

# Projet IPI: chemins de poids minimum

Romain PEREIRA

04/12/2017

## Sommaire

<b>1</b>	<b>Recherche de chemin le plus court</b>	<b>2</b>
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 remonté	3
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)	5
<b>2</b>	<b>Application: résolution labyrinthe</b>	<b>5</b>
<b>3</b>	<b>A propos de l'implémentation</b>	<b>6</b>
3.1	Structures de données	6
3.2	Qualité logiciel	8
<b>4</b>	<b>Références</b>	<b>9</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

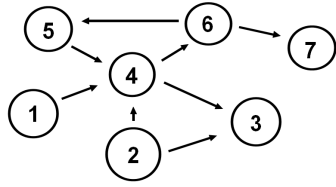


Figure 1:  
*graphe*  $G$   $n=7$ ,  
 $X=\{1, 2, \dots, 7\}$ ,  
 $A=\{(1, 4), (4, 6), \dots, (6, 7)\}$

## 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' '

#### 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)

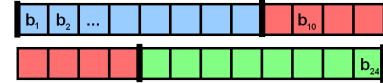


Figure 2: *Schéma bits*

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 bit. 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 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 necessite que 1 multiplication, 1 addition, et quelques operations 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.

#### 1.1.2 2ème approche

La complexité ('spatial') 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 structures de données. En ajoutant un attribut liste à chaque sommet, qui contient l'indice de ses sommets voisins, la complexité spatial devient donc (au plus) en  $m * O(n)$ , où 'm' est le

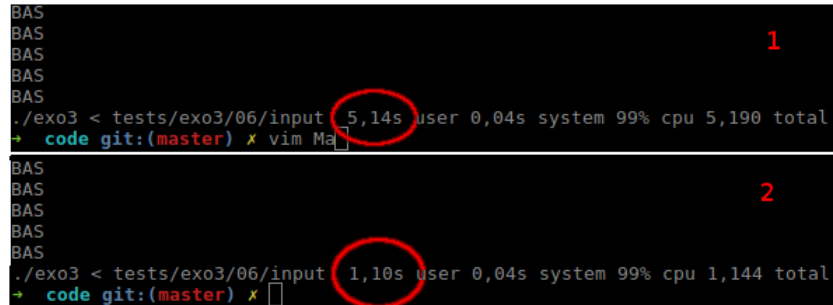


Il s'est avéré que mon programme passe (en moyenne) plus de 70% de son temps d'exécution à chercher le prochain sommet à visiter. (l'opération 'trouver un sommet 'u' non visité minimisant d(u)' dans l'algorithme fourni).

Ainsi, j'ai décidé d'implémenter des 'files de priorités', et plus précisément des 'tas binaire', afin d'optimiser cette partie du programme. 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 trouvé en annexe [1], ou directement dans mon implementation. (voir *pqueue.c,h*)



```
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

BAS
BAS
BAS
BAS
BAS
./exo3 < tests/exo3/06/input 1.10s user 0.04s system 99% cpu 1.144 total
+ code git:(master) x
```

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 l'implémentation de Dijkstra (file de priorité)

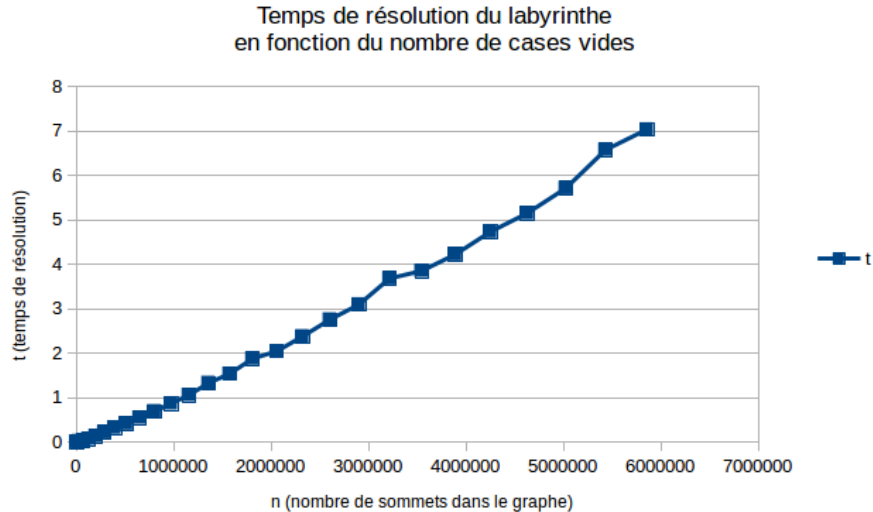


Figure 6: *Temps de résolution de labyrinthe avec Dijkstra, allure en  $O(n * \log(m))$*

### 1.3 Algorithme A\* (graphes pondérés et fonctions heuristiques)

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

## 2 Application: résolution labyrinthe

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

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

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

L'idée est dans un premier temps, de tenter de résoudre avec A\* ('Distance de Manhattan').

