Rapport projet IPI

Corentin Juvigny

15 Janvier 2018

I Algorithme en largeur

Pour implementer l'algorithme de parcours en largeur, j'ai choisi de diviser l'algorithme en deux fonctions distinctes.
L'une va regrouper les sommets selon la distance (en nombre de sommets reliées accessibles) qui les séparent du point de départ de la recherche et ceci de façon à ce que chaque sommet soit présent une fois au maximum au total. On arrête cette fonction dès que l'on a regroupé le sommet d'arrivé ou qu'il n'y ai plus de sommets non classés qui soient reliés au sommets triés (et donc que le sommet d'arrivé soit au final inaccessible. Pour l'implementer j'ai donc choisi d'utiliser un tableau dynamique de listes chaînées de taille n = nombre de sommets du graphe car l'on a dans le pire des cas (le graphe est une chaîne de n éléments avec pour sommet de départ un bout de la chaîne et sommet d'arrivée l'autre bout). Le choix des listes chaînées s'explique par le faîte qu'il est impossible à l'avance de connaître le nombre de sommets de même distance. La liste chaînée est donc le meilleur moyen de retenir ce sommet (on enregistre le numéro du sommet) avec la meilleur utilisation de la mémoire possible. Enfin si le sommet d'arrivée a été trié alors la fonction enregistre dans des variables entrées en paramètres le tableau des Vlistes ainsi que le nombre d'éléments du tableau de liste qui ont réellement été utilisé. Dans le cas où la fonction s'arrête sans avoir trouvé le sommet d'arrivée c'est donc qu'il est inaccessible et enregistre dans la variables correspondant à la taille utilisé du tableau la valeur n + 1 (qui est impossible car le tableau est de taille réelle n). Une telle valeur caractérise donc un echec.
La deuxième fonction elle reconstruit le chemin entre le point de départ et celui d'arrivée. Pour cela il se base sur le faite que si le point d'arrivée est dans le tableau au m-ième rang, c'est qu'au rang m - 1 il se situe au moins un sommet qui est relié au sommet d'arrivée. De même on est sûr qu'il existe un sommet au rang m - 2 qui est relié à ce sommet et ainsi de suite jusqu'au sommet de départ. On parcours donc notre tableau de liste et on regarde quel élément est relié au précédent en partant du sommet d'arrivée jusqu'au sommet de départ. On enregistre alors cette suite de sommets dans un tableau de taille size (qui correspond au nombre d'éléments du tableau de listes utilisés) car si le sommet d'arrivé est distant de size point du sommet de départ, il faudra size sommets pour les reliés. Par contre s'il reçoit une longueur supérieur au nombre de sommets du graphe, alors il renvoie un tableau NULL (pour absence de chemin possible).
Finalement dans le main affiche les éléments du tableau en partant de la fin (car les sommets sont enregistré du dernier au premier dans le tableau) car on veut afficher le chemin en partant du sommet de départ. Enfin dans le cas où le chemin est inexistant, alors c'est que le tableau vaut NULL et alors on affiche "Not connected".

II Algorithme de Dijkstra

Pour implementer l'algorithme de Dijkstra j'ai divisé l'algorithme en deux fonctions distinctes.
La première réalise le pseudo-algrithme de Dijkstra. Pour cela on a besoin de la matrice d'adjacence avec les poids des chemins ainsi qu'un tableau de taille égale ou supérieur à la taille de la matrice d'adjacence et dans lequel on va enregistrer les poids des sommets. Enfin on enregistre les sommets visités dans une liste visite car la taille du chemin n'est pas connu à l'avance. Pour choisi le maximum j'ai implémenter une fonction minimum qui recherche dans tous les sommets restant ie ceux qui n'ont pas encore été ajoutés à ma liste visite celui dont la distance (enregistré dans d) est minimal. Enfin comme valeur d'infini, qui va être donné à tous les sommets qui ne sont pas encore accessible, j'a décidé d'utiliser le plus grand entier possible en c, et pour cela j'ai utilisé la librairie standart <limits.h> qui contient la valeur de cet entier sous le nom de INT_MAX. Ansi je suis sur qu'un sommet qui n'est pas accessible ne pourra jamais être considéré comme minimal dans un chemin. Si mon algorithme ne rencontre jamais t c'est qu'il n'y a pas de chemin possible entre s et t et je renvoie INT_MAX pour le signifier, et dans la fonction main avant de reconstruire le chemin avec la seconde fonction je teste s'il existe et dans le cas contraire j'affiche "Not connected" et j'arrête là le programme.
La seconde reconstruit le chemin à partir de la fin. Pour celà on utilise la matrice d'adjacence ainsi que des poids de d en se basant sur le faite que le poid du sommet d'arrivé t correspond à la distance entre le noeud de départ s et t. Donc il existe un noeud u tel que la distance entre d[t] - d[u] = G[u][v]. De même de ce point il existe un noeud v tel que d[u] - d[v] = G[u][V]. En remontant ainsi de proche en proche on atteint s. Mais ne connaissant pas ainsi à l'avance le nombre de noeux entre s et t, j'ai dû utilisé des listes chaînées pour minimiser la place mémoire nécessaire. Ensuite il ne reste plus qu'à dépiler la liste chaînée en affichant ses éléments pour afficher le chemin à parcourir.

