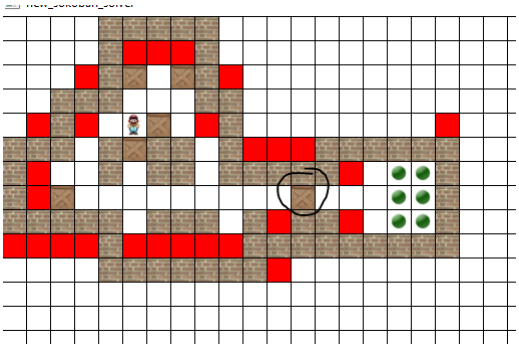
H(n)=(box la plus proche du goal)*nombre de box-0+(deuxième box la plus proche du goal)*nombre de box-1+(deuxième box la plus proche du goal)*nombre de box-2 …..

Cette fonction va pousser les Best first search à se concentrer sur les situations permettant de rapprocher les boxes de goal, avec une préférence pour les boxes qui sont déjà proches du goal.

Nous avons également rajouté à H(n) un critère prenant en compte le nombre de box déjà placées. Et nous avons donné un coefficient de 50 à cette composante, cela a permis de faire en sorte que l'algorithme préfère placer une boxe sur le goal plutôt que d'en rapprocher une autre.

Nous avons enfin rajouté un critère permettant de prendre en compte l'accessibilité des goals par le personnage.

Par exemple dans le cas suivant, le goal n'est pas accessible rendant impossible la résolution du niveau.



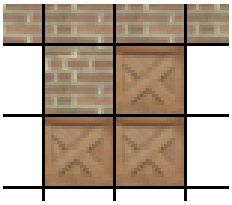
Au final nous obtenons la fonction suivante :

H(n)=2*(distance_des_boxes)+50*(box_placées)+(accessibilité_des_goal)

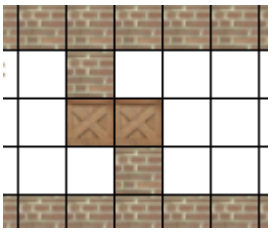
DeadLocks Dynamiques

Nous avons également appliqué un système de détection des deadlocks dynamique. Nous avons repéré deux cas de formation de deadlocks dynamiques :

- Un carré formé de boxes et de murs



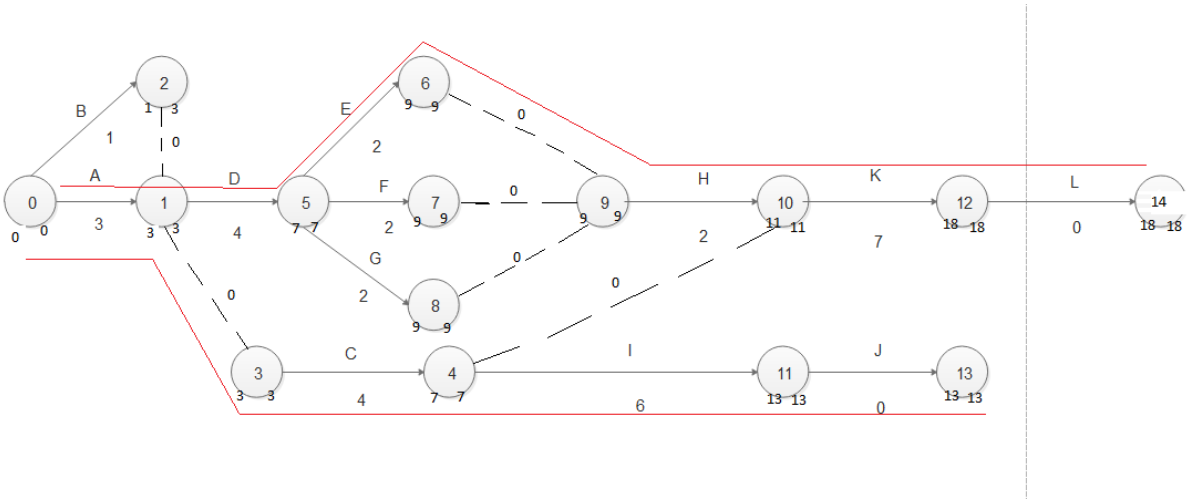
- Un Z formé de deux boxes et deux murs



Nous avons appliqué cet algorithme de détection des deadlocks à notre Best First Search. Cela a fait baisser le nombre de nœuds explorés de 3500 à 1250 pour le niveau medium_1.

