2013

Université de Limoges
1ère année Master
CRYPRIS

Pierre Bailhache
Julien Barraud

[PROJET IA : LABYRINTHE]

[Le but de ce projet est de créer une application qui peut charger des labyrinthes, et utilise divers algorithme pour essayer de trouver la sortie d'une manière optimale.]

Sommaire

Présentation

Nous devions réaliser une application sous la forme d'un labyrinthe, où un personnage doit atteindre la sortie en jouant le moins de coups possibles.

Le personnage avance d'une case à chaque coup et le score final obtenu est en fonction du nombre de coups joués pour atteindre la sortie et il possède la vision.

Le programme peut charger n'importe quel labyrinthe, du moment qu'il respecte le mode de construction. De plus une interface graphique permet de voir le personnage se déplacer dans le labyrinthe, prouvant ainsi qu'il passe bien par le chemin menant à la sortie.

Une fois arrivé à destination, le programme indique le nombre de coup qu'il a été nécessaire pour pouvoir trouver la sortie en fonction de l'algorithme choisit. De plus le chemin le plus court est également affiché pour l'utilisateur.

Nous devions implémenter trois algorithmes différents , avec des heuristiques distincts :

- Meilleur d'Abord ;
- Cout Uniforme ;
- A *.

Etude

Case par case

Tout d'abord une étude pour pouvoir respecter les différentes contraintes citées précédemment. Il est dit que le personnage avance d'une case à chaque coup, par conséquent nous avons fait deux parties :

- Next ;
- Cycle.

La partie Next appelle les algorithmes mais ces derniers ne font qu'un seul tour, ainsi le personnage s'avance bien d'une case et attend la prochaine action de l'utilisateur. Grâce à cela, l'utilisateur peut parfaitement prévoir les différents mouvements en même temps que l'algorithme s'exécute. La partie Cyle appelle simplement Next dans une boucle qui s'exécute jusqu'à qu'on trouve la sortie. En effet nous avons trouvé cela utile de programmer pour voir le déplacement d'une seule case par case, mais cela peut vite devenir lassant et répétitif pour l'utilisateur. On peut appeler Cycle quand l'on veut, par conséquent on peut commencer notre programme pas à pas avec Next puis passer en mode cycle pour gagner du temps.

De plus il est dit que le personnage possède la vision, il n'est pas un simple automate sans vision et sans réflexion. Nous avons modélisé cette contraintes par le fait que le personnage voit s'il emprunte un chemin menant à une impasse, ou si il voit la sortie. Bien évidemment le personnage ne peut voir à travers les murs, mais il peut apercevoir dans les virages. Donc si le personnage voit une impasse, il

saura que le chemin n'est pas bon, par conséquent il ne l'empruntera pas. De la même manière dès qu'il voit la sortie, il empruntera le chemin le plus court !

Algorithme

Ensuite il a fallu réfléchir les heuristiques pour chaque algorithme :

- Meilleur d'Abord : nous avons effectué une distance de Manhattan en partant de la sortie et se propageant dans tout le labyrinthe, chaque case voisine au père reçoit l'heuristique du père plus un, ainsi chaque case (quelle soit un mur ou un couloir) possède une distance propre par rapport à la sortie. Bien évidement de cette manière plusieurs cases peuvent avoir la même distance, donc si on est malchanceux cet algorithme peut explorer toutes les cases.
- Cout Uniforme : cet algorithme est le plus simple, car il explore uniformément le labyrinthe. Pas d'heuristique spécial pour celui-ci, en effet on aurait pu changer le coût de certaines cases si elles étaient instables, non traversable,… Mais notre labyrinthe présente toujours les mêmes couloirs, par conséquent pour se déplacer nous avons un coût de un entre deux cases. Cela aurait pu être plus attractif si certaines cases avaient un coup supérieur aux autres, mais cela n'était pas demandé dans le sujet.
- A* : le A* est un mélange entre les deux algorithmes précédemment, en effet il prend en compte la distance de Manhattan, ainsi que le coût du déplacement pour accéder à une case. Cet algorithme est censé être le plus rapide parmi les trois, nous comparerons leurs résultats en détails dans un autre chapitre.

Déplacement

On est censé avoir une interface et pouvoir voir le déplacement du personnage dans le labyrinthe. Pour cela nous avons affiché notre labyrinthe, avec un simple tableau de JPanel auquel nous donnons une couleur. En revanche pour pouvoir respecter la condition, nous devions voir le déplacement du personnage pas à pas, par conséquent case par case. Donc celui-ci ne doit pas se téléporter, et on doit le voir revenir en arrière pour aller sur la case déterminer par les algorithmes, donnant ainsi vraiment l'impression du déplacement « normal » du personnage.