III Labyrinthe

Pour résoudre cet exercice, j'ai décidé d'utiliser l'algorithme a* [1] car celui-ci est dans de nombreux cas plus rapide que le parcours en largeur ou l'algorithmde de Dijkstra pour trouver un chemin optimal.
La première étape de mon algorithme consiste à interpréter le labyrinthe passé en entrée de façon à pouvoir utiliser l'algorithme a* dessus. Tout d'abord j'identifie chaque caractère 'X', '.', '&', etc... comme un noeud d'un graphe, en considérant que tous les sommets sont reliés avec leur voisins, sauf s'il est un 'X' ou un 'A'. J'utilise alors ma fonction a_voisination qui va, pour un tableau à m éléments [2] créée un tableau de taille m*8 (8 car au maximum chaque élément du labyrinthe ne peut pas avoir plus de 8 voisins ; les 4 sommets qui lui sont adjacents plus si c'est un téléporteur les 4 sommets adjacents à l'autre entrée du téléporteur) qui va contenir à la position tab[i] un tableau les sommets qui sont accessibles à partir du sommet i (par exemple si l'on a $tab[7] = \{2, 12, -1, -1, -1, -1, -1, -1\}$ on a donc que le noeud 7 est relié aux noeuds 2 et 12). J'ai préféré un tel tableau à un tableau de voisins standarts (c'est à dire de taille nxn comme dans l'exo 1 et 2) car ce tableau n'est qu'en $\Theta(m)$ et non $\Theta(m^2)$ ce qui est nécessaire pour stocker de grands labyrinthes. Pour relier les téléporteurs entre eux je retiens la position du premier téléporteur que je stocke dans un tableau (de taille 10 car il y a 10 types distincts de téléporteurs) et lorsque je rencontre la deuxème version du même téléporteur je relie tous leurs voisins respectifs en utilisant la position du premier contenue dans le tableau. Enfin s'il y a une clé je retiens sa position ainsi que la position de la porte dans un tableau d'une struct pos_cle qui contient 3 éléments : le numéro du noeud, sa coordonnée en x et celle en y. De plus je considère que A n'est pas un sommet traversable dans cette fonction.
Je réalise ensuite l'algorithme a* sur tab. Pour l'implémener j'ai dû coder un certain nombre de fonctions qui sont nécessaires au bon fonctionnement de l'algorithme. Tout d'abord en ce qui concerne la distance j'ai décidé d'utiliser la distance de manathan :

$$\forall (Z1, Z2) \in [0, n-1]^2 \, \text{distance\_manathan}(Z1, Z2) = |x2-x1|+|y2-y1|$$

car cette distance ne surévalue pas la distance dans notre cas et a l'avantage de plus de renvoyer un entier par rapport à la distance euclidienne par exemple qui elle renvoie un flottant, ce qui agmenterai les temps de calcule. Il me fallait aussi un moyen d'insérer un sommet dans une liste (qu'on suppose triée) de façon à ce que la liste soit triée en fonction du coût heuristique qui est la somme du coût pour aller au sommet (comme dans Dijkstra) plus l'heuristique de ce point. Ce trie me permet d'obtenir un chemin optimal ; par contre si le labyrinthe ne contient pas de mur [3], ce trie est plus lent et j'ai préféré trier selon l'heuristique et non le coût heuristique, ce qui ne m'assure pas de trouver le chemin optimal dans tous les cas mais qui est plus rapide lors de l'execution du programme.

1. une version modifiée de l'algorithme de Dijkstra qui rajoute un coût suplémentaire appelé heuristique lié à la distance entre le point étudié et le point d'arrivé
2. soient n lignes et n colonnes
3. ici nbr_de_'.' = n - 2

