

TP6 : Algorithme de DIJKSTRA

L'objectif de ce TP est d'implanter l'algorithme de DIJKSTRA vu en cours, ainsi qu'une variante appelée *algorithme A**. Les fichiers fournis sont :

- un fichier `tests.cpp` qui contient le `main`;
- deux fichiers `Affichage.h/cpp` pour l'affichage graphique;
- deux fichiers `Structures.h/cpp` pour les structures de données de base;
- deux fichiers `Dijkstra.h/.cpp` (`Dijkstra.cpp` est l'unique fichier à modifier);
- un `Makefile`.

Représentation des graphes

On représente les graphes **par listes d'adjacence**. La liste d'adjacence d'un sommet u est une liste de couples (v, p) où v est un sommet 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 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
```

Graphes de tests

On utilise pour les tests trois types de graphes :

- deux graphes prédéfinis, codés dans des fichiers `Graphe5.txt` et `Graphe10.txt`, représentés en figure 1;
- des graphes à *obstacles*, dont un exemple est représenté en figure 1;
- des graphes aléatoires dont la création fait l'objet des premières questions.

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

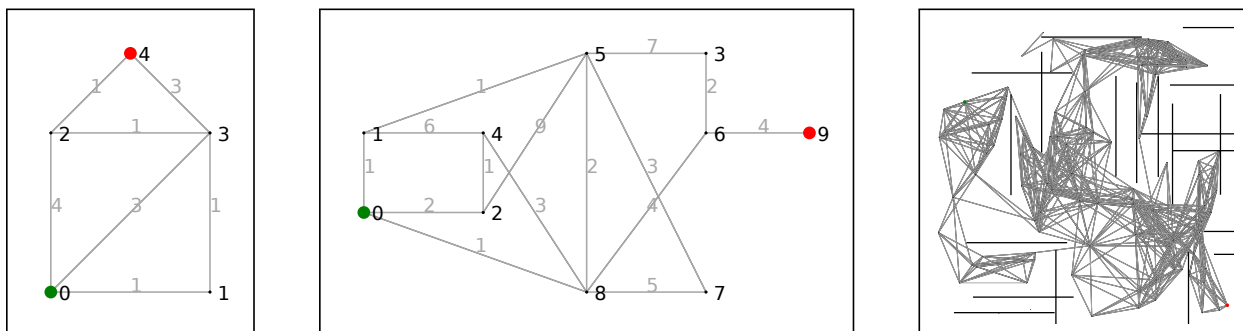
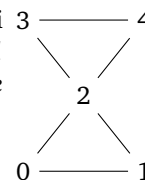


FIGURE 1 – Les deux graphes exemples fixés, avec le sommet de départ en vert et l'arrivée en rouge. Les poids des arêtes sont indiqués en grisé; un exemple de graphe à obstacle à 100 sommets, dont les arêtes sont grisées et les obstacles représentés en noir.

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 coordonnées, 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 $d_{\max} = 8$.



Deuxième partie – algorithme de DIJKSTRA

On implante l'algorithme de DIJKSTRA à l'aide d'une file de priorité. La classe File est illustrée ci-dessous.

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(); // Extraît 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
```

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 2). 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 3). 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.

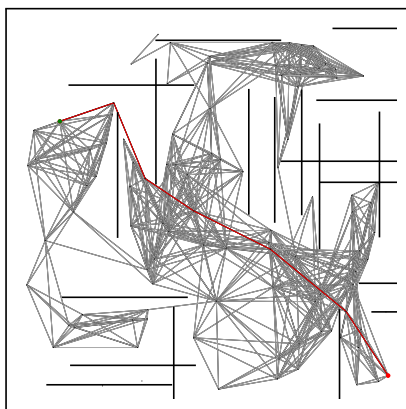


FIGURE 2 – Chemin le plus court

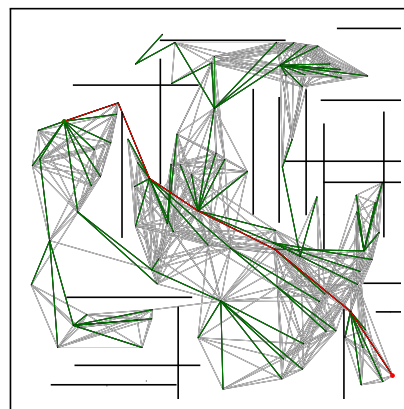


FIGURE 3 – Arbre des plus courts chemins

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 4). Visualiser la diminution du nombre de sommets visité par l'algorithme A* comparé à l'algorithme de DIJKSTRA.

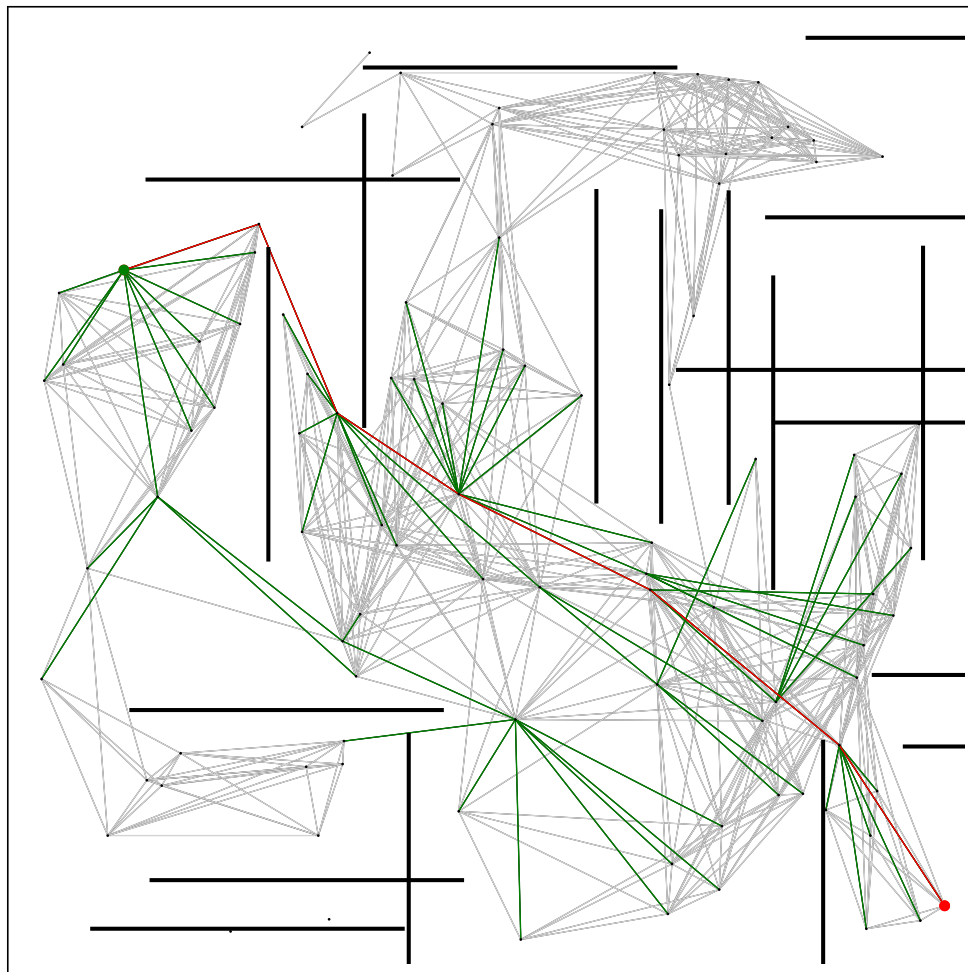


FIGURE 4 – Exploration avec l'algorithme A*