TP6 : Algorithme de DIJKSTRA

L'objectif de ce TP est d'implanter l'algorithme de DIJKSTRA vu en cours. Il est constitué de trois parties : une première partie pour créer des graphes aléatoires ; une deuxième partie pour l'algorithme de DIJKSTRA ; une troisième partie (bonus) pour implanter une variante de l'algorithme, appelée *algorithme A**. **Les parties 1 et 2 sont indépendantes et peuvent être traitées dans n'importe quel ordre.**

Les fichiers fournis sont :

— un fichier `tests.cpp` qui contient le `main` ;

— des fichiers `Structures.h/cpp` et `Affichage.h/cpp` pour les structures de données et l'affichage graphique ;

— deux fichiers `Dijkstra.h/.cpp` pour vos implantations ;

— un `Makefile`.

Le fichier `Dijkstra.cpp` est l'unique fichier à modifier et à rendre !

Représentation des graphes

On utilise une représentation des graphes par *listes d'adjacence*. La liste d'adjacence d'un sommet u est une liste de couples (v, p) où v est un voisin de u, et p est le poids de l'arête entre u et v. *Chaque arête est représentée deux fois* : une arête de poids p entre u et v apparaît aussi dans la liste d'adjacence du sommet v, comme le couple (u, p).

Informatiquement, une liste d'adjacence est une liste chaînée, composée de *maillons* (`class Voisin`). Chaque `Voisin` contient contient un `sommet` (`int`), un `poids` (`float`) et un pointeur `suivant` (`Voisin*`) vers le maillon suivant. Le constructeur `new Voisin(s, p, L)` crée un nouveau maillon dont le `sommet` est `s`, le poids `p` et `suivant` est le pointeur `L` vers le maillon suivant.

On nomme `listeAdj` le type `Voisin*`. Un graphe G à n sommets est représenté par un tableau de n listes d'adjacence (`listeAdj`), et est donc de type `listeAdj*`. Les sommets de G sont numérotés de 0 à $n-1$.

EXEMPLE

```
listeAdj* G = new listeAdj[n];           // Graphe à n sommets
for (int i=0; i < n; i++) G[i] = NULL;  // Initialisation des listes
G[5] = new Voisin(2, 1.5, G[5]);        // Ajout de l'arête de poids 1.5...
G[2] = new Voisin(5, 1.5, G[2]);        // ... entre les sommets 2 et 5
```

Files de priorité

Afin d'implanter l'algorithme de Dijkstra, une classe `File` est fournie pour les listes de priorités. La structure de donnée utilisée est un tas. Son utilisation est illustrée ci-dessous. En particulier, noter que le constructeur crée une liste qui contient tous les sommets 0 à $n-1$, chacun avec priorité $+\infty$.

EXEMPLE

```
File* F = new File(n); // File avec les sommets 0 à n-1, de priorités +INF chacun
F->est_vide();             // true si F est vide, false sinon
int u = F->extraire_min();   // Extrait et renvoie le sommet de priorité minimale
F->changer_priorite(7, 3.5); // Passe la priorité du sommet 7 à 3.5
F->afficher();             // Affiche l'état de la file
```

Graphes de tests

Dans les deuxième et troisième parties, quatre types de graphes peuvent être utilisés pour les tests :

— deux graphes prédéfinis, codés dans des fichiers `Graphe5.txt` et `Graphe10.txt`, représentés en figure 1 ;

— des graphes aléatoires à *obstacles*, où le poids d'une arête est la distance euclidienne entre les sommets (exemple en figure 2) ;

— des grilles aléatoires, où chaque arête est de poids 1 (exemple en figure 2) ;

— les graphes aléatoires qui sont l'objet de la première partie.

Tester les algorithmes avec des graphes de tailles croissantes, de différents types, jusqu'à quelques milliers de sommets.

