**Informatique** *Cours*
**Algorithmique des graphes - Pt2** Ipesup

## Objectifs

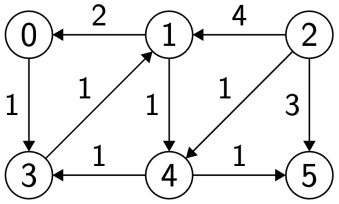
À l'issue de cette leçon, l'étudiant doit être capable de :
- Comprendre et implémenter le parcours en profondeur (`dfs`) sur un graphe.
- Déterminer les sommets atteignables, tester l'existence d'un chemin et analyser la complexité $O(V + E)$.
- Calculer les composantes connexes d'un graphe non orienté.
- Exploiter l'ordre postfixe et l'appliquer au tri topologique des graphes acycliques.

# 1 Plus court chemin

Dans cette section, on considère des graphes orientés pondérés, où les poids sont assimilés à des distances et donc considérés comme *positifs ou nuls*. On s'intéresse au problème de trouver le plus court chemin d'un sommet à un autre sommet, la longueur n'étant plus le nombre d'arcs mais la somme des poids le long du chemin.

### Exemple

Si on considère le graphe

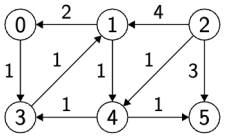



le plus court chemin du sommet 2 au sommet 0 est de longueur 5. Il s'agit du chemin $2 \to 4 \to 3 \to 1 \to 0$. En particulier, il est plus court que le chemin $2 \to 1 \to 0$, de longueur 6, même si celui-ci contient moins d'arcs. De même, le plus court chemin du sommet 2 au sommet 5 est de longueur 2, en passant par le sommet 4. Dans certains cas, il n'y a pas de chemin, et donc pas de plus court chemin. C'est le cas par exemple entre le sommet 5 et tout autre sommet du graphe, ou encore entre le sommet 4 et le sommet 2.

Dans ce qui suit, on note $u \xrightarrow{d*} v$ le fait qu'il existe un chemin de $u$ à $v$ de longueur $d$.
On appelle *distance* d'un sommet $u$ à un sommet $v$ la longueur d'un plus court chemin de $u$ à $v$. S'il existe au moins un chemin de $u$ à $v$, alors la distance est bien définie, car les poids ont été supposés positifs ou nuls et par conséquent les longueurs des chemins sont minorées par 0. On note qu'il peut exister plusieurs chemins de longueur minimale. La distance n'est pas définie lorsqu'il n'existe pas de chemin entre les deux sommets.

## 1.1 Algorithme de Floyd–Warshall

On commence par un algorithme qui détermine la distance pour toute paire de sommets, lorsqu'elle existe, sous la forme d'une matrice. La figure suivante donne une matrice pour le graphe pris plus haut en exemple.




| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 2 | ∞ | 1 | 3 | 4 |
| 1 | 2 | 0 | ∞ | 2 | 1 | 2 |
| 2 | 5 | 3 | 0 | 2 | 1 | 2 |
| 3 | 3 | 1 | ∞ | 0 | 2 | 3 |
| 4 | 4 | 2 | ∞ | 1 | 0 | 1 |
| 5 | ∞ | ∞ | ∞ | ∞ | ∞ | 0 |

On note en particulier que la diagonale contient la valeur 0. On remarque également la colonne 2 qui ne contient que la valeur $\infty$ (sauf pour la ligne 2), car aucun chemin ne mène au sommet 2, de même que la ligne 5 qui ne contient que la valeur $\infty$ (sauf pour la colonne 5), car aucun chemin ne part du sommet 5.

Pour calculer cette matrice, on va utiliser l'algorithme de Floyd–Warshall, dont le principe consiste à chercher des chemins passant par de plus en plus de sommets différents. Initialement, on ne considère que les chemins réduits à un seul arc, c'est-à-dire sans aucun sommet intermédiaire. Puis, dans un deuxième temps, on considère les chemins qui empruntent uniquement le sommet 0 comme sommet intermédiaire, c'est-à-dire les chemins de la forme $i \to 0 \to j$. Si cela constitue une amélioration par rapport aux chemins que l'on connaît déjà, on met à jour notre matrice. Puis on considère les chemins qui peuvent également emprunter le sommet 1 comme sommet intermédiaire. Et ainsi de suite, jusqu'à avoir autorisé les chemins à passer par n'importe quels sommets intermédiaires [1].

