École Publique d'Ingénieurs en 3 ans

Rapport

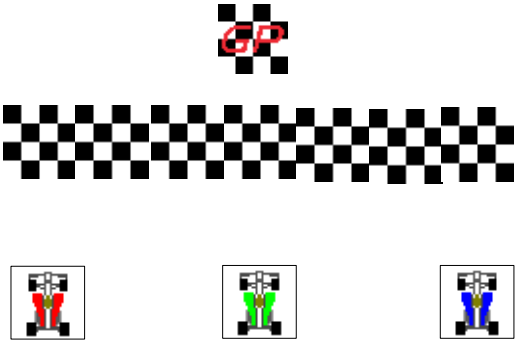
CHALLENGE DE PROGRAMMATION

le 31 mai 2022

Ayoub ED-DAHMANY
ayoub.ed-dahmany@ecole.ensicaen.fr
Youssef AGHZERE
youssef.aghzere@ecole.ensicaen.fr

ENSI CAEN
ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

TABLES DE MATIÈRES



1. INTRODUCTION

Le challenge de programmation est un projet dédié aux étudiants 1A de l'ENSICAEN. Il permet l'application des acquis en algorithmique avancée en utilisant le langage C.

Il s'agit de développer un pilote qui va pouvoir parcourir une carte de course fournie par le gestionnaire pour atteindre l'arrivée.

Pour cela, le pilote doit envoyer les accélérations convenables au gestionnaire de la course en chacune des positions.

Certaines contraintes doivent être respectées afin de rendre le pilote compatible avec le gestionnaire de la course.

2. DÉFINITIONS

2.1. La carte :

Une carte est caractérisée par la longueur, la hauteur, et le carburant réservé pour le pilote ainsi qu'une suite de lignes contenant 4 types de caractères organisés de façon qu'ils construisent un chemin de course.
Les 4 caractères sont les suivants :

. : Il représente le mur infranchissable.
: Il représente la piste que le pilote est vivement conseillé de la préférer.
~ : Il représente le sable qui consomme plus de carburant que la piste, en le parcourant.

= : Elle représente l'arrivée que doit atteindre le pilote.

2.2. Le pilote :

Un pilote est caractérisé par sa position courante, sa vitesse, et son accélération. Il n'échange avec le gestionnaire que les valeurs de l'accélération suivant les deux axes à chaque tour de rôle.

3. DÉMARCHE DE RÉSOLUTION

Notre démarche pour la création du pilote consiste à trouver un chemin optimal en termes de consommation de carburant. À chaque tour, le pilote calcule la vitesse qu'il lui faudra pour arriver au prochain point en calculant la différence des deux points successifs. Sachant qu'il mémorise sa vitesse actuelle, il déduit

donc l'accélération nécessaire en sou tractant sa vitesse actuelle de la nouvelle vitesse.

3.1 CHEMIN OPTIMAL :

Pour trouver le chemin optimal, nous avons utilisé l'algorithme A* (A star), c'est un algorithme itératif qui utilise deux listes, l'une nommée ouverte et l'autre fermée, chaque nœud dans la carte est caractérisé par un parent et un coût, voici un résumé de ses étapes :

- On commence par considérer le nœud de départ dans la carte comme nœud courant.
- On itère sur ses 8 voisins qui l'entourent :
  - Si le voisin est un obstacle, on l'oublie.
  - Si le voisin est déjà présent dans la liste fermée, on l'oublie.
  - Si le voisin est déjà présent dans la liste ouverte, on modifie alors son parent et son coût si ce dernier est inférieur à celui déjà existant dans la liste ouverte.
  - Sinon, on ajoute le voisin dans la liste ouverte avec comme parent le nœud courant.
- On cherche le meilleur nœud de la liste ouverte en termes de coût, Si la liste ouverte est vide, il n'y a pas de solution, fin de l'algorithme.
- On ajoute ce meilleur nœud dans la liste fermée en le retirant de la liste ouverte.
- On recommence avec ce nœud comme nœud courant jusqu'à ce que le nœud courant soit le nœud d'arrivée.

