Modélisation Projet C – Graphes et applications

Tristan Ngounou 2A-24

Christophe Guillou 2A-23

Problèmes Rencontrés :

Partie Front :

Problème principal réside dans l'adaptation de notre structure pour afficher le graph via un fichier jSON. En effet, M. Yolin nous a laissé un lien vers son GitHub où le code pour afficher le graph était expliqué. M. Yolin a utilisé une structure propre à lui mais dès que nous essayer d'adapter justement ses fonctions avec notre structure, la compilation bug. Le point dérangeant est que nous n'avons aucune indication concernant l'erreur. Nous avons juste les informations concernant la librairie Raylib.

De plus, la transition pour passer de la page d'accueil au labyrinthe puis du labyrinthe au graph avec la librairie Raylib. Nous avons fixé le problème grâce au type Enum. Nous avons ajouté les états des différentes fenêtres nécessaires. Le labyrinthe de l'étape 1,2,3,4 ainsi que les graphes de l'étape 1,2,3,4. Lorsque l'utilisateur clique sur l'option qu'il souhaite (lab1, graph1…), l'état de la fenêtre est mise à jour. Enfin, une condition suffit pour nous rediriger ensuite vers la fenêtre souhaitée.

Ensuite, nous avons dû faire face au problème concernant la relation utilisateur, interface. Lorsque l'utilisateur entrait son résultat, deux possibilités apparaissaient. La réponse correcte ou bien un mauvais résultat. Donc on à afficher « WRONG » et « RIGHT » en fonction des cas. Mais le problème a commencé à se corser lorsque nous sommes passé au niveau deux car la box de réponse de situait au même endroit dans les deux fenêtres distinctes. Donc quand le résultat du niveau1 était bon, s'affichait le RIGHT du niveau 1 et le WRONG du niveau 2. Pour pallier ce problème, nous avons séparé les fonctions par niveau et nous les avons insérés dans la condition du changement de fenêtre.

Enfin, un autre problème concernant les fichiers Json. M. Yolin nous avait conseillé de changer de format le fichier graph.json car il était sur une seule ligne et donc pour des questions d'efficacité, c'était mieux pour nous d'avoir le fichier sur plusieurs lignes. Après cela, nous avons essayé de créer un autre fichier Json mais cette fois ci avec notre graphe. Et donc vient le problème, la main ne compile pas et il y a une erreur de fichier. Or, les variables utilisées étaient déclarées correctement, les espaces étaient supprimés et les accolades bien indentées. Ce problème s'est étalé sur plusieurs jours car plusieurs groupes étaient dans le même cas que nous. Nous avons donc compris par la suite que les fichier Json compile uniquement lorsque le code est écrit sur une seule ligne. Nous sommes donc aller sur internet chercher des convertisseurs de fichier Json en une seule ligne et le graph s'est afficher.

Partie Back :

Afin d'optimiser la réalisation du back-end, nous avons procédé étape par étape. En premier lieu, il nous était demandé de vérifier la réponse de l'utilisateur lors de la résolution du plus court chemin en utilisant différent algorithmes (niv.1 = Bellman, niv.2 = Dijkstra, etc..)

Le premier objectif a donc été d'implémenter ces algorithmes vus en cours en C, et de les tester sur le premier labyrinthe. Le premier algorithme que nous avons implémenté est celui de Dijkstra, nous n'avons pas eu de problème particulier au niveau de l'implémentation.

Cependant, lors de l'implémentation du 2^{e} algorithme Bellman, nous avions remarqué que les 2 algorithmes donnaient des résultats différents ce qui n'était pas normal, nous avons alors commencé à investiguer là-dessus. Il s'avérait que la cause du problème provenait de notre structure de graphe, où un champ « adjency » servait à stocker la matrice d'adjacence des sommets pondérés du graphe correspondant au labyrinthe.

Le problème venait de l'équivalence entre 2 structures utiles pour les algorithme : celle de l' « adjency » et celle de la « arcsList » (liste des arcs stocké dans un tableau 1D) On a rapidement vu que cela venait de l'indexation des sommets dans le tableau 2D, effectivement, pour l'algorithme de Dijkstra, nous utilisions la matrice d'adjacence mais pour Bellman, uniquement la liste des arcs. Tandis que dans la liste des arcs, le premier sommet commençait par 1 (exemple : 1->3), dans la matrice d'adjacence, le premier sommet commençait à 0.