Le programme suivant contient une fonction OCaml qui réalise cette idée.

**Algorithme de Floyd–Warshall**

```
1  let floyd_warshall (g: wdigraph) : float array array =
2    let n = size g in
3    let dist = Array.make_matrix n n infinity in
4    List.iter (fun (d, i, j) -> dist.(i).(j) <- d) (edges g);
5    for i = 0 to n - 1 do dist.(i).(i) <- 0. done;
6    for k = 0 to n - 1 do
7      for i = 0 to n - 1 do
8        for j = 0 to n - 1 do
9          let x = dist.(i).(k) +. dist.(k).(j) in
10         if x < dist.(i).(j) then dist.(i).(j) <- x
11       done
12     done
13   done;
14   dist
```

La fonction reçoit un graphe orienté pondéré `g` en argument et renvoie une matrice donnant, pour chaque paire de sommets, la distance entre ces deux sommets, lorsqu'un chemin existe, et la valeur `infinity` sinon. Le code commence par construire cette matrice (ligne 3), la remplir avec les poids associés à chaque arc (ligne 4) et initialiser sa diagonale avec la distance

1. L'algorithme de Floyd–Warshall est un exemple de programmation dynamique.

0 (ligne 5). On note que lorsque le graphe est une matrice d'adjacence, cela revient donc à faire une simple copie de cette matrice. Mais ici on a écrit un code qui ne préjuge pas de la représentation de la matrice étant, même lorsque le graphe est une matrice d'adjacence, il faut faire cette copie, ce qui n'est pas moins coûteux. Puis le programme se compose de trois boucles imbriquées, qui vont petit à petit considérer des chemins qui passent par de plus en plus de sommets différents. À chaque étape de la boucle sur `k` (ligne 6), on considère la possibilité que le chemin de `i` à `j` passe par `k`. Si cela donne une distance plus petite que celle que l'on connaît pour l'instant, on met à jour la matrice (ligne 10).

**Correction.**

Il est clair que l'algorithme de Floyd–Warshall termine, car il est constitué uniquement de boucles `for`.
Pour montrer qu'il calcule bien les longueurs des plus courts chemins, on defini l'invariant de cet algorithme comme un invariant de la boucle `for` sur l'indice `k`.
Pour toute paire de sommets $i$ et $j$, la valeur `dist.(i).(j)` est la longueur d'un plus court chemin de $i$ à $j$ qui n'emprunte que des sommets intermédiaires strictement inférieurs à $k$, si un tel chemin existe, et $\infty$ sinon.

**Démonstration.**

— Initialement, c'est-à-dire pour $k = 0$, l'invariant est vrai car `dist.(i).(j)` coïncide avec les arcs du graphe et un chemin n'empruntant aucun sommet intermédiaire est donc nécessairement réduit à un arc.

— Supposons l'invariant vrai pour une certaine valeur de $k$ et exécutons le corps de la boucle, c'est-à-dire les deux boucles imbriquées sur $i$ et $j$. Soient $i$ et $j$ deux sommets et un plus court chemin de $i$ à $j$ qui n'emprunte que des sommets intermédiaires strictement inférieurs à $k+1$. Si aucun de ces sommets n'est égal à $k$, alors `dist.(i).(j)` contient déjà la longueur de ce chemin. Sinon, le chemin est de la forme

$$i \to v_0 \to \cdots \to v_{n-1} \to k \to w_0 \to \cdots \to w_{m-1} \to j$$

avec tous les sommets $v_\ell$ et $w_\ell$ strictement inférieurs à $k$ car on peut considérer le chemin sans cycle – rappelons que les poids sont positifs ou nuls. Dès lors, la longueur du chemin de $i$ à $k$ est égale à `dist.(i).(k)` par l'invariant de boucle, et de même la longueur du chemin de $k$ à $j$ est égale à `dist.(k).(j)`. C'est bien la somme de ces deux longueurs qui est affectée à `dist.(i).(j)`. Si en revanche il n'existe pas de chemin avec des sommets intermédiaires strictement inférieurs à $k+1$, alors il n'existait pas de chemin avec des sommets intermédiaires strictement inférieurs à $k$, et `dist.(i).(j)` conserve donc la valeur $\infty$.

