

INFO-F-203 – Algorithmique 2

Manipulation de graphs

Pakkuman

Gérard TIO NOGUERAS Alexis DEFONTAINE

18/12/2014

1 Introduction

Ce projet d’algorithmique de deuxième année porte sur les graphs et sur la recherche du plus court chemin dans un graph. Il est demandé aux étudiants de concevoir un programme capable de traduire un fichier texte en une structure représentant le labyrinthe et ensuite de trouver le plus court chemin entre l’élément *Pakkuman* et la sortie, en prenant en compte des cases bloquante dites ”monstre” opposées aux cases ”bonbons”.

2 Parsage, création du graph et son optimisation, algorithme du plus court chemin et affichage des différentes situations

Le projet a été divisé en 5 étapes : le parsage du fichier, la création du graph et son optimisation, l’algorithme du plus court chemin et finalement son affichage initial/final.

2.1 Parsage

Cette section consiste en l’analyse du fichier contenant toute l’information nécessaire pour lancer le projet. Cette analyse va se diviser en 2 grosses parties : Le parsing de la structure du labyrinthe et du parsing des éléments contenus dans le labyrinthe. Avant

de lancer ces deux parties on initialise une matrice qui va contenir les cases du labyrinthe ainsi que ses murs d'où sa taille qui vaut `matrice[2*lines+1][2*columns+1]`, *lines* et *columns* étant parsés sur la première ligne du fichier.

2.1.1 Structure du labyrinthe

La fonction qui va s'occuper de remplir la matrice initialisée au début est `maze_parser()`, cette fonction va lire caractère par caractère pour chaque ligne et chaque caractère va être analysé avec la fonction `analyse_case` qui va renvoyer un entier qu'on va introduire dans notre matrice.

2.1.2 Les éléments contenus

Après le passage du labyrinthe, on s'attaque au passage de Pakkuman, des monstres et des bonbons. On parse tout d'abord le nombre de monstres et de bonbons contenus, on lit la ligne contenant cette information, on la split par l'espace qui se trouve après les : et on garde le 2ème element qui contient le nombre désiré. Ensuite on attaque les coordonnées :

- **Coordonnée Pakkuman** : On commencer par split la ligne par l'espace qui se trouve après les : , ce qui nous laisse avec (x,y) ensuite on remplace les parenthèses par rien (équivaut à les supprimer) et on split ce qu'il reste par , et on obtient les coordonnées désirées.
- **Coordonnées Monstres et bonbons** : C'est exactement la même idée que pour la coordonnée de Pakkuman si ce n'est qu'on va supprimer toutes les parenthèses et ensuite effectuer la même méthode pour le nombre de bonbons et monstres (parsé plus haut dans la méthode).

On va directement remplir les coordonnées de la matrice par la valeur parsée lui correspondant pour éviter des boucles inutiles.

2.2 Optimisation et création du graph

2.2.1 Graph

Cette section constitue l'analyse du labyrinthe représenté sous forme de matrice et la mise en place des nœuds et arrêtes formant le dit graphe. Le principe développé ici est celui du *backtraking* servant de base au parcours complet de la matrice. Les nœuds représentent ici les monstres et bonbons, le pakkuman, la sortie et les intersections présentes dans le labyrinthe. Ces nœuds sont enregistrés dans un dictionnaire dont les clés représentent leur position dans la matrice.

Noeud et arc sont des objets qui contiennent leurs propres méthodes et attributs. Les arcs sont enregistrés dans une liste appartenant aux deux nœuds concernés. Afin de ne traiter qu'une seule donnée au lieu de deux correspondant aux coordonnées de la matrice nous avons créé une position dite *cryptée* permettant de ne gérer et stocker qu'une donnée pour chaque position testée.

2.2.2 Création du graph

Une multitude de méthodes interviennent pour la création du graph. Hormis certains détails nécessaires à la compréhension du code, seul `create_graph()` et `optimisation_graph()`, constituant le cœur même de la class, seront détaillés.

