VECTOR RACER

TABLE DES MATIÈRES

TABLE DES FIGURES

LISTE DES TABLEAUX

INTRODUCTION

Vector Racer est originellement un jeu de course de voitures, que l'on peut déplacer sur des cases d'une grille. Le jeu gère les notions de vitesses et d'accélération des voitures, si bien qu'il devient difficile de tourner rapidement avec une vitesse trop élevée. L'objectif de ce jeu est d'arriver en tête de la course tout en évitant les obstacles comme les murs par exemple.

Comme toute course, le joueur cherchera à minimiser son temps de parcours et de fait, à maximiser la vitesse tout en minimisant la distance parcourue. On comprend déjà que notre algorithme de résolution va chercher un parcours qui va correspondre en tout point à une optimisation à la fois de la distance parcourue et de la vitesse.

Nous allons donc poser dans un premier temps les problèmes à résoudre ainsi que le vocabulaire que nous allons employer pour caractériser le jeu. Nous verrons ensuite comment nous avons implémenté les types abstraits de données qui apparaissent dans les problèmes évoqués ci-dessus puis nous terminerons par présenter l'algorithme de résolution et l'ensemble des tests réalisés pour vérifier sa validité.

Capacité du programme réalisé

Notre programme permet d'afficher, de jouer et de résoudre une carte donnée en paramètre. La résolution de la carte n'est pas une résolution optimale. Il s'agit d'une solution non triviale de parcours du circuit.

1 ALGORITHMIQUE DU JEU

1.1 DÉFINITIONS

Les mots que nous allons utiliser pour décrire un objet ou un concept précis mis en oeuvre dans la résolution du problème sont des mots pouvant donner lieu à de nombreuses lectures et approches différentes qui peuvent fausser les démonstrations qui s'appuient dessus. Pour fixer les idées, nous nous devons d'imposer des bases, seul appui pour nos raisonnements ultérieurs.

1.1.1 Cartes

Une *case* c sera considérée comme un élément admettant plusieurs états possibles, représentés par les cas suivants :

— une case associée au symbole $\star$ représente une position de départ possible pour la course,

— une case associée à un entier de 0 à 9 fait partie de la piste du circuit, et est donc la zone parcourable par les voitures,

— une case est enfin associée au symbole # lorsqu'elle représente un obstacle (mur bordant la piste par exemple), qui n'est pas accessible au voitures.

Une *carte* $\mathcal{C}$ (c.f. exemple Figure 1) est une structure contenant des cases, organisées sous forme d'une matrice de dimension $n \times m$, et sur laquelle est définie une fonction $\eta_{\mathcal{C}}$ à valeur dans l'ensemble $\Omega = \{\star, \#, [1-9]\}$ telle que $\forall c \in \mathcal{C}, \exists \omega \in \Omega, \eta_{\mathcal{C}} = \omega$. Cette fonction permet donc d'accéder à l'état d'une case.

Un case sera dite *valide* si son état est dans l'ensemble $\Omega_V = \{\star, [1-9]\}$.

On défini la notion de *chemin* sur la carte comme une séquence de cases - donc de positions - prises par un véhicule. Un *parcours* est un chemin partant et aboutissant dans une case étiquetée par $\star$. Un parcours est dit *valide* si et seulement si les cases sont parcourue dans l'ordre croissant depuis la position de départ, à partir d'une case étiquetée par 1 et que les coups élémentaires sont valides (voir Section 1.1.2 page suivante). Enfin, un parcours est *complet* - appelé aussi *tour* - si il est valide et que l'ensemble des positions élémentaires fait apparaitre au moins une fois chaque chiffre visible sur la carte. La position d'arrivé est soit une case étiquetée par $\star$, soit une case étiquetée par 1.

