

Projet IPI: chemins de poids minimum

Romain PEREIRA

04/12/2017

Sommaire

1	Recherche de chemin le plus court	2
1.1	Parcours en largeur (graphes non-pondérés)	2
1.1.1	1ère approche	2
1.1.2	2ème approche	2
1.1.3	Algorithme de remontée	3
1.2	Algorithme de Dijkstra (graphes pondérés positivement)	3
1.2.1	1ère approche	3
1.2.2	File de priorité ('Priority Queue')	4
1.3	Algorithme A* (graphes pondérés et fonctions heuristiques)	5
2	Application: résolution labyrinthe	5
2.1	1ère approche	5
2.2	Optimisation	5
2.3	Parallélisation	6
3	A propos de l'implémentation	7
3.1	Types / 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 qui ont été pris. Soit (X, A) un graphe. On note:

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

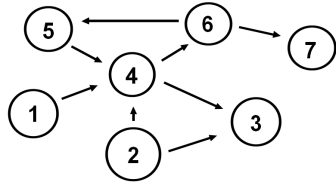


Figure 1:
graphe G $n=7$,
 $X=\{1, 2, \dots, 7\}$,
 $A=\{(1, 4), (4, 6), \dots, (6, 7)\}$

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 est minimum. On souhaite coder l'information:

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

1.1.1 1ère approche

Cette information est un booléen, et peut donc être codée sur un 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)

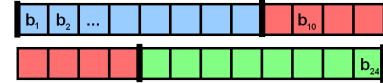


Figure 2: *Schéma bits*

on s'assure de l'unicité,

- $b = 8 * b_o + b_b$

De plus, étant donné deux sommets 'u' et 'v', en posant

- $b = n * v + u$

on peut cartographier les arcs sur le tableau de bits. Le bit vaut alors 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 utilise donc 8 fois moins de mémoire que si l'information était stockée sur un octet. De plus, on ne perd pas (ou très peu) de temps en lecture / écriture, 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, le programme est rendu plus 'cache-friendly' [1], lorsque l'on teste le voisinage entre 2 arcs en parcourant la matrice.

1.1.2 2ème approche

La complexité ('spatiale') de stockage des arcs est donc en $1/8 * O(n^2)$. Plus précisément, pour $n = 10^6$ (cas labyrinthe exo3/tests/06), l'espace mémoire nécessaire est de 125Go... On va donc changer de structure de données. En ajoutant un attribut liste à chaque sommet, qui contient l'indice de ses sommets voisins, la complexité spatiale devient donc (au plus) en $m * O(n)$, où 'm' est le nombre maximal de successeurs par sommet. (Remarque : pour $m = n$, on retrouve $O(n^2)$) Il en résulte que dans la résolution de labyrinthe (exo3/tests/06), on a $m \leq 5$, donc pour $n = 10^6$, on passe donc de 125Go à 5Mo. Les transformations précédentes vont cependant être ré-utilisées dans Dijkstra et A*.

1.1.3 Algorithme de remontée

Chaque sommet 'u' du 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 '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.

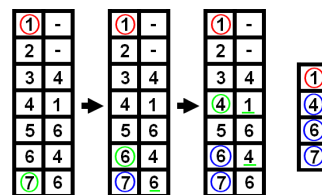


Figure 3: Schéma de l'algorithme de remontée (voir Figure 1)

Cette modélisation permet de réduire les coûts de stockage, et la remontée est d'un coût négligeable devant le temps de résolution du chemin. Elle sera réutilisée 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

L'implémentation reprend le squelette du parcours en largeur. Cependant, une différence fondamentale 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 de 's'. Ainsi, on pourrait parcourir tous les sommets non-visités, et en extraire un minimum (dans le sens de sa distance à 's'). Si la file contient 'm' sommets non visités, la complexité de cette opération est en $O(m)$.

Après avoir implementé Dijkstra avec cette 2nde approche, j'ai étudié le temps d'exécution.

%	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 4: résultat de 'gprof' sur 'tests/exo3/06'

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)' dans l'algorithme fourni).