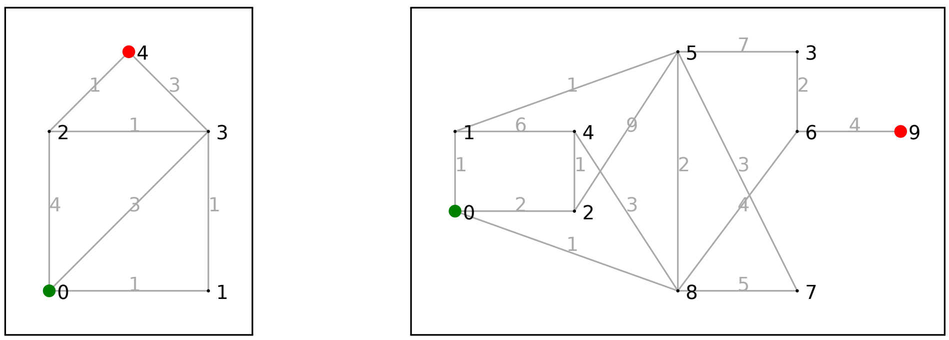


FIGURE 1 – Les deux graphes exemples fixés, avec le sommet de départ en vert et l'arrivée en rouge. Les poids des arêtes sont indiqués en grisé.

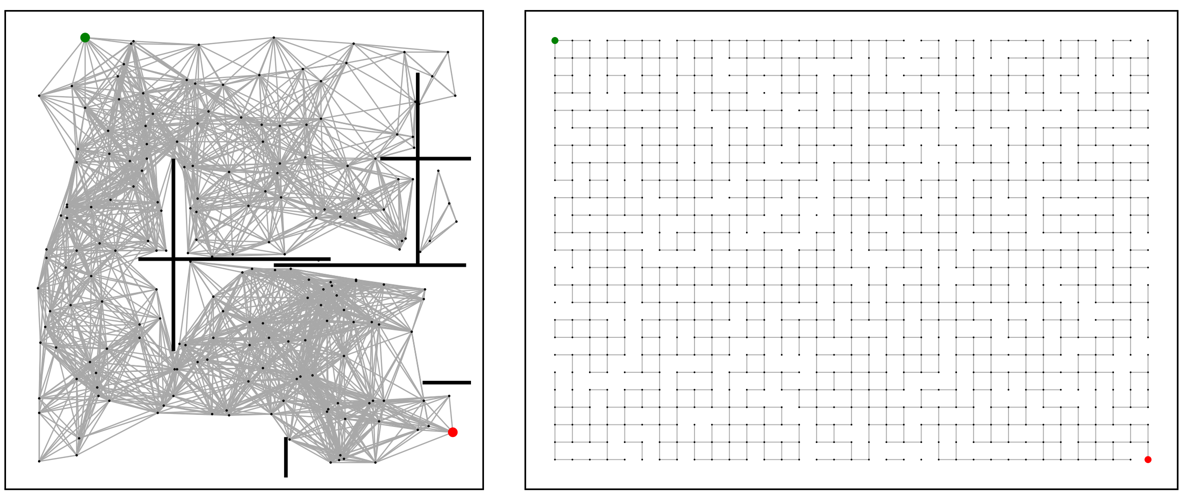


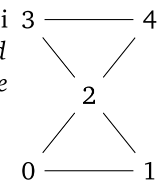
FIGURE 2 – Un exemple de graphe à obstacle à 200 sommets (arêtes grisées et obstacles noir) et une grille de dimension 35×25 avec 75% des arêtes.

Première partie – création de graphes aléatoires

1. Compléter la fonction `coord* sommetsAleatoires(int n, int l, int h)` qui tire n sommets aléatoires dont l'abscisse est comprise entre 10 et $l-10$ et l'ordonnée entre 10 et $h-10$.
   Un sommet `s` *est représenté par ses* `coord`*onnées, définies par son abscisse* `s.x` *et son ordonnée* `s.y`.

2. Compléter la fonction `float distance(coord* S, int i, int j)` qui calcule la distance euclidienne entre les sommets d'indices i et j de S. *On peut utiliser la fonction* `sqrt` *de la bibliothèque* `cmath` *pour cela.*

3. Compléter la fonction `listeAdj* graphe(int n, coord* S, float dmax)` qui crée un graphe dont les sommets sont les éléments de S, et qu'il existe une arête de poids d entre i et j si la distance euclidienne entre `S[i]` et `S[j]` est $d \leq d_{\max}$. *Ci-contre : exemple avec* $S = [(0,0),(8,0),(4,5),(0,10),(8,10)]$ *et* $d_{\max} = 8$.



