Projet : Algorithmique et Programmation

Faculté des Sciences et Techniques
Master 1 - Maths. CRYPTIS

Rush Hour

A l'attention de :
M. Arnault
M. Dusart

Rédigé par :
CARVAILLO Thomas
FAURIAT Mattieu
JACQUET Raphaël
PIARD Arthur
PONS Hugo

Table des matières

Introduction

Le Rush Hour est un jeu casse-tête de déplacement, simulant une congestion automobile dans un parking. Le parking est modélisé par une grille, qui est de taille 6x6 et les véhicules occuperont, suivant leur type, un nombre différents de cases. Le but du jeu est de faire sortir la voiture rouge en déplaçant les autres véhicules.
Ce projet consiste à programmer le jeu en langage $\mathcal{C}$.
La dimension du plateau choisi est de 6x6 et la sortie se trouve en la coordonnée $(3\,;\,6)$. Les autres véhicules sont soit des voitures, qui occuperont 2 cases, soit des camions, qui occuperont quant à eux 3 cases. Ils ne peuvent se déplacer que dans une direction, horizontale ou verticale, d'un nombre quelconque de cases ; dans la limite du plateau.

1 Quelques généralités

Le fil conducteur du projet est le suivant :
Nous avons commencé par créer la structure fondamentale de notre code, le *struct vehicule*. Cette structure contient en dernier élément un pointeur vers le véhicule suivant. Les véhicules sont ainsi représentés par une liste chaînée.
Ensuite, nous avons dû décider du type de plateau, c'est-à-dire un tableau à une dimension ou une matrice... Notre choix s'est porté sur un tableau à une dimension composé de caractère, contenant 36 éléments - les cases -. L'affichage utilise les caractères "|" et des soulignements pour délimiter le plateau et les cases entre elles, ainsi que des espaces pour représenter les cases vides.
La suite découle naturellement : nous avons mis en place, plus ou moins en parallèle et suivant les affinités de chacun, les fonctions effectuant les opérations suivantes :

- déplacement des véhicules,
- améliorations de l'affichage pour améliorer la visibilité,
- détections des collisions et/ou des déplacements illicites,
- lire et charger en mémoire des cartes de jeu,
- résoudre le problème, dire si une carte admet ou non une solution,
- détecter la fin du jeu,

dont nous présenterons les détails prochainement.
Pour plus de maniabilité le programme est capable de lire les entrées claviers pour les déplacements des véhicules.

2 Le coeur du projet : le code et ses explications

Nous détaillerons dans cette partie les fonctions constituant notre projet. Nous entrerons dans les détails pour certaines d'entre elles et exhiberons les différents problèmes rencontrés, ainsi que les solutions trouvées.

2.1 Fonction *deplace(char*, char, int)*

Cette fonction permet, comme son nom l'indique, de déplacer un véhicule sur la grille, tout en gérant les éventuels problèmes de collisions.
Elle prend pour premier paramètre un *char**, correspondant au nom du véhicule que l'on souhaite déplacer. Le second correspond à la direction, H ou B pour un déplacement vertical et D ou G pour un déplacement horizontal. Le dernier paramètre correspond quant à lui au nombre de cases du déplacement.
Cette fonction est une de celles qui nous ont présentées le plus de problèmes lors de la conception. Au départ, les origines de nombreux problèmes de taille de déplacements, de collisions, de chevauchements ou même de sorties du plateau restaient incomprises.
Regardons un peu plus en détails son fonctionnement.

L'idée est de parcourir la liste chainée des véhicules jusqu'à arriver sur celui correspondant au nom entré en paramètre. Pour cela, la fonction *strcmp* est utilisée, elle retournera 0 si les deux chaînes sont égales, 1 sinon.

```
1  char deplace(char * name, char dir, int nbrDep) {
2      while(nbrDep < 0 || nbrDep > 6){
3          printf("%d n'est pas accepté car il n'est pas compris entre 0 et
       6. Choisissez un nouveau nombre !\n",nbrDep);
4          scanf("%d", &nbrDep);
5      }
6      int verifDeplacement = 0;
7      vehicule *ptr = head;
8      int distMax = 0;
9      while (ptr != NULL){
10         if(strcmp(ptr->name , name) == 0){
```

