

Exploration algorithmique d'un problème

SAÉ S2.02 – IUT Charlemagne

Alexandre PERROT – Nathan PAULIN

Sommaire

1. Représentation d'un graphe.....	2
2. Calcul du plus court chemin par point fixe	3
3. Calcul du meilleur chemin par Dijkstra.....	4
4. Validation et expérimentation	5
5. Extension : Intelligence Artificielle et labyrinthe	11
6. Conclusion	14

1. Représentation d'un graphe

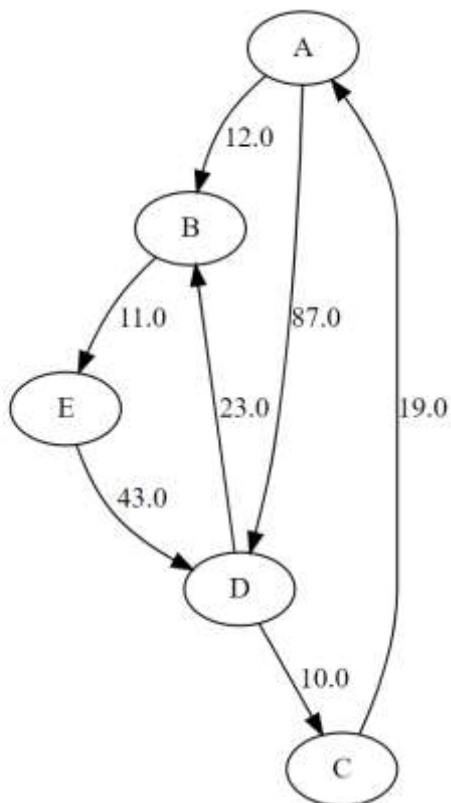
Dans cette partie, nous devons implémenter le type abstrait de donnée **Graphe** en Java.

Afin d'implémenter un graphe en Java, nous avons d'abord implémenté une classe « **Nœud** » représentant un nœud quelconque du graphe et également une classe « **Arc** » afin de créer des chemins avec un poids entre chaque nœud du graphe.

Comme un graphe est un TAD, nous avons créé une interface « **Graphe** » afin d'avoir plusieurs implémentations possibles, nous avons également créé la classe « **GrapheListe** » représentant un graphe sous forme concrète (avec une liste contenant les nœuds)

Nous avons créé différents constructeurs afin de pouvoir créer des graphes, soit manuellement (en ajoutant les nœuds dans le code un par un) ou bien à partir d'un fichier texte.

Nous avons également implémenté des méthodes afin d'afficher un graphe sous forme de chaîne de caractère et sous forme d'un GraphViz.



Question 10 : Représentation d'un graphe avec un GraphViz avec notre implémentation en Java

Nous avons fait des tests unitaires afin de s'assurer que les graphes créés sont affichés correctement et qu'ils sont reliés entre eux comme demandé.

Et enfin nous avons créé un main pour tester « **GrapheListe** »

2. Calcul du plus court chemin par point fixe

Dans cette partie, nous devons implémenter l'algorithme de Bellman-Ford ou plus communément appelé l'algorithme du point fixe, c'est un algorithme permettant de rechercher le plus court chemin dans un graphe et l'algorithme le recherche jusqu'à qu'il n'y ait plus de chemin plus court (point fixe)

Question 13 : Un algorithme du point fixe sous forme de pseudo-code

```
fonction pointFixe(g : Graphe InOut, depart : Noeud)
début
  Pour x dans g faire
    L(X) <- +inf
  Fpour
  L(depart) <- 0

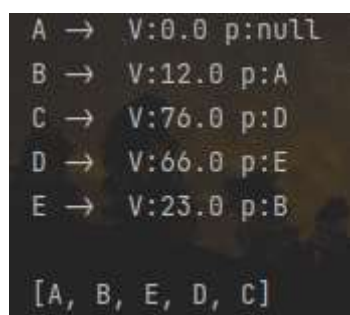
  Tant que non point fixe faire
    Pour x dans g faire
      Pour arcX dans x faire
        si L(x) + d(arcX, x) < L(arcX) alors
          L(arcX) <- L(x) + d(arcX, x)
          parent(arcX) <- x
        Fsi
      Fpour
    Fpour
  Ftant
Fin
```

Ensuite nous avons implémenté l'algorithme du point fixe en Java avec une méthode *resoudre(Graphe g, String noeudDepart)* et qui renvoie un objet « **Valeur** » contenant chaque nœud avec le poids minimal et son nœud parent.

Nous avons créé un main afin de tester l'algorithme sur le graphe fourni dans la figure 1 du sujet à partir du nœud A et nous avons également fait un test unitaire afin de tester l'algorithme également et vérifier si ses valeurs correspondent aux valeurs que nous avons trouvé manuellement (nous avons appliqué la méthode du point fixe sur papier afin d'être sûr d'avoir le même chemin court que l'algorithme)

Et enfin, nous avons implémenté une méthode supplémentaire dans la classe « **Valeur** » permettant de faire du backtracking, c'est-à-dire retracer le chemin du nœud de départ vers le nœud de destination, et comme on stocke le parent de chaque nœud, c'est très facile de retrouver son chemin.