Normaliser les 2 structures de données nous a donc permis de résoudre ce problème et donc d'obtenir les mêmes résultats entre les 2 algorithmes.

Le second objectif était de travailler sur les différentes épreuves, en analysant le sujet, nous avons rapidement compris que la première épreuve serait de concevoir un algorithme de coloration d'arêtes (ennemis), et non de sommets comme l'algorithme de Welsh & Powell le permet. Nous avons donc fait nos recherches sur internet, sans trouver d'algorithme mais uniquement des hypothèses mathématiques, telles que le maximum de couleurs (ennemis) était (<= maximum(degré de sommet)) + 1 dans le graphe donc dans un labyrinthe, cela correspond à 5 couleurs.

Nous avons donc conçu un algorithme glouton permettant de colorier au fur et à mesure les différentes arêtes, le processus est assez simple : pour chaque arc (arête) dans le graphe (après avoir supprimé tous les doublons ex : 1->2 et 2->1), nous récupérons tous les arcs adjacents et vérifions s'ils ont déjà une couleur attribué puis nous lui attribuons une couleur qui n'a pas déjà été attribué à ses voisins. Cet algorithme ne permet pas de couvrir toutes les possibilités mais en essayant avec comme sommet de départ chacun des sommets du graphe, nous pouvons conclure à une solution assez fiable avec une complexité assez fable de

l’ordre d’un O(n²) ce qui est assez faible par rapport à une autre solution que nous avions étudié en utilisant un algorithme récursif, qui s’élevait à O(n !), cependant, cet algorithme aurait couvert toutes les solutions possibles.

Voici le processus que suit notre algorithme pour attribuer un type d’ennemi distinct à chaque arc adjacent.

```
Color of edge (1 -> 2): Red
Color of edge (1 -> 3): Green
Color of edge (2 -> 4): Green
Color of edge (2 -> 10): Blue
Color of edge (2 -> 11): Yellow
Color of edge (3 -> 4): Red
Color of edge (3 -> 16): Blue
Color of edge (4 -> 5): Blue
Color of edge (5 -> 6): Red
Color of edge (5 -> 14): Green
Color of edge (6 -> 12): Green
Color of edge (6 -> 13): Blue
Color of edge (7 -> 10): Red
Color of edge (8 -> 9): Red
Color of edge (9 -> 10): Green
Color of edge (9 -> 15): Blue
Color of edge (10 -> 11): Purple
Color of edge (11 -> 12): Red
Color of edge (12 -> 13): Yellow
Color of edge (13 -> 14): Red
Color of edge (14 -> 15): Yellow
Color of edge (14 -> 16): Purple
Color of edge (15 -> 16): Red
```

Structure de données :

Dans ce projet, nous avons utilisé beaucoup de structures de données différentes :

- Listes chaînées
- Tableaux/Tableaux 2D
- Structures

L’utilisation des structures nous a permis de matérialiser un graphe en C : structure Graphe, structure Arc et les structures pour les listes chaînées.

La structure Graphe se compose des champs suivants :

- Son ordre (int)

- Son nombre d'arcs (int)
- La matrice d'adjacence pondérée (int**)
- Un tableau des arcs (arc*)
- Une liste chaînée des arcs (ListSuccessor*)

```
typedef struct Graphe {
    int order;
    int nbArcs;
    int **adjency;
    Arc *arcs;
    ListSuccessor *arcsList;
} Graphe;
```

Le tableau des arcs a été utilisé dans l'implémentation de l'algorithme de Bellman.

La matrice d'adjacence pondérée a été utilisée dans l'implémentation de l'algorithme de Dijkstra.

La liste chaînée des arcs a été utiliser pour la création de l'algorithme de coloration des arcs, ce qui est plus pratique qu'un tableau 1D pour effectuer des changements dedans.

La structure Arc se compose des champs suivants :

- Sommet 1 - Sommet 2 (int)
- Son poids (int)
- Sa couleur (int) -> (pour l'algorithme de coloration)
- Un identifiant unique pour les comparer plus facilement (int)

```
typedef struct Arc {
    int sommet1;
    int sommet2;
    int weight;
    int color;
    int id;
} Arc;
```

Utilisée dans la plupart des algorithmes du projet.