LASSOUDIERE Anthony
NICOLAS Arthur
BERRY Yanis

Enseignants :
Christophe Saint-Jean
Laurent Mascarilla

Projet d'étude de cas :
Jeu de collecte rapide d'item en 2D

Sommaire

Les objectifs

1- Aller d'un point A à un point B de la carte :

La position de départ et l'objectif du joueur lui sont donnés au début du jeu. Ces positions sont générées aléatoirement et prennent en compte le type des cases donc le fait que le joueur doit pouvoir être sur sa case de départ et sur sa case d'arrivée. C'est la fonction "genDepartArrivee" dans jeu.c qui s'occupe de faire ça.

2- Environnement :

Les cases sur le terrain sont de 2 types, libre et infranchissable, l'algorithme en tient compte et fonctionne avec cette contrainte.

3- Ressources :

A l'aide d'un menu option on peut choisir le nombre de ressources que l'on veut sur le terrain (le nombre de ressources peut être égal à zéro). Si il y en a, le joueur a pour objectif de les ramasser avant d'aller à son point d'arrivée, donc l'algorithme le prend en compte. Les ressources sont générées avant le début de la partie et la position donnée prend en compte le fait que le joueur doit pouvoir être sur la case pour récupérer la ressource. La fonction "genObjet" dans jeu.c gère l'attribution des positions des ressources.

4- Fin de partie :

On vérifie à la fin de chaque tour si le joueur actif est sur sa case d'arrivée et si toutes les ressources ont été ramassées, dans ce cas on lui donne la victoire.

Le code est le suivant:

```
if (jeu.players[i]->position[0] == jeu.players[i]->arrivee[0]
     && jeu.players[i]->position[1] == jeu.players[i]->arrivee[1]
     && jeu.nbRessource == 0){ //ttes les ressources sont ramassees
  choix = victoire;
}
```

5- Le score :

Le nombre de déplacements en 4-connexité est compté et affiché a l'écran en tant que score. La fonction "afficherScore" de affichage.c affiche ce score.

6- Bonus 1 :

On a codé un générateur de terrain aléatoire qui produit un chemin cohérent, toute case libre est reliée aux autres donc un joueur peut atteindre n'importe quelle case libre malgré les cases infranchissables, pour peu qu'il soit sur une case libre. C'est la fonction "genTerrain" dans jeu.c qui gère cela.

La fonction initialise toutes les cases en infranchissable puis en prend une au hasard pour la rendre franchissable. Pour chaque case à côté dans les quatre directions on vérifie qu'elle est dans le terrain et pas déjà franchissable, si c'est le cas on la garde en mémoire dans un tableau. Une fois les quatre directions vérifiées on prend une des cases du tableau aléatoirement pour la rendre franchissable et recommencer la vérification des quatre directions. Si à un moment donné le tableau ne contient aucune case (toutes en dehors du terrain ou déjà franchissable) on prend une des cases déjà franchissables au hasard sur la carte pour recommencer le processus. Cela est effectué en boucle jusqu'à ce que l'on ait le nombre de cases franchissables voulu par l'utilisateur.

7- Bonus 2 :

Un mode multijoueur est intégré au jeu, on peut le choisir dans le menu des options, on a limité le nombre de joueurs à 9 pour des raisons de mémoire. Les joueurs effectuent alternativement leurs tours.

On utilise la bibliothèque SDL pour que l'utilisateur puisse interagir avec le jeu à l'aide du clavier, pour gérer le chargement, l'affichage des images (SDL_image) et l'affichage de texte a l'écran (SDL_ttf) comme le score ou pour déclarer la victoire du joueur.

Une fonction est appelée pour faire toutes les allocations mémoire avant le début du jeu et les initialisations (la fonction "new_Game") et à la fin de la partie on appelle une fonction qui se charge de libérer la mémoire (la fonction "free_Jeu"). Hormis ça toutes les mémoires allouées sont libérées lorsque l'on en a plus l'utilité.

Les structures de données

Notre code se décompose en plusieurs fichiers (un .c et un .h pour chaque, excepté le *main.c*) :

- Affichage :
  Gère toute la partie graphique du jeu : de l'affichage de la carte à celui des joueurs, ressources, score etc...

- Algorithme :
  Contient les algorithmes servant à calculer le chemin le plus court pour le joueur visant à récupérer l'intégralité des ressources et atteindre la case d'arrivée avec le moins de "pas" possible.

- Arrete :
  Une arête représente un arc entre un point A et un point B. Elle contient le chemin à parcourir et la distance de celui-ci.

- Case :
  Une case est soit libre soit infranchissable et son sprite varie en fonction de ce paramètre ;

- EtatPosition :
  Contient les flags d'informations liés à une position.