Voici un exemple de résultat de l'algorithme de Bellman-Ford sur le graphe de la figure 1 (la première partie correspond aux valeurs de chaque nœud après convergence de l'algorithme, et la seconde partie est le chemin le plus court entre le nœud A et le nœud C)



```
A → V:0.0 p:null
B → V:12.0 p:A
C → V:76.0 p:D
D → V:66.0 p:E
E → V:23.0 p:B

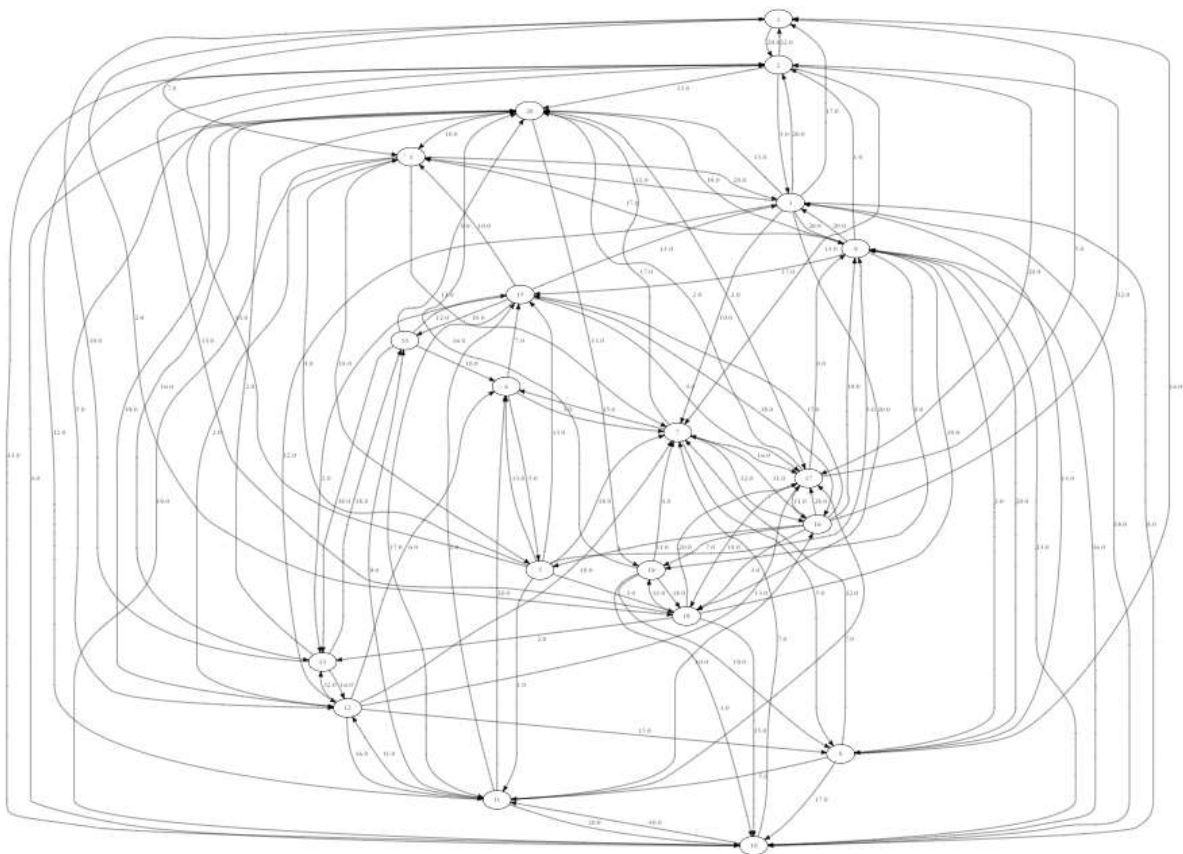
[A, B, E, D, C]
```

3. Calcul du meilleur chemin par Dijkstra

Dans cette partie, nous devons implémenter l'algorithme de Dijkstra, c'est un algorithme permettant de rechercher le plus court chemin dans un graphe et l'algorithme le recherche à partir du premier nœud ayant la plus petite valeur et parcourt qu'une fois dans tous les nœuds du graphe.

Tout d'abord, nous avons implémenté l'algorithme de Dijkstra en Java dans la classe Dijkstra à partir de l'algorithme du sujet.