Etape initiale : Comme cité plus tôt, nous appliquons un *backtracking* sur la matrice contenant l'ensemble du labyrinthe. Nous effectuerons `create_graph()` uniquement si la position actuelle reçue en paramètre n'est pas un nœud ayant déjà été traité. Cette condition est importante, elle évite que le traitement d'un nœud se fasse plus d'une fois, dans le cas par exemple d'un chemin effectuant un retour sur lui même. Une fois cette condition passée, une première méthode sera appelée `data_ofPosition()`. Cette fonction va récolter les informations propres à la position actuellement traitée en rapport avec les quatre directions (Nord, Sud, Est, Ouest) vers lequel on pourrait se diriger. Il est important ici de préciser que la matrice contient aussi les murs présents dans le labyrinthe, les positions testées ici se font par saut de deux cases évitant ainsi les lignes et colonnes dédiées uniquement à la présence ou non de murs. On récupère donc une liste contenant 1 ou 0 pour chaque direction possible, le dernier élément nous permet de savoir si plusieurs directions sont possibles indiquant immédiatement si c'est un nœud. Cette première étape est commune à toute position testée.

Nouveau nœud : On sait donc savoir si la position possède plusieurs directions possibles via la liste que l'on vient de récupérer. Un deuxième test sera effectué au cas où la position ne soit pas multidirectionnelle afin de récupérer l'élément présent dans la matrice, un monstre ou un bonbon par exemple.

- Si cette condition est remplie par l'un ou l'autre test, on fait appel au constructeur de *Node* et on clôture l'arc courant pour le connecter au nouveau nœud créé. Ce dernier élément ne se fait que si on ne situe pas dans le premier appel à `create_graph()` qui n'a nullement besoin de clôturer un arc inexistant.
- Sinon on doit simplement ajouter la position actuelle à l'arc courant.

Traitement des directions possibles : On retourne dans le tronc commun ou l'on traitera les différentes directions vers lequel on peut se diriger à partir de la position actuelle. On tombe alors sur une boucle while dont la condition d'entrée est celle que le nombre de direction possible et testée, initialement égale à zéro, est strictement inférieur au nombre maximal de direction possible vers lequel on peut se diriger, variable contenue à la fin de la liste créée par `data_ofPosition()`. Cette condition permet de ne relancer la boucle que quatre fois dans le pire des cas. A l'intérieur de cette boucle on retrouve une condition créant un nouvelle objet *Arc*, on modifie la position actuellement traité et on effectue l'appel récursif à `create_graph()`. Ces étapes ne sont bien entendu effectuées que lorsque la direction choisit nous le permet. Enfin si nous sommes en présence d'un nouveau nœud nous allons faire appel à `optimisation_graph()` appeler après le traitement de toutes les directions possible nous permettant d'assurer que ce nœud est complet et contient toutes les connexions qu'il aurait pu avoir.

2.2.3 Optimisation du graph

L'optimisation du graphe est une étape importante permettant de ne compter que sur les nœuds utiles à *la recherche du plus court chemin*. Plusieurs méthodes ont été pensés pour réaliser cette optimisation, bien que pas parfaite elle est permet de supprimer et fusionné une multitude de nœuds sans effectuer de boucle ou de parcours supplémentaire au graphe. Afin d'obtenir un graph dont les nœuds sont tous utiles à la prochaine étape, il aurait fallut effectuer des étapes supplémentaires qui coulerai plus cher en complexité que l'algorithme qui cherche le plus court chemin. Nous avons donc opté pour une optimisation se faisant dans le même cycle d'appel que la création du nœud concerné.

Nœud unidirectionnel : Lorsqu'on se situe dans le cas d'un nœud en bout de chemin, il ne possède qu'une seule direction. L'unique cas qui permet au nœud de rester dans le graphe, à l'exception du pakkuman et de la sortie, est un bonbon qui n'aurait pas comme lien direct un monstre. Si un monstre était sa seule connexion, cela signifierait qu'on aurait sacrifié un bonbon pour en récupérer un autre, c'est inutile. Tout les autres cas d'impasse sont donc à supprimer.

