INF421-a
Bases de la programmation et de l'algorithmique

(Bloc 5/ 9)

Philippe Baptiste

CNRS LIX, École Polytechnique

22 septembre 2006

Aujourd'hui

Parcours

Les arbres d'arité quelconque

Union Find

Arbre enraciné → arbres binaires

Def récursive des arbres : Arbre = couple formé de sa racine et d'un ensemble d'arbres. C'est encore un peu compliqué. Regardons pour le moment les arbres binaires.

> Un arbre binaire sur un ensemble fini est soit vide, soit l'union disjointe d'un nœud appelé sa racine, d'un arbre binaire appelé sous-arbre gauche, et d'un arbre binaire appelé sous-arbre droit

Représentation naturelle des arbres binaires : $A = (A_g, r, A_d)$.

```
class Arbre {
    int val; Arbre gauche, droite;
    Arbre (Arbre gauche, int val, Arbre droite) {
        this.gauche = gauche;
        this.val = val;
        this.droite = droite; }}
```

Comment parcourir un arbre

Objectif : parcourir un arbre (pour imprimer les sommets ou pour les numéroter). Une première réponse, le parcours en profondeur d'abord (DFS) . Plusieurs variantes mais 3 étapes essentielles :

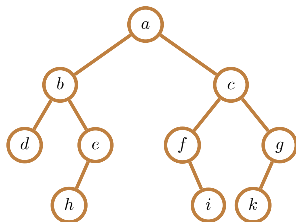
- Récursion sur le sous-arbre gauche
- Récursion sur le sous-arbre droit
- Impression (ou numérotation) du sommet courant

L'ordre dans lequel on effectue ces opérations est déterminant :

- Préfixe : sommet courant puis sous-arbre gauche puis sous-arbre droit
- Infixe : sous-arbre gauche puis sommet courant puis sous-arbre droit
- Postfixe : sous-arbre gauche puis sous-arbre droit puis sommet courant

Parcours DFS "préfixe"

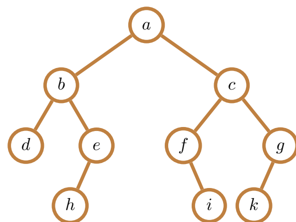
- Visiter la racine
- Visiter le sous-arbre gauche
- Visiter le sous-arbre droit
- Implémentation récursive (ou dérécursivée avec une pile)



Ordre de visite
$a, b, d, e, h, c, f, i, g, k$

Parcours DFS "préfixe" : Codage

```
static void prefixe(Arbre x) {
   if (x == null)
      return;
   System.out.println(x.val);
   prefixe(x.gauche);
   prefixe(x.droite);
}
```

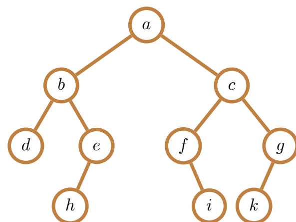


Ordre de visite
$a, b, d, e, h, c, f, i, g, k$

DFS itératif

On veut écrire un DFS préfixe dérécursivé (*i.e.*, pas d'appel récursif)

- Empiler la racine
- Tant que la pile n'est pas vide
  - Afficher le sommet de pile et le supprimer
  - Empiler le fils droit
  - Empiler le fils gauche



La pile

		d							
	b	e	e	h		f	i		
a	c	c	c	c	c	g	g	g	k

Une révision idéale des Piles

```
class Pile {
  final static int maxP = 50;
  int hauteur;
  Arbre[] contenu;
  Pile() {
    hauteur = 0;
    contenu = new Arbre[maxP];
  }
  boolean estVide() {
    return hauteur == 0;
  }
  void ajouter(Arbre x) {
    contenu[hauteur++] = x;
  }
  Arbre valeur() {
    return contenu[hauteur-1];
  }
  void supprimer() {
    hauteur--;
  }
```

DFS itératif avec une pile (variante)

```
static void dfsPref1(Arbre a) {
  if (a == null) return;
  Pile p = new Pile();
  p.ajouter(a);
  while (!p.estVide()) {
    a = p.valeur();
    p.supprimer();
    System.out.print(a.val + " ");
    if (a.droite ≠ null)
      p.ajouter(a.droite);
    if (a.gauche ≠ null)
      p.ajouter(a.gauche);
  }
}
```

```
static void dfsPref2(Arbre a) {
  Pile p = new Pile () ;
  p.ajouter (a) ;
  while (!p.estVide ()) {
    a = p.valeur () ;
    p.supprimer () ;
    if (a ≠ null) {
      System.out.print (a.val + " ");
      p.ajouter (a.droite) ;
      p.ajouter (a.gauche) ;
    }
  }
}
//
```

Complexité d'un DFS

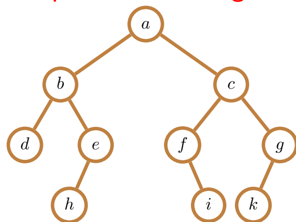
- Un sommet n'est empilé qu'une fois
- On emprunte les arêtes une et une seule fois
- Il y a exactement n itérations `while (!p.estVide())`
- Complexité linéaire en $O(n)$

Arbres et expressions arithmetiques

...

BFS (itératif évidemment)

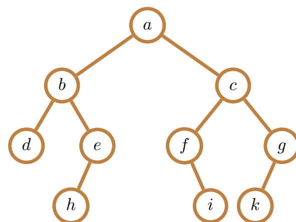
Un parcours en largeur énumère les nœuds niveau par niveau.



Ordre de visite
$a, b, c, d, e, f, g, h, i, k$

- Enfiler la racine
- Tant que la file n'est pas vide
  - Afficher le début de file et le supprimer
  - Enfiler le fils gauche
  - Enfiler le fils droit

BFS (itératif évidemment)



La file

a b c d e f g h i k

c d e f g h i k

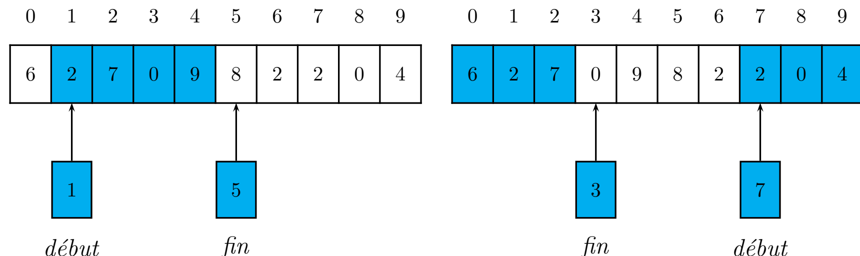
e f g h i k

g

Une révision idéale des Files

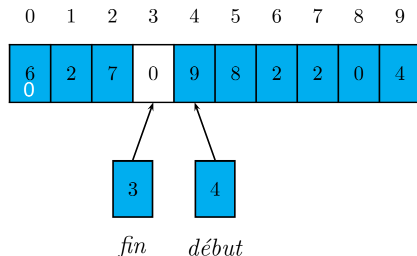
Idée simple : Utiliser un tableau pour enregistrer les éléments de la file.

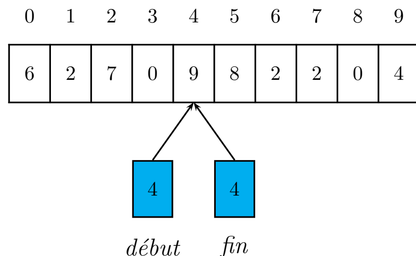
- Attention : Le début ET la fin de la file changent
- Utilisation d'un tableau `contenu` "circulaire" de taille `maxF`
  - Les éléments de la file sont ceux compris entre `début` et `fin` modulo `maxF`



Une révision idéale des Files

- Convention pour une file vide : `début` $=$ `fin`
- Convention pour une file pleine : `fin + 1` $\equiv$ `début` $\bmod n$
- On perd ainsi une place dans le tableau





Une révision idéale des Files

```
class File {
    final static int MaxF = 30;
    int debut;
    int fin;
    Arbre[] contenu;
    File() {
        debut = 0;
        fin = 0;
        contenu = new Arbre[MaxF];
    }
    static int successeur(int i) {
        return (i+1) % MaxF;
    }
    boolean estVide()  {
        return (debut == fin);
    }
```

```
Arbre valeur() {
    return contenu[debut];
}
void ajouter(Arbre x) {
    contenu[fin] = x;
    fin = successeur(fin);
}
void supprimer() {
    debut = successeur(debut);
}
```