J'ai aussi une fonction qui me permet de mettre à jour ma liste triée ; lorsqu'un élément de cette liste change de valeur je le supprime de la liste et je l'insère avec ma fonction d'insertion triée et ainsi ma liste triée le reste, car celà est un gain de temps considérable pendant l'exécution, on n'a pas à parcourir l'entièreté de closedList pour trouver le plus petit, on a juste à dépiler le premier élément. Enfin la dernière fonction importante est la fonction de guidage. Cette fonction récupère en entrée le chemin sous forme d'un tableau d'entier, qui sont les numéros des sommets qui composent le chemin, et affiche sur stdout 'HAUT' 'BAS' 'GAUCHE' 'DROITE' 'TP' en se basant sur le faite que : si la différence entre un sommet et son prédécesseur vaut 1, c'est que ces deux sommets sont voisins sur la même ligne (donc il est soit à droite soit à gauche de son prédecesseur) tandis que si la différence vaut n [4] alors c'est que son prédecesseur se situe au dessus ou n dessous de lui. Sinon c'est qu'il y a eu téléportation entre les 2 sommets. On retrouve ainsi facilement la suite d'actions à effectuer pour parcourir le labyrinthe selon le chemin donné par l'algorithme a*. Tous cela m'a permis d'implementer le pseudo-algrithme du a* [5]. Dans le cas standart (ie sans clé) il suffit de réaliser une unique fois l'algorithme. Celui-çi enregistre dans un tableau de prédecesseurs quel était le noeud qui précède celui que l'on étudie, et ceci à chaque étape. On peut ainsi reconstruire le chemin entre la sortie et l'entrée en partant du sommet d'arrivé, en récupérant son prédecesseur dans le tableau de prédecesseurs, puis de trouver le prédecesseur de ce point et ainsi de suite jusqu'à atteindre le sommet d'arrivé. On obtient alors une liste de sommets qui forme un chemin entre le départ et l'arrivée du labyrinthe. Il suffit alors d'utiliser la fonction guidage sur cette liste pour avoir la suite d'instructions nécessaires pour atteindre la sortie du labyrinthe.
Dans le cas où le labyrinthe contient une porte avec une clé, on se retrouve avec le problème suivant. Soit il existe un chemin plus rapide sans passer par la porte (et donc par la clé) soit il faut passer par la clé et dans ce cas le chemin se divise en deux étapes indépendantes : aller du départ à la clé puis aller de la clé à la fin. J'ai donc décidé de réaliser une fonction qui, lorsque si après la a_voisination il y a présence d'une clé (dont la position est enregistrée dans une struct pos_clé) dans le labyrinthe, va s'occuper de trouver le plus court chemin. Dans un premier temps elle va tenter de trouver un chemin entre le début et la fin du labyrinthe sans passer par la clé et donc en considérant toujours la porte comme un mur). Si ce chemin aboutit et que la durée du chemin est inférieur à la durée maximal t qu'on se laisse pour réaliser l'algorithme, alors la fonction affiche le résultat grâce à la fonction de guidage. Sinon ces deux conditions ne sont pas remplis, il faut donc passer par la clé pour réussir à sortir à temps du labyrinthe. Mais pour accélérer les temps de calculs, au lieu de calculer le chemin vers la clé puis celui vers la porte à la suite dans un seul processus, je calcule les 2 chemins en même temps en utilisant un processus lourd fils [6] qui va calculer le chemin entre l'entrée et la clé tandis que le père va calculer le chemin entre la clé et la sortie en considérant maintenant que la porte 'A' est ouverte et est donc

4. la longueur d'une ligne
5. voir annexe
6. grâce à la fonction fork() en C

traversable. Le père attend [7] que son fils est affiché son résultat grâce à la fonction de guidage et qu'il se soit terminé sans erreur avant de lui-même utiliser guidage et de se terminer. On obtient ainsi le chemin compler du début à la fin. Le limitation de cette fonction est que l'algorithme a* peut mettre du temps, surtout si le labyrinthe est très grand, à se rendre compte qu'il n'existe pas de chemin directe entre l'entrée et la sortie sans passer par la clé. J'ai donc décider de ne pas tester ce chemin si le graphe posséde plus de 100 000 noeuds. Il aurait été possible pour contourner ce problème d'executer le test du chemin sans passer par la clé dans un processus fils. S'il trouve un chemin valable alors il aurait écrit son résultat dans un pipe. Les deux autres auraient aussi écrit leur résultat dans un même pipe et le père aurait lu le résultat des trois et aurait choisi d'utiliser guidage sur le plus court. De plus si au moment où une des solutions est donné l'un des processus fils tourne encore, le père l'aurait tué. Et ainsi pourrait tester le chemin même sur de grand test. Mais pour des raisons de temps je n'ai pas pu l'implémenter.
Au final cette algorithme a* est un algorithme vraiment très performant pour trouver son chemin dans un graphe dont on connait la position de la sortie car il diminue forteent le nombre d'éléments que l'on a à tester. Et cela se ressent dans les performances de l'algorithmes qui sont bien supérieurs à celle d'un Dijkstra pure, particulièrement lorsque le nombre de noeuds du labyrinthe est très élevée.

7. en utilisant la fonction wait() du C

Annexe

```
function aetoile(g, objectif, depart) :
    visite = File()
    closedList = FilePrioritaire(comparateur)
    openList.ajouter(depart)
    tant que closedList n'est pas vide :
        u = closedList.depiler()
        si u.x == objectif.x et u.y == objectif.y :
            reconstituerChemin(u)
            terminer le programme
        pour chaque voisin v de u dans g :
            si v existe dans closedList avec un cout inférieur ou si v
            existe dans visite avec un cout inferieur :
                passer
            sinon :
                v.cout = u.cout
                v.cout_heuristique = v.cout + v.heuristique
                closedList.ajouter(v)
        visite.ajouter(u)
    terminer le programme
```