Puis dans le MainDijkstra, nous testons l'algorithme sur un graphe créé depuis un fichier texte qui est de taille 20.



```
1 → V:0.0 p:null
10 → V:17.0 p:18
11 → V:21.0 p:8
12 → V:9.0 p:4
13 → V:4.0 p:16
14 → V:22.0 p:13
15 → V:22.0 p:6
16 → V:19.0 p:17
17 → V:8.0 p:20
18 → V:2.0 p:1
19 → V:16.0 p:18
2 → V:13.0 p:9
20 → V:6.0 p:13
3 → V:17.0 p:2
4 → V:7.0 p:1
5 → V:20.0 p:6
6 → V:15.0 p:12
7 → V:18.0 p:6
8 → V:16.0 p:9
9 → V:12.0 p:18
[1, 18, 13, 14]
```

Ensuite nous recherchons le chemin le plus court à partir du nœud « 1 »

Et on affiche la valeur minimal de chaque nœud du graphe et nous affichons le chemin le plus court grâce au backtracking entre le nœud « 1 » et le nœud « 14 »

Nous avons également fait un test unitaire pour valider l'algorithme et qu'il renvoie bien le plus court chemin.

4. Validation et expérimentation

Dans cette partie, nous allons expérimenter avec les deux algorithmes afin de pouvoir les comparer en termes d'efficacité.

Question 21 : Les images qui vont suivre représente les itérations des deux algorithmes afin de déterminer lequel demande le plus d'itération.

```
BellmanFord - Initialisation → :  
A → V:0.0 p:null  
B → V:1.7976931348623157E308 p:null  
C → V:1.7976931348623157E308 p:null  
D → V:1.7976931348623157E308 p:null  
E → V:1.7976931348623157E308 p:null  
F → V:1.7976931348623157E308 p:null  
G → V:1.7976931348623157E308 p:null  
  
BellmanFord - Étape 1 → :  
A → V:0.0 p:null  
B → V:9.0 p:C  
C → V:7.0 p:D  
D → V:3.0 p:A  
E → V:38.0 p:F  
F → V:35.0 p:G  
G → V:30.0 p:B  
  
BellmanFord - Étape 2 → :  
A → V:0.0 p:null  
B → V:9.0 p:C  
C → V:7.0 p:D  
D → V:3.0 p:A  
E → V:27.0 p:F  
F → V:24.0 p:G  
G → V:19.0 p:B  
  
BellmanFord - Étape 3 → :  
A → V:0.0 p:null  
B → V:9.0 p:C  
C → V:7.0 p:D  
D → V:3.0 p:A  
E → V:27.0 p:F  
F → V:24.0 p:G  
G → V:19.0 p:B
```

Algorithme de Bellman-Ford sur le graphe de la figure 10, Initialisation et trois itérations

```

Dijkstra - Initialisation → :
A → V:0.0 p:null
B → V:1.7976931348623157E308 p:null
C → V:1.7976931348623157E308 p:null
D → V:1.7976931348623157E308 p:null
E → V:1.7976931348623157E308 p:null
F → V:1.7976931348623157E308 p:null
G → V:1.7976931348623157E308 p:null

Dijkstra - Noeud A (arc: B) → :
A → V:0.0 p:null
B → V:20.0 p:A
C → V:1.7976931348623157E308 p:null
D → V:1.7976931348623157E308 p:null
E → V:1.7976931348623157E308 p:null
F → V:1.7976931348623157E308 p:null
G → V:1.7976931348623157E308 p:null

Dijkstra - Noeud A (arc: D) → :
A → V:0.0 p:null
B → V:20.0 p:A
C → V:1.7976931348623157E308 p:null
D → V:3.0 p:A
E → V:1.7976931348623157E308 p:null
F → V:1.7976931348623157E308 p:null
G → V:1.7976931348623157E308 p:null

Dijkstra - Noeud D (arc: C) → :
A → V:0.0 p:null
B → V:20.0 p:A
C → V:7.0 p:D
D → V:3.0 p:A
E → V:1.7976931348623157E308 p:null
F → V:1.7976931348623157E308 p:null
G → V:1.7976931348623157E308 p:null

```

```

Dijkstra - Noeud C (arc: B) → :
A → V:0.0 p:null
B → V:9.0 p:C
C → V:7.0 p:D
D → V:3.0 p:A
E → V:1.7976931348623157E308 p:null
F → V:1.7976931348623157E308 p:null
G → V:1.7976931348623157E308 p:null

Dijkstra - Noeud B (arc: G) → :
A → V:0.0 p:null
B → V:9.0 p:C
C → V:7.0 p:D
D → V:3.0 p:A
E → V:1.7976931348623157E308 p:null
F → V:1.7976931348623157E308 p:null
G → V:19.0 p:B

Dijkstra - Noeud G (arc: F) → :
A → V:0.0 p:null
B → V:9.0 p:C
C → V:7.0 p:D
D → V:3.0 p:A
E → V:1.7976931348623157E308 p:null
F → V:24.0 p:G
G → V:19.0 p:B

Dijkstra - Noeud F (arc: E) → :
A → V:0.0 p:null
B → V:9.0 p:C
C → V:7.0 p:D
D → V:3.0 p:A
E → V:27.0 p:F
F → V:24.0 p:G
G → V:19.0 p:B