BFS

```
static void bfs(Arbre a) {
    File file = new File();
    file.ajouter(a);
    while (!file.estVide()) {
        a = file.valeur();
        file.supprimer();
        if (a ≠ null) {
           System.out.print(a.val + " ");
           if (a.gauche ≠ null)
              file.ajouter(a.gauche);
           if (a.droite ≠ null)
               file.ajouter(a.droite);
        }
    }
}
```

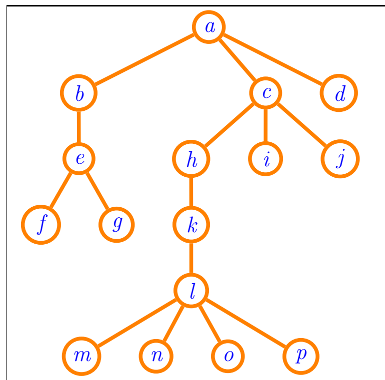
Aujourd'hui

Parcours

Les arbres d'arité quelconque

Union Find

Les arbres d'arité quelconque



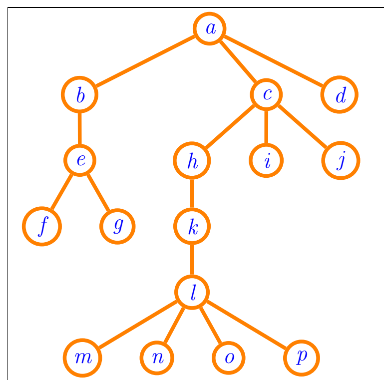
- Le nombre de fils est quelconque
- Représentation des fils par une liste d'arbres

```
class Liste {
    Arbre arbre;
    Liste suivant;
    Liste (Arbre a,
           Liste l) {
        arbre = a;
        suivant = l;
    }
}
```

Les arbres d'arité quelconque

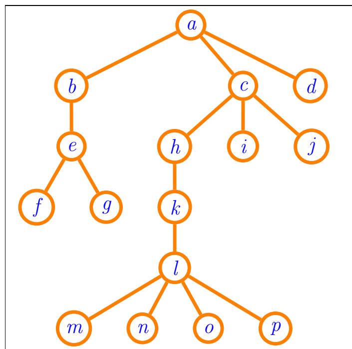
```
class Arbre {
    char val;
    Liste fils;
    Arbre(char val, Liste fils) {
        this.val = val;
        this.fils = fils;
    }
    Arbre(char val) {
        this.val = val;
        this.fils = null;
    }
    // pour construire facilement des arbres
    void ajouteFils(Arbre f1) {
        fils = new Liste(f1, fils); }
    void ajouteFils(Arbre f1, Arbre f2) {
        fils = new Liste(f1, new Liste(f2, fils)); }
}
```

Les arbres d'arité quelconque




```
Arbre a = new Arbre('a');
Arbre b = new Arbre('b');
// de même pour c, d, e ...
Arbre p = new Arbre('p');
a.ajouteFils(b, c, d);
b.ajouteFils(e);
e.ajouteFils(f, g);
c.ajouteFils(h, i, j);
h.ajouteFils(k);
k.ajouteFils(l);
l.ajouteFils(m, n, o, p);
```

Parcours d'arbres d'arité quelconque



DFS Préfixe :

a d c j i h k l p o n m b e g f

BFS :

a b c d e h i j f g k l m n o p

Simple transposition des arbres binaires

Les arbres d'arité quelconque

```
// DFS
static void
dfsPrefixe (Arbre a) {
  Pile pile = new Pile () ;
  pile.ajouter (a) ;
  while (!pile.estVide ()) {
    a = pile.valeur () ;
    pile.supprimer () ;
    System.out.print(a.val + " ") ;
    for (Liste fl = a.fils;
          fl ≠ null;
          fl = fl.suivant)
       pile.ajouter(fl.arbre) ;
  }
}
```

```
// BFS
static void
bfs(Arbre a) {
  File file = new File();
  file.ajouter(a);
  while (!file.estVide()) {
    a = file.valeur();
    file.supprimer();
    System.out.print(a.val + " ") ;
    for (Liste fl = a.fils;
          fl ≠ null;
          fl = fl.suivant)
       file.ajouter(fl.arbre) ;
  }
}
```

Aujourd'hui

Parcours

Les arbres d'arité quelconque

Union Find

Union-Find

Partition de E = ensemble de parties non vides de E, deux à deux disjointes et dont la réunion est E

- Étant donné une partition de l'ensemble $\{0, \ldots, n-1\}$, on veut
- trouver la classe d'un élément (*find*)
- faire l'union de deux classes (*union*).

En général, on part d'une partition où chaque classe est réduite à un singleton, puis on traite une suite de requêtes union / find

Union-Find : Une idée naïve

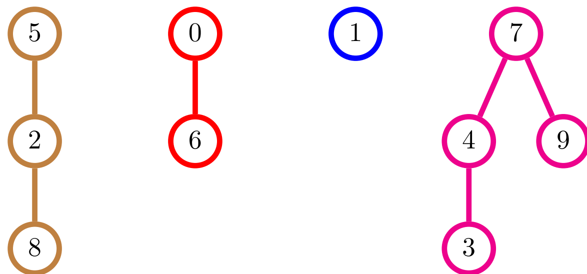
- La partition est représentée par un tableau `classe`
- Chaque classe est identifiée par un entier, et `classe[x]` contient le numéro de la classe de l'élément `x`
- Trouver la classe d'un élément se fait en temps constant,
- Fusionner deux classes prend ? ? ?

x	0	1	2	3	4	5	6	7	8	9
classe[x]	2	3	1	4	4	1	2	4	1	4

Partition $\{2, 5, 8\}, \{0, 6\}, \{1\}, \{3, 4, 7, 9\}$

Union-Find par les arbres

- choisir un représentant dans chaque classe
- Fusionner deux classes = changer de représentant pour les éléments de la classe fusionnée
- La partition = une forêt
  - Chaque classe de la partition = un arbre de cette forêt.
  - La racine de l'arbre est le représentant de sa classe

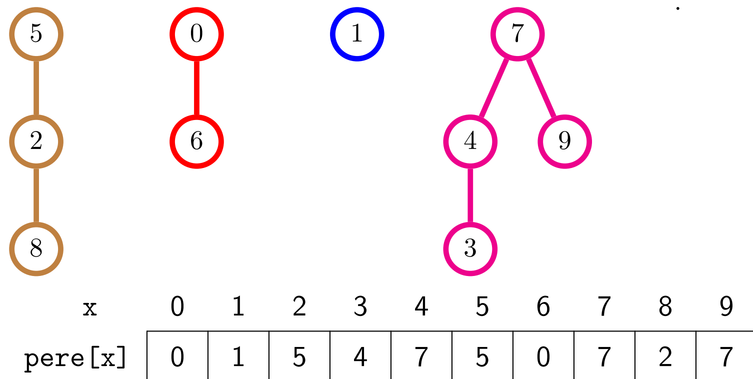


Union-Find par les arbres

- Pour cet algorithme, on peut se contenter de représenter un arbre par le tableau des "pères"
- Chaque nœud est représenté par un entier
- `pere[x]` est le père du nœud x
- Une racine r n'a pas de père et `pere[r] = r`

Attention, cette représentation des arbres est "insuffisante" pour de nombreux algorithmes

Union-Find par les arbres



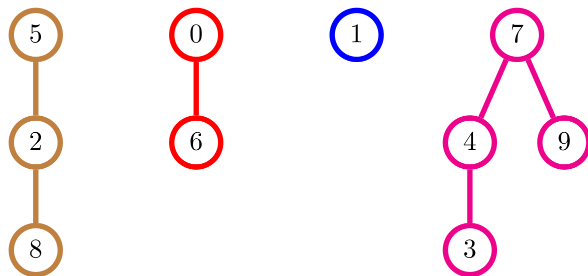
x	0	1	2	3	4	5	6	7	8	9
pere[x]	0	1	5	4	7	5	0	7	2	7

Union-Find par les arbres

```
class UF {
    int[] pere;
    UF(int n) { /* classes = singletons */
        pere  = new int[n];
        for (int i = 0; i < pere.length ; i++)
            pere[i] = i; }
    /* Chercher le représentant de la classe contenant un élément
       revient à trouver la racine de l'arbre contenant l'élément */
    int trouver(int x) {
        while (x ≠ pere[x]) x = pere[x];
        return x; }
    /* L'union de deux arbres = ajout de  la racine de l'un des
       comme nouveau fils à la racine de l'autre */
    void union(int x, int y) {
        int r = trouver(x);
        int s = trouver(y);
        if (r ≠ s) pere[r] = s; }}
```

Union-Find par les arbres

```
UF uf = new UF(10);
uf.union(8, 2); uf.union(2, 5); uf.union(6, 0);
uf.union(3, 4); uf.union(9, 7); uf.union(3, 9);
for (int i = 0; i < 10; i++)
    System.out.println("classe de " + i + " = " + uf.trouver(i));
```



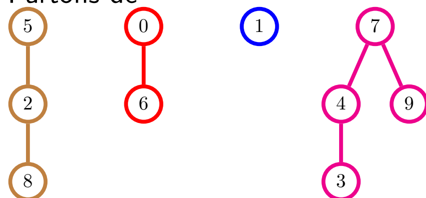
Union-Find par les arbres

- Complexité de l'union ?
- Complexité du find ?