Nœud bidirectionnel : Ce deuxième cas est plus complexe, il permet de fusionner deux nœuds qui seraient connecté par un nœud libre (possédant comme attribut ni un monstre, ni un bonbon, ni le pakkuman, ni la sortie). On va alors récupérer les deux arcs sous forme de liste et les traiter pour qu'ils puissent s'ajouter l'un à l'autre.

Ce traitement vise à garder la structure de l'arc permettant d'arriver d'un nœud à l'autre en itérant dans l'ordre chaque élément présent dans celui-ci. Il suffit ensuite de clôturer l'arc et de mettre à jour les nœuds ayant été modifiés.

Une information sur le nombre de nœuds supprimés lors de l'optimisation est affichée dans le terminal afin de voir l'importance de celui ci.

2.3 Algorithme du plus court chemin

Cette section porte sur l'algorithme mis en place pour trouver le plus court chemin dans le graphe créé à la section précédente. Nous nous sommes basés sur l'algorithme connu sous le nom de dijkstra qui très facilement trouvera le chemin le plus court vers la sortie, le souci ici est le paramètre bonbon. De nombreuse manière de faire ont été pensées de l'enregistrement de chaque élément parcouru pour chaque nœud testé jusqu'à un parcours effectué uniquement pour les bonbons avant l'appel à dijkstra ou après... La solution ici est la meilleure qu'on est trouvée. Nous l'avons testé sur une 50aine de labyrinthes différents, tous ont été concluants. Ici aussi de nombreuses méthodes ont été développées seul dijkstra() et dijkstra_sweet() verront leur fonctionnement détaillé ici.

2.3.1 Dijkstra

Notre algorithme suivant le principe de dijkstra a été ajusté pour correspondre à nos besoin. On y retrouve une boucle while ne s'arrêtant que lorsque le nœud choisit correspond à la sortie ou que le nœud actuel est le même que le précédent. Plusieurs listes sont essentielles à la réalisation de cette recherche :

Distance : contient les distances minimales séparant le nœud de départ avec chaque nœud présent dans le graph en fonction du chemin qu'on a parcouru jusqu'ici.

Predecessor : chaque nœud à une liaison avec le nœud qui le précède.

Visited : contient chaque nœud ayant été testé au fur et à mesure du parcours.

Nb_sweet : contient pour chaque nœud le nombre de bonbons actuellement encore utilisables jusqu'à ce nœud.

Nous entrons ensuite dans deux parties différentes :

- L’actualisation des distances possibles en rapport avec les connexions directes avec le nœud actuel.
- La sélection de la distance la plus courte présente dans la liste de sauvegarde.

Connexions directes : Nous allons parcourir l’ensemble des nœuds connectés à celui qu’on a actuellement choisi ($O(4)$ au pire des cas). Pour chaque nœud nous testerons si la distance qu’on a actuellement dans `Distance` est la bonne si ce n’est pas le cas et que le nœud est un monstre on vérifiera dans `Nb_sweet` si le chemin contient un bonbon sinon, on devra faire appel à `dijkstra_sweet()` qui trouvera un bonbon plus haut dans le chemin. Cette distance trouvée pour le bonbon sera ajoutée à la distance d’origine et si elle est toujours plus petite que ce qui se trouve dans `Distance`, alors `Predecessor`, `Nb_sweet` et `Distance` seront mis à jour.

Sélection : Un simple appel à `find_lightWay()` nous permettra de récupérer le nœud suivant qui possède la valeur la plus petite dans `Distance`. On parle bien ici de la plus petite distance actuellement dans `Distance`, non pas de la distance la plus courte avec le nœud suivant. Cette technique permet d’avoir pour tous les nœuds du graphe le chemin le plus court pour accéder à celui ci, testant ainsi toute les possibilité. Si celui-ci est un bonbon ou un monstre nous devrons modifier la valeur dans `Nb_sweet`.