1.2.2 File de priorité ('Priority Queue')

Ainsi, j'ai décidé d'implémenter des 'files de priorités', et plus précisément des 'tas binaires' [2], afin d'optimiser cette partie du programme. Cette structure de données est une file, permettant de définir des priorités parmi les éléments, et d'effectuer les 4 opérations élémentaires suivantes (avec 'm' le nombre d'éléments de la file):

- '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 de cette structure peuvent être trouvés en annexe [2], et son implementation dans les fichiers *pqueue.[c,h]*.

On passe alors d'une complexité $O(n^2)$ à $O(n * \log(n))$ pour l'algorithme de Dijkstra.

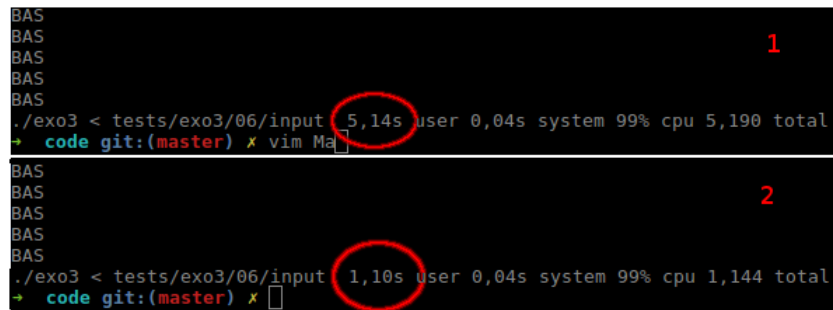


Figure 5: Temps d'exécution avec et sans une file de priorité, sur le test *exo3/06*

- 1 : Dijkstra sans file de priorité
- 2 : Dijkstra avec file de priorité

Voici une courbe d'étude sur les performances de Dijkstra (avec file de priorité)

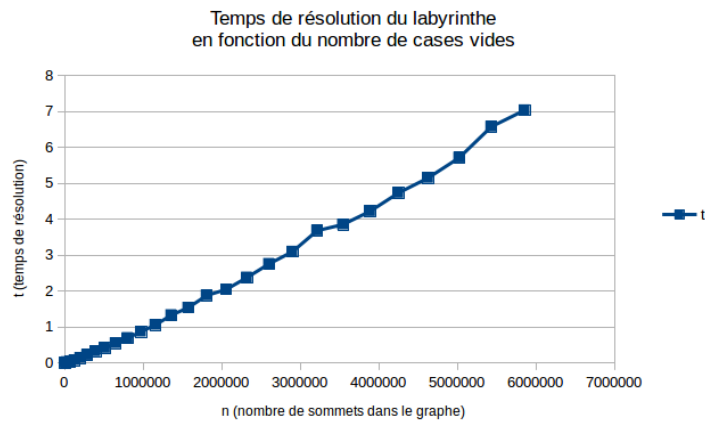


Figure 6: Temps de résolution (en sec.) de labyrinthe avec Dijkstra, allure en $O(n * \log(m))$

1.3 Algorithme A* (graphes pondérés et fonctions heuristiques)

L'algorithme A* est une extension de l'algorithme de Dijkstra [4] [5]. 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 [6] aux poids des arcs.

Cette heuristique permet de modifier l'ordre de priorité dans lequel les sommets seront visités dans le graphe. Bien qu'elle fait perdre l'optimalité, elle permet d'orienter la recherche dans le graphe, rendant la convergence vers un chemin plus rapide. (et avec une heuristique bien conçu au problème, on s'assure d'une solution acceptable). De plus, on peut très bien décider de laisser la recherche s'affiner, Par exemple, dans la résolution de labyrinthe, une heuristique intéressante est la distance de Manhattan. [7] Remarquons également qu'en utilisant une heuristique nulle (qui renvoie toujours '0'), on obtient très exactement l'algorithme de Dijkstra.

2 Application: résolution labyrinthe

Dans l'exercice 3, on nous propose de résoudre des labyrinthes.

Intuitivement, le labyrinthe peut être modélisé par ce type de graphe (voir Figure 7). On crée un sommet pour chaque case 'non-mur' (cases vides, téléporteurs, porte, clef). Pour chaque sommet, on crée des arcs entre lui et ses voisins.