Quelle "forme" doivent donc avoir les arbres ?

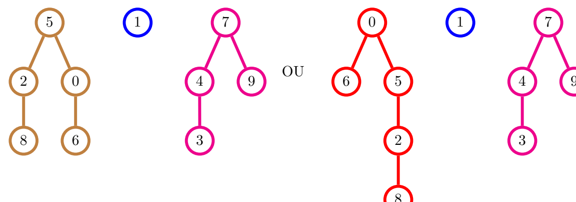
Union-Find par les arbres, un peu de stratégie

Partons de



.

et faisons l'union des classes 5 et 0 → 2 "solutions" possibles



Laquelle choisir ?

Union-Find par les arbres, un peu de stratégie

Lors de l'union de deux arbres, la racine du petit arbre devient fils de la racine du grand

→ utiliser un tableau supplémentaire qui mémorise la taille des arbres (initialisé à 1)

```
class UF {
    int[] taille; int[] pere;
    UF(int n) {
        pere   = new int[n];
        taille  = new int[n];
        for (int i = 0;
             i < pere.length ;
             i++) {
            taille[i] = 1;
            pere[i] = i; }}
```

```
void union(int x, int y) {
    int r = trouver(x);
    int s = trouver(y);
    if (r == s) return;
    if (taille[r] > taille[s]) {
        pere[s] = r;
        taille[r] += taille[s]; }
    else {
        pere[r] = s;
        taille[s] += taille[r]; }}}
```

Union-Find par les arbres, un peu de complexité

La hauteur d'un arbre à n nœuds créé par union pondérée est au plus $1 + \lfloor \log_2 n \rfloor$

- Pour $n = 1$, trivial
- Soit l'arbre obtenu par union pondérée d'un arbre à m nœuds et d'un arbre à $n - m$ nœuds, avec $1 \leqslant m \leqslant n/2$
  - sa hauteur h est telle que

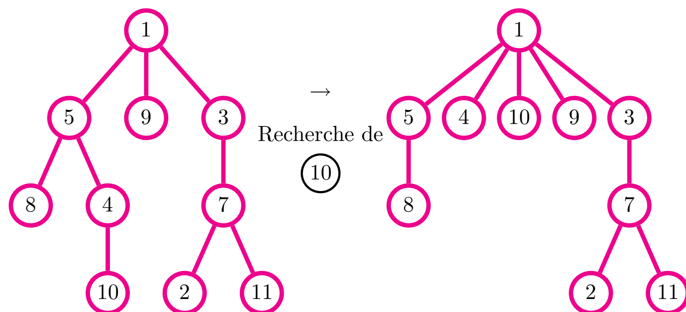
$$h \leq \max(1 + \lfloor \log_2(n - m) \rfloor, 2 + \lfloor \log_2 m \rfloor) .$$

  - or $\log_2 m \leqslant \log_2(n/2) = \log_2 n - 1$, et donc

$$h \leq 1 + \lfloor \log_2 n \rfloor$$

Plus fort : Compresser les chemins

Après être remonté du nœud x à sa racine r, on refait le parcours en faisant de chaque nœud rencontré un fils de r.



Plus fort : Compresser les chemins

```
/* Version précédente */
int trouverSimple(int x) {
    while (x ≠ pere[x])
        x = pere[x];
    return x; }
/* Avec compression */
int trouver(int x) {
    int r = trouverSimple(x);
    while (x ≠ r) {
        int y = pere[x];
        pere[x] = r;
        x = y;
    }
    return r;
}
```

n unions + m finds ($m \geqslant n$) en $O(n + m\alpha(n, m))$, où α est une fonction qui croit très très lentement (inverse d'une fonction d'Ackermann)

En pratique, comme un algorithme linéaire en $n + m$ (mais attention, il *n'est pas* linéaire)

La fonction d'Ackermann

On peut définir la fonction d'Ackermann $A(m, n)$ récursivement

- $A(0, n) = n + 1$
- $A(m, 0) = A(m - 1, 1)$
- $A(m, n) = A(m - 1, A(m, n - 1))$

$m \backslash n$	0	1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9	10	11
1	2	3	4	5	6	7	8	9	10	11	12
2	3	5	7	9	11	13	15	17	19	21	23
3	5	13	29	61	125	253	509	1021	2045	4093	8189

La fonction d'Ackermann croît très rapidement : $A(0, n) = n + 1$; $A(1, n) = n + 2$; $A(2, n) = 3 + 2n$; $A(3, n) = 5 + 8(2^n - 1)$

$\alpha(n, m) = \min\{i \geq 1 : A(i, \lfloor \frac{m}{n} \rfloor) > \log n\}$. En pratique $\alpha(n, m) \leq 4$ pour $n \leq m \leq 10^{80}$

Union / find : Arbre couvrant de poids minimal

- $G = (V, E)$ un graphe et w_e le poids de ses arête ($e \in E$).
- Définition : Un arbre couvrant = un arbre (V, T) avec $T \subseteq E$
- Définition : le poids d'un sous-graphe est la somme des poids de ses arêtes

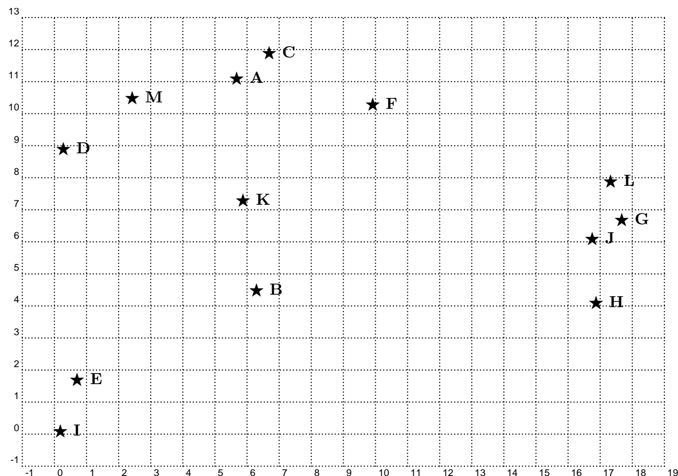
Union / find : Arbre couvrant de poids minimal

Un exemple (MST Euclidien) :

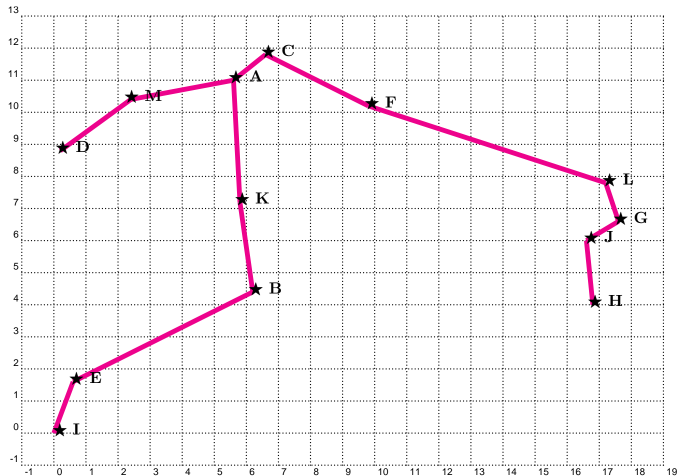
- Des villes à connecter par un réseau
- Coût d'une arête = longueur
- On ne peut pas créer de nouveaux noeuds
- Mettre en réseau les villes pour un coût total minimal

Attention, $\exists$ des algorithmes spécifiques (plus efficaces) pour ce problème basés sur les triangulations de Delaunay (cf, les dernières planches)