2.3.2 Dijkstra sweet

Cet algorithme est très proche de celui précédemment détaillé. Il possède les mêmes parties et ne s’arrête que lorsqu’on trouve un bonbon qui n’a pas encore été utilisé dans le parcours actuel. A la fin de chaque recherche, `dijkstra_sweet()` stockera dans `Index_sweet` l’indice du bonbon trouvé, `Data_sweet` stockera pour chaque nœuds testés la liste `Predecessor` et `Distance` déjà rencontré plus haut mais propres à l’appel de `dijkstra_sweet()` pour le nœud choisi. Le grand changement se fait principalement avant l’appel à `dijkstra_sweet()`.

Préparation à Dijkstra sweet : L’astuce ici est de réaliser une sélection intelligente des nœuds que l’on va tester pour aller chercher des connexions avec d’éventuels bonbons. Pour cela nous allons faire appel à `state_actuWay()` qui va se charger de parcourir le chemin que nous avons pour l’instant choisit et qui se trouve dans `Predecessor` qui est cette fois ci global à la classe. Nous allons parcourir et sélectionner uniquement les nœuds multidirectionnels et copier dans `visited_real` les nœuds qu’on

parcourt réellement pour l'instant. Une fois cela fait, nous allons faire autant d'appels à `dijkstra_sweet()` qu'il y a de nœuds multidirectionnels.

Sélection du bonbon : Une fois les appels finis nous allons parcourir `data_sweet` contenant toutes les informations dont nous avons besoin pour savoir quel bonbon est le plus proche du chemin initial. On renvoie ensuite cette distance multipliée par deux pour correspondre à un aller-retour, celle-ci sera ajoutée à la distance qu'il existe entre le nœud actuel et le monstre avec lequel il est connecté dans `dijkstra()`. Si un bonbon a été trouvé nous faisons aussi appel à `updateData_newSweet()` qui mettra à jour `way_supp` contenant pour chaque chemin une liste semblable à `Predecessor` contenant donc les nœuds supplémentaires permettant de trouver un bonbon. On ajoutera aussi à `matrix_sweet` une information permettant de savoir quel bonbon a été utilisé pour le parcours actuel. Ces dernières listes se transmettront de nœuds en nœuds au fur et à mesure que l'on parcourt les connectivités de celle-ci dans la première étape de `dijkstra()`.

2.3.3 Conclusion algorithme plus court chemin

On se retrouve avec un `dijkstra` faisant appel à une méthode similaire qui reste en $N.O(\log N)$, on doit déplorer le fait que `dijkstra_sweet()` sera appelé plusieurs fois à chaque monstre trouvé pouvant ainsi alourdir le traitement. Tout ceci a été pensé pour réduire au maximum les appels inutiles, ou les parcours supplémentaires qui n'auraient plus de sens d'être recherché. N'oublions pas qu'on travaille avec un graphe qui a été aussi optimisé contenant un minimum de nœuds sans intérêt. Beaucoup de conteneur sont présent dans cette classe afin de réduire un maximum le temps d'accès aux données et de stocker les nœuds déjà visités.

Dans l'énoncé du projet il est affiché un chemin impossible s'arrêtant sur un monstre, il est malheureusement impossible de reproduire cela dans tous les cas pour notre version car un chemin qui va jusqu'à un monstre sans bonbon est un chemin dit impossible et possède donc une distance maximale qui n'est pas traitée. Ce détail permet de ne prendre en compte que les chemins faisables, plusieurs méthodes ont été développées pour récupérer un nœud de départ permettant l'affichage d'un chemin vers un monstre ou vers le parcours le plus long faisable. Ces étapes ne sont pas utilisées si le chemin existe, elles ne sont là que pour obtenir un affichage quand bien même le labyrinthe est impossible.