On fait ensuite une disjonction de cas suivant la direction du véhicule. Nous présenterons ici le cas où la direction du véhicule est *droite*, i.e. *ptr->dir == 'D'* ; les autres cas étant, excepté pour la variable *distMax*, analogues.
La variable *distMax* est ensuite décidée en fonction de l'orientation du véhicule et du paramètre de direction entré.
Les problèmes liés à un déplacement trop important ou à une direction non autorisée sont immédiatement gérés.

```
1           if (ptr->dir == 'D'){
2               distMax = 6 - (ptr->y + ptr->type);
3               if(ptr->dir != dir)
4                   distMax = 6 - (distMax + ptr->type);
5               if(nbrDep > distMax){
6                   ThrowAnException("Vous sortez de la grille !\n");
7                   break;
8               }
9               if(dir == 'H' || dir == 'B'){
10                  ThrowAnException("Le véhicule ne va pas dans cette
    direction !\n");
11                  break;
12              }
```

Une nouvelle comparaison entre la direction du véhicule et celle entrée est nécessaire. Par exemple, il est évident que le déplacement vers la gauche d'un véhicule orienté à gauche ne s'effectuera pas de la même manière que le déplacement à gauche d'un véhicule orienté vers la droite.
Quelque soit la direction, il est en premier lieu géré le problème de collision : on vérifie si les cases sont «libres», i.e. si elles contiennent un caractère vide.
Il va maintenant entrer en jeu la variable *verifDeplacement*. Initialisée à zéro, elle est incrémentée de 1 chaque fois qu'une case est considérée comme libre.

```
1               if(ptr->dir != dir){
2                   for (int i = 1 ; i<nbrDep+1 ; i++){
3                       if(grid[ptr->y - i + 6*ptr->x] != ' '){
4                           ThrowAnException("Déplacement impossible, un vé
    hicule se trouve sur votre chemin !\n");
5                           return grid[ptr->y - i + 6*ptr->x];
6                       }
7                       else
8                           verifDeplacement += 1;
9                   }
10              }
11              else{
12                  for (int i = 0 ; i<nbrDep ; i++){
13                      if(grid[ptr->y + ptr->type + 6*ptr->x + i] != ' '){
14                          ThrowAnException("Déplacement impossible, un vé
    hicule se trouve sur votre chemin !\n");
15                          return grid[ptr->y + ptr->type + 6*ptr->x + i];
16                      }
17                      else
18                          verifDeplacement += 1;
19                  }
20              }
```

Si elle est supérieure ou égale au nombre de déplacements souhaité, le déplacement est possible et réalisé. Sinon, cela signifie qu'il y a un problème de collision, le déplacement n'est pas réalisé.

```
1               if (verifDeplacement >= nbrDep){
2                   if(ptr->dir != dir)
3                       ptr->y -= nbrDep;
4                   else
5                       ptr->y += nbrDep;
6               }
```

2.2 Fonction *dlconfig(int)*

Cette fonction lit et charge en mémoire la configuration d'une carte stockée dans un fichier annexe, selon son numéro. Pour ce faire, les cartes doivent être présentées sous une forme bien spéciale, illustrée par l'exemple suivant :

```
1 Yellow 2 0 2 D
2 Orange 3 3 5 H
3 Magenta 2 5 2 G
4 Fuchsia 3 1 3 B
5 Red 2 2 0 D
```

Chaque ligne correspond à la disposition d'une voiture sur le plateau. Elle commence par la couleur de la voiture (la première lettre sera choisie pour représenter le véhicule sur la grille). Ensuite, séparé par un espace, vient la taille du véhicule (2 pour une voiture, 3 pour un camion). Puis, toujours séparé par un espace, les deux nombres suivants seront respectivement la coordonnée en X et en Y de la tête du véhicule. Enfin, après un dernier espace, se trouve une lettre qui donnera la direction du véhicule (H pour haut, B pour bas, D pour droite et G pour gauche).
A noter que le fichier texte doit comporter une ligne vide pour permettre la lecture totale de la carte.
Pour le fonctionnement on initialise d'abord le nom du fichier que l'on voudra charger à « Carte X », ensuite on remplace le X par l'argument de la fonction qui est le numéro de la carte voulue (par exemple « dlconfig(2) » chargera le fichier « Carte 2 »).

```
1 void dlconfig(int n)
2 {
3     char fichier[] = "Carte X";
4     fichier[6]=n + '0';
5     FILE* config=fopen(fichier,"r");
6     char ligne[200];
```