Union / find : Arbre couvrant de poids minimal



Union / find : Arbre couvrant de poids minimal

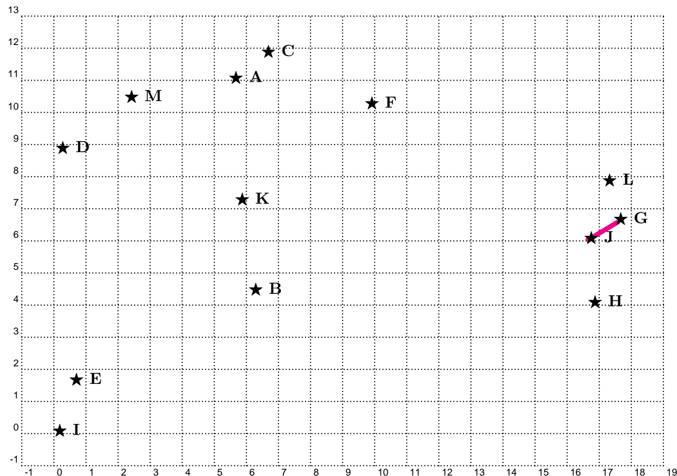


Union / find : Arbre couvrant de poids minimal

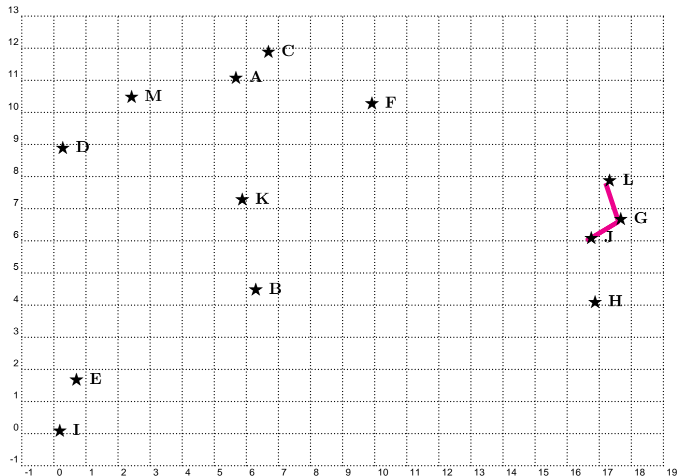
$G = (V, E)$ un graphe et w_e le poids de ses arête ($e \in E$).
Calculons l'arbre couvrant de poids minimal (Algo. de Kruskal)

- Soit $\mathcal{L}$ la liste des arêtes triées par poids croissant
- Construisons itérativement l'arbre
  - Initialement $T = (V, \emptyset)$
  - Pour chaque arête e de $\mathcal{L}$ ajouter e à T ssi on ne crée pas de cycle

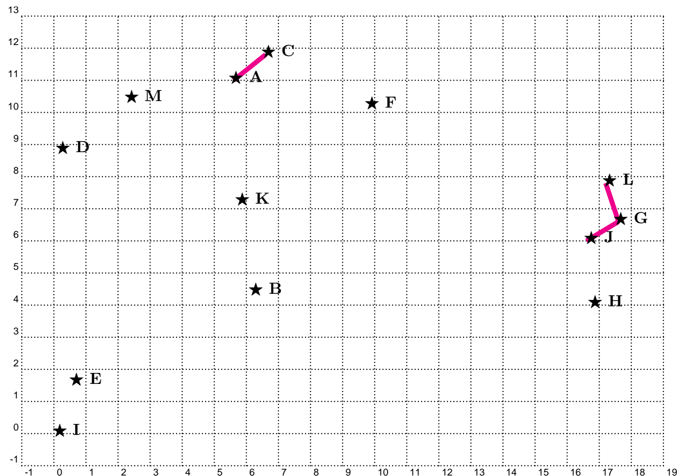
Union / find : Arbre couvrant de poids minimal



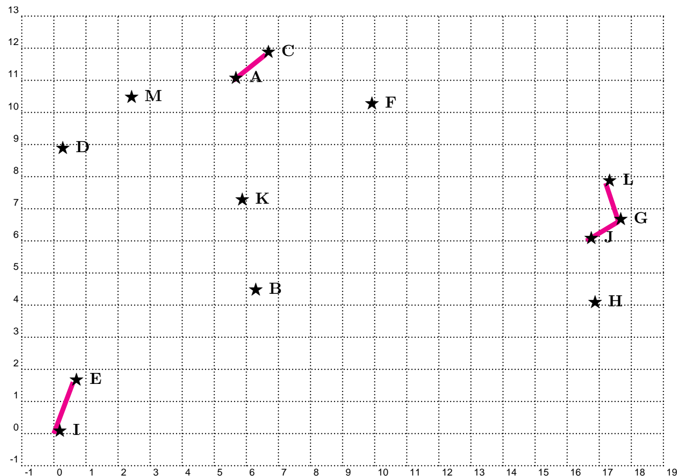
Union / find : Arbre couvrant de poids minimal



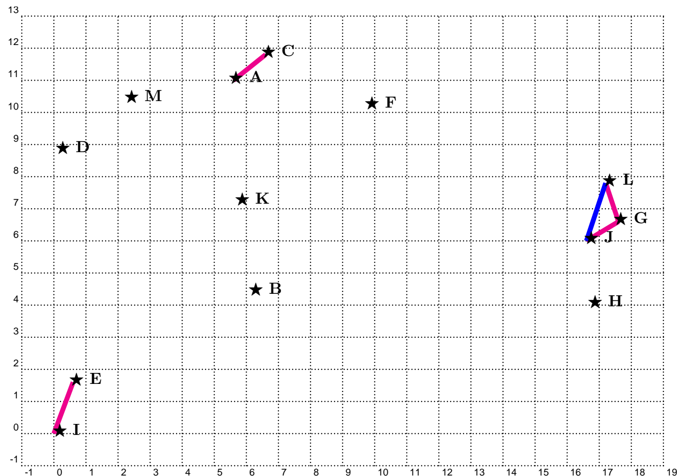
Union / find : Arbre couvrant de poids minimal



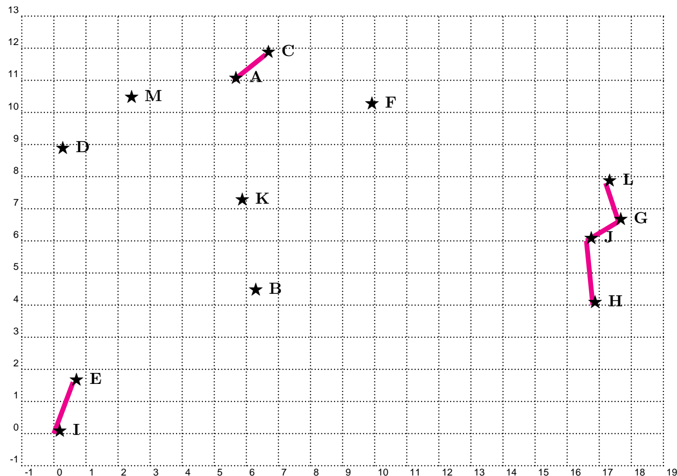
Union / find : Arbre couvrant de poids minimal



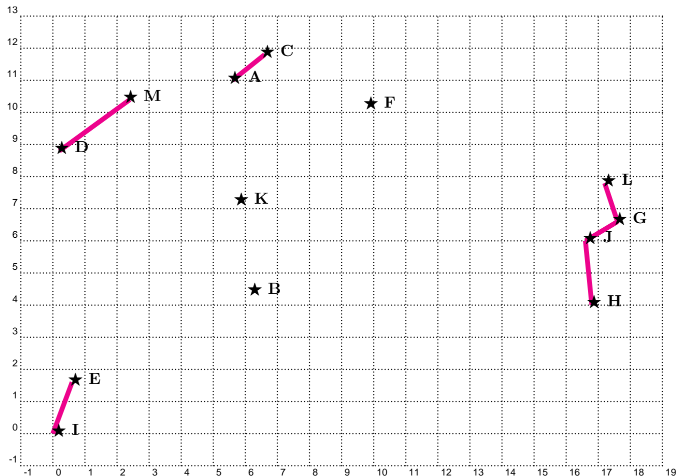
Union / find : Arbre couvrant de poids minimal



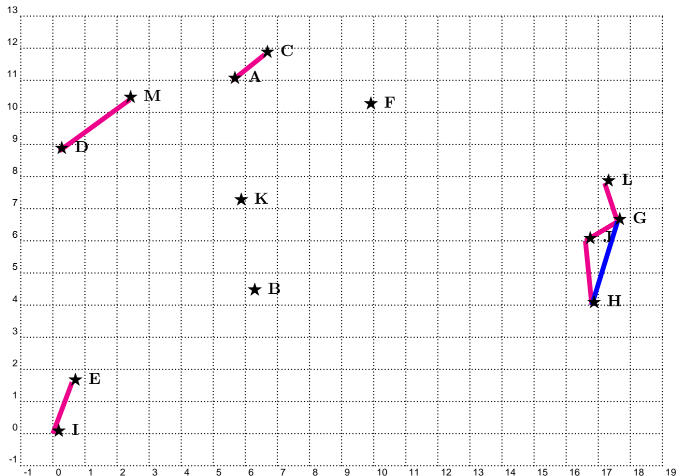
Union / find : Arbre couvrant de poids minimal



