

# Projet IPI: chemins de poids minimum

Romain PEREIRA

04/12/2017

## Sommaire

<b>1 Recherche de chemin le plus court</b>	<b>2</b>
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 2ème approche	3
1.2.3 File de priorité ('Priority Queue')	3
1.3 Algorithme A* (graphes pondérés et fonctions heuristiques)	4
<b>2 Application: resolution labyrinthe</b>	<b>5</b>
<b>3 A propos de l'implémentation</b>	<b>7</b>
3.1 Structures de données	7
3.2 Qualité logiciel	7
<b>4 Références</b>	<b>7</b>

## 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

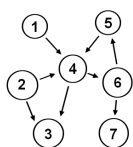


Figure 1:  
*graphe*  $G$   $n=7$ ,  
 $X=\{1, 2, \dots, 7\}$ ,  
 $A=\{(1, 4), (4, 6), \dots, (6, 7)\}$

# 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' '

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,

- $b = 8 * b_o + b_b$

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

- $u = b\%n$
- $v = b/n$
- $b = n * v + u$

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 stocké sur un octet. De plus, je ne perd pas de temps en lecture / écriture dans le tableau des arcs, car ces changements de coordonnées ne nécessite 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 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.

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.

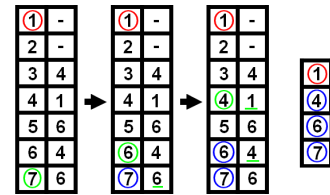


Figure 2: Schéma de l'algorithme de remonté

Cette modélisation permet de réduire les coûts de stockage, et la remonté 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 reprends le squelette du parcours en largeur.

On remplace le tableau de bit par une matrice d'entier (4 octets), stockant le poids des arcs.

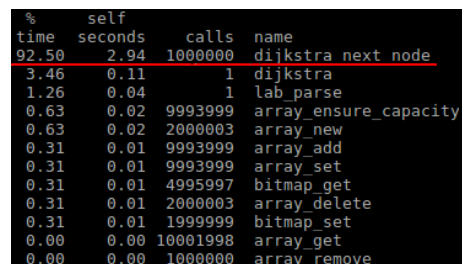
Cependant, une différence réside dans le choix du prochain sommet à visiter. Dans le parcours en largeur, les sommets sont visités dans leur ordre d'apparition dans la liste (1er entré, 1er sorti). Trouver le prochain sommet à visiter est d'une complexité  $O(1)$ . Dans l'algorithme de Dijkstra, les 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 2ème approche

En utilisant une matrice, la complexité ('spatial') de stockage des arcs est donc en  $O(n^2)$ . Plus précisément, si le poids est stocké sur 4 octet, pour  $n = 10^6$  (cas labyrinthe exo3/tests/06), l'espace mémoire nécessaire est de 400Go... On va donc changer de structures de données. En ajoutant un attribut liste à chaque sommet, qui contient l'indice de ses sommets voisins, la complexité spatial devient 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 400Go à 5Mo.

### 1.2.3 File de priorité ('Priority Queue')

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 3: 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).

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):

- '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 trouvés en annexe [1], ou directement dans mon implementation. (voir *pqueue.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 [4] 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 le sommet de destination plus rapide. (et avec une heuristique bien conçue au problème, on s'assure tout de même une solution proche de l'optimal). Par exemple, dans la résolution de labyrinthe, une heuristique intéressante est la distance de manhattan [5], ou encore l'heuristique 'diagonal' (on oriente la recherche sur la diagonale du labyrinthe)

Remarquons également qu'en utilisant une heuristique nulle (qui renvoie toujours 0), on obtient très exactement l'algorithme de Dijkstra. (voir 'dijkstra.c.aster'). Pour des raisons d'optimisations j'ai cependant préféré garder 2 implementations complètes distinctes. (avec une heuristique nulle, on peut économiser son stockage mémoire et quelques additions)

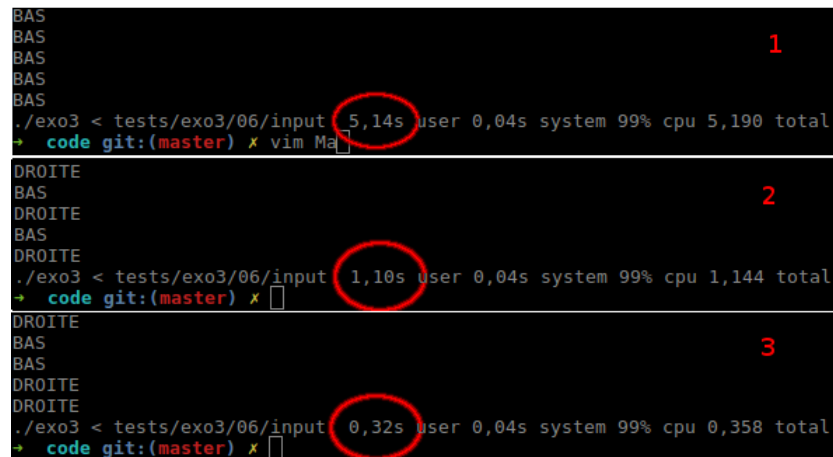


Figure 4: Temps d'exécution des algorithmes

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)