On crée aussi un compteur qu'on mettra à 0.

```
1       int compteur=0;
```

L'idée est de parcourir les lignes du fichier une à une, et d'insérer au fur et à mesure les véhicules sur la grille. La lecture du fichier s'arrêtera après la ligne vide que l'on aura laissée au préalable.
Ici, la fonction *sscanf()* permet de récupérer à la fois la couleur, la taille et les coordonnées du véhicule. La direction étant récupérée à part, il ne reste plus qu'à insérer le véhicule avec la fonction insert.

```
1       while(fgets(ligne,200,config) != NULL)
2       {
3           char dire;
4           int taille, abs, ord;
5           sscanf(ligne, "%s%d%d%d", listecouleurs[compteur], &taille, &
      abs, &ord);
6           dire=ligne[strlen(ligne)-2];
7           insert(listecouleurs[compteur],taille,abs,ord,dire);
8           compteur += 1;
9       }
```

A la fin de la fonction, on referme le fichier texte.

```
1       fclose(config);
```

2.3 Fonction *resolution()* et *libcase()*

Cette fonction tente de résoudre une configuration donnée. Commençons par la fonction *resolution()*.
On commence par définir le int *geneur*, qui servira à stocker ce que renverra le déplacement, i.e. percuter un mur, un autre véhicule ou un déplacement réussi. *Vgen* sera utilisé pour stocker le nom du véhicule que l'on aura percuté.

```
1   void resolution2(){
2       int geneur = 0;
3       char* Vgen=(char*) malloc(15*sizeof(char));
```

On entre ensuite dans une boucle, qui tournera tant que le jeu ne sera pas considéré comme gagné. Elle avance le plus possible la voiture rouge vers la droite, jusqu'à ce qu'il y ait collision.

```
1       while(!ended()) {
2           while(geneur == 0){
3               geneur = deplace("Red" , 'D' , 1);
4               grille();
```

```
5                 printf("%d\n", geneur);
6           }
```

Si on a pas directement gagné, on récupère le nom du véhicule "géneur", et la fonction libcase est exécuté.

```
1           if (geneur < 37){
2               Vgen = assimilateur(geneur);
3               printf("%s\n",Vgen);
4               libcase(geneur);
5           }
```

Détaillons maintenant la fonction *libcase(int a)* Elle prends pour argument la position du tableau où la collision a eu lieu ; son but étant de libérer la case de la dite collision.
Tant que la case n'est pas libérée, on va déplacer les autres véhicules afin de la rendre libre.

```
1       while (grid[a] != ' '){
2           if (geneur != 0)
3               V = assimilateur(a);
4           int d = 1;
```

2.4 Fonction *insert(char*, int, int, int, char)*

Cette fonction sert à insérer un véhicule dans la liste chaînée. Les arguments correspondent aux caractéristique du véhicule que l'on souhaite insérer, et sont respectivement : sa couleur, son type (voiture ou camion), les coordonnées x et y de sa tête, et sa direction (horizontale ou verticale).
Originellement, cette fonction insérait le véhicule «sans réfléchir», c'est à dire ne gérait pas le fait qu'une case était déjà occupée par un autre véhicule. Nous avons donc, dans un premier temps réglé le problèmes en insérant le véhicule que si la place était libre, mais ensuite, nous somme revenu à notre première version. La fonction *insert()* étant utilisée avec la lecture de carte, on obtient une erreur seulement si la carte est erronée.

2.5 Fonctions *grille()* et *couleur(char)*

Ce sont deux fonctions utilisées pour l'affichage.
La fonction *grille()* génère le plateau du jeu et l'affiche. Nous avons hésité entre un tableau unidimensionnel, constitué de 36 caractères, et un tableau bidimensionnel représentant une matrice de taille 6x6. Nous avons choisis la première option, cette dernière étant plus simple à utiliser.