```

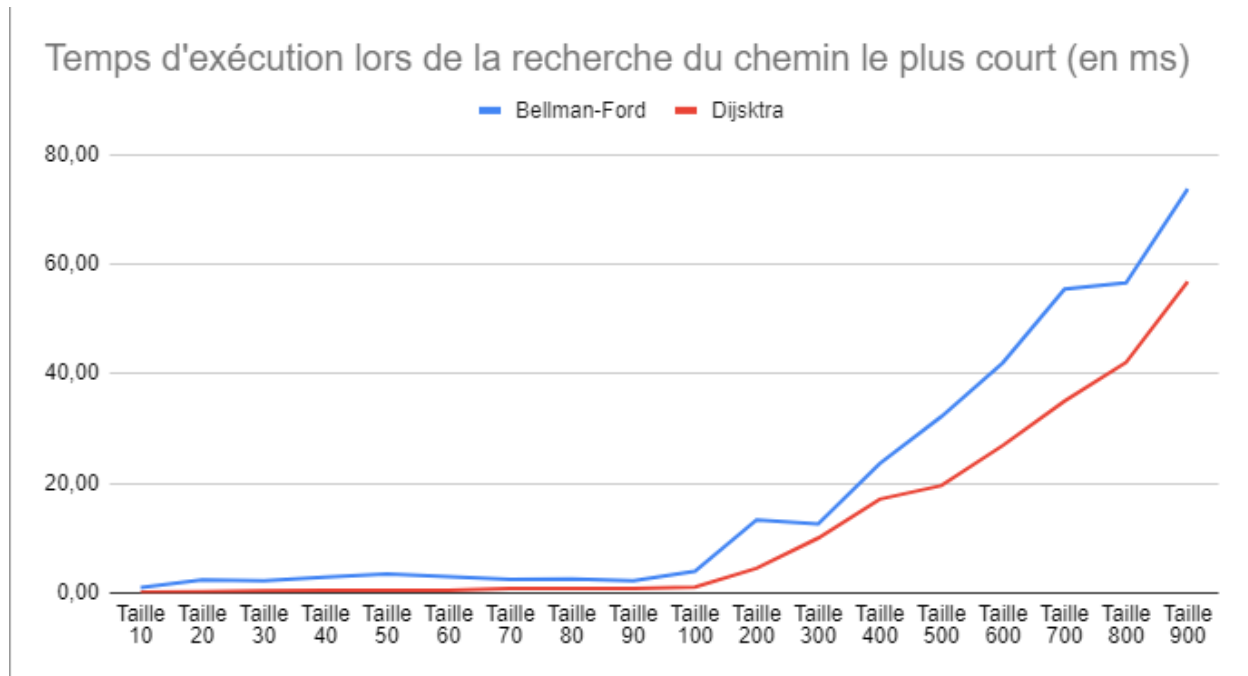
Algorithme de Dijkstra sur le graphe de la figure 10, Initialisation et sept itérations

On remarque qu'entre les deux algorithmes :

- Dijkstra prend le plus petit nœud de la liste des nœuds à traiter et vérifie chaque arc de chaque nœud du graphe même si le plus court chemin a été trouvé et ne passe qu'une fois sur chaque arc
- Bellman-Ford vérifie également chaque arc de chaque nœud mais s'arrête UNIQUEMENT quand l'algorithme a atteint un point fixe (càd quand il n'y a plus de différence entre l'itération précédente et l'actuelle), il peut donc passer plusieurs fois sur un même arc du graphe.

Question 22 : On peut donc conclure que Dijkstra est plus rapide sur le papier car il passe qu'une seule fois sur chaque arc de chaque nœud tandis que Bellman-Ford doit recalculer plusieurs fois le même arc pour avoir le plus petit chemin.

Question 23 :

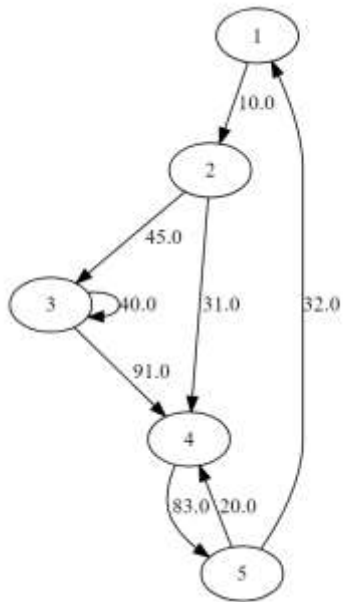


D'après nos tests effectués sur les graphes fournis par le sujet, on remarque que sur les graphes de taille 10 jusqu'à 900 nœuds, l'algorithme de Dijkstra est plus rapide que Bellman-Ford, cela peut s'expliquer par le fait que Dijkstra passe qu'une fois sur chaque arc de chaque nœud pour estimer le plus court chemin en partant du plus petit nœud d'une liste de nœuds à traiter. Tandis que Bellman-Ford, on parcourt plusieurs fois le même arc de chaque nœud en partant d'un nœud qui n'est pas le plus petit afin de déterminer le plus court chemin et donc le gain de performance s'explique par l'optimisation que propose Dijkstra en parcourant moins les nœuds.