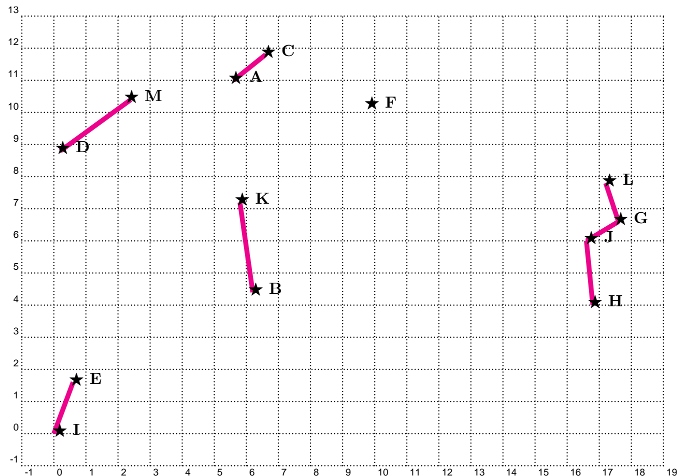
Union / find : Arbre couvrant de poids minimal



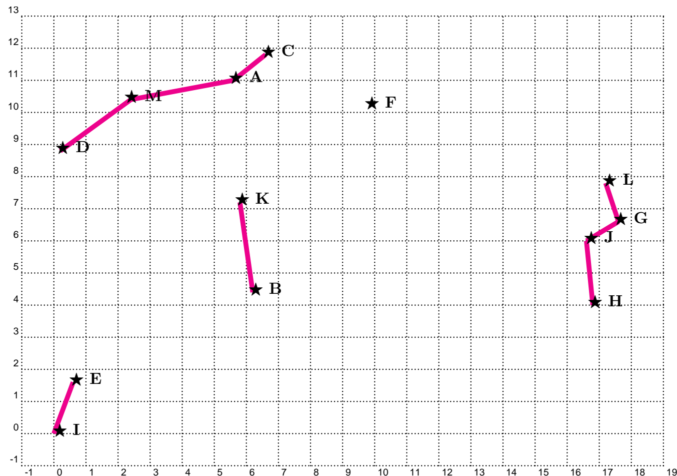
Union / find : Arbre couvrant de poids minimal



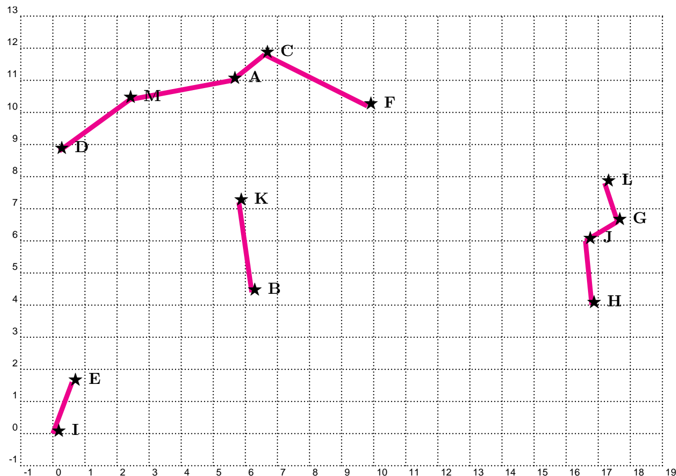
Union / find : Arbre couvrant de poids minimal



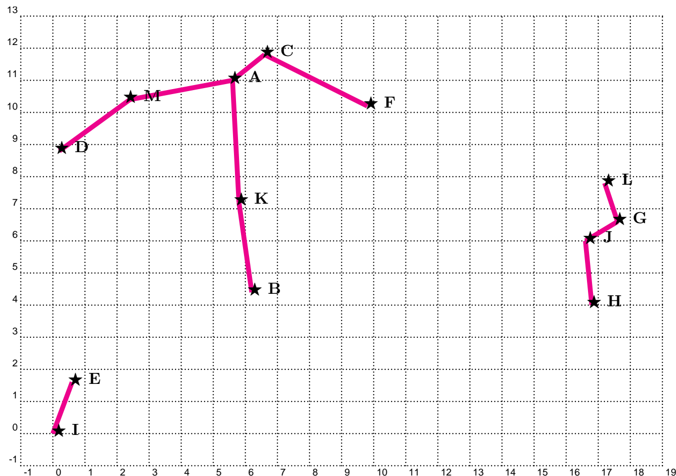
Union / find : Arbre couvrant de poids minimal



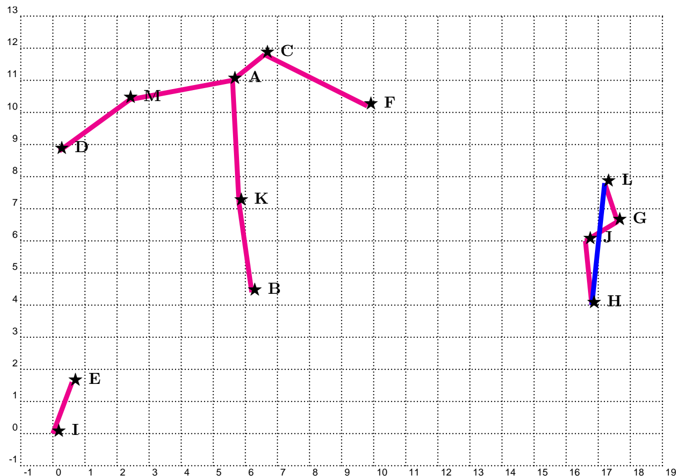
Union / find : Arbre couvrant de poids minimal



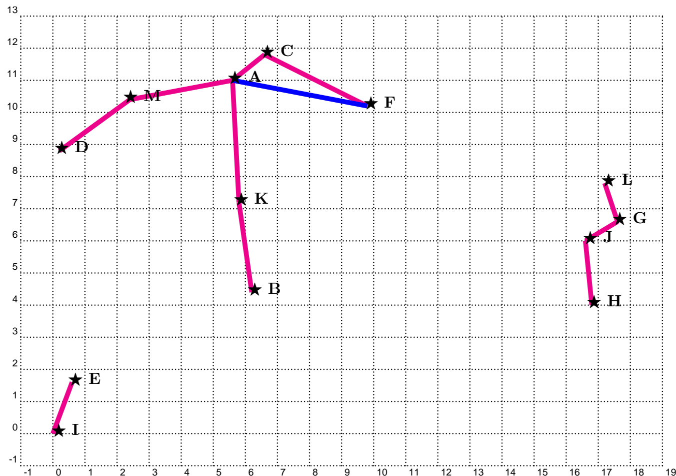
Union / find : Arbre couvrant de poids minimal



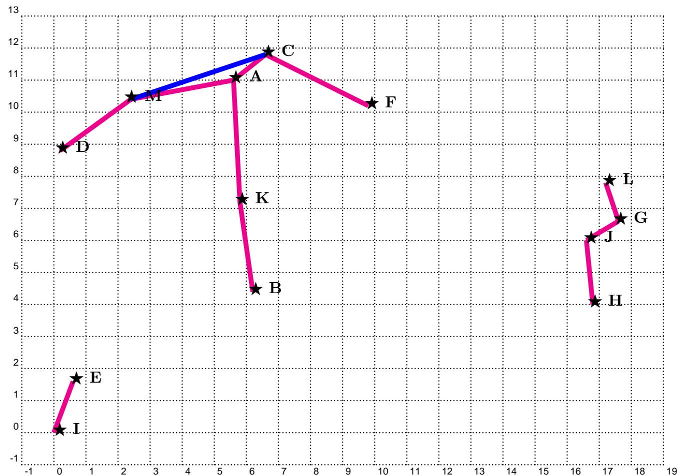
Union / find : Arbre couvrant de poids minimal



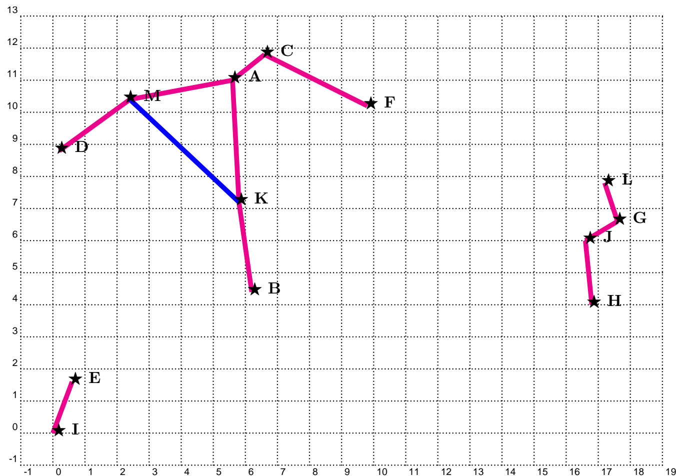
Union / find : Arbre couvrant de poids minimal