Quant à la fonction *couleur()*, elle permet de changer la couleur du véhicule en fonction de la première lettre de celui-ci, et donc d'ajouter des couleurs dans la sortie console pour rendre l'affichage plus élégant, dans la mesure de ce que permet le C.

En premier lieu est généré dynamiquement le tableau de caractères : *grid*. On occupe toutes les cases par un caractère vide ' '.
On va ensuite parcourir tous les véhicules et un *switch* est utilisé en fonction de la direction de celui-ci.
Nous concentrerons les explications sur le cas *case 'D'*, les autres fonctionnant de manière similaire.

```
1  void grille()
2  {
3      grid = (char*) malloc(36*sizeof(char));
4      vehicule *ptr = head;
5      char c = ' ';
6      for(int i=0;i<36;i++)
7      {
8          grid[i]=c;
9      }
10     while(ptr != NULL)
11     {
12         grid[6*(ptr->x)+(ptr->y)]=*(ptr->name);
13         switch(ptr->dir)
14     ...
```

Le type du véhicule est ensuite considéré, et suivant cela, une ou plusieurs cases sont occupées par la première lettre de la couleur du véhicule.

```
                case 'D':
                {
                    switch(ptr->type)
                    {
                        case 2:
                        {
                            grid[6*(ptr->x)+1+(ptr->y)]=*(ptr->name);
                            break;
                        }
                        case 3:
                        {
                            for (int i = 1; i < 3; ++i)
                            {
                                grid[6*(ptr->x)+i+(ptr->y)]=*(ptr->name);
                            }
                            break;
                        }
                    }
                    break;
                }
```

La suite correspond à l'affichage de la grille. Il est utilisé des simples *printf()*, agrémentées par des espaces et des balises pour les couleurs ; et des appels à la fonction *couleur()*.

2.6 Fonction *ThrowAnException(char*)*

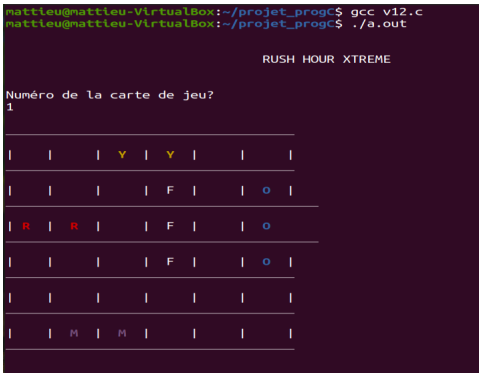
Cette fonction sert à afficher les éventuelles erreurs lors de l'exécution de certaines fonctions. Ces erreurs peuvent être des problèmes d'insertions, de déplacements non autorisés, etc.

2.7 Fonction *ended()*

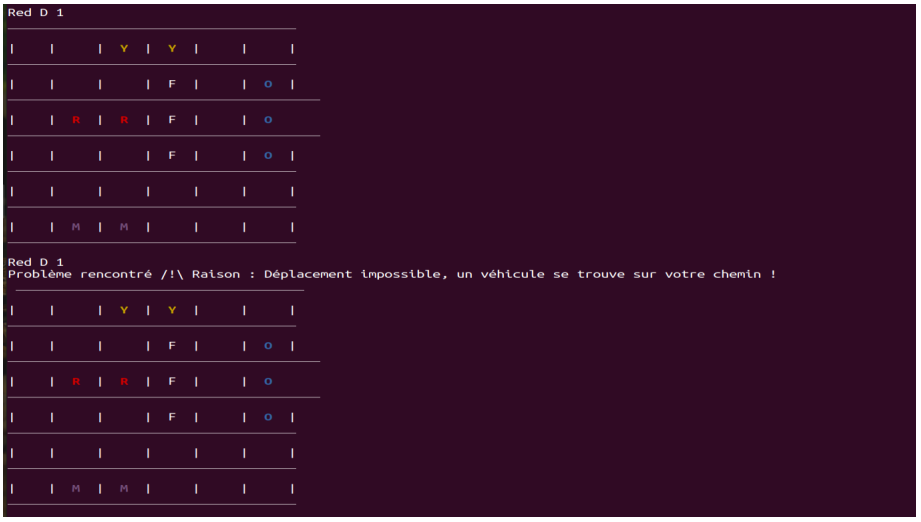
Cette fonction renvoie *false* tant que l'on n'a pas gagné le jeu, i.e. tant que la voiture rouge n'est pas arrivée à la sortie. Une fois le jeu gagné, un message de félicitations est affiché.