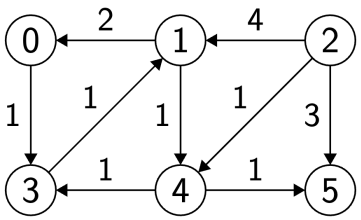
Enfin, cet invariant permet bien de conclure car, en sortie de boucle, on a $k$ égal au nombre de sommets du graphe. Dès lors, `dist.(i).(j)` est la longueur d'un plus court chemin de $i$ à $j$, si un tel chemin existe, car les sommets intermédiaires ne sont plus limités. Et s'il n'existe pas de chemin entre $i$ et $j$, alors `dist.(i).(j)` vaut bien $\infty$. □

**Complexité.** La complexité de l'algorithme de Floyd–Warshall est clairement en $O(V^3)$. En effet, la création de la matrice `dist` et son initialisation sont respectivement en $O(V^2)$ et en $O(E)$, soit $O(V^2)$. Mais ceci est dominé par les trois boucles `for` imbriquées, dont le coût est clairement $O(V^3)$.

**Construire le chemin.** Notre programme calcule les distances entre les sommets mais il ne donne pas pour autant un *chemin* qui réalise cette distance. Il est cependant facile de modifier le programme pour être en mesure de donner un chemin de longueur minimale entre deux sommets, lorsqu'il existe. Il suffit de conserver, pour chaque paire $(i, j)$, l'indice $k$ qui a permis d'obtenir une distance plus courte à la ligne 10.

## 1.2 Algorithme de Dijkstra

On considère maintenant le problème de déterminer tous les plus courts chemins à partir d'une source donnée. L'algorithme que nous présentons ici pour résoudre ce problème est dû à Edsger W. Dijkstra et il date de 1959. C'est une variation du parcours en largeur. Comme pour ce dernier, on procède par « cercles concentriques ». La différence est ici que les rayons de ces cercles représentent une distance en terme de poids total et non en terme du nombre d'arcs. Si on prend l'exemple du graphe precedente,




en partant de la source 2, on atteint d'abord les sommets à distance 1 (à savoir 4), puis à distance 2 (à savoir 3 et 5), puis à distance 3 (à savoir 1), puis enfin à distance 5 (à savoir 0). La difficulté de mise en œuvre vient du fait qu'on peut atteindre un sommet avec une certaine distance, par exemple le sommet 5 avec l'arc $2 \to 5$, puis trouver plus tard un chemin plus court en empruntant d'autres arcs, par exemple $2 \to 4 \to 5$. On ne peut plus se contenter d'une file comme dans le parcours en largeur, on va utiliser une *file de priorité*. Elle contient les sommets déjà atteints, ordonnés par distance à la source. Lorsqu'un meilleur chemin est trouvé, le sommet est remis dans la file avec une plus grande priorité, c'est-à-dire une distance plus petite.

Le programme suivante contient un code OCaml qui implémente cette idée, sur la base du type `wdigraph` des graphes orientés pondérés.

**Algorithme de Dijkstra**

```
1  let dijkstra (g: wdigraph) (source: int) : float array =
2    let n = size g in
3    let dist = Array.make n infinity in
4    let pqueue = Pqueue.create () in
5    let visited = Array.make n false in
6    let add v d = dist.(v) <- d; Pqueue.insert pqueue (d, v) in
7    add source 0.;
8    while not (Pqueue.is_empty pqueue) do
9      let dv, v = Pqueue.extract_min pqueue in
10     if not visited.(v) then (
11       visited.(v) <- true;
12       (* on vient de dterminer la distance de v *)
13       List.iter
```

```
14          (fun (w, dvw) ->
15            let d = dv +. dvw in
16            if d < dist.(w) then add w d)
17          (succ g v)
18      )
19    done;
20    dist
```

La fonction `dijkstra` renvoie la distance de chaque sommet à la source, sous forme d'un tableau. Comme pour l'algorithme de Floyd–Warshall, la valeur `infinity` est utilisée pour désigner un sommet pour lequel il n'y a pas de chemin. Prenons le temps de détailler et d'expliquer le code.