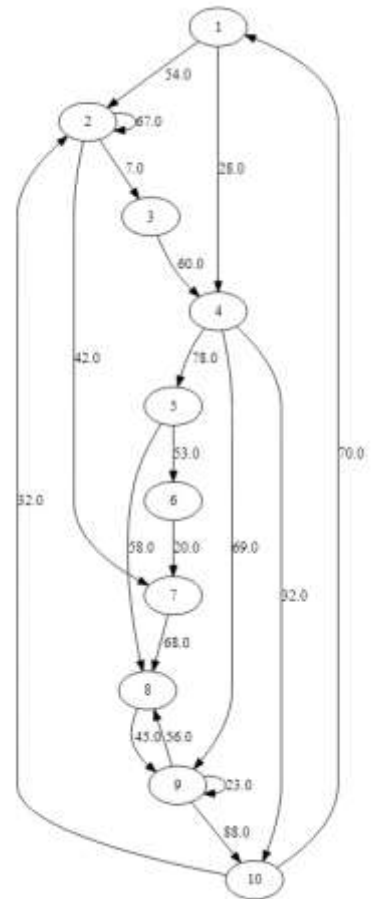
Question 24 et 25 : Nous avons décidé de générer des graphes d'une manière successive, c'est-à-dire que pour n nœuds donné, nous allons commencer le graphe au nœud 1 et on relie le nœud 1 à son nœud suivant, donc le nœud 2 et ce jusqu'au nœud n afin de garantir d'avoir un chemin passant par tous les nœuds du graphe et ensuite nous générons des arcs aléatoirement entre les nœuds du graphe avoir d'avoir de la diversité, il y'a autant d'arc que de nœuds qui sont générés aléatoirement pour avoir un nombre respectable de chemins.

Voici quelques exemples de graphes générés aléatoirement

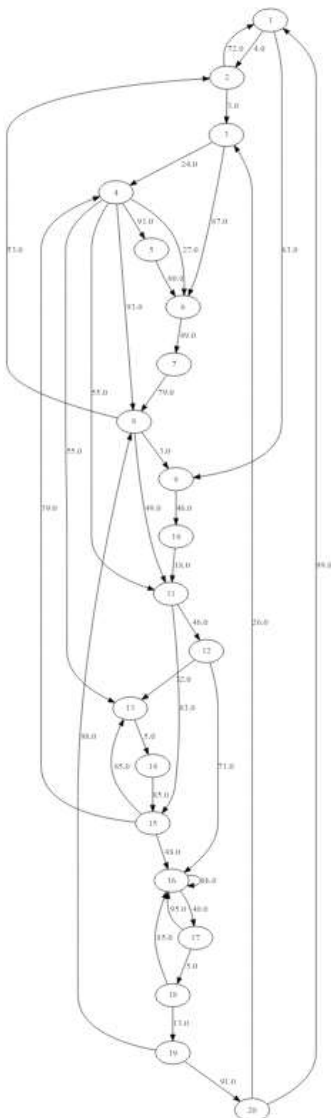
Graphe de taille 5



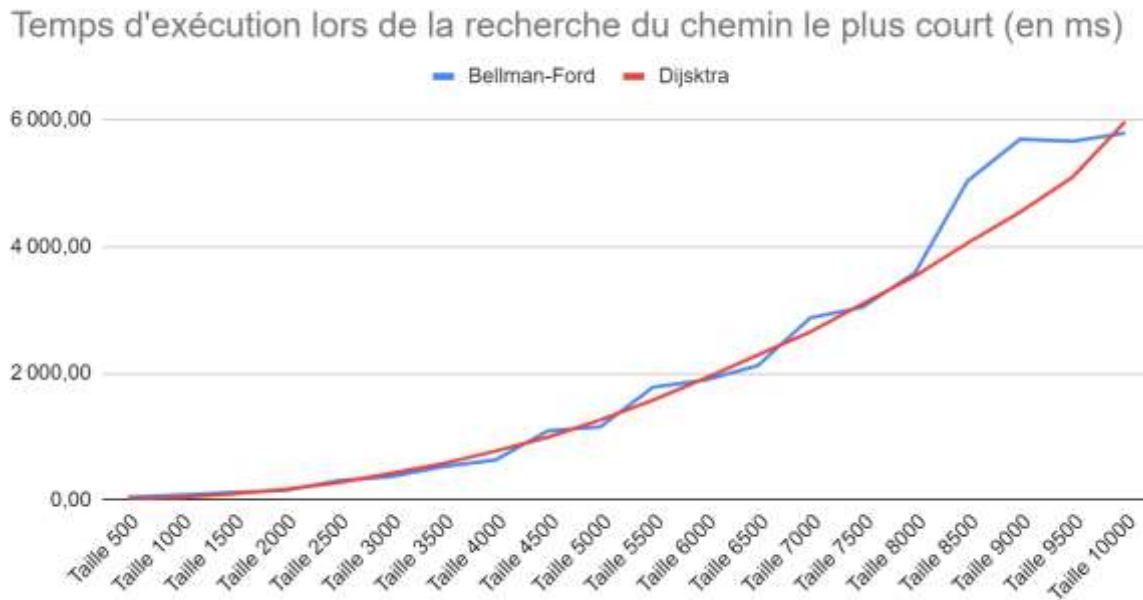
Graphe de taille 10



Graphe de taille 20



Question 26 :