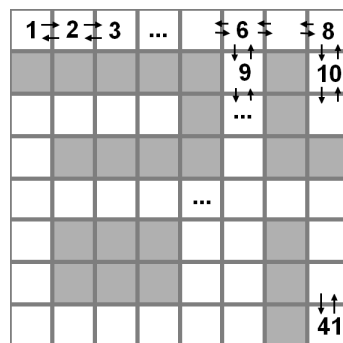


Figure 7: *Modélisation du labyrinthe*

2.1 1ère approche

Une fois le labyrinthe modélisé, il ne reste plus qu'à le résoudre à l'aide des algorithmes préalablement implémentés.

Cependant, cette approche demande de générer un graphe dans le bon format, et il s'est avéré que générer un telle en mémoire et en temps (requiert beaucoup de mémoires avec cette modélisation). La 1ère implementation est disponible dans le dossier '.bkp' de l'archive, on ne s'attardera pas dessus.

2.2 Optimisation

Après avoir ré-implementer A* spécifiquement au problème, en travaillant directement sur la grille, voici les résultats obtenus:

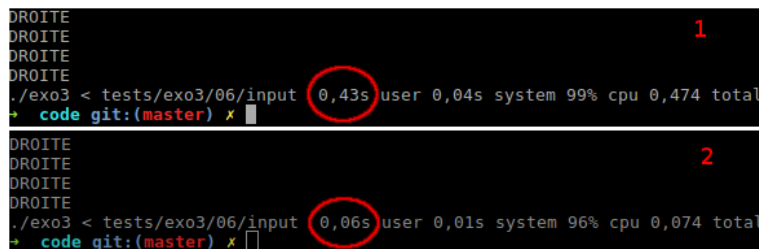


Figure 9: *performances sur 'tests/exo3/06'*

- 1: (2.1) avec A* et la distance de Manhattan. : Le programme passe 90 (0.4sec)% de son temps à générer le graphe, 10% (0.05 sec) à résoudre.
- 2: (2.2) avec A* et la distance de Manhattan : le programme ne génère pas de graphes, il résout en travaillant sur la grille directement. (on utilise également 4 fois moins de mémoire qu'avec la 1ère approche).

2.3 Parallélisation

Pour paralléliser la recherche, on crée un modèle 'client/serveur'. Le processus père est le 'serveur', et les processus fils sont les 'clients'.

Coté client, chaque processus calcule une partie du chemin à l'aide de l'algorithme A* une partie du chemin ((entrée, clef), (clef, porte), (porte, sortie)) Dès qu'un chemin (ou un chemin plus court que le précédent) est trouvé, il notifie le serveur en lui envoyant des données via un 'pipe'.

Coté serveur, le processus père ne fait que lire les chemins des clients, et essaye de trouver une solution en concaténant les chemins. Si une solution convient, il notifie ces clients (via des 'signaux') d'afficher leur chemin, et de s'arrêter.

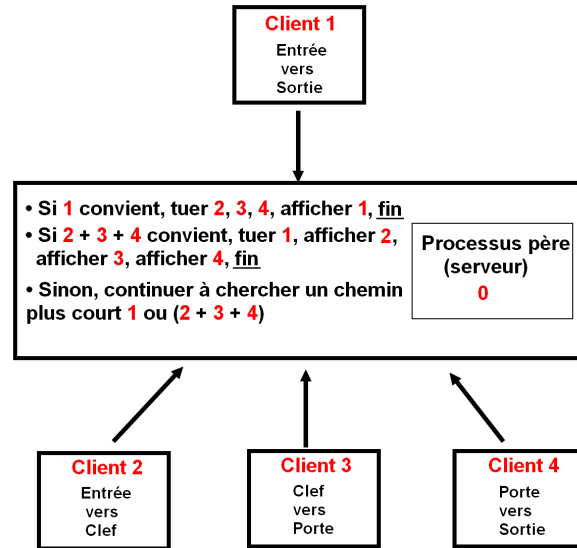


Figure 8: *Modèle client/serveur*

Ce modèle n'est pas optimal. Par exemple, si un chemin est trivial, et que le client le trouve rapidement, il passera le reste du temps à attendre que les autres aient fini pour savoir s'il doit afficher son chemin ou non.