- 1 : Dijkstra sans file de priorité
- 2 : Dijkstra avec file de priorité
- 3 : A\* avec file de priorité, et fonction heuristique ('rester sur la diagonale du labyrinthe')

## 2 Application: resolution labyrinthe

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

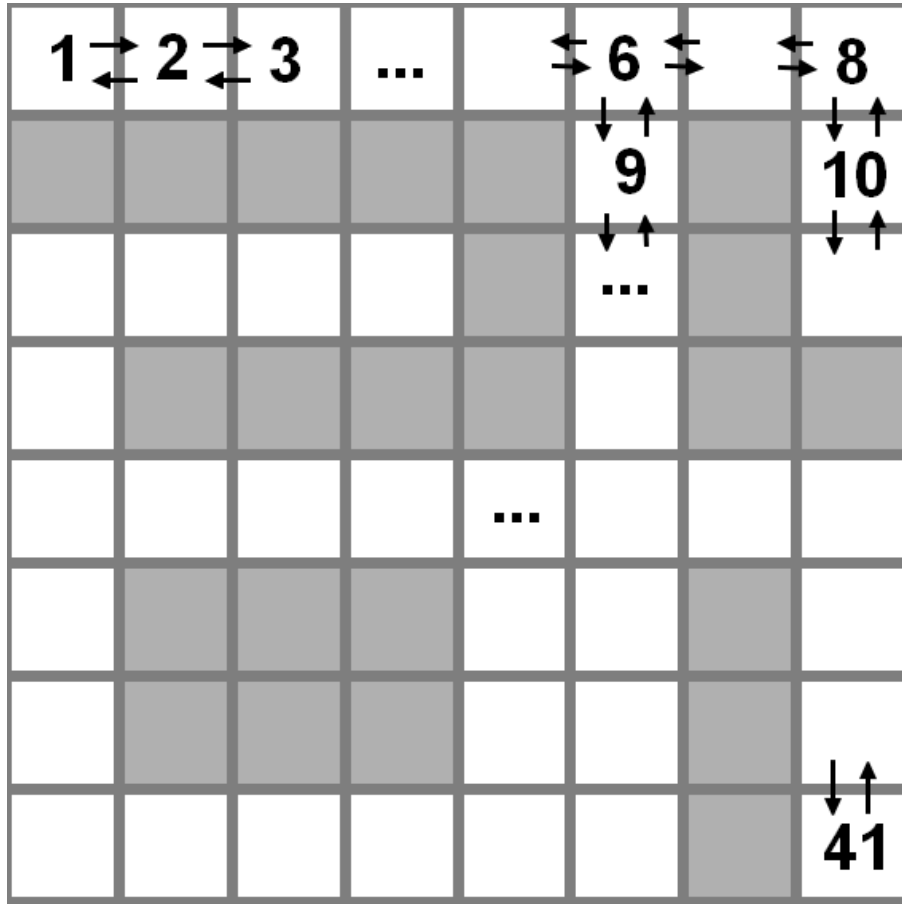


Figure 5: *Représentation du labyrinthe sous forme de graphe*

Voici les résultats

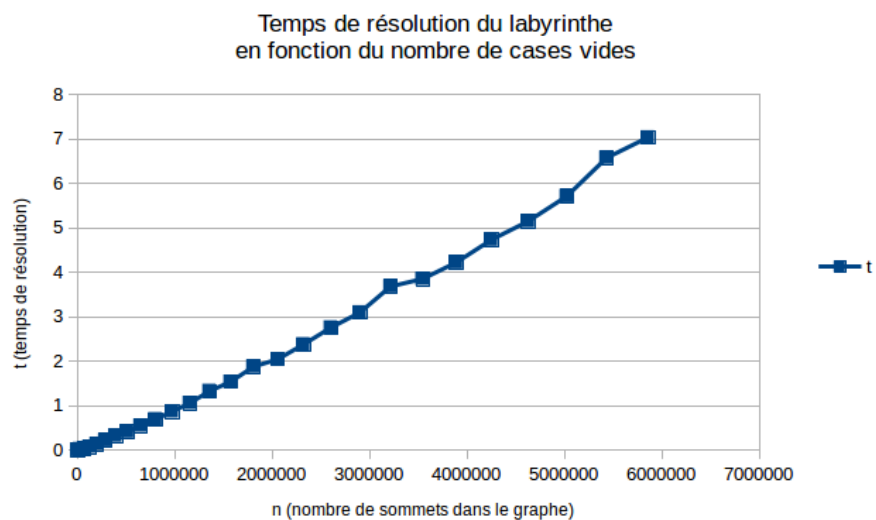


Figure 6: *Temps de résolution de labyrinthe avec Dijkstra, allure en  $O(n * \log(n))$*

## 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 vivement de regarder les fichiers '.c' et '.h', qui sont commentés.

### 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, avec des fichiers bien formatés).

```
==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==
==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 7: résultat de 'valgrind' sur 'tests/exo3/06')

J'ai optimisé mon programme à l'aide de l'outil 'gprof'. D'où par exemple, l'implémentation des tas binaires, ou du 'define' ARRAY\_ITERATE ('array.h')

## 4 Références

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