On commence par créer le tableau qui sera le résultat final, avec la valeur `infinity` pour chaque sommet, et une file de priorité.

```
let dijkstra (g: wdigraph) (source: int) : float array =
  let n = size g in
  let dist = Array.make n infinity in
  let pqueue = Pqueue.create () in
  let visited = Array.make n false in
```

Cette file de priorité va contenir des paires $(d, v)$ où $v$ est un sommet et $d$ sa distance à la source. On suppose que la file de priorité ordonne les éléments selon la première composante de la paire, par exemple en utilisant un ordre lexicographique sur les paires. C'est en pratique ce qui se passe si on utilise la comparaison structurelle polymorphe d'OCaml. Ceci aura bien l'effet de parcourir les sommets par distance croissante à la source. On se donne également un tableau `visited` pour marquer les sommets pour lesquels on a déjà trouvé un plus court chemin.

```
  let visited = Array.make n false in
```

Enfin, on se donne une fonction `add` qui affecte une distance $d$ au sommet $v$ et met la paire $(v, d)$ dans la file de priorité. On applique immédiatement cette fonction à la source, avec la distance 0.

```
let add v d = dist.(v) <- d; Pqueue.insert pqueue (d, v) in
add source 0.;
```

Comme pour un parcours en largeur, on procède alors à une boucle, tant que la file n'est pas vide. Le cas échéant, on extrait le premier élément de la file, $v$, avec sa distance $dv$.

```
while not (Pqueue.is_empty pqueue) do
  let dv, v = Pqueue.extract_min pqueue in
```

Si $v$ est marqué dans `visited`, c'est que l'on a déjà trouvé un plus court chemin jusqu'à ce sommet et il n'y a rien à faire. Cette situation peut effectivement se produire lorsqu'un

premier chemin est trouvé puis un autre, plus court, trouvé plus tard. Ce dernier passe alors dans la file de priorité devant le premier. Lorsque le chemin plus long finit par sortir de la file, il faut l'ignorer. Si en revanche le sommet $v$ n'est pas marqué dans `visited`, c'est qu'on vient de déterminer la distance du sommet $v$ à la source. On le marque donc dans `visited`.

```
if not visited.(v) then (
  visited.(v) <- true;
```

Puis on examine chaque successeur $w$ de $v$. La distance à $w$ en empruntant l'arc correspondant est la somme de la distance à $v$, c'est-à-dire $dv$, et du poids $dvw$ de l'arc.

```
List.iter
  (fun (w, dvw) ->
     let d = dv +. dvw in
```

Plusieurs cas de figure sont possibles pour le sommet $w$. Soit c'est la première fois qu'on l'atteint, soit on connaît déjà une distance `dist.(w)`. Dans ce dernier cas, on peut ou non améliorer la distance à $w$ en passant par $v$. Le seul test $d <$ `dist.(w)` suffit à déterminer si la distance $d$ mérite d'être considérée, car le tableau `dist` a été initialisé avec `infinity`. Le cas échéant, on ajoute $w$ à la file de priorité.

```
     if d < dist.(w) then add w d)
  (succ g v)
)
```

Une fois tous les successeurs traités, on réitère la boucle principale. Une fois qu'on est sorti de celle-ci, tous les sommets atteignables ont leur distance à la source renseignée dans `dist`. C'est ce que l'on renvoie.

```
done;
dist
```

**Complexité.** Évaluons la complexité de l'algorithme de Dijkstra, dans le pire des cas. La file de priorité peut contenir jusqu'à $E$ éléments, car l'algorithme visite chaque arc au plus une fois, et chaque considération d'un arc peut conduire à l'insertion d'un élément dans la file. En supposant que les opérations `insert` et `extract_min` de la file de priorité ont un coût logarithmique (c'est le cas pour les files de priorité), chaque opération sur la file a donc un coût $O(\log E)$, c'est-à-dire $O(\log V)$ car $E \leq V^2$. D'où un coût total $O(E \log V)$.

**Correction**

Il n'est pas complètement évident de se persuader que l'algorithme de Dijkstra est correct. Montrons qu'à la fin de la fonction `dijkstra`, le tableau `visited` contient exactement les sommets atteignables depuis la source et le tableau `dist` donne pour ces sommets la longueur d'un plus court chemin.
On le fait en établissant des invariants de boucle : on note $v \in$ `visited` pour signifier que le sommet $v$ est marqué à `true` dans le tableau `visited`. De même, on note $v \in$ `pqueue` pour signifier que le sommet $v$ apparaît dans la file de priorité (pour une certaine

distance).
Les deux premiers invariants stipulent que la source fait toujours partie des sommets déjà considérés et que sa distance est toujours égale à 0.