Notons qu'une carte est *valide* si et seulement si elle respecte les contraintes suivantes :

1. elle comporte au moins une case à l'état $\star$,
2. l'ensemble formé par les cases d'étiquette dans Ω_V est connexe,
3. il existe au moins un tour.

```
#######
#33332#
#4###2#
#4###2#
#*1111#
#######
```

FIGURE 1 – Exemple de carte 7×6

1.1.2 Coups

Le jeu *Vector Racer* présente des règles particulières quant aux coups possibles à chaque étape, du fait de la considération de la vitesse et de l'accélération du véhicule. Le principe général est le suivant :

1. on considère le vecteur vitesse du véhicule à chaque étape : $\begin{pmatrix} x \\ y \end{pmatrix}$ (composantes dans le repère orthonormé direct associé à la grille),
2. étant donné une séquence d'accélérations possibles (dépendantes des performances du véhicule), par exemple $\begin{pmatrix} dx \\ dy \end{pmatrix}$, le vecteur vitesse à l'étape suivante est $\begin{pmatrix} x + dx \\ y + dy \end{pmatrix}$.

Ce vecteur vitesse et la séquence des accélérations possibles conditionnent ainsi l'accès à certaines cases à chaque étape.

Un coup est *valide* si et seulement si :

1. les positions de départ et d'arrivée sont valides sur la grille (c.f. Section 1.1.1 page précédente),
2. la variation du vecteur vitesse induite par ce coup correspond à une accélération possible (parmis celle de la séquence d'accélérations que l'on considère).

Accélérations possibles	Coups effectuées	Variation de vitesse	Case
1,1			⋆
-1,-1	1,0	1,0	1
-1,0	2,0	1,0	1
-1,1	1,1	-1,1	1
0,-1	0,1	-1,0	2
0,0	-1,1	-1,0	2
0,1	-2,0	-1,-1	2
1,-1	-1,-1	1,-1	3
1,0	0,-1	1,0	3
	0,-1	0,0	4
	1,0	1,1	⋆

TABLE 1 – Exemple de liste de coups possibles pour la carte Figure 1 page précédente

1.2 PROBLÈMES

1.2.1 Spécifications

```
Problème : Résolution
Entrée : Une grille G représentant la carte réalisable.
Sortie : Une liste de coups valide.
```

1.3 RÉSOLUTION

1.3.1 Choix de l'algorithme

Le but du jeu Vector Racer est de trouver le plus court chemin faisant le parcours d'une grille. Les notions de chemins et parcours nous ont poussés à considérer plusieurs algorithmes connues sur les graphes susceptibles de résoudre ce problème. Il s'agit en effet du parcours en largeur, du parcours en profondeur, de Dijkstra et de A*. Ces algorithmes garantissent tous des solutions, néamoins ils sont plus ou moins optimales. Le parcours en profondeur par exemple ne renvoie pas le plus court chemin. A* quand à lui nécessite des heuristiques très difficles à généraliser vu le grand nombre de cartes possibles. Dijkstra et le parcours en largeur à l'opposée semblent très adéquats au problème, ce sont d'ailleurs les deux algorithmes que nous avons choisis d'implémenter pour chercher les solutions.

En effet, ces deux algorithmes évoluent progressivement vers la solution en minimisant le nombre de coups. Dans le cas du jeu Vector Racer, la connaissance seulement de la position et la vitesse permet à chaque étape de générer un ordre convenable de successeurs, vers lequels le joueur peut se déplacer en maximisant sa vitesse tout en se rapprochant de la destination. Dans la section qui suit, nous allons présenter l'implémentation de l'algorithme de résolution. Nous allons aussi détailler quelques fonctions auxiliaires essentielles dans la prise de décision.

```
Data: carte
Result: solution
f ← file-vide();
Enfiler le noeud position de départ;
while no-vide(f) do
    Défiler;
    Colorer le noeud;
    Générer les 8 successeurs possibles;
    Parmis ces successeurs, choisir ceux qui maximisent l'accélération;
    Chaîner les successeurs séléctionnés au noeud coloré;
    for Chaque successeur do
        if Le successeur est le noeud d'arrivée then
            Retourner le noeud;
        else
            Enfiler le noeud;
        end
    end
end
```

Algorithm 1: Algorithme de résolution d'une grille

2 RÉALISATION DU PROGRAMME

2.1 STRUCTURES DE DONNÉES

Certaines des structures de données ne sont utilisées que par certains modules du programme. Il n'est donc pas nécessaire de les abstraire. En revanche, certaines de ces structures sont utilisées par plusieurs modules, c'est le cas de la liste chaînée.

