

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 ('Priority Queue')	3
1.3	Algorithme A* (graphes pondérés et fonctions heuristiques)	4
2	Application : resolution labyrinthe	6
3	A propos de l'implémentation	7
3.1	Structures de données	7
3.2	Qualité logiciel	7
4	Références	8

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 : $\text{Card}(X)$
- 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 correspond à une famille d'arcs, dont le cardinal minimum.
On souhaite coder l'information

- 'il existe un arc entre le sommet 'u' et le sommet 'v' '

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

Soit 'b' l'indice d'un bit dans un tableau d'octet. Pour y accéder, il faut récupérer 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,

- $b = 8 * b_o + b_b$

SCHEMA On peut aussi faire une transformation entre 2 coordonnées ('u', 'v') vers un bit 'b'.

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

Le bit 'b' vaut 1 s'il existe un arc entre 'u' et 'v', 0 sinon. On a besoin de n^2 bits, et donc de $n^2/8 + 1$ octets. On gagne donc 8 fois plus de mémoire que si l'information était stockée sur un octet. De plus, je ne perds pas de temps en lecture / écriture dans le tableau des arcs, car ces changements de coordonnées ne nécessitent que 1 multiplication, 1 addition, et quelques opérations sur les bits (diviser par 8 \Leftrightarrow décaler les bits de 3 vers la droite)

Egalement, en réduisant la mémoire utilisée, 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.

Pour enregistrer les sommets en fonction de leur profondeur dans le graphe, et ainsi les visiter dans l'ordre de leur profondeur, j'utilise une file FIFO (1er entré, 1er sorti).

(VOIR STRUCTURE DE DONNEES 'list')

Chaque sommet 'u' de mon graphe possède un attribut pointant vers un autre sommet du graphe. Une fois l'algorithme de parcours terminé, cet attribut pointe vers le prédécesseur de 'u', dans le chemin le plus court allant de 's' à 't', et passant par 'u'. Pour reconstruire le chemin, il suffit de regarder récursivement les prédécesseurs, en partant du sommet 't' jusqu'à ce que l'on ait atteint 's'. La complexité de la reconstruction est en $O(m)$, où 'm' est la longueur du chemin.

(SCHEMA DE LA REMONTE)

Cette modélisation permet d'optimiser les coûts de stockage, et la remontée est d'un coût négligeable devant le temps de résolution du chemin. Elle sera réutiliser dans Dijkstra et A^*

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 correspond 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'apparition 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 ('Priority Queue')

Après avoir implémenté Dijkstra avec cette 1ère approche, j'ai étudié le temps d'exécution du programme.

Il s'est avéré que mon programme passe, en moyenne, plus de 70% de son temps d'exécution à chercher le prochain sommet à visiter. (l'opération 'trouver un sommet 'u' non visité minimisant 'd(u)').

Ainsi, j'ai décidé d'implémenter des 'files de priorités', et plus précisément des 'tas binaire'. Cette structure de donnée est une file, permettant de définir des priorités parmi les éléments, et d'effectuer les 4 opérations élémentaires suivantes (avec 'm' le nombre d'élément dans la file) :

%	self		
time	seconds	calls	name
92.50	2.94	1000000	dijkstra next node
3.46	0.11	1	dijkstra
1.26	0.04	1	lab_parse
0.63	0.02	9993999	array_ensure_capacity
0.63	0.02	2000003	array_new
0.31	0.01	9993999	array_add
0.31	0.01	9993999	array_set
0.31	0.01	4995997	bitmap_get
0.31	0.01	2000003	array_delete
0.31	0.01	1999999	bitmap_set
0.00	0.00	10001998	array_get
0.00	0.00	1000000	array_remove

FIGURE 1 – résultat de 'gprof' sur 'tests/exo3/06'

- 'insérer un élément' : $O(\log m)$
- 'extraire l'élément ayant la plus grande priorité' : $O(\log m)$
- 'tester si la file de priorité est vide' : $O(1)$
- 'diminuer la priorité d'un élément déjà inséré' : $O(\log m)$

les détails techniques peuvent être trouvé en annexe [3], ou directement dans mon implementation. (voir *queue.c,h*)

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 effectue la même opération, mais 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.'* [2]

Cette heuristique permet de modifier l'ordre de priorité dans lequel les sommets seront visités dans le graphe. Bien que cela 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 cela que je l'ai re-implementé Dijkstra, une fois l'algorithme A* implementé et optimisé. (une version de l'algorithme de Dijkstra 'stand-alone' est toujours disponible dans le fichier 'dijkstra.c.bkp')