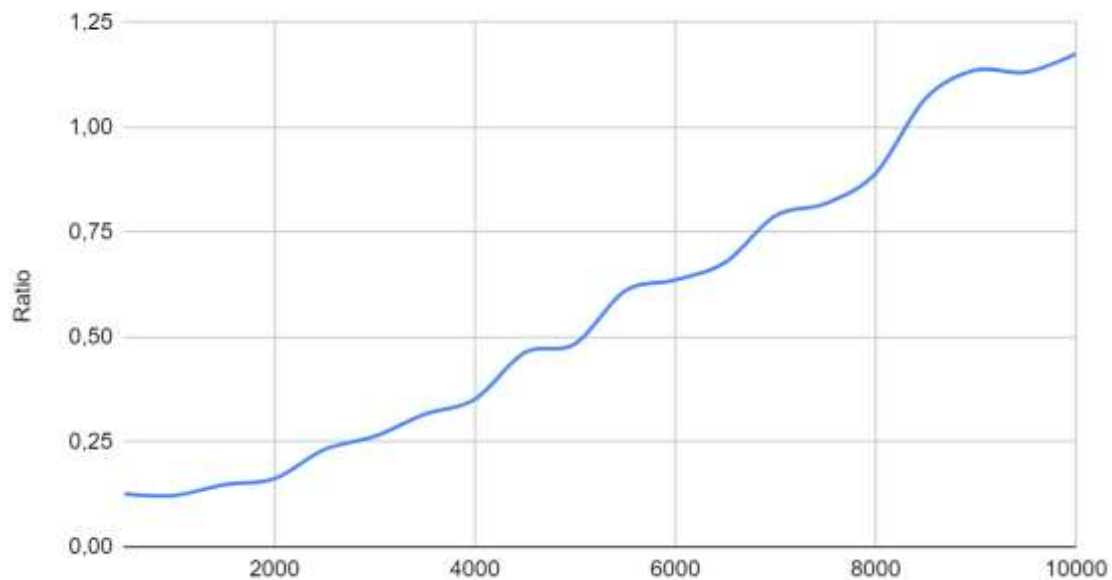


D'après nos tests effectués sur les graphes générés aléatoirement, on remarque que sur les graphes de taille 500 jusqu'à 10 000 nœuds, l'algorithme de Dijkstra est légèrement plus efficace que Bellman-Ford avec une moyenne respective de 1,98 secondes de temps d'exécution sur l'ensemble des nœuds des graphes pour Dijkstra et de 2,01 secondes de temps d'exécution sur l'ensemble des nœuds des graphes pour Bellman-Ford.

Nota Bene : Nous avons constaté que durant l'expérimentation qu'avec un graphe généré aléatoirement, l'algorithme de Bellman-Ford était plus efficace que Dijkstra lorsqu'il y avait peu de chemins entre les nœuds du graphe (quand le graphe commence à avoir beaucoup de nœuds) mais à partir du moment où nous avons rajouté plus de chemins lors de la génération du graphe, l'algorithme de Dijkstra était le plus efficace. Mais il faut noter que notre réglage pour la génération de graphes aléatoires est qu'il y a au moins un chemin passant par tous les nœuds du graphe et nous rajoutons autant de chemins que de nœuds dans le graphe aléatoirement. Et donc on peut voir que Dijkstra est légèrement plus efficace mais cela dépend énormément de comment le graphe est généré (si y'a beaucoup de chemins ou non) et on peut voir qu'à certains moments, l'algorithme de Bellman-Ford est plus rapide que Dijkstra sur un graphe de taille 6500.

Question 27 :

Ratio de performances



Le ratio de performances (qui est calculée par cette formule : $\text{TempsExecDijkstra} + \text{TempsExecBellman} / \text{TailleNoeud}$) n'est pas constant et est croissante. Cela signifie que les algorithmes sont plus performants quand il y a moins de nœuds et sont plus lents quand il y a plus de nœuds. Donc on peut dire que ces deux algorithmes dépendent du nombre de nœuds.

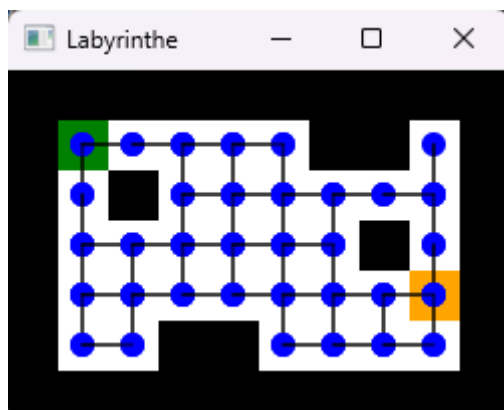
Question 28 : On peut donc conclure que Dijkstra est légèrement plus efficace que Bellman-Ford dans un contexte où le graphe contient beaucoup de chemins.