On aurait pu imaginer que dès qu'un processus ait fini, il aide un autre client à résoudre une autre partie du chemin (avec une autre heuristique par exemple, ou bien encore en divisant le chemin en 2 à l'aide d'un point subsidiaire à 'mi-chemin'). Mais pour des raisons de simplicité, on reste sur ce modèle qui offre déjà de belles performances:

3 A propos de l'implémentation

Ci-joint, vous trouverez mon implémentation en langage 'C'.

3.1 Types / Structures de données

Les structures de données de manière générique, afin de pouvoir être ré-utilisé. (voir les fichiers '.c' et '.h', qui sont commentés).

3.2 Qualité logiciel

Mes programmes passent les tests fournis. J'ai également créé quelques tests supplémentaires pour mettre en défaut mon programme. (notamment pour l'illustration 6)

De plus, j'ai débuggé 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 initialisée. (avec des fichiers bien formatés du moins).

```
+ code git:(master) x valgrind ./exo3 < tests/exo3/18/input > /dev/null
==12759== Memcheck, a memory error detector
==12759== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==12759== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==12759== Command: ./exo3
==12759==
==12761==
==12761== HEAP SUMMARY:
==12761==   in use at exit: 0 bytes in 0 blocks
==12761==   total heap usage: 56,548 allocs, 56,548 frees, 61,429,488 bytes allocated
==12761== All heap blocks were freed -- no leaks are possible
==12761==
==12762== HEAP SUMMARY:
==12762==   in use at exit: 0 bytes in 0 blocks
==12762==   total heap usage: 61,036 allocs, 61,036 frees, 61,600,296 bytes allocated
==12762== All heap blocks were freed -- no leaks are possible
==12762==
==12763== HEAP SUMMARY:
==12763==   in use at exit: 0 bytes in 0 blocks
==12763==   total heap usage: 66,467 allocs, 66,467 frees, 61,603,992 bytes allocated
==12763== All heap blocks were freed -- no leaks are possible
==12763==
==12759== HEAP SUMMARY:
==12759==   in use at exit: 0 bytes in 0 blocks
==12759==   total heap usage: 1,003 allocs, 1,003 frees, 4,016,320 bytes allocated
==12759== All heap blocks were freed -- no leaks are possible
```

Figure 10: *résultat de 'valgrind' sur 'tests/exo3/06'*

Les performances du programme ont été étudié à l'aide de l'outil 'gprof'. C'est notamment ce qui m'a dirigé vers l'implémentation de tas binaires par exemple, ou encore de la 2ème approche dans la résolution du labyrinthe.

4 Références

- [1] 'Writting Cache-Friendly code', Gerson Robboy, Portland State University
<http://web.cecs.pdx.edu/~jrb/cs201/lectures/cache.friendly.code.pdf>.
- [2] Wikipédia, Binary Heap, 15 Décembre 2017,
https://en.wikipedia.org/wiki/Binary_heap.
- [3] Mary K. VERNON, Priority Queues, 3 Septembre 2016,
<http://pages.cs.wisc.edu/~vernon/cs367/notes/11.PRIORITY-Q.html>.
- [4] Dr. Mike POUND, Sean RILEY, 24 Février 2017,
Maze Solving - Computerphile,
<https://www.youtube.com/watch?v=rop0W4QDOUI>.
- [5] Dr. Mike POUND, Sean RILEY, 15 Février 2017,
A* (A Star) Search Algorithm - Computerphile,
<https://www.youtube.com/watch?v=ySN5Wnu88nE>.
- [6] Wikipédia, Heuristique, 31 Octobre 2017,
[https://fr.wikipedia.org/wiki/Heuristique_\(mathématiques\)](https://fr.wikipedia.org/wiki/Heuristique_(mathématiques)).
- [7] Wikipédia, Distance de Manhattan, 31 Octobre 2017,
https://fr.wikipedia.org/wiki/Distance_de_Manhattan.