$$\texttt{source} \in \texttt{visited} \cup \texttt{pqueue} \qquad (1)$$

$$\texttt{dist[source]} = 0 \qquad (2)$$

Le troisième invariant stipule que `dist` contient effectivement la longueur d'un chemin pour tout sommet déjà considéré.

$$\forall v \in \texttt{visited} \cup \texttt{pqueue}, \quad \texttt{source} \xrightarrow{\texttt{dist[v]}*} v \qquad (3)$$

Pour les sommets dans `visited`, le quatrième invariant stipule plus précisément qu'il s'agit de la longueur d'un plus court chemin.

$$\forall v \in \texttt{visited}, \ \forall d, \ \texttt{si source} \xrightarrow{d*} v \ \texttt{alors dist[v]} \leq d \qquad (4)$$

Le cinquième invariant stipule que, pour tout arc $v \to w$ déjà considéré, la distance à $w$ n'excède pas celle du chemin passant par $v$.

$$\forall v \in \texttt{visited}, \ \forall w \ \texttt{t.q.} \ v \xrightarrow{d} w, \quad w \in \texttt{visited} \cup \texttt{pqueue et dist[w]} \leq \texttt{dist[v]} + d \qquad (5)$$

Enfin, le sixième invariant indique que tout sommet $v$ à une distance inférieure au plus petit élément de `pqueue` est nécessairement déjà dans `visited`.

$$\forall v, \ \texttt{si source} \xrightarrow{d*} v \ \texttt{et} \ d < \min(\texttt{pqueue}) \ \texttt{alors} \ v \in \texttt{visited} \qquad (6)$$

Montrer que ces six propriétés sont effectivement des invariants de boucle nécessite de montrer que d'une part elles sont établies initialement (*i.e.*, avant la boucle) et que d'autre part elles sont préservées par toute exécution du corps de la boucle. La première partie de cette preuve est simple, car initialement, `visited` est vide et `pqueue` ne contient que le sommet `source`. La préservation des invariants est plus subtile.

- Les deux premiers invariants sont clairement préservés, car la source passe de `pqueue` à `visited` à la première itération, puis y reste. Par ailleurs, sa distance est nulle et donc ne peut être améliorée par la suite.
- L'invariant (3) est préservé car chaque mise à jour de distance correspond à la somme de la longueur d'un chemin jusqu'à un nœud, pour lequel l'invariant est supposé, et du poids d'un arc sortant de ce sommet.
- Pour montrer la préservation de l'invariant (4), considérons un sommet $u$ et l'instant où `dist[u]` est fixée, c'est-à-dire l'instant où $u$ sort de la file pour être ajouté à `visited`. Un chemin $\texttt{source} \xrightarrow{*} u$ strictement plus court que `dist[u]` sortirait de `visited` par un certain arc $v \to w$. Mais alors on aurait `dist[w]` < `dist[u]` ce qui contredit le choix de $u$.
- La préservation de l'invariant (5) découle directement du fait que, lorsqu'un sommet est ajouté à `visited`, tous les arcs sortant de ce sommet sont examinés.
- Enfin, l'invariant (6) est préservé par un argument analogue à celui de la préservation de l'invariant (4) : un chemin plus court que min(`pqueue`) vers un sommet qui n'est pas dans `visited` devrait nécessairement sortir de `visited` par un arc dont l'extrémité est dans `pqueue`, en vertu de l'invariant (5), et contredirait donc la minimalité de min(`pqueue`).

Il reste à déduire de ces invariants de boucle la correction de l'algorithme de Dijkstra. On sort de la boucle lorsque la file de priorité `pqueue` est vide. L'invariant (3) nous assure alors que `visited` ne contient que des sommets atteignables depuis la source. Inversement, tout sommet atteignable depuis la source est nécessairement dans `visited`. En effet, la source y appartient, en vertu de l'invariant (1), et un chemin de la source à un sommet $v$ en dehors de `visited` devrait donc sortir de `visited` par un certain arc. Mais cela contredirait alors l'invariant (5). L'ensemble `visited` contient donc exactement les sommets atteignables depuis la source et l'invariant (4) stipule que `dist` contient bien la longueur d'un plus court chemin pour chacun de ces sommets. □

**Construire le chemin.** Notre programme calcule les distances de la source à chaque sommet mais il ne donne pas pour autant un *chemin* qui réalise cette distance. Il est facile de modifier le programme pour être en mesure de donner un chemin de longueur minimale vers un sommet, lorsqu'il existe. Il suffit de conserver, pour chaque sommet $w$, le sommet $v$ qui a permis d'obtenir sa distance.