TEST

```
Question à tester (0 pour sortir) : 1
Entrer le nombre de sommets : 50
Sommets dessinés dans graphe.svg

Question à tester (0 pour sortir) : 2
Entrer l'abscisse et l'ordonnée du premier sommet : 0 0
Entrer l'abscisse et l'ordonnée du second sommet : 1 1
Les sommets de coordonnées (0,0) et (1,1) sont à distance 1.41421

Question à tester (0 pour sortir) : 3
Entrer le nombre de sommets : 50
Entrer la distance maximale (recommandé : env. 42.4264) : 40
Graphe dessiné dans graphe.svg

Question à tester (0 pour sortir) : 0
```

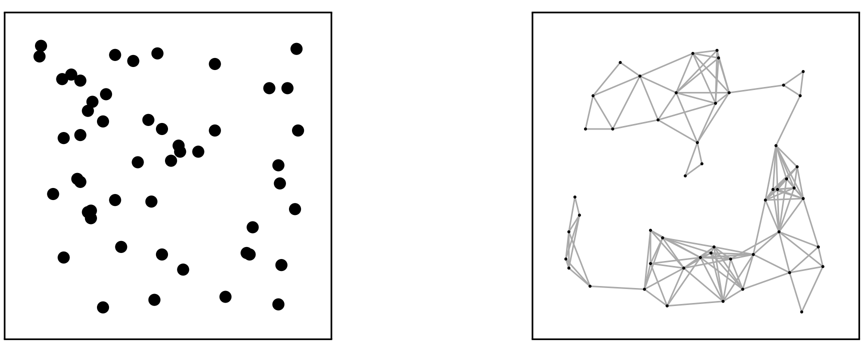


FIGURE 3 – Exemples de résultats des questions 1 et 3.

Deuxième partie – algorithme de Dijkstra

4. Compléter la fonction `void dijkstra(int n, listeAdj* G, int s, float*& D, int*& P)` qui implante l'algorithme de Dijkstra sur le graphe G à partir du sommet s. À la fin de l'algorithme, `D[i]` doit contenir la distance entre les sommets s et i, et `P[i]` le prédécesseur du sommet i dans le plus court chemin de s à i. *Il faut allouer de la mémoire pour* `D` *et* `P` *et les initialiser. La macro* `INFINITY` *représente* $+\infty$. *S'il n'existe pas de chemin entre s à i,* `P[i]` *doit valoir* -1.

5. Compléter la fonction `listeAdj chemin(int n, listeAdj* G, int* P, int s, int v)` qui renvoie le plus court chemin de s à v calculé grâce à `dijkstra` (figure 4). *On représente le chemin par une* `listeAdj` *qui contient la liste des sommets. Remarque : on réutilise ici la structure de donnée* `listeAdj` *pour représenter un chemin, même si elle n'est pas prévue pour cela à l'origine. En particulier, on peut ignorer les poids en les mettant tous à* 0.

6. Compléter la fonction `listeAdj* arbre(int n, listeAdj* G, int* P, int s)` qui renvoie l'arbre des plus courts chemins depuis s (figure 4). *On représente l'arbre des plus courts chemins par un graphe. On peut ignorer les poids des arêtes dans cet arbre, en les mettant par exemple tous à* 0.

TEST

```
Question à tester (0 pour sortir) : 4
Choix du graphe :
    1. Graphe n°1 (5 sommets)
    2. Graphe n°2 (10 sommets)
    3. Graphe aléatoire avec obstacles
    4. Grille aléatoire
    n. Graphe aléatoire à n sommets (question 3)
Choix (entier) : 3
Nombre de sommets : 200
Choix des sommets de départ et d'arrivée :
  1. Sommets aléatoires
  2. Départ au centre, arrivée aléatoire
  3. Sommets dans deux coins opposés
Choix (entier) : 3
Graphe dessiné dans graphe.svg

  Algorithme de Dijkstra appliqué en temps : 140 µs
  Distances : [408.729,272.121,40.8534,399.174,445.043,457.484,328.219,269.5,167.0
  Prédecesseurs : [89,57,23,108,24,0,196,63,12,157,... (+ 190 éléments)]
  Longueur du plus court chemin entre les sommets 23 et 137 : 484.105

Reconstruire le chemin (question 5) ? [o/n] o
  Chemin : 23→56→135→15→178→42→189→154→137
  Chemin représenté dans graphe.svg.

Construire l'arbre des plus courts chemins (question 6) ? [o/n] o
  Arbre représenté dans graphe.svg.
```