3 Jeu d'essais

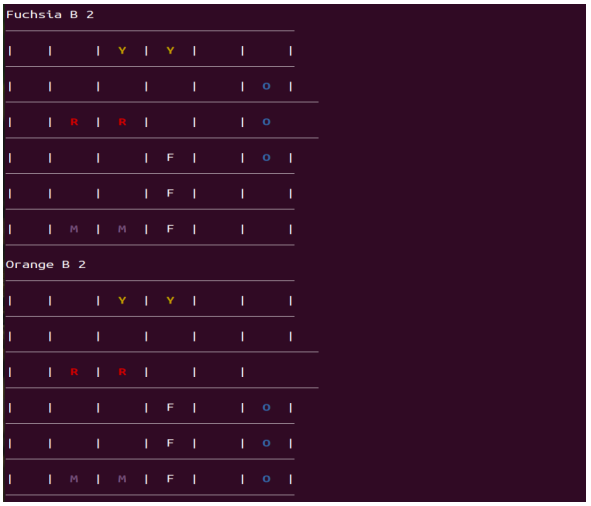
Chargement de la carte 1.



Déplacement de la voiture Rouge de 1 case vers la droite. Le deuxième déplacement illicite est levé.

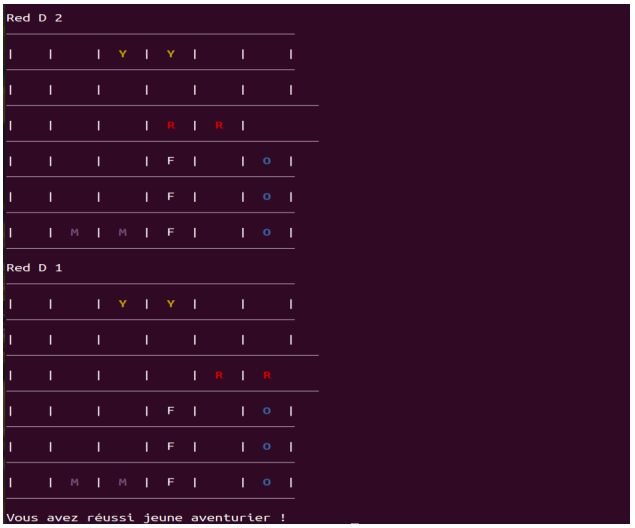


Déplacements des camions Fuchsia et Orange pour libérer le passage.



La voiture Rouge peut enfin sortir du plateau. Un message apparaît lorsque la partie

est gagnée.



Dans les deux cas les déplacements illicites sont repérés, et rien ne se passe.

Conclusion

Bien que les fonctions ne soient pas des plus optimisées, le travail demandé a été réalisé au mieux. Les optimisations concernent principalement la gestion des disjonctions de cas et l'utilisation des *switch*. La fonction *resolution()* n'est pas totalement aboutie. Elle est fonctionnelle dans une majorité des cas, mais les cartes les plus compliquées font boucler l'algorithme. Il a été mis en place une certaine intéraction, la fonction *deplace()* permet de rendre le jeu jouable, c'est à dire de gérer manuellement les déplacements.
Concernant l'organisation du temps de travail et de la répartition des tâches, chacun a apporté sa pierre à l'édifice selon ses compétences (recherche des idées, facilitées pour implémenter ces dernières en C, etc...). Ne pouvant nous voir en présentiel, de nombreuses sessions ont été organisées - en partage d'écran - sur notre précieux allié

en période de confinement : Discord.
Finalement, nous sommes satisfaits du rendu final, malgré le non-aboutissement de la fonction *resolution()*.