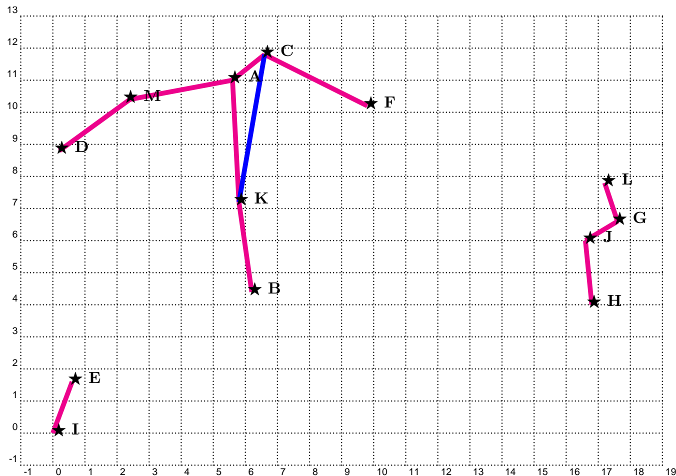
Union / find : Arbre couvrant de poids minimal



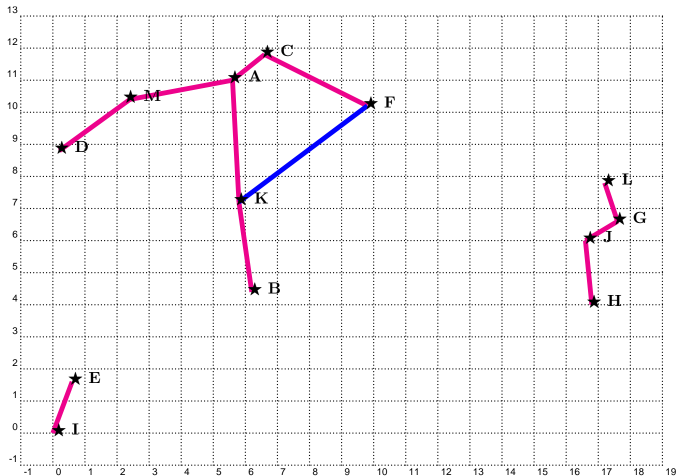
Union / find : Arbre couvrant de poids minimal



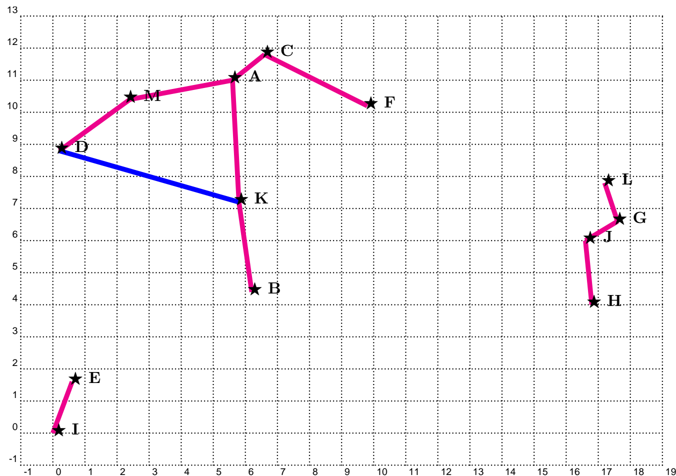
Union / find : Arbre couvrant de poids minimal



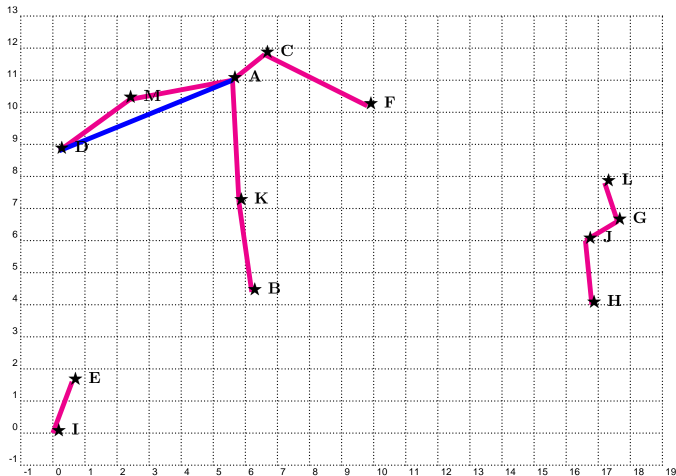
Union / find : Arbre couvrant de poids minimal



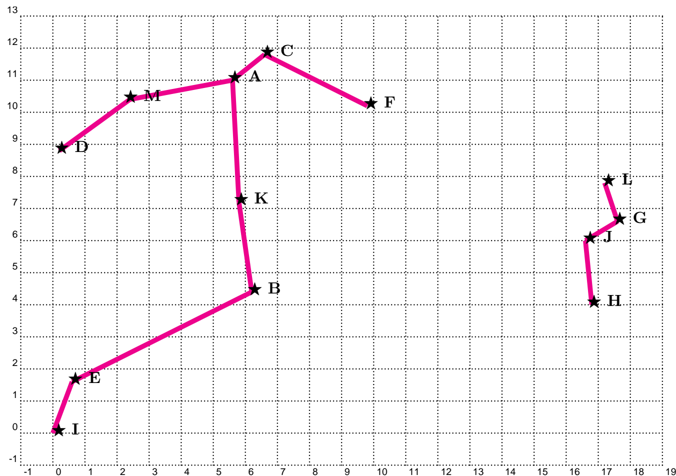
Union / find : Arbre couvrant de poids minimal



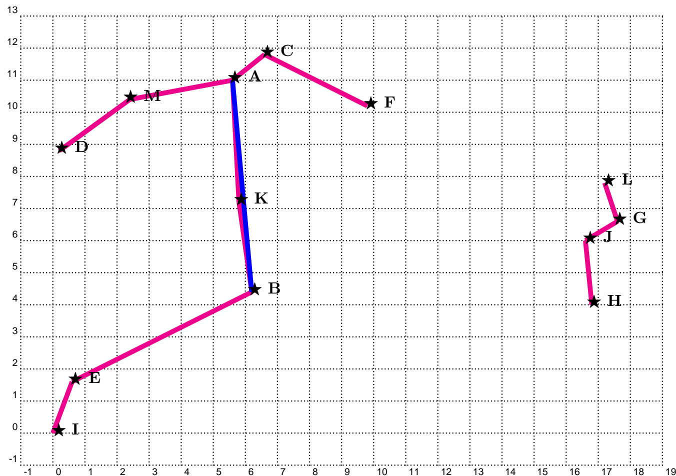
Union / find : Arbre couvrant de poids minimal



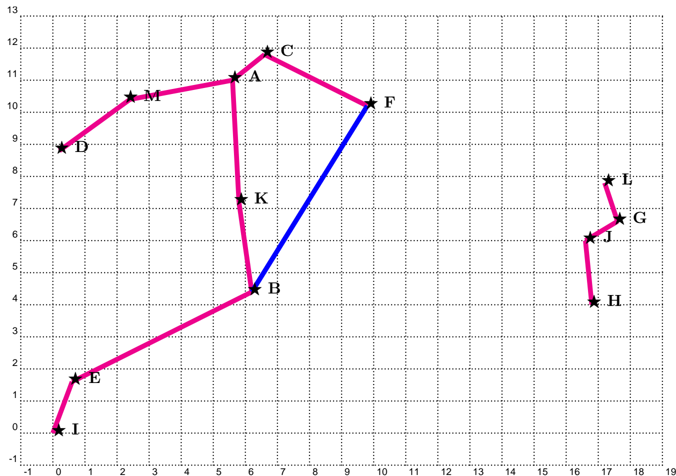
Union / find : Arbre couvrant de poids minimal