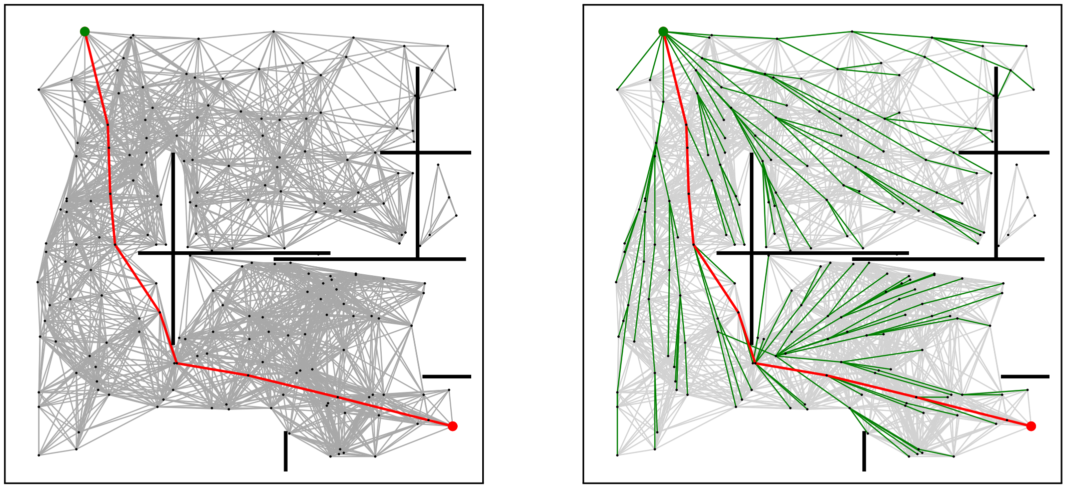


FIGURE 4 – Exemples de résultats des questions 5 et 6.

Troisième partie (bonus) – algorithme A*
Le but de cette partie est de voir et d'implanter une variante de l'algorithme de DIJKSTRA. Cet algorithme est à la base des calculs d'itinéraires sur des cartes en ligne.
L'objectif est de calculer le plus court chemins entre deux sommets s et t. On remarque que dans les graphes dont les poids des arêtes sont des distances euclidiennes, on connaît pour tout sommet u une borne inférieure sur la distance dans le graphe entre u et l'arrivée t : en effet, le mieux qu'on puisse espérer est un chemin en ligne droite, donc la longueur est simplement la distance euclidienne entre u et t. L'algorithme A* utilise cette information pour accélérer la recherche du plus court chemin entre s et t.
Plus précisément, il suit le cadre général de l'algorithme de DIJKSTRA avec deux modifications :

— puisqu'on cherche le plus court chemin entre s et t, on peut arrêter l'algorithme dès que le sommet t a été traité, plutôt que d'attendre que la file de priorité soit vide ;

— si on a calculé une distance $D_{[u]}$ entre s et u et qu'on note $\delta(u,t)$ la distance euclidienne entre u et t, on attribue la priorité $D_{[u]} + \delta(u,t)$ au sommet u plutôt que $D_{[u]}$ comme dans l'algorithme de DIJKSTRA ;

7. Compléter la fonction `void a_etoile(int n, listeAdj* G, coord* sommets, int s, int t, float*& D, int*& P)` qui implante l'algorithme A* (figure 5). *Visualiser la diminution du nombre de sommets visité par l'algorithme A* comparé à l'algorithme de* DIJKSTRA.