2.1.1 Structures simples

La structure map

La structure map est construite par le parseur de cartes, contenu dans le module de parsing. les champs de cette structure sont :

— Un entier hauteur

— Un entier largeur

— Un tableau de caractères

— Une structure position pour repérer le point de départ

— Un entier pour le maximum de checkpoints

La structure steps

La structure steps est un ensemble de coups relatifs à une carte. Elle est générée par le parseur de coups. Elle peut être utilisée,par exemple, pour de comparer les performences d'autres intelligences artificelles.
Elle contient :

— Un entier pour le nombre d'accélérations

— Un entier pour le nombre d'étapes

— Une structure point pour la position de départ

— Deux tableaux de structures point contenant les positions et vitesses correspondantes

La structure point

Cette structure est exclusivement utilisée par les modules de moteur de jeu, ainsi que l'affichage graphique.
La structure point est une structure simple à deux champs entiers. Ces deux nombres représentent une position sur la grille. Le moteur de jeu utilise cette structure pour communiquer une position à l'interface.

La structure node

La structure node est utilisée par le module de résolution d'une carte. Elle contient quatre champs entiers, correspondants à une position et une vitesse. Elle contient de plus un champs pointeur sur le noeud précédent.

Grâce à l'abscence de typage des données dans la liste chaînée (pointeur void), nous pouvons stocker des données de n'importe quel type dans une liste chaînée, notemment les structures point (pour le moteur de jeu) et node (pour le solveur).

2.1.2 Structures abstraites

La seule structure ayant été sujet d'une abstraction est la liste chaînée.

Mise en oeuvre

Le type de données utilisée içi est une liste doublement chainée

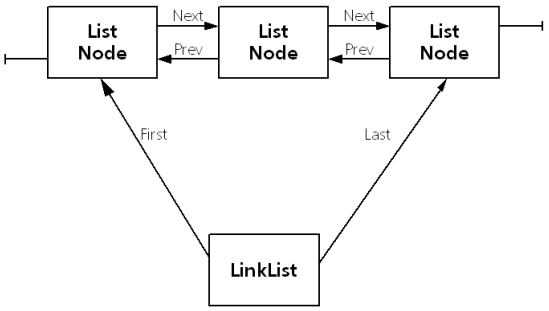

FIGURE 1 – Double linked list

Les structures sont donc :
`liste`

- pointeur sur l'élément tête
- pointeur sur l'élément queue
- entier nombre d'éléments (à 0 initialement)

`element`

- pointeur sans type sur une donnée
- pointeur élément sur l'élément précédent
- pointeur élément sur l'élément suivant

Primitives

Les primitives utilisées pour gérer la liste sont les suivantes :
`list_empty` qui teste si la liste est vide.
`list_head` renvoie la tête de la liste.
`list_tail` renvoie la queue de la liste.
Celles relatives aux éléments sont : `list_data` qui renvoie un pointeur sans type sur le champ donné de l'élément passé en paramètre.
`list_next` renvoie le pointeur sur l'élément suivant.
`list_previous` renvoie le pointeur sur l'élément précédent.

Intérêt

L'abstraction de la liste chainée et de ses élements la rend utilisable par n'importe module sans qu'il n'ai connaissance de son implémentation. Cela permet notemmment de l'utiliser pour lister des coups jouables avec le solveur, ou de générer une liste de positions jouées.

Le double chaînage permet de réduire la complexité de la fonction `list_add` (qui effectue un ajout en queue). Avec une liste simple elle aurait été $\theta(n)$ en la taille de la liste, içi elle est en $\theta(1)$. De même pour la fonction de retrait en queue.