Une fois que la destination est atteinte, il faut retrouver le chemin en suivant à chaque fois les parents des nœuds présents dans la liste fermée. On remonte le fil jusqu'à arriver au point de départ.

Dans notre situation, voici le tableau des couts de chaque nœud voisin selon sa position : (un obstacle est modélisé par un cout infini)

Nœud	Coût correspondant
Voisin latéral dans la piste	0
Voisin latéral dans le sable	100
Voisin diagonal dans la piste	100
Mur	Infini
Voisin diagonal dans le sable	Infini

3.2 CALCUL DES ACCÉLÉRATIONS :

Une fois le chemin trouvé, le pilote calcule donc à chaque tour l'accélération nécessaire pour arriver au point suivant :

Supposons que le pilote est dans le n-iéme point p_n du chemin de coordonnées

x_n et y_n : p_n (x_n, y_n).

La nouvelle vitesse qu'il lui faut pour arriver à p_n+1(x_n+1, y_n+1) est :

v_n+1 = (x_n+1 – x_n, y_n+1 – y_n)

Sachant qu'il mémorise sa vitesse actuelle v_n (v_x, v_y), l'accélération qu'il lui faut est a_n (a_x , a_y) = (x_n+1 – x_n - v_x, y_n+1 – y_n – v_y)

3.3 GESTION DES COLLISIONS :

Le pilote calcule son chemin optimal au premier tour, mais dans la suite de la course, si 2 pilotes décident en même tour d'aller tous les deux au même point, alors l'un des deux sera arrêté par le gestionnaire de course (invalid acceleration) et lui remet sa vitesse à zéro, si c'est le cas pour notre pilote, on doit donc lui remettre sa vitesse à zéro pour ne pas faire de faux calculs dans les prochains tours.

Si en plus, le pilote est resté bloqué dans un point donné car le prochain point est occupé pour une longue durée par un autre pilote concurrent, dans ce cas le pilote doit recalculer un autre chemin en considérant le point occupé comme étant un obstacle.

4. CHOIX DES STRUCTURES DE DONNÉES :

4.1. point

Dans un premier abord, on définit une énumération bol comme suivant : typedef enum {FALSE, TRUE} bol;

Voici la structure point :

```
typedef struct _point{ int x, y;}point;
```

Des fonctions permettant son utilisation sont :

bol isEquals(point* a, point* b); qui vérifie l'égalité de a et b.

point* createpoint_2D(int x, int y); qui retourne un pointeur sur un point.

point* addPoint(point* a, point* b); qui additionne 2 points.

4.2. node

```
typedef struct _node{
    point p;
    int cost;
    struct _node* parent;
    struct _node* next;
}node;
```

La structure node contient un point de la carte, le coût qu'on lui a affecté, un pointeur sur le prochain nœud pour la gestion des listes simplement chaînées, et un pointeur sur le nœud parent pour la retrouvaille du chemin recherché.
La fonction permettant son utilisation est :

node* createNode(point* p); elle retourne un pointeur sur un nœud dont le point correspondant est celui passé en paramètre, son coût, son nœud parent et son nœud prochain sont initialement nuls.

4.3. list

```
typedef struct _list
{node* head;
    int size;
}list;
```

Une liste est un ensemble enchaîné de nœud, la structure contient un pointeur sur la tête de la liste et un entier dans lequel on stocke le nombre de nœud contenu dans la liste.

Des fonctions permettant son utilisation sont :

list* createlist ();

bol isEmpty (list* l);
void add(list* l, node* n); qui ajoute le nœud n après le dernier nœud de la liste l, puis incrémente la taille de cette dernière.
node* search(list *l, node* n);
void add(list* l, node* n);
void delete(list *l, node *n);
node* bestNode(list* l); qui retourne le nœud contenant le coût minimal dans la liste.
void add_adjacent_squares(list* openList, list* closedList, node *n, map *m); qui cherche les 8 voisins du nœud n dans la carte m, puis teste les 4 conditions de l'algorithme A* citées dans la troisième page.
void addClosedList(node *n, list *openList, list *closedList); qui ajoute le nœud n à la liste closedList en le retirant de openList.