Ces résultats sont ceux appliqués au test exo3/06 (graphe de 1 000 000 de sommets, on cherche le plus court chemin entre les 2 sommets les plus distants)

```
BAS
BAS
BAS
BAS
BAS
./exo3 < tests/exo3/06/input 5,14s user 0,04s system 99% cpu 5,190 total
+ code git:(master) x vim Ma

DROITE
BAS
DROITE
BAS
DROITE
./exo3 < tests/exo3/06/input 1,10s user 0,04s system 99% cpu 1,144 total
+ code git:(master) x

DROITE
BAS
BAS
DROITE
DROITE
./exo3 < tests/exo3/06/input 0,32s user 0,04s system 99% cpu 0,358 total
+ code git:(master) x
```

FIGURE 2 – *Temps d'exécution des algorithmes*

- 1 : Dijkstra sans file de priorité
- 2 : Dijkstra avec file de priorité
- 3 : A* avec file de priorité, et fonction heuristique (distance de Manhattan)

2 Application : resolution labyrinthe

Dans l'exercice 3, on nous propose de résoudre des labyrinthes. Nous allons bien évidemment nous servir des fonctions algorithmes préalablement implémenté.

```

==11950== Memcheck, a memory error detector
==11950== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==11950== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==11950== Command: ./exo3
==11950==
==11950== HEAP SUMMARY:
==11950==    in use at exit: 0 bytes in 0 blocks
==11950==   total heap usage: 5,000,027 allocs, 5,000,027 frees, 161,171,949 bytes allocated
==11950==
==11950== All heap blocks were freed -- no leaks are possible
==11950==
==11950== For counts of detected and suppressed errors, rerun with: -v
==11950== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
exo3 06 (END)

```

FIGURE 3 – résultat de 'valgrind' sur 'tests/exo3/06')

3 A propos de l'implémentation

Ci joint, vous trouverez mon implémentation en langage '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 réserver plus tard dans d'autre projet. Si vous souhaitez plus de détail, je vous conseille de regarder les fichiers '.h' correspondants.

- 'list' : liste doublement chaînée. Elle sont plus efficace que des listes simplement chaînées (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 ('Binary 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 explicite implémenté. Mes graphes sont représentés sous forme de tableau de sommet.

3.2 Qualité logiciel

Mes programmes passent les tests fournis. De plus, j'ai debuggé l'intégralité 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 optimisé mon programme à l'aide de l'outil 'gprof'. D'où par exemple, l'utilisation du 'define' ARRAY_ITERATE implémenté dans le fichier 'array.h'.

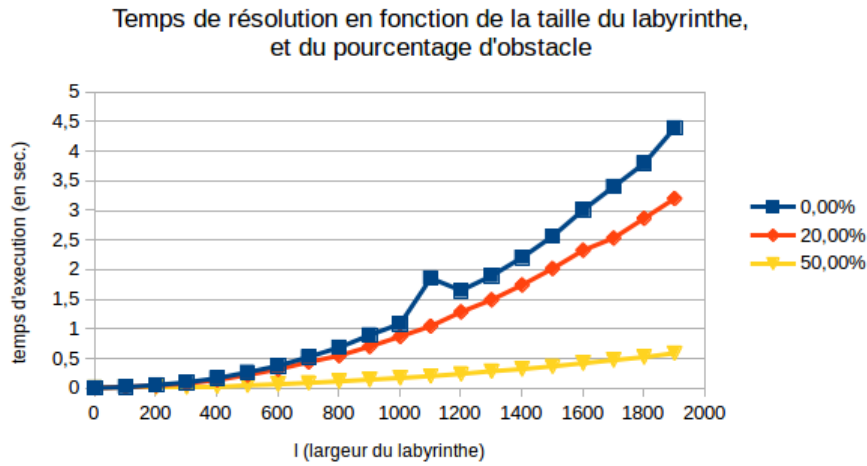


FIGURE 4 – *Temps de résolution de labyrinthe avec Dijkstra (A^* heuristique nulle)*

Ce 'define' me permet d'itérer rapidement sur les tableaux, en utilisant l'arithmétique des pointeurs.

J'ai généré des labyrinthes aléatoires, afin d'obtenir ces courbes : Dans le cas externe à 0% d'obstacles, on a $l * l$ sommets (case vide du labyrinthe) dans le graphe, d'où l'allure quadratique de la courbe.

4 Références

- [1] Dr. Mike POUND, Sean RILEY, 24 Février 2017,
Maze Solving - Computerphile,
<https://www.youtube.com/watch?v=rop0W4QDOUI>.
- [2] Wikipédia, Heuristique, 31 Octobre 2017,
[https://fr.wikipedia.org/wiki/Heuristique_\(mathématiques\)](https://fr.wikipedia.org/wiki/Heuristique_(mathématiques)).
- [3] Wikipédia, Binary Heap, 15 Décembre 2017,
https://en.wikipedia.org/wiki/Binary_heap.
- [4] Mary K. VERNON, Priority Queues, 3 Septembre 2016,
<http://pages.cs.wisc.edu/~vernon/cs367/notes/11.PRIORITY-Q.html>.