5. Extension : Intelligence Artificielle et labyrinthe

Nous avons décidé de faire la partie optionnelle et bonus en générant un graphe à partir du Labyrinthe et aussi d'implémenter une solution d'Adaptateur.

Question 30 : Après avoir créé la méthode *genererGraphe* dans le labyrinthe, nous devons tester la méthode afin de savoir si elle génère bien le graphe correctement et aussi de vérifier que la solution du chemin le plus court est bien valide avec les deux algorithmes. Mais cela est plus difficile de savoir si le graphe est bien généré et de vérifier si le chemin le plus court des deux algorithmes est bien le plus court avec uniquement des String.

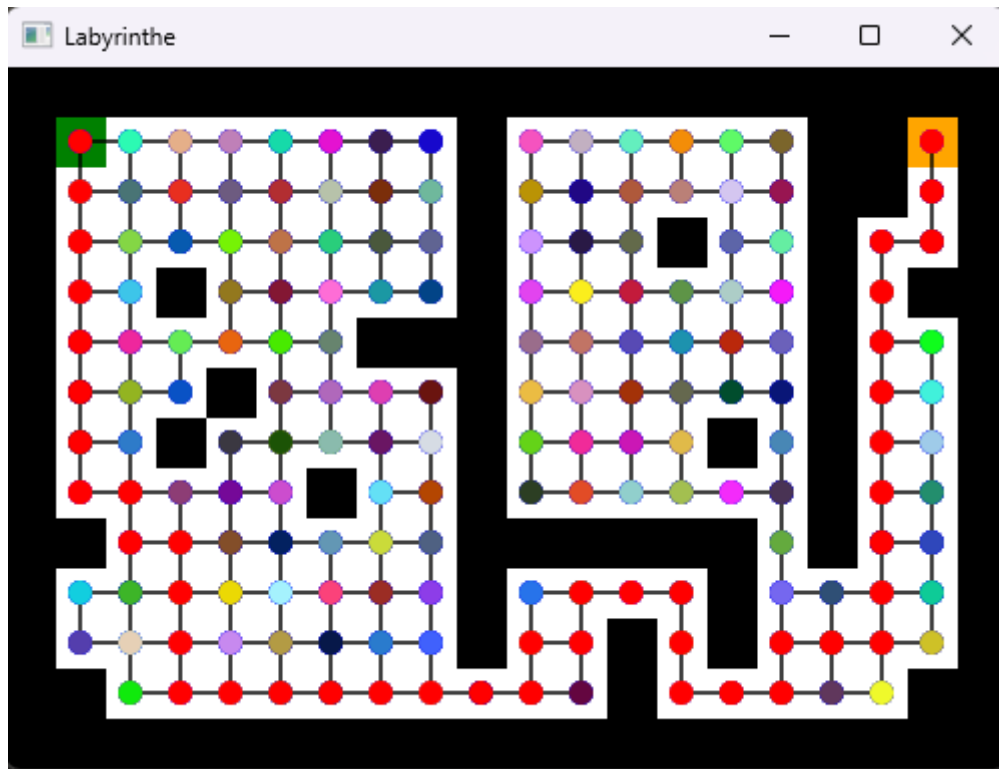
J'ai décidé de faire un outil de visualisation d'un labyrinthe et d'un graphe sous IHM afin d'avoir un retour visuel agréable et plus facile pour déterminer si le graphe est bien généré et que le chemin est bien le plus court.



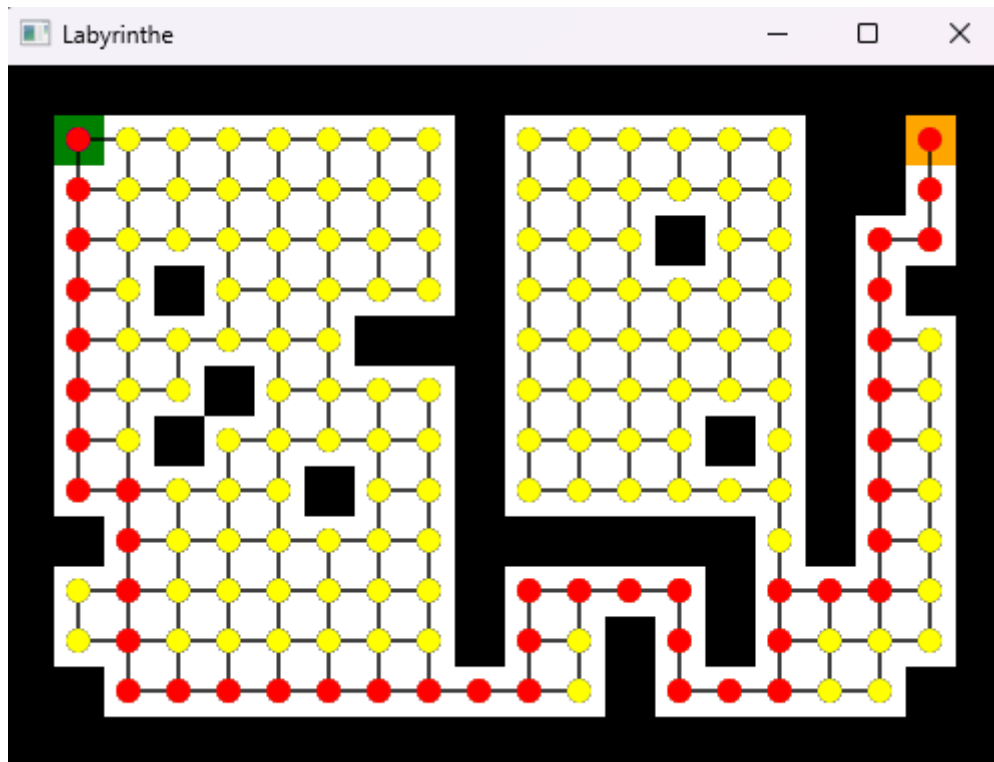
Les points en bleu représentent les nœuds du graphe et les traits noirs représente les arcs entre les nœuds, la case verte est le nœud de départ et la case orange est le nœud de destination.