## 1.3 Algorithme A*

L'algorithme de Dijkstra, que nous venons de voir, détermine un plus court chemin, depuis une source donnée, pour tous les sommets du graphe qui sont atteignables. Supposons maintenant que l'on cherche uniquement un plus court chemin entre la source et une *destination* donnée. Un bon exemple d'application serait le navigateur GPS d'une voiture. On pourrait se servir de l'algorithme de Dijkstra, mais il est totalement inutile de déterminer les plus courts chemins entre Bordeaux et toutes les villes de France si on souhaite aller à Lyon. Bien entendu, il est facile de modifier l'algorithme de Dijkstra pour s'interrompre dès lors qu'on a trouvé un plus court chemin jusqu'à la destination. Il suffit de s'arrêter lorsque la destination sort de la file de priorité. Mais cela reste très inefficace car l'algorithme de Dijkstra va partir dans toutes les directions.

Il y a fort à parier que, lorsqu'un plus court chemin jusqu'à Lyon aura été trouvé, beaucoup de villes de France auront été explorées.

L'algorithme A* permet de déterminer un plus court chemin entre une source `src` et une destination `dst` données avec une meilleure efficacité que l'algorithme de Dijkstra pourvu que l'on guide l'algorithme pour qu'il aille dans la bonne direction. Cette assistance prend la forme d'une fonction d'heuristique $h$ qui, pour chaque sommet $v$, estime la distance entre $v$ et la destination `dst`. L'heuristique $h$ doit prendre des valeurs positives ou nulles et vérifier $h(\texttt{dst}) = 0$. Par ailleurs, pour que l'algorithme A* détermine bien un plus court chemin, l'heuristique doit avoir la propriété suivante.

**Heuristique admissible**

La fonction d'heuristique $h$ est dite *admissible* si, pour tout sommet $v$ tel qu'il existe un chemin de $v$ à `dst` de longueur $d$, alors $h(v) \leq d$. Autrement dit, une heuristique admissible ne surestime jamais la distance à la destination. On note que l'hypothèse $h(\texttt{dst}) = 0$ est cohérente avec cette propriété.

Un exemple naturel d'heuristique admissible est la « distance à vol d'oiseau » dans un graphe où les sommets sont des points dans le plan et où la distance d'un arc $v \to w$ est la distance

euclidienne entre les points $v$ et $w$. En effet, toute succession d'arcs entre un sommet et la destination ne pourra jamais être plus courte que la distance en ligne droite.

L'algorithme A* procède d'une façon très similaire à l'algorithme de Dijkstra. On utilise toujours une file de priorité contenant des sommets qui ont été atteints par un chemin depuis la source. Cependant, la priorité n'est plus la distance à la source, mais la somme de la distance à la source et de l'estimation donnée par l'heuristique. L'autre différence est que l'on s'interrompt dès que la destination est atteinte. Le programme suivante contient un code OCaml qui met en œuvre l'algorithme A*.

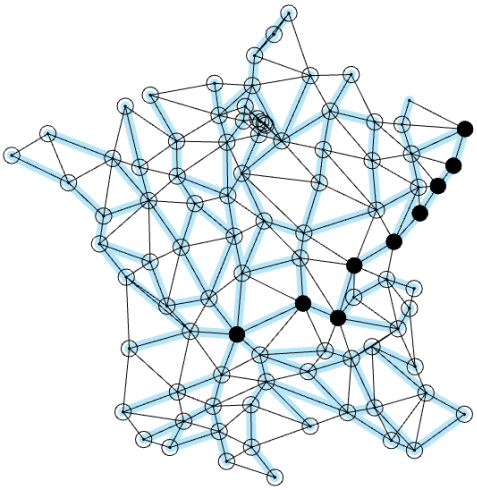
### Algorithme A*

Détermine la longueur d'un plus court chemin de `src` à `dst` dans le graphe `g`, en utilisant l'heuristique `h`, si un chemin existe, et lève `Not_found` sinon.

```
let astar (g: wdigraph) (src: int) (dst: int)
    (h: int -> float) : float =
  let n = size g in
  let dist = Array.make n infinity in
  let pqueue = Pqueue