2.2 LE MODULE SDL

Un jeu pour qu'il en soit un a besoin d'une interface. Cette interface est indispensable pour le joueur. Sans elle il n'y aurait pas d'échange d'information entre l'utilisateur et la machine.

Nous avons fait le choix d'utiliser la bibliothèque SDL pour réaliser l'interface de vector racer. L'avantage de cette bibliothèque est qu'elle permet à la fois une gestion de l'affichage,du temps et des évènements (principalement les choix de l'utilisateur). Nous avions de plus de l'expérience avec cette bibliothèque, ce qui a facilité notre choix.

Lien pour la bibliothèque SDL : http ://www.libsdl.org

Les fonctions utilisant directement la SDL sont regroupées dans le module vectorSDL. A l'exception du module vectorGame, aucun autre module ne touche directement aux fonctions de la bibliothèque SDL. Nous avons fait ce choix afin de simplifier son utilisation.

2.3 MOTEUR DE JEU

Le joueur va parcourir le circuit et traverser les différentes zones. Les évènements (les choix de l'utilisateur) sont traités grâce à des fonctions de la bibliothèque SDL. Pour déterminer si le joueur perd,gagne ou triche, il faut que l'on vérifie à chaque étape la situation du joueur.

La première situation à vérifier est la colision avec un mur. On vérifie donc que le joueur ne tombe pas sur une case mur. Nous n'avons pas traité le fait qu'un tracé peut traverser un mur sans s'y arrêter.

La deuxième situation est la fin du circuit. Pour gérer l'évènement, on considère le circuit comme terminé si le joueur est bien passé par toute les zones de checkpoint et qu'il est revenu soit sur la case départ, soit sur un checkpoint inférieur en valeur aux deux dernier checkpoint.

Enfin, il faut vérifier que le joueur ne rate pas un checkpoint. Pour cela on crée un compteur qui vérifie que la valeur du dernier checkpoint atteint augmente bien de manière régulière (sans sauts).

2.4 MISE EN OEUVRE DE L'ALGORITHME DE RÉSOLUTION

2.4.1 Implémentation

Nous avons implémenté l'algorithme de recherche en largeur en utilisant une structure noeud (cf. FIgure 2). Cette structure contient 4 entiers représentant la position dans la grille et la vitesse en cette position. De plus elle contient un pointeur sur un noeud qui va servir à retracer le chemin qui a généré ce noeud.

```
Struct node{
int x,y,dx,dy;
struct node * parent;
}
```

FIGURE 2 – Définition de struct noeud

Pour stocker ces noeud nous utilisons une structure liste. L'algorithme manipule deux listes pricipales : "queue" et "colored" en plus d'une liste temporaire "successors". La liste "queue" sert à stocker les noeuds 'candidats' qui potentiellement font

avancer le chemin le plus vers la destination. "colored" quand à elle sert à mémoriser les noeuds déjà essayés. "successors" est une liste qui stocke temporairement tous les successeurs de la position courante.

Au début de l'algorithme, la liste "queue" contient uniquement la position de départ correspondant à une '*' dans la grille. Une fonction "generate-successors" s'occupe de générer les noeuds successeurs de cette position, il y'en a au maximum neuf, correspondant aux neuf accélérations possibles. Ces noeud successeurs sont générés dans l'ordre faisant maximiser la norme de la vitesse. Dans ce sens où le noeud de norme la plus grande est généré à la fin. Ensuite les champs parents de ces successeurs pointeront sur le noeud qui les a générés (le noeud source). Pour chacun de ces successeurs, l'algorithme va tester si sa position corrspond à une position destination. Si ce n'est pas le cas et qu'il n'a pas déjà été visité, il sera ajouté à "queue". L'étape suivante consiste à rajouter le noeud source à "colored" et continuer la recherche à partir du dernier élément de "queue". Ce processus sera répété jusqu'à ce que l'algorithme tombe sur un noeud destination et que le chemin parcouru soit valide, ou alors la liste "queue" devienne vide.