- Jeu :
  Contient les informations relatives au jeu (joueurs, dimensions de la carte, ressources, etc…), permet de créer le jeu, de créer la carte, de libérer l'espace mémoire alloué au jeu, et de jouer un tour.

- Joueur :
  Un joueur est caractérisé par sa position, son sprite, et le nombre de ressources qu'il a récupéré. Il peut de déplacer de case en case en vérifiant s'il n'y a pas une ressource à chaque case.

- Main :
  Construit la fenêtre et lance le jeu.

- Ressource :
  Une ressource est seulement caractérisée par un sprite et sa position sur la carte.

- Sprite :
  Toutes les informations utilisées pour l'affichage d'une image.

L'Algorithme

Il s'agit d'un algorithme maison qui se rapproche des algorithmes les plus connus pour ce problème. Il se décompose en deux algorithmes distincts.

Le premier algorithme consiste en la création d'arètes représentant les arcs entre les différents points (positions) de la carte et leur tri croissant selon la distance calculée. La liste triée est ensuite parcouru et on en retire les arcs du plus court au plus long de manière à former un chemin entre un point A et un point B, passant par tous les points de ressources de la carte.

Le second algorithme décide le chemin entre un point A et B pour la création d'une arête. Il tente les différentes solutions qui s'offrent pour atteindre un point en évitant les obstacles (cases infranchissables) et cherche à trouver le meilleur chemin possible de cette manière. C'est la distance du chemin trouvé qui est utilisé dans le premier algorithme pour déterminer le chemin final.

➢ Déroulement type de l'algorithme 1 : chemin le plus court par les arcs.

- Vérification du nombre de ressources. Création d'une simple arête entre départ et arrivée s'il n'y a aucune ressource et retour de celle-ci.
- Création de la liste de toutes les arêtes possibles (grâce au second algorithme) en les rangeant de manière ordonnée dans un tableau.
- Recherche du chemin le plus court en sélectionnant les arêtes de la plus courte à la plus longue : Vérifications de boucles et oublis afin de créer un chemin passant par chaque ressource.
- Tri du chemin obtenu afin de ranger les arêtes dans l'ordre du parcours.
- Retour de la liste.

➢ Déroulement type de l'algorithme 2 : chemin le plus court par contraintes.

Fonction récursive. Teste les directions les plus plausibles pour atteindre son objectif. Tente ensuite les autres directions. S'arrête si le point est atteint ou si l'avance actuelle dépasse un chemin trouvé précédemment.

- Comparaison avec l'avancée actuelle, sortie si dépassement.
- Comparaison avec l'objectif. Mise à jour du meilleur résultat et sortie si correspondance.
- Test des cases dans les deux directions X et Y les plus susceptibles de mener à l'objectif : appel de l'algorithme sur les nouvelles cases.
- Test des autres directions appel de l'algorithme sur les nouvelles cases.
- Libération de la mémoire sur la case actuelle et retour.

➢ Analyse de l'algorithme

De par la création de l'intégralité des Arêtes l'espace mémoire nécessaire est exponentiel au nombre de ressources sur la carte. Cependant c'est un problème sans grande conséquence avec de multiples joueurs puisque le calcul s'effectue joueur par joueur et l'espace non utilisé est correctement libéré.

A l'inverse, le temps d'exécution est surtout affecté par la taille de la carte et paradoxalement sa simplicité. L'algorithme créé vise à se rapprocher le plus de la recherche du meilleur chemin possible de telle manière que presque l'intégralité des chemins sont explorés. Plus il y a d'espace libre, plus il peut se révéler long. C'était un choix entre optimal et efficace.

➢ En bref

L'algorithme créé dans ce projet répond aux attentes visant à trouver un chemin court tout en prenant en compte des contraintes. Cependant, le fait qu'il ait été créé de toute pièce avec des ambitions probablement trop gourmandes trop rapidement, il reste des défauts et il arrive que le programme ne réponde plus dans certains cas non identifiés. Pour conclure : l'algorithmique c'est très intéressant mais comme on est en informatique, il faut se souvenir d'une règle importante : il vaut mieux prendre ce que d'autres ont déjà fait (généralement mieux) plutôt que de créer de toute pièce.

Améliorations

- Pour le cas où il y a plusieurs joueurs on pourrait recalculer le chemin à prendre pour un joueur lorsqu’un autre joueur a pris une ressource avant lui, car pour l'instant il suit son chemin pré-calculé sans voir si une ressource est déjà prise ou non.

- Pour le multijoueur la victoire va à celui qui rentre en premier sur sa case d'arrivée mais il serait plus juste que le nombre de ressources ramassées entre en compte également.

- Par le multijoueur on pourrait aussi proposer la possibilité d’utiliser des algorithmes différents afin de comparer l’efficacité de plusieurs variantes (préalablement codées).

- L'algorithme a encore quelques problèmes dans certaines situations, on pourrait terminer de le corriger, et également l’optimiser.