4.4. Map

```
typedef struct _map
{
    int height, width, gas;
    char** tab;
}map;
```

Cette structure stocke les données d'une carte à savoir la hauteur, la longueur, le carburant et un tableau de 2 dimensions qui contient les caractères composants la carte.

Des fonctions permettant son utilisation sont :

map* loadMap(char* f);

bol isRoad(node *n, map *m);
bol isSand(node *n, map *m);
bol isWall(node *n, map *m);
bol isFinish(node *n, map *m);

4.5. Path

```
typedef struct _path{
point* traject;
int size;
}path;
```

Cette structure permet le stockage du trajet qui sera un tableau de type point, et un entier size qui contient le cardinal de ce tableau.

Des fonctions permettant son utilisation sont :

path* findPath(point* begin, map* m);

C'est la fonction qui englobe l'ensemble des étapes de l'algorithme A*.

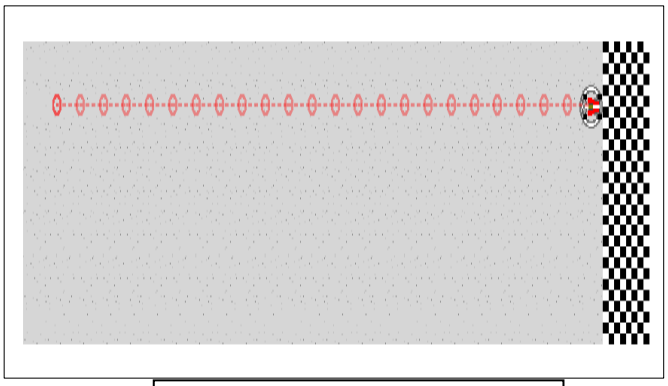
Elle fait appel à la fonction add_adjacent_squares

path* reconstructPath(list *traject);
Cette fonction complète la tache de findPath, elle donne, à partir de la liste retournée par la précédente, un pointeur sur path *pa dont le pa→traject contient l'ensemble des points bien ordonnés.

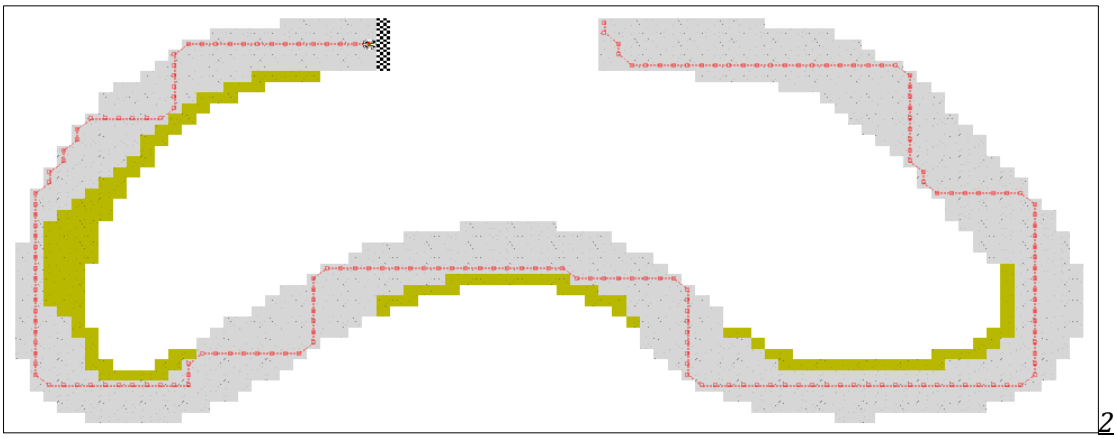
5. RÉSULTATS D'EXÉCUTION :

Voici l'application de notre pilote sur quelques cartes de différentes difficultés.