Option supplémentaire

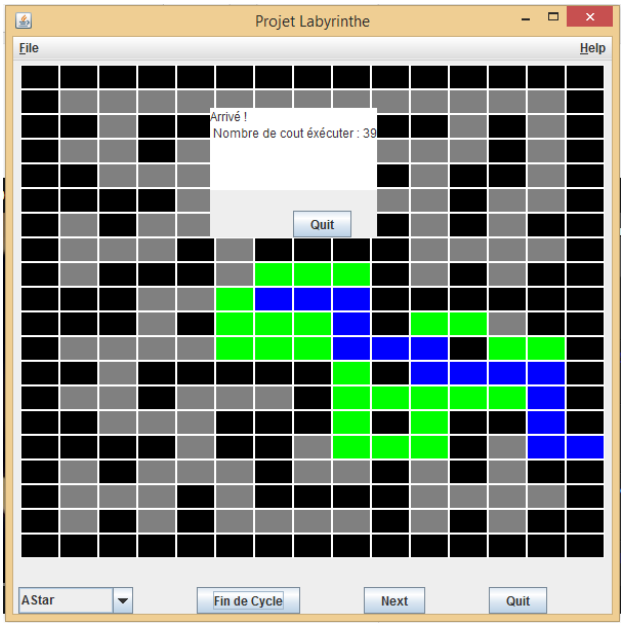
Nous avons également pensé à quelques options qui peuvent être utiles pour le projet :

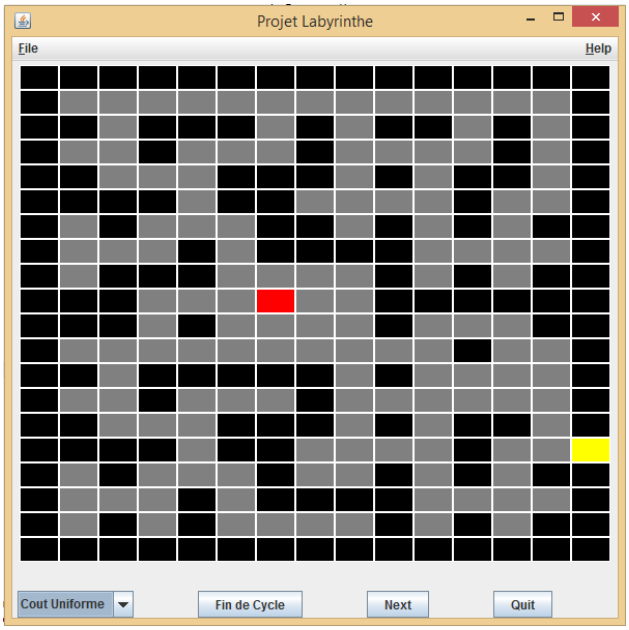
- Reset : permet de remettre le labyrinthe dans son état d'origine, par conséquent celui lors de son chargement dans le logiciel.
- Load : Permet de charger un autre labyrinthe, mais le fichier doit respecter le format de lecture.
- Help : un simple Pop-Up pour aider l'utilisateur avec les différents boutons et commandes, même si ce logiciel n'est pas très compliqué il est important d'avoir ce menu pour indiquer le but du logiciel ainsi que les différentes options.

Création de labyrinthe aléatoire

Nous avons pensé qu'il fallait un petit programme pour pouvoir créer aléatoirement des labyrinthes, et ensuite de les tester dans notre programme.

Interface





Voici l'interface du logiciel, tout d'abord à droite l'affichage lorsqu'on lance le logiciel, on observe un qu'un labyrinthe est déjà pré charger, il s'agit du labyrinthe qui est présent sur le sujet du projet. Le personnage est représenté par la couleur rouge, et la sortie par la couleur jaune. Il y a un menu déroulant en bas à gauche, permettant le choix entre les trois algorithmes. Il faut savoir que lorsqu'on lance un algorithme, on ne peut plus en changer tant que l'on n'a pas reset le labyrinthe. Il y a le bouton « Fin de Cyle » pour effecteur les déplacements automatiquement jusqu'à ce que l'on trouve la sortie. Le bouton « Next » permet d'avancer pas à pas, tant qu'on n'appuie pas sur le bouton le personnage n'avance pas, permettant ainsi de mieux étudier si le personnage prend le bon chemin. Et enfin le bouton « Quit » qui permet de quitter l'application.

Sur la gauche on peut voir un labyrinthe dont la sortie a été trouvé avec un algorithme A*. En vert on observe les cases qui ont étés visités par le personnage, en bleu le chemin le plus court allant de la position initial à la sortie du labyrinthe. On observe le Pop-Up qui nous indique le nombre de case qui a été visité avant de trouver la sortie.