Répartition des tâches

Pour ce projet, nous avons, comme suggéré par le sujet, réalisé un réseau PERT afin d'établir un programme d'avancement du projet. Voici le réseau en question :



Avec les tâches suivantes :

A → compréhension du sujet

B → création d'un projet bitbucket

C → commencement du rapport

D → compréhension du code fourni et début de code

E → BFS

F → DFS

G → Brute Force

H → visualisation des résultats (performances, etc..)

I → fin du rapport

J → rendre le rapport

K → heuristique (niveau 3) et derniers détails

L → rendre le projet

Les dates au plutôt sont indiquées en bas à gauche de chaque nœud et les dates au plus tard en bas à droite. Toutes les marges valent 0 mis-à-part la tâche B qui a une marge de deux.

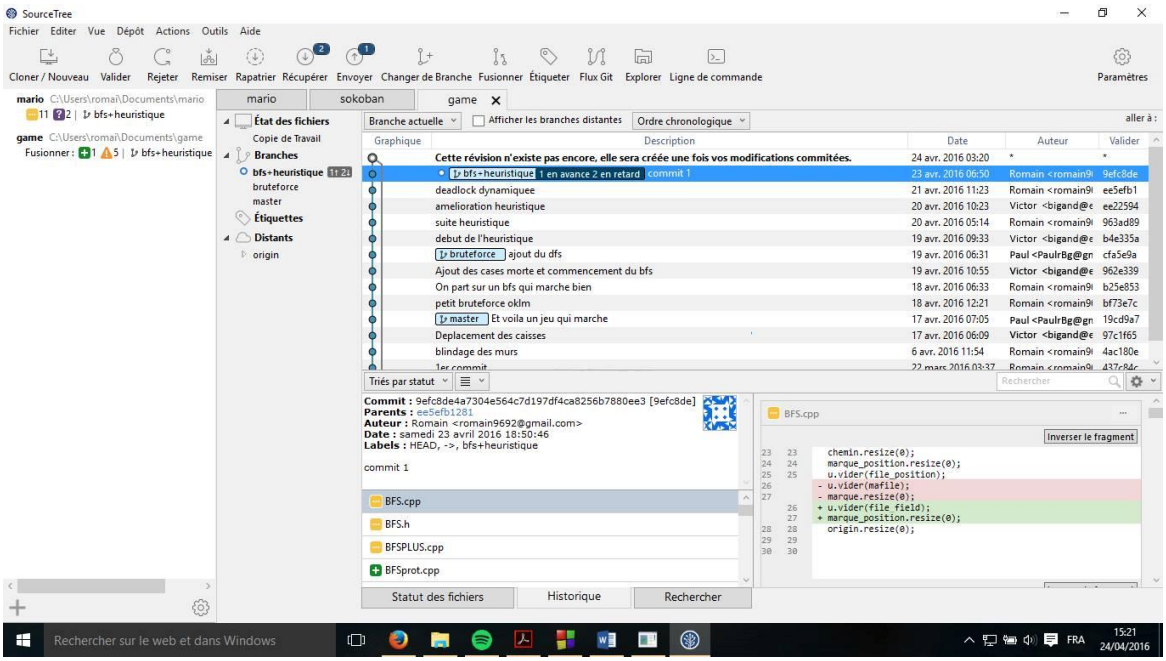
Les chemins critiques sont indiqués en rouge, ce sont donc tous les chemins qui ne comprennent pas la tâche B.

Nous avions réalisé ce réseau au préalable et avons essayé de le respecter au mieux. Cependant, plusieurs contraintes ont fait qu'il a été impossible de respecter intégralement les durées que nous nous étions imposées. En effet, certains algorithmes ont été plus durs que nous le pensions et nous avons donc été plus longs sur certaines tâches, comme plus cours sur d'autres.