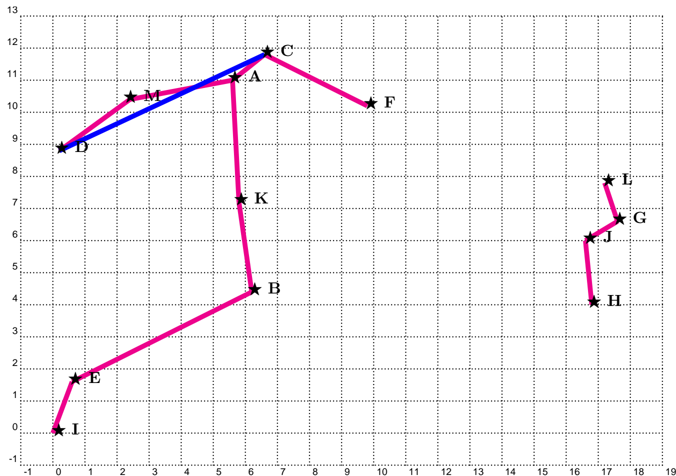
Union / find : Arbre couvrant de poids minimal



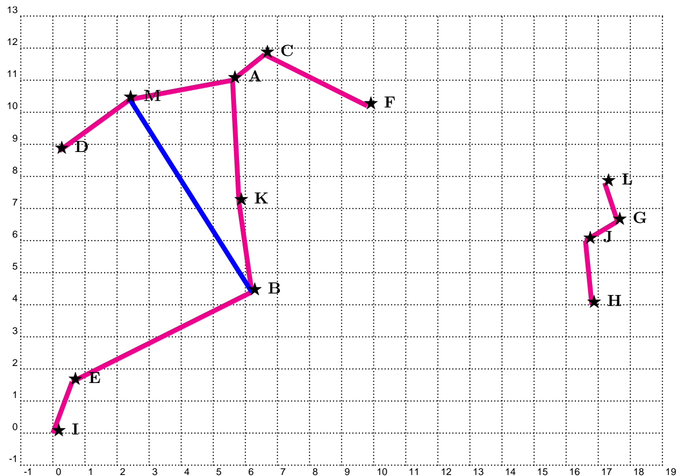
Union / find : Arbre couvrant de poids minimal



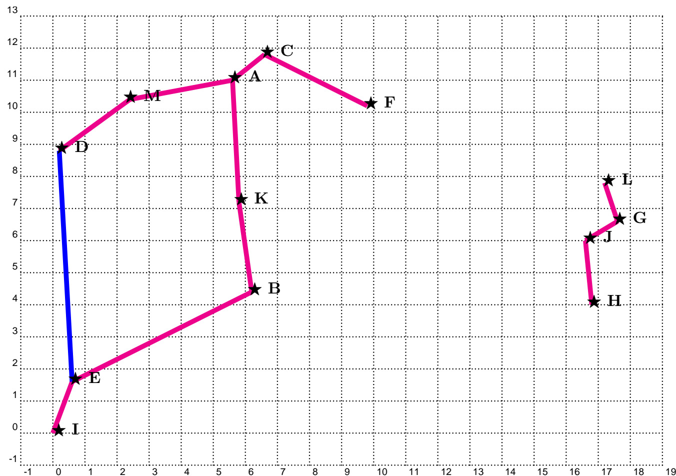
Union / find : Arbre couvrant de poids minimal



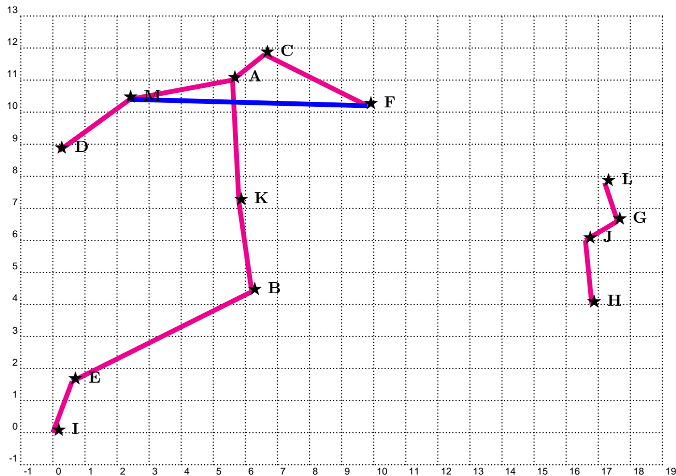
Union / find : Arbre couvrant de poids minimal



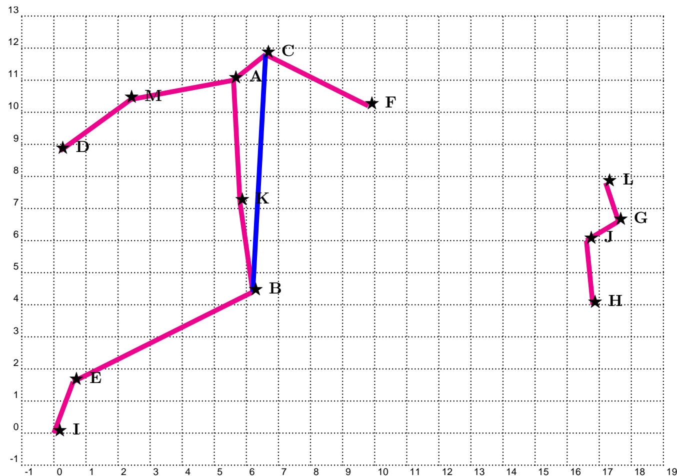
Union / find : Arbre couvrant de poids minimal



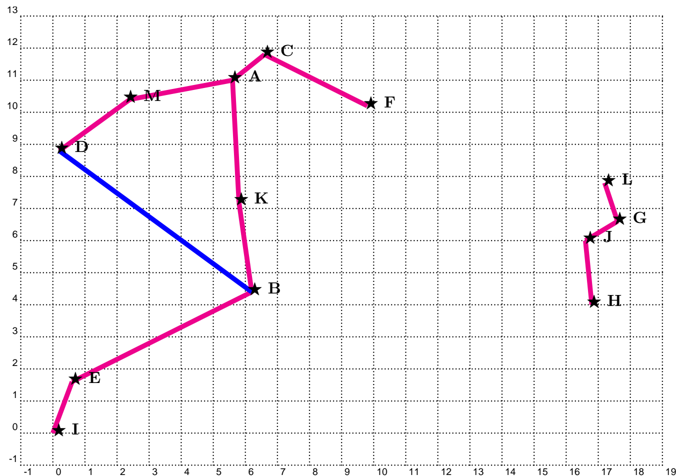
Union / find : Arbre couvrant de poids minimal



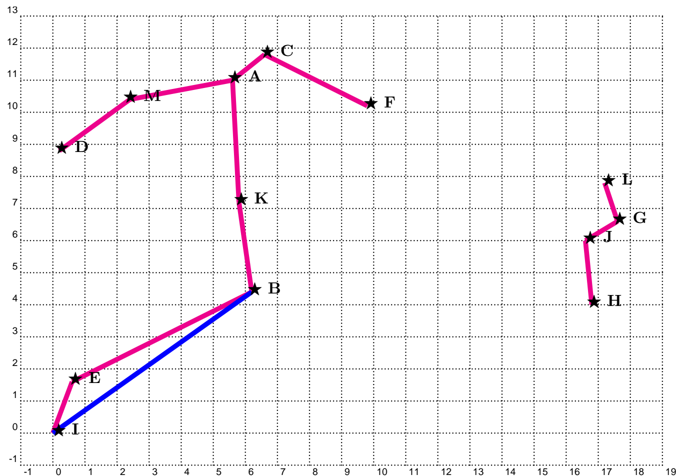
Union / find : Arbre couvrant de poids minimal



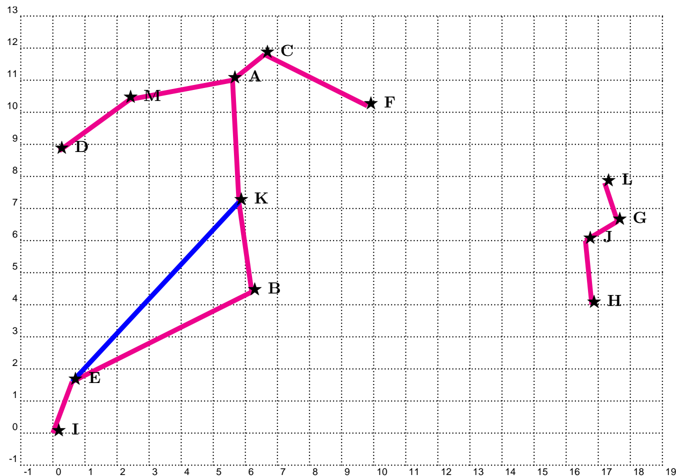
Union / find : Arbre couvrant de poids minimal