Il y a également le menu en haut de la fenêtre, il ne possède que deux menu « File » et « Help » car nous n'avions pas assez de contenu pour en remplir davantage. Le menu File comporte les items « Load », « Reset » et « Quit », chacun ayant un raccourci clavier pour une utilisation plus rapide et simple pour l'utilisateur. Le menu « Help » ne comporte que le Pop-Up d'aide.

Code

Projet Labyrinthe

Nous avons créé différentes classes pour le bon fonctionnement du programme :

- Algorithme : cette classe permet d'initialiser les différents tableaux pour ensuite permettre le bon fonctionnement des algorithmes. Elle a des fonctions qui permettent d'initialiser le tableau des déplacements dans le labyrinthe, permettant ainsi de savoir où on est, et où on a déjà visité. De plus elle a une fonction qui permet de tester si les cases sont des impasses, l'initialisation d'un tableau avec leur distance par rapport à la sortie (la distance de Manhattan). Pour voir le déplacement du personnage dans le labyrinthe, nous avons créais deux fonctions qui permet à partir d'un nœud de parcourir tous ses pères jusqu'à la position initiale, ou un nœud spécifique. Ainsi nous obtenons une liste de tous les nœuds Pères parcourus, et donc nous pouvons retracer son déplacement. Et enfin il y a une fonction qui permet d'ordonner la liste passée en paramètres.
- AStar : cette classe permet de créer une instance de l'algorithme A*. Elle hérite de la classe Algorithme, obtenant ainsi toutes les fonctions utiles pour son bon fonctionnement. Astar ne contient que son constructeur ainsi que la méthode pour lancer l'algorithme.
- CoutUniforme : cette classe permet de créer une instance de l'algorithme Cout Uniforme. Elle hérite de la classe Algorithme, obtenant ainsi toutes les fonctions utiles pour son bon fonctionnement. CoutUniforme ne contient que son constructeur ainsi que la méthode pour lancer l'algorithme.
- Création : cette classe permet de créer tout d'abord l'interface, donc elle hérite de la classe JFrame pour faire apparaitre la fenêtre, et contient toutes les méthodes pour afficher le labyrinthe. En effet cette classe permet de charger un labyrinthe (donc tout simplement de lire un fichier et de stocker les différentes variables), de reset un labyrinthe, l'affichage en temps réel des déplacements. De plus cette classe gère les boutons, donc elle fait des appels aux algorithmes pour qu'ils s'exécutent. En outre elle contient les diffèrent PopUp qui s'affiche si le personnage a trouvé la sortie ou si on exécute un algorithme alors qu'un autre est déjà e cours d'exécution.
- MeilleurDAbord : cette classe permet de créer une instance de l'algorithme Meilleur d'Abord. Elle hérite de la classe Algorithme, obtenant ainsi toutes les fonctions utiles pour son bon fonctionnement. MeilleurDAbord ne contient que son constructeur ainsi que la méthode pour lancer l'algorithme.
- Menu_Haut : cette classe permet de créer la barre de menu présent en haut de l'interface. C'est un JMenu avec des JMenuItem, et chaque JMenuItem a son propre raccourci clavier.
- My_Laby : Permet de remplir un JPanel avec d'autre JPanel ayant des couleurs, qui sera ensuite afficher sur l'interface.
- Nœud : Un nœud possède sa position, une heuristique ainsi que son père. Grâce à la dernière information, nous pouvons retourner en arrière dans le labyrinthe, évitant ainsi de se téléporter. Elle contient la fonction « Comparator » qui permet de comparer l'heuristique de deux nœuds, pour savoir lequel est le plus petit. Cette fonction est très utile lorsqu'on veut trier la liste des nœuds en fonction de leurs heuristiques.

Générateur de labyrinthe

Pour obtenir une série de tests rapidement, nous avons décidé de coder un générateur de labyrinthe. Celui-ci est codé en Python dans un souci de portabilité (utilisation dans des projets personnels).

Son fonctionnement est plutôt simple :

- On prend un tableau de taille n x m.
- On choisi un entier aléatoire i entre n et m.
- On fait un mur vertical en coordonnée i.
- On génère un passage aléatoire dans le mur.

On répète sur les deux sous-parties du tableau (à droite et à gauche du mur), récursivement en alternant entre les murs horizontaux et verticaux.

On s'arrête quand on arrive sur une taille de 2x2 pour éviter de faire des « blocs » de murs. Après la génération, il y a un nettoyage du labyrinthe pour s'assurer que chaque endroit de celui-ci soit accessible au joueur. Les labyrinthes générés ont un aspect assez « géométrique » et les résultats obtenus lors des tests sont dus à cet aspect général plutôt qu'aux algorithmes en eux-mêmes.

