Projet IPI: chemins de poids minimum

Romain PEREIRA

04/12/2017

Table des matières

1 Recherche de chemin le plus court **2**
1.1 Parcours en largeur (graphes non-pondérés) 2
1.2 Algorithme de Dijkstra (graphes pondérés positivement) 3
1.2.1 1ère approche 3
1.2.2 File de priorité ('Pairing Priority Queue') 3
1.3 Algorithme A* (graphes pondérés et fonctions heuristiques) . . . 4

2 Application : resolution labyrinthe **5**

3 A propos de l'implémentation **6**
3.1 Structures de données 6
3.2 Qualité logiciel 6

4 Sources **7**

Préambule

Ce projet est réalisé dans le cadre de mes études à l'ENSIIE.
Le but est d'implémenter des algorithmes de recherche de 'chemin le plus court', dans des graphes orientés.
Ce document rapporte mon travail, et explique les choix techniques que j'ai pris.

On considère (X, A) un graphe.

- n : Card(X)
- X : sommets du graphe
- A : arcs du graphe
- s : sommet 'source', celui à partir duquelle les chemins sont construits
- t : sommet 'target', celui vers lequel on souhaite construire un chemin

1 Recherche de chemin le plus court

1.1 Parcours en largeur (graphes non-pondérés)

On considère ici un graphe où les arcs sont non pondérés.
'Le chemin le plus court' entre 2 sommets corresponds au chemin à une famille d'arc de cardinal minimum.
On souhaite codder l'information

- il y a existence d'un arc entre le sommet 'u' et le sommet 'v' '

Cette information est un booléen, et peut donc être coddée sur un bit. J'économise ainsi beaucoup de mémoire. (8 fois plus que si l'arc était coddé sur un octet), en representant mes arcs sur un tableau de bit.

Soit 'b' l'indice d'un bit dans un tableau d'octet. Pour y accéder, il faut récuperer l'indice b_o de l'octet correspondant dans le tableau, et l'indice b_b du bit sur cet octet.

En posant :

- b_o = b / 8 (quotient de la division euclidienne de 'b' par 8)
- b_b = b % 8 (reste de la division euclidienne de 'b' par 8)

on s'assure de l'unicité,

- b = 8 b_o + b_b

De plus, on peut également faire une transformation entre 2 coordonnées ('u', 'v') et le bit 'b'. On souhaite que le bit 'b' soit à 1 s'il y a un arc entre 'u' et 'v', sur 0 sinon. Pour cela, j'utilise la bijection :

- u = b % n
- v = b / n
- b = n * v + u (Rappel : 'n' est le nombre de sommet dans le graphe)

Je ne perd pas de temps en lecture / écriture dans le tableau des arcs, car ces changements de coordonnées ne necessite que 1 multiplication, 1 addition, et quelques operations sur les bits (diviser par 8 <=> décaler les bits de 3 vers la droite)

De plus, en réduisant la mémoire utilisé, je rends mon programme plus 'cache-friendly', le rendant plus rapide. Le processeur écrit des blocs mémoire du programme dans sa mémoire cache : plus les données sont compactes, moins il aura à faire des allés/retours entre la mémoire du programme et sa mémoire cache.

Le reste de l'implémentation du parcours en largeur suit l'algorithme traditionelle. J'utilise une file FIFO (1er entré, 1er sorti), afin d'enregistrer les sommets à visiter en fonction de leur profondeur dans le graphe.
(VOIR STRUCTURE DE DONNEES 'list')

1.2 Algorithme de Dijkstra (graphes pondérés positivement)

On considère ici un graphe où les arcs sont pondérés avec des poids positifs. 'Le chemin le plus court' entre 2 sommets corresponds au chemin avec la somme des poids de ses arcs minimum.
L'algorithme de Dijkstra nous est fourni dans le sujet. Remarquons que si le poids de tous les arcs sont identiques, on retrouve l'algorithme de parcours en largeur.