Grâce à cette application, j'ai pu repérer de nombreux problèmes quant à ma génération de graphe du Labyrinthe, j'avais oublié que lors du parcours du labyrinthe, toutes les cases étaient un nœud (mur compris) et donc tout était relié, j'ai pu résoudre ça en rajoutant une condition de si la case X,Y était un mur alors on ne fait rien

Maintenant, il fallait déterminer si le chemin des algorithmes était bien le chemin le plus court (ou parmi les plus courts), j'ai utilisé mon application visuelle pour déterminer cela également, voici quelques exemples



Ceci est le visuel de l'algorithme de Bellman-Ford, les différentes couleurs sur les nœuds me permet de savoir si l'algorithme est passé plusieurs fois sur le même nœud (il change de couleur lors de l'exécution) et le chemin rouge est la solution la plus courte selon l'algorithme (c'est bien l'une des plus courtes)

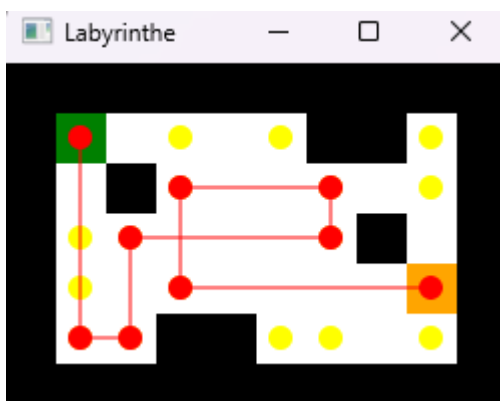


Ceci est le visuel de l'algorithme de Dijkstra, la couleur jaune me permet de savoir si l'algorithme a traité ce nœud (cela fait une propagation lors de l'exécution du programme) et le chemin rouge est la solution donnée par l'algorithme (le meilleur chemin possible)

Question 31 : Nous avons ensuite implémenté notre labyrinthe sous forme d'adaptateur avec un Graphe nommé *GrapheLabyrinthe*, et ses méthodes *listeNoeuds* représente toutes les cases libres du labyrinthe et *suivants(String n)* représente les déplacements possible à partir d'un nœud (Gauche, Droite, Haut, Bas).

Et grâce à cette implémentation sous forme d'adaptateur, nous avons pu réutiliser ce que nous avons fait en Qualité de Développement (labyrinthe ricochet) et nous avons pu trouver le plus court chemin sous un labyrinthe ricochet (donc qui ne peut s'arrêter de s'avancer jusqu'à tomber devant un mur)

Voici un exemple de chemin le plus court sous forme de labyrinthe ricochet



C'est également le chemin le plus court sous forme de ricochet.

6. Conclusion

Ce projet nous aura appris énormément sur les graphes d'un point de vue pratique, nous avons compris le fonctionnement des deux algorithmes et nous comprenons maintenant leur réel intérêt et leur efficacité sur des graphes de taille différentes.

Nous avons rencontré quelques difficultés sur ce projet comme l'algorithme de Dijkstra qui au début devait utiliser une file pour les nœuds à traiter mais il se trouve qu'il fallait utiliser une liste.

Et aussi le *genererGraphe* pour générer un graphe aléatoirement, la génération au début ressemblait plus à un arbre qu'un graphe et les graphes étaient mal reliés entre eux (Boucle entre 0 et 1 et connecté à rien d'autres donc la résolution du plus court chemin était impossible)

Et nous avons du mal à comprendre pourquoi Bellman-Ford était plus rapide que Dijkstra sur certains graphes et nous avons compris que Dijkstra dépend du nombre d'arcs dans le graphe, si le graphe était complet (tous les nœuds sont reliés entre eux), Dijkstra serait le plus efficace.

A l'issue de cette SAÉ, nous avons appris comment implémenter des graphes et les manipuler et d'utiliser des algorithmes qui nous serviront à par exemple, résoudre des labyrinthes, utiliser des GPS...