8. *(non évaluée)* L'algorithme A* est-il toujours correct sur la grille ? Pourquoi ?

TEST

```
Question à tester (0 pour sortir) : 7
Choix du graphe :
    1. Graphe n°1 (5 sommets)
    2. Graphe n°2 (10 sommets)
    3. Graphe aléatoire avec obstacles
    4. Grille aléatoire
    n. Graphe aléatoire à n sommets (question 3)
Choix (entier) : 3
Nombre de sommets : 200
Choix des sommets de départ et d'arrivée :
  1. Sommets aléatoires
  2. Départ au centre, arrivée aléatoire
  3. Sommets dans deux coins opposés
```

```
Choix (entier) : 3
Graphe dessiné dans graphe.svg

Appliquer l'algorithme de Dijkstra (question 4) ? [o/n] n
  Algorithme A* appliqué en temps : 396 µs
  Longueur du plus court chemin entre les sommets 42 et 55 : 491.685
  Chemin et arbre représentés dans graphe.svg
```

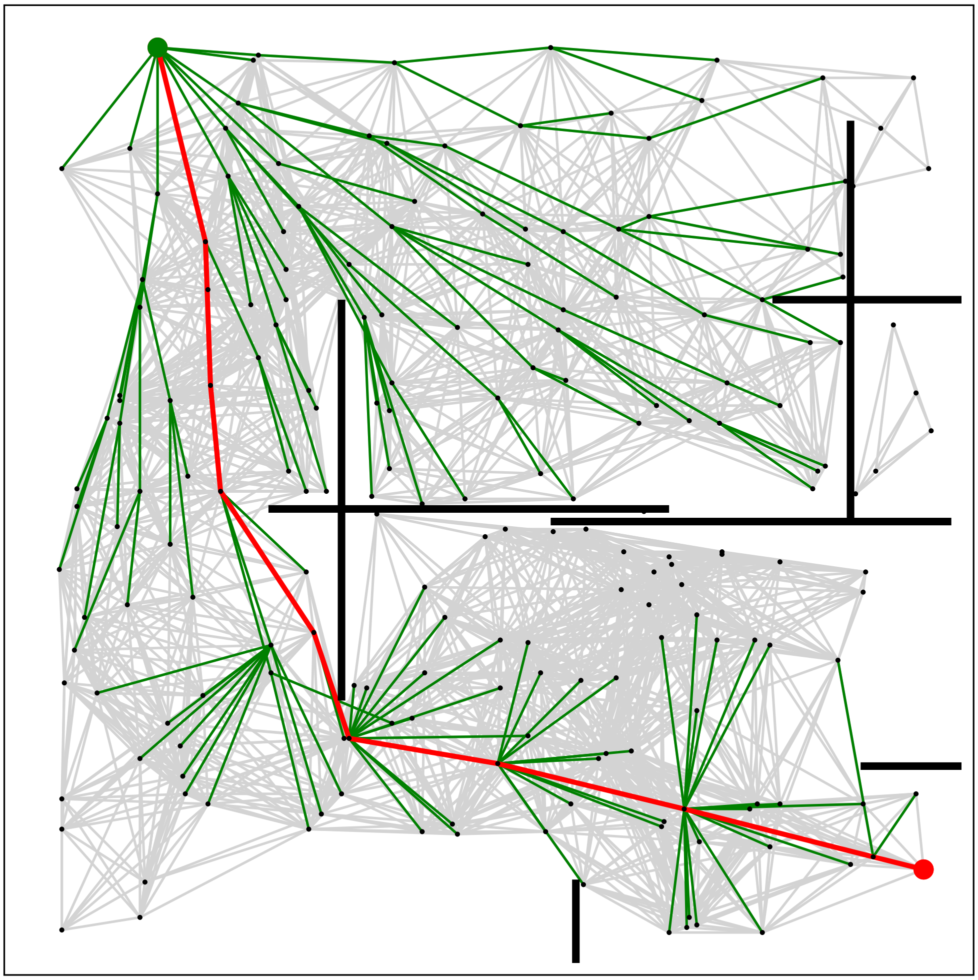


FIGURE 5 – Exploration avec l'algorithme A*