- Si la solution convient, alors on a (probablement) gagné du temps en utilisant A\* plutôt que Dijkstra. C'est le cas dans le test exo3/06.

```

DROITE
DROITE
DROITE
DROITE
./exo3 < tests/exo3/06/input 0,43s user 0,04s system 99% cpu 0,474 total
+ code git:(master) x

BAS
DROITE
BAS
DROITE
./exo3 < tests/exo3/06/input 1,16s user 0,03s system 99% cpu 1,193 total
+ code git:(master) x

```

Figure 8: *performances sur 'tests/exo3/06'*

- 1: avec A\* et la distance de Manhattan. (on obtient alors une solution optimal)
- 2: avec Dijkstra seulement.
- Si la solution ne convient pas, on résout avec Dijkstra. (on a alors perdu du temps, car la résolution avec A\* a été 'inutile'). C'est le cas dans exo3/18. Si Dijkstra échoue, il n'a pas de solutions.

```

DROITE
DROITE
DROITE
DROITE
./exo3 < tests/exo3/18/input 1,79s user 0,05s system 99% cpu 1,844 total
+ code git:(master) x

DROITE
DROITE
DROITE
DROITE
./exo3 < tests/exo3/18/input 1,18s user 0,06s system 99% cpu 1,244 total
+ code git:(master) x

```

Figure 9: *performances sur 'tests/exo3/18'*

- 1: avec A\* et la distance de Manhattan. (on n'obtient pas une solution valide), puis Dijkstra.
- 2: avec Dijkstra directement. (Remarque: A\* prends donc environ 0.61 secondes à s'exécuter sur ce test)

Voici l'algorithme de résolution:

### 3 A propos de l'implémentation

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

#### 3.1 Structures de données

J'ai conçu mes structures de données de manière générique, afin de pouvoir m'en réserver plus tard dans d'autre projet. (cela ajoute un peu 'd'overhead' à mon programme, il peut donc encore être optimisé) Si vous souhaitez plus de détail, voir les fichiers '.c' et '.h', qui sont commentés.

---

Algorithm 1: Résolution labyrinthe

---

**Ensure:** Résout le labyrinthe.

**Require:**  $G = (X, A); E, S, C, P \in X; t \in \mathbb{R}$  ;  $E$  l'entrée,  $S$  la sortie,  $C$  la clef,  $P$  la porte,  $t$  le nombre d'action. On suppose que les fonctions 'Dijkstra' et 'A\*' renvoie le nombre d'action nécessaire pour accomplir le chemin trouvé

**Begin**

Fermer porte.

**if**  $A^*(G, E, S) \leq t$  **or**  $Dijkstra(G, E, S) \leq t$  **then**

**print** Il existe un chemin valide sans passer par la porte. **Renvoyer Vrai**

**end if**

**print** Il n'existe pas de chemin valide sans passer par la porte.

$t_1 \leftarrow A^*(G, E, C)$

**if**  $t_1 < t$  **then**

    Ouvrir porte ,  $t_2 \leftarrow t - t_1$

**if**  $A^*(G, C, S) < t_2$  **or**  $Dijkstra(G, C, S) \leq t_2$  **then**

**print** Il existe un chemin valide passant la porte. **Renvoyer Vrai**

**end if**

    Fermer porte

**end if**

$t_1 \leftarrow Dijkstra(G, E, C)$

**if**  $t_1 < t$  **then**

    Ouvrir porte ,  $t_2 \leftarrow t - t_1$

**if**  $A^*(G, C, S) < t_2$  **or**  $Dijkstra(G, C, S) \leq t_2$  **then**

**print** Il existe un chemin valide passant la porte. **Renvoyer Vrai**

**end if**

**end if**

**print** Il n'existe pas de chemin valide pour aller chercher la clef. **Renvoyer Faux**

---

## 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 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. (avec des fichiers bien formatés du moins).

1	2	3	...	6	7	8
				9		10
				...		
			...			

```

==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)

```

rinthe

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

J'ai optimisé mon programme à l'aide de l'outil 'gprof'. C'est ce qui m'a dirigé vers l'implémentation de tas binaires par exemple, ou du 'define' ARRAY\_ITERATE ('array.h') utilisant l'arithmétique des pointeurs.



## 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\*](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\*](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).