1.2.1 1ère approche

Ma 1ère implementation (voir 'dijkstra.c.bkp') reprends le squelette du parcours en largeur. La seule différence réside dans le choix du prochain sommet à visiter.
Dans le parcours en largeur, les sommets sont visités dans leur ordre d'apparation dans la liste (1er entré, 1er sorti). Trouver le prochain sommet à visiter est d'une complexité **O(1)**.

Dans l'algorithme de Dijkstra, les sommets sont visités par ordre du poids de leur chemin à 's'. Ainsi, si la file de visite contient 'm' sommets, trouver le prochain sommet à visiter devient en complexité **O(m)**

1.2.2 File de priorité ('Pairing Priority Queue')

Je me rends compte que l'operation 'trouvé sommet 'u' suivant minimisant 'd(u)' est pré-dominante dans le temps d'execution => j'optimiser

1.3 Algorithme A* (graphes pondérés et fonctions heuristiques)

L'algorithme A* est une extension de l'algorithme de Dijkstra.
Avec l'algorithme de Dijkstra, on parcourt le graphe en largeur selon le poids de ses arcs. Avec A*, on ajoute une heuristique aux poids des arcs.
'Une heuristique est une méthode de calcul qui fournit rapidement une solution réalisable, pas nécessairement optimale ou exacte, pour un problème d'optimisation difficile.'
Cette heuristique permet de modifier l'ordre de priorité dans lequel les sommmets seront visités dans le graphe. Bien quel fait perdre l'optimalité, elle permet d'orienter la recherche dans le graphe, rendant la recherche plus rapide. (et avec une heuristique bien conçu au problème, on s'assure tout de même une solution proche de l'optimal)

Remarquons également qu'en utilisant une heuristique nulle (qui renvoie toujours '0'), on obtient très exactement l'algorithme de Dijkstra. C'est d'ailleurs comme celà que je l'ai re-implementé une fois l'algorithme A* implementé et optimisé.
(une version de l'algorithme de Dijkstra 'stand-alone' est toujours disponible dans le fichier 'dijstra.c.bkp')

2 Application : resolution labyrinthe

3 A propos de l'implémentation

Ci joint, vous trouverez mon implémentation en language 'C'.

3.1 Structures de données

J'ai implémenté plusieurs structures de données, afin d'optimiser les algorithmes. Je les ai conçu de manière générique, afin de pouvoir m'en reservir plus tard dans d'autre projet. Si vous souhaitez plus de détail, je vous conseille de regarder les fichiers '.h' correspondants.

- 'list' : liste doublement chainée. Elle sont plus efficace que des listes simplement chainées (insertion et deletion). Elles me servent file FIFO et LIFO.
- 'array' : tableau dynamique générique. Le tableau grossit automatiquement si besoin lors de l'insertion.
- 'bitmap' : tableau d'octet. Cette structure facilite les opérations sur les bits.
- 'pqueue' : une file de priorité ('Pairing Priority Queue' ci-dessus)
- 'node' : sommet d'un graphe. Cette representation m'a permis de créer un système d'héritage, rendant mon code plus modulaire. (n'importe quel structure héritant de 'node' est compatible avec tous les algorithmes implémentés.)
- graphes : pas de structure explicit implémenté. Mes graphes sont representés sous forme de tableau de sommet.

3.2 Qualité logiciel

Mes programmes passent les tests fournis.

De plus, j'ai debuggé l'integralité du code à l'aide de l'outil 'valgrind'. Il ne semble y avoir ni fuite mémoire, ni dépassement de tampon, ni accès à de la mémoire non initialisé. (tester sur le set de test fourni).

J'ai également utilisé l'outil 'gprof' afin d'optimiser mes algorithmes. C'est nottement ce qui m'a fait implémenter des files de priorités, car en utilisant une file standart, mon programme passait la majeur partie de son temps d'execution à chercher le sommet optimal à visiter AVANT : //TODO APRES : //TODO

4 Sources