Les tests ont été effectués sur 19 labyrinthes générés via notre outil.

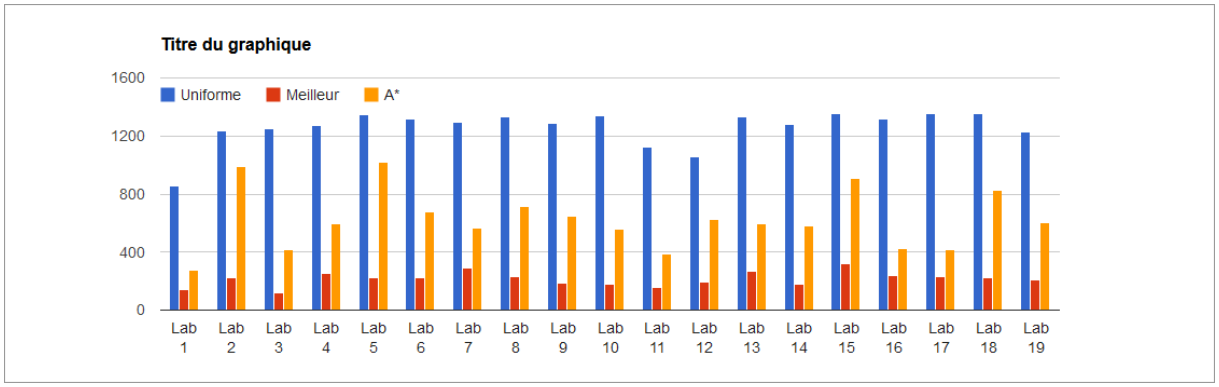
Résultat

Nous avons effectué une batterie de test, pour montrer les différences entre chaque algorithme de recherche. On se doute que le coût uniforme sera le moins optimal et que le A* sera le plus optimal, mais nous allons vérifier cette hypothèse. Pour faire les tests, nous avons utilisé notre générateur de labyrinthe, ainsi ils seront tous différents et créé aléatoirement.

Nous avons créais des labyrinthes de 40 par 40, le personnage est toujours situé en haut à gauche, et la sortie en bas à droite du labyrinthe. Voici le tableau correspondant aux différents algorithmes de recherche, avec le nombre de coût qu'il a fallu faire avant de trouver la sortie :

	Uniforme	Meilleur	A*
Lab 1	859	138	278
Lab 2	1236	221	992
Lab 3	1252	122	416
Lab 4	1269	251	593
Lab 5	1344	223	1017
Lab 6	1320	225	675
Lab 7	1297	292	562
Lab 8	1334	231	715
Lab 9	1290	185	648
Lab 10	1343	180	558
Lab 11	1122	155	387
Lab 12	1057	196	624
Lab 13	1329	266	594
Lab 14	1279	176	577
Lab 15	1358	317	910
Lab 16	1317	237	422
Lab 17	1352	233	417
Lab 18	1358	225	828
Lab 19	1229	210	605
MIN	859	122	278
MAX	1358	317	1017
MOYENNE	1260,26315789474	214,894736842105	622

Voici le résultat sous forme d'un graphique :



Exploitation

Nous pouvons voir que les résultats ne correspondent pas aux hypothèses établies précédemment, en effet dans tous les cas le meilleur d'abord obtient un meilleur résultat que les deux autres algorithmes. Ceci peut s'expliquer par le fait que les labyrinthes sont tous créés à partir du programme fait par nos soins, donc peut être qu'il ne créait pas de « vrai » labyrinthe, vu qu'ils sont tous basés sur la même méthode de construction, par conséquent ils se ressemblent tous. Donc si leurs constructions correspondent à une bonne performance du meilleur d'abord, alors il est normal que dans tous les labyrinthes l'algorithme meilleur d'abord ait les meilleurs résultats.

Pour être sûr de nos résultats, il aurait fallu trouver d'autre labyrinthe, ou encore les créer à la main, ou encore prendre les mêmes labyrinthes entre les différents groupes et que chacun puisse ainsi comparer leurs résultats.

Conclusion

Dans ce projet nous avons pu constater les différences entre des algorithmes de recherche. En effet même si ils ont le même but, certains algorithmes sont beaucoup plus rapides que d'autres, tout dépend de leurs heuristiques. D'après nos essais, c'est l'algorithme meilleur d'abord qui est le plus optimal, or nous avons vu en cours que c'est l'algorithme A* qui est censé avoir les meilleurs résultats.

Il y a juste un petit défaut avec notre programme, c'est que l'affichage ralenti la programme, donc on obtient le résultat après un certains délai. De plus, plus le labyrinthe est grand, plus l'affichage a quelques soucis.