2.4 Affichage des différentes situations

Cette section va concerner la création de 2 fichiers différents qui vont contenir la situation initiale et finale du labyrinthe ainsi qu'un affichage en terminal de la situation finale.

2.4.1 Situation initiale

`initial_situation()` est très simple et consiste en une lecture de la matrice créée avec le parsing et de l'utilisation de la fonction `output_analyse()` qui va convertir les entiers contenus dans la matrice en caractères ou strings reproduisant le labyrinthe en son état initial dans le fichier *InitialSituation.txt*.

2.4.2 Situation finale

`final_situation()` peut être considéré comme divisé en deux parties mélangées car utilisent les mêmes données mais les affichent soit en terminal soit dans un fichier. Le but de cette fonction sera d'afficher le chemin suivi par Pakkuman et de s'occuper de l'affichage des cas impossibles également.

Affichage en terminal : On commence avec la méthode `intro_situationFinale()`, qui va afficher l'information sur les éléments contenus dans le labyrinthe et l'information de base parsée au début (taille labyrinthe, nombre de monstres, nombre de bonbons et position des différents éléments). Ensuite on attaque l'affichage des déplacements de Pakkuman, celle-ci consiste en deux boucles imbriquées, l'une qui parcourt les noeuds principaux par lesquels passe Pakkuman, et la deuxième qui parcourt les cases contenues dans les arcs de ces noeuds principaux pour retrouver le chemin complet.

Parcours noeuds principaux : C'est la méthode `printing_journey()` qui s'occupe de cette partie, elle commence par déchiffrer la position du noeud, ensuite elle affiche les coordonnées, les enregistre dans `complete_way` (utilisé pour l'interface graphique) et les enregistre également dans une string qu'on utilisera plus tard lors de l'écriture sur le fichier (pour éviter de refaire toutes les boucles). Un dernier affichage est ajouté pour savoir la direction vers laquelle Pakkuman s'est dirigé et quand est-ce qu'il prend un bonbon ou qu'il passe par un monstre, ce dernier affichage sera effectué avec la fonction `print_way()`.

Pourcours arcs : C'est la méthode `printing_way()` qui va s'occuper de cette partie, elle fait principalement la même fonction que `printing_journey()` mais au lieu de parcourir des sommets, elle parcourt des éléments d'arc. Cette fonction va donc nous donner les chemins intermédiaires entre les noeuds principaux sauvegardés pendant l'optimisation du graph.

Ecriture fichier : C'est la méthode `outro_situationFinale()` qui va écrire la situation finale dans le fichier *FinalSituation.txt*. La fonction va tout d'abord écrire la matrice dans son état final c'est à dire avec le chemin parcouru par Pakkuman en utilisant la même méthode `matrix_printer()` utilisé dans `initial_situation()`. Ensuite selon si il existe une sortie ou pas, la fonction va afficher la longueur du chemin ou un message indiquant qu'il n'a pas réussi à sortir. Et pour finir indique le chemin pris par Pakkuman sous forme d'une suite de coordonnées.

3 Conclusion

Ce projet est une première en son genre dans notre parcours d'étudiants en informatique, pour la première fois nous avons du prendre en compte des aspects comme la vitesse d'exécution (la complexité), la mémoire et d'autres aspects pour optimiser au maximum notre code. Chaque conception d'algorithme a du être discuté et optimisé dans la mesure du possible, tant au niveau de sa complexité et que de son implémentation, enfin les structure de données on du être pensées afin d'utiliser celle qui conviendrait le mieux à nos attentes. Nous nous sommes principalement focalisé sur la vitesse car au final avec les technologies actuelles la mémoire est importante mais souvent en suffisance alors que la vitesse d'exécution peut vite se dégrader lorsqu'on traite de grande quantité de donnée. C'est aussi un projet qui donnait énormément de libertés, il a donc nécessité une documentation théorique importante pour résoudre au mieux le problème initial.