1.Chemin tracé par le caractère '7'

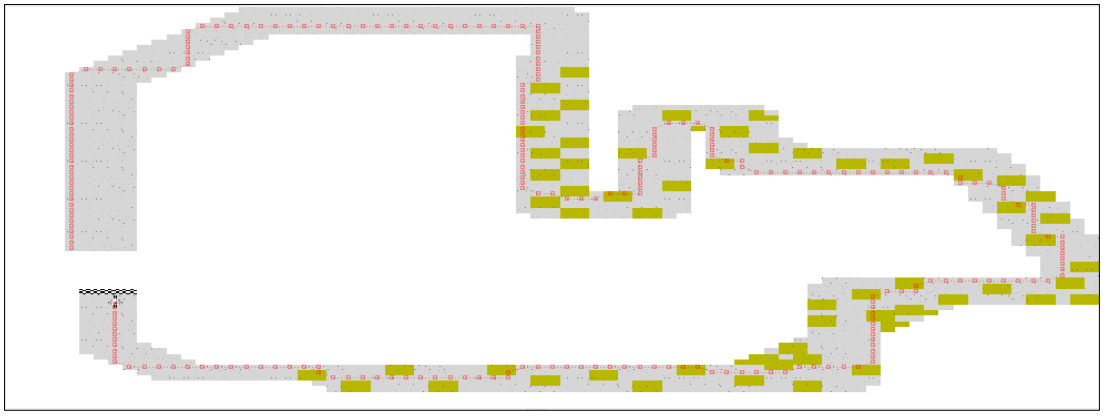


2.La carte droit au but (facile)



2

3. starter_virage_sable (difficile)



4. f-zero_landmine(moyenne mais astucieuse)

6- OPTIMISATION :

Pendant l'exécution, nous avons remarqué que le pilote lui reste toujours du carburant non utilisé, cela est dû au fait que le chemin retourné par la fonction findPath est constitué d'un ensemble des points adjacents, cela signifie que la différence entre deux points quelconque est un point dont les coordonnées ne sont jamais supérieurs à 1, donc, pendant toute la course, la vitesse du pilote est la vitesse minimale avec laquelle il peut arriver, ce qui nuit bien sûr à ses performances en terme de temps d'arrivée.

Pour cela, nous avons décidé d'optimiser le pilote en lui donnant le droit au cours de la course de retirer de son chemin quelque points bien soumises à des conditions :

1/ Son point actuel n'est pas du sable (pour que la norme de sa vitesse ne dépasse pas 1 dans le sable).

2/ Les prochains points seront alignés (On peut régler le nombre de points à notre guise).

3/ Le carburant restant est suffisant pour terminer le trajet :

Soit Ci : le carburant initial donné au pilot, Ni : le nombre de points du trajet.

Et soit Ri le rapport suivant : Ri = Ci / Ni,

Ri est donc une constante qui caractérise chaque carte et son trajet.

Soit, à un instant t :

C(t) le carburant qui reste au pilote.

N(t) le nombre de points qui lui restent à parcourir dans son trajet.

Et soit R(t) = C(t)/N(t)

Nous avons pu déduire expérimentalement le fait que si R(t) > (5/3)Ri
alors le pilote lui reste suffisamment du carburant pour terminer le trajet.

Donc en comparant à chaque tour son rapport instantané au rapport initial, le pilote peut lui-même décider s'il doit accélérer encore ou non.

Voici une comparaison, dans la carte f-Zero-Death-Wind, entre l'ancienne version et la version optimisée de notre programme :

On remarque, à partir de la deuxième moitié de la carte, le pilote voit qu'il lui reste suffisamment du carburant et augmente sa vitesse.

sans optimisation

Rounds : 410

Avec optimisation
Accélération à la 2ème moitié du trajet
Rounds : 330

7. CONCLUSION

Ce projet a été une occasion extrêmement importante dans notre cursus en tant qu'étudiants en première année en informatique. En effet, il englobe non seulement la recherche d'un algorithme convenable et son implémentation en langage C, mais aussi un bon entraînement en gestion de projet.

ENSI CAEN

ÉCOLE PUBLIQUE D'INGÉNIEURS
CENTRE DE RECHERCHE

École Publique d'Ingénieurs en 3 ans

ministère éducation nationale enseignement supérieur recherche

Normandie Université

Cti Commission des titres d'ingénieur

cdefi Conférence des Directeurs des Écoles Françaises d'Ingénieurs

European Accreditation of Engineering Programmes EUR-ACE®

CONFÉRENCE DES GRANDES ÉCOLES