

INFO-F-203 – Algorithmique 2

Manipulation de graphes

Pakkuman

G rard TIO NOGUERAS Alexis DEFONTAINE

17 d cembre 2014

1 Introduction

Ce projet d’algorithmique de deuxi me ann e porte sur les graphes et sur la recherche du plus court chemin. Le probl me est pos  aux  tudiants comme ceci : il faut  crire un programme qui re oit la structure d’un labyrinthe ainsi que la position des  l ments importants se trouvant dans celui-ci. Apr s avoir cr   une structure repr sentant ce labyrinthe, il nous est demand  de trouver le plus court chemin entre l’ l ment *Pakkuman* et la sortie, en sachant qu’il existe des monstres qui nous bloquent le passage et des bonbons qui nous permettent de passer par les cases monstres.

2 Parsage, optimisation de la cr ation de graphe, algorithme du plus court chemin et affichage des diff rentes situations

On a divis  le projet en 5  tapes : le parsage du fichier, la cr ation du graphe, son affichage initial/final, son optimisation et finalement l’algorithme du plus court chemin.

2.1 Parsage

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 de nœuds et arrêtes formant le dit graphe. Le principe développé ici est celui du *backtracking* servant de base pour un 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.

Chaque nœud est un objet qui comme arc contient ses 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 graphe. Hormis certains détails nécessaires à la compréhension du code, seuls `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 actuellement 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 mur. On récupère donc une liste contenant 1 ou 0 pour chaque direction possible ou non, 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 qu'on vient de récupérer. Un deuxième teste sera effectué au cas ou la position n'est pas multidirectionnel 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 teste, 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 clôturé 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 réalisé 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 resté 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 traités 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 modifié.

Une information sur le nombre de nœuds supprimé 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 mit en place pour trouver le plus court chemin dans le graphe créer 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 nombreuses manières de faire ont été pensé 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 50 de labyrinthe différent, tous ont été concluant. Ici aussi de nombreuses méthodes ont été développées seul dijkstra() et dijkstra_sweet() verrons le 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 bonbon actuellement encore utilisable jusqu'à ce nœud. Nous entrons ensuite dans deux parties différentes :

- L'actualisation des distances possible 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 choisit ($O(4)$ 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 bon ou un monstre nous devrons modifier la 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 propre 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 qu'on va testé pour aller chercher des connexions avec d'éventuelles bonbons. Pour ca 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 class. Nous allons parcourir et sélectionné uniquement les nœuds multidirectionnels et copié dans `visited_real` les nœuds qu'on parcourt réellement pour l'instant. Une fois cela fais nous allons faire autant d'appel 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 renvoi ensuite cette distance multiplié par deux pour correspondre à un aller retour, celle ci sera ajouté 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 supplémentaire permettant de trouver un bonbon. On ajoutera aussi à `matrix_sweet` une information permettant de savoir quel bonbon a été utilisé pour le parcours actuelle. Ces dernières liste se transmettrons de nœuds en nœuds au fur et a mesure qu'on parcours 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 travail 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 au données et de stocké 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 tout 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 maximal qui n'est pas traitée. Ce détail permet de ne prendre en compte que les chemins faisables, plusieurs méthodes ont été développé pour récupérer un nœud de `emphfin` 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 la que pour obtenir un affichage quand bien même le labyrinthe est impossible.

2.4 Affichage des différentes situations