Au niveau de la répartition même des tâches, nous avions décidé dès le début de comment nous organiser. Victor a commencé par comprendre le code fourni et coder le restant de jeu, puis à faire l'algorithme de force brute ainsi qu'une partie du rapport. Romain a travaillé sur les algorithme BFS, DFS et la visualisation des performances de chacun, ainsi qu'une partie du rapport. Paul a travaillé sur l'heuristique avec Romain. Nous avons tous travaillé ensuite sur les morceaux restants.

Dans l'ensemble, le travail en équipe s'est bien déroulé, avec une bonne cohésion et une bonne entente.

Voici une capture d'écran de Source Tree, l'outil de versionning que nous avons utilisé.



Bilans

Bilan collectif

Ce projet a été très bénéfique pour notre approfondissement de nos connaissances en théorie des graphes. En effet, c'est une branche de l'informatique nouvelle pour nous et son aspect théorique nous rebutait un peu au début. Nous avons eu quelques difficultés à s'imprégner de toutes les notions et tous les algorithmes mis à notre disposition. Néanmoins, le côté ludique du jeu nous a plu et permis de mieux comprendre les notions vues en cours.

Nous pensons qu'une des plus grandes difficultés a été de se baser sur un code déjà écrit. En effet, programmer à partir de fonctions, de variables, et de classes qui ne sont pas les notre est plutôt compliqué et demande un certain temps d'adaptation. Nous n'avons pas aimé ce principe au début mais avec du recul, nous avons compris son intérêt. Il est en effet beaucoup plus rapide de partir d'un stade déjà avancé, il suffit juste de s'imprégner du code. De plus, le but de ce projet n'était pas de créer le jeu mais de coder les différents aspects de théorie des graphes concernés.

Nous avons trouvé intéressant le fait d'améliorer les performances d'un programme. C'est un point que nous avions que très peu aborder au cours de nos enseignements informatiques ces deux dernières années et c'est un sujet intéressant qui est omniprésent dans l'informatique aujourd'hui.

Bilan Victor BIGAND

Je dois dire que ce projet était un réel défi. En effet de par le code de base et les notions complexes étudiées, j'ai trouvé ce projet compliqué et l'ai peut-être moins apprécié que les précédents.

Cependant, il s'est avéré bénéfique puisque je ne m'étais pas vraiment investi dans le module de théorie des graphes auparavant et j'ai pu combler mes lacunes et consolider mes acquis.

Pour la seconde fois j'ai trouvé que travailler avec un outil de versionning est performant bien que nous ayons eu quelques soucis avec. Le travail en équipe s'est bien déroulé et nous avons chacun pu apporter nos visions des choses ce qui a été positif.

Comme abordé dans le bilan collectif, j'ai beaucoup apprécié la notion d'heuristique et le fait d'améliorer les performances du programme, ceci étant directement mesurable au niveau de la charge mémoire du programme.

Bilan Romain MICHAU

J'ai personnellement beaucoup aimé ce projet. En effet nous avions des objectifs clairs à remplir, c'est-à-dire résoudre les niveaux. Cela permet de transformer le projet en réel défi. Par ailleurs nous avons pu appliquer les notions théoriques vues en cours de théorie des graphes, et ainsi voir quelles applications pratiques peuvent être données à ce domaine. Je n'avais pas du tout imaginé que la théorie des graphes puisse servir à la résolution de problème de ce genre.

J'ai également pu améliorer mes compétences au niveau du versionning, qui est vraiment un outil formidable pour le travail en équipe.

Bilan Paul RENOUIL

Cette approche de la théorie des graphes est je pense un complément nécessaire aux cours théoriques reçus au cours de ce semestre. Notamment en ce qui concerne la transformation d'un problème situationnel en graphe mathématique afin d'y appliquer les algorithmes vus en amphi. Ce projet nous a également amené à manipuler la notion d'heuristique, qui peut être un concept compliqué à aborder mais que l'application pratique aide à saisir. Etant Ing2 nouveau, ce projet fut également mon premier projet utilisant les bibliothèques d'Allegro, et toute expérience est bonne à apprendre en programmation. Ce projet nous a aussi montré, plus que les précédents, l'importance de la planification et des schémas avant de s'attaquer au code. Ainsi que les utilisations aux allures limitées de prime abord des différents conteneurs de la STL (difficile de comprendre l'utilité d'avoir des files ou des piles alors que l'on avait les vecteurs avant ce projet).