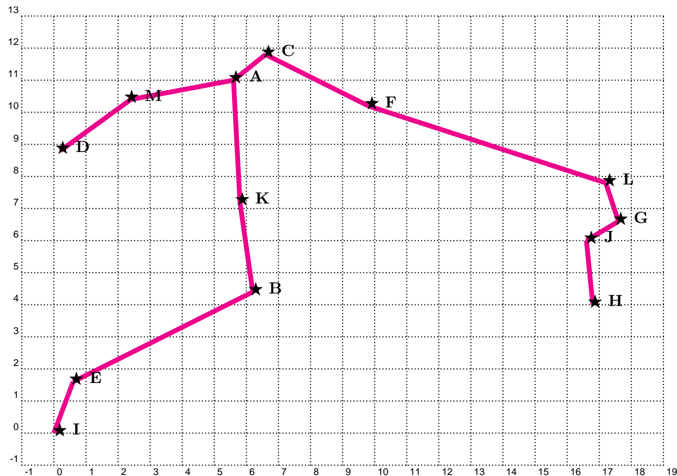
Union / find : Arbre couvrant de poids minimal



Union / find : Arbre couvrant de poids minimal



Union / find : Arbre couvrant de poids minimal



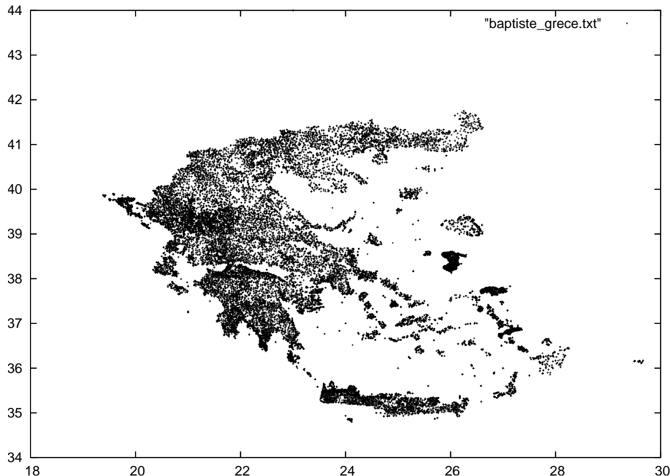
Union / find : Arbre couvrant de poids minimal

```
class Arete implements Comparable<Arete> {
    int i;
    int j;
    double distance;
    Arete(int i, int j, double distance) {
        this.i = i;
        this.j = j;
        this.distance = distance;
    }
    public int compareTo(Arete autre) {
        return (distance - autre.distance < 0) ? -1 : 1;
    }
}
```

Union / find : Arbre couvrant de poids minimal

```
// Dans le main un peu plus loin
UF uf = new UF(n);
Arete[] aretes = new Arete[nbAr];
int n = 0; int nbAr = 0; // le nb de sommets et d'arêtes
// lecture du fichier (omis)
Arrays.sort(aretes);
for (int e = 0; e < nbAr; e++)
    if (uf.trouver(aretes[e].i) ≠ uf.trouver(aretes[e].j)) {
        uf.union(aretes[e].i, aretes[e].j);
        totalCostPC += aretes[e].w;
    }
```

Union / find : 44346 sommets 6023607 arêtes



Union / find : Arbre couvrant de poids minimal

- Villes = 44 346
- Arêtes = 6 023 607
- MST naïf 30 185 ms
- MST Pondéré 2 169 ms
- MST Pondéré + Compressé : 1 671 ms

Union / find : Arbre couvrant de poids minimal (complexité)

- Graphe de départ $G = (V, E)$
- Tri des arêtes : $O(|E| \log |E|)$
- Ajout et détection de cycle :
  - Un arbre contient $|V| - 1$ arêtes et donc
  - $O(|V|)$ unions et $O(|E|)$ recherches
  - soit donc $O(|V| + |E|\alpha(|V|, |E|))$
- Au total :

Union / find : Arbre couvrant de poids minimal (Validité)

Montrons que l'algorithme de Kruskal construit bien un arbre couvrant de poids minimal.

Lemme : Soient T_1 et T_2 deux arbres couvrants de G, soit une arête $a \in T_2$ et $a \notin T_1$ alors il existe une arête $b \in T_1$ telle que $T_2 - a + b$ soit un arbre recouvrant.

- Soit $a = (x, y)$
- Il existe un chemin de x à y dans T_1
- Soit b une arête de ce chemin qui relie les deux sous arbres $T_2 - a$

Union / find : Arbre couvrant de poids minimal (Validité)

- Soit T_K l'arbre de Kruskal et soit T^* un arbre couvrant de poids minimal dont le nombre d'arêtes distinctes de T_K est minimal
- Hypothèse : $T_K \neq T^*$
- Soit $a \in T^*$ et $a \notin T_K$ et soit $b \in T_K$ l'arête du lemme précédent ($T^* - a + b$ est un arbre couvrant)
- T^* est minimal et donc $w(T^*) \leq w(T^* - a + b)$ soit $w_a \leq w_b$
- Kruskal commence par les arêtes de poids petit. Or b est dans T_K et pas a. Donc $w_b \leq w_a$.
- L'arbre $T^* - a + b$ a le même poids que T^* et est plus "proche" de T_K que T^*
- Absurde

Arbre couvrant de poids minimal (pb. Euclidien)

- n points dans le plan $P_1, ..., P_n$
- Distance Euclidienne (poids d'une arête = sa longueur)
- Objectif : Clacul d'un arbre couvrant de poids minimal

Notre mauvaise réponse : Calcul du graphe complet (toutes les paires de points sont considérées), tri puis Kruskal $\rightarrow O(n^2 \log n)$

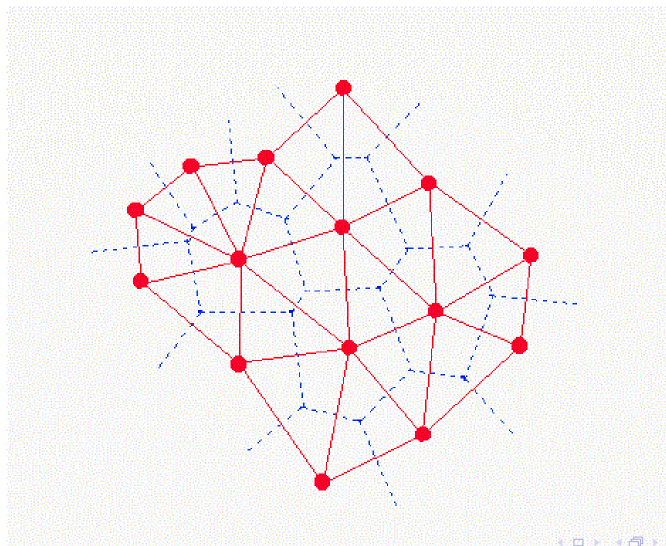
Mieux en exploitant les propriétés géométriques ?

Le diagramme de Voronoï associé à n points P_i du plan est l'ensemble des n régions V_i tq

$$V_i = \{x | \forall j \neq i, d(x, P_i) \leq d(x, P_j)\}$$

La triangulation de Delaunay est un graphe $(\{P_1, ..., P_n\}, D)$ où $(P_i, P_j) \in D$ ssi ils appartiennent à des régions de Voronoï adjacentes.

Arbre couvrant de poids minimal (pb. Euclidien)



Arbre couvrant de poids minimal (pb. Euclidien)

Un algorithme rapide en deux étapes :

1. Triangulation de Delaunay
   - Le nombre d'arêtes de la triangulation croit linéairement avec n
   - Calcul en $O(n \log n)$
2. Calcul l'arbre couvrant de poids minimal sur le graphe de Delaunay

Cet arbre couvrant est un arbre couvrant du graphe initial et il est optimal !

Quelques propriétés d'une triangulation de Delaunay

Soit D un disque de diamètre $[P_i, P_j]$. Si D ne contient aucun autre point alors l'arête (P_i, P_j) est dans le graphe de Delaunay

En effet, le centre O du disque est équidistant de P_i et de P_j et de plus, tous les autres points sont plus éloignés de O que P_i, P_j. Et donc $O \in V_i \cap V_j$

Arbre couvrant de poids minimal (pb. Euclidien)

- Soit T un arbre couvrant de poids minimal et soit (a, b) une arête de T qui n'est pas dans le graphe de Delaunay.
- Il existe donc un point c dans le disque de diamètre $[P_i, P_j]$
- Enlevons (a, b) de l'arbre T. Nous avons alors deux sous-arbres
- Sans perte de généralité, supposons que c est dans le même sous-arbre que a
- Relions les deux sous arbres par l'arête (b, c)
- Nous avons alors un arbre couvrant T'
- Son poids est égal à celui de $T + |bc| - |ab|$. Absurde