Pour décider de la validité du chemin parcouru, une fonction "valid-path" a été implémenté. Elle retrace les neouds parcourues grâce au champs parent, et vérifie que le chemin passe par toutes les zones dans l'ordre croissant. Elle vérifie aussi si toutes les positions sont valides dans la grille. Par contre elle ne vérifie pas le fait que l'on puisse sauter sur des obstacles.

D'autres fonctions auxiliaires, mais très importantes, ont été impléméntés. Notamment "identify-zones" qui identifie les zones d'une map et "order-of-accelerations" qui ordonne les accélérations selon la norme.

2.4.2 Tests et performances

Nous avons testé notre algorithme sur plusieurs grilles de tailles différentes. La Figure 3 montre un exemple de résolution sur une carte de taille 30x30. La liaison avec le module SDL permet de visualiser la validité de la solution proposée par l'algorithme, cependant nous ne disposons d'aucune outil permettant de confirmer l'optimisation.

Généralement, notre solveur propose des solutions instantanée pour des grilles d'ordre inférieur à 100x100. Il faut présiser que notre solver ne vérifie pas si une grille est valide avant de la résoudre.

Nous avons aussi envisgé de tester toutes les parties de notre algorithme. Ou à la limite ses fonctions pricipales. Mais la réalisations de ses tests s'est avérée insuffisante pour en tier des conclusions, puisque nous nous sommes basés sur des cas très particuliers qui ne peuvent être généralisés.

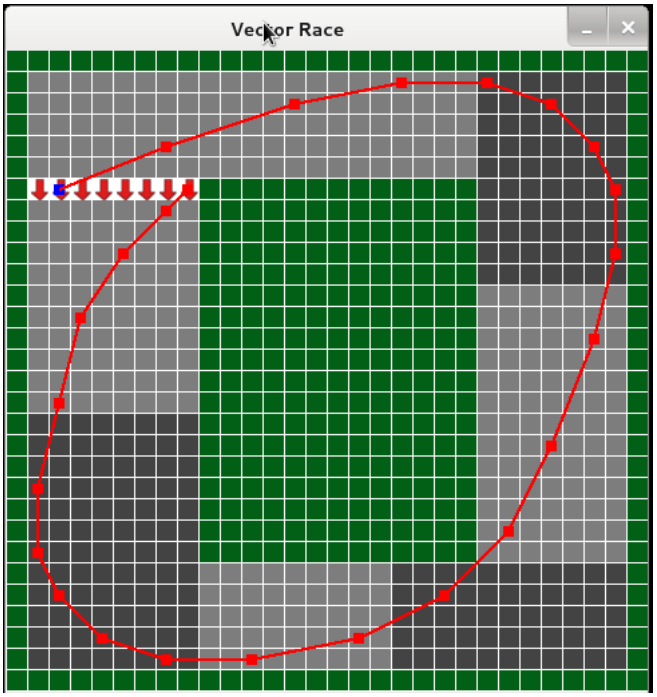

FIGURE 3 – Exemple de résolution

3 REMARQUES ET AMÉLIORATIONS POSSIBLES

3.1 FUITES MÉMOIRE

Malheureusement il reste des fuites mémoire.

La plupart sont dûes à la SDL. Valgrind indique que des allocations de la fonction SDL_Init() n'ont pas été libérées. Notre connaissance de SDL n'étant pas parfaite nous ne savons pas comment régler ces fuites mémoires.

Il reste cependant des fuites mémoires qui correspondent à la libération de la structure map et à la libération de structures dans le solveur. Nous n'avons pu régler ce problème avant la limite de temps malgrès nos tentative de correction. C'est donc un travail qu'il reste à faire.

3.2 IDÉES SUPPLÉMENTAIRES

De nombreuses possibilités d'amélioration existent. En voici quelques unes :

— L'optimisation de la résolution.

— La possibilité en jouant de revenir en arrière sur les choix effectués.

— L'affichage de la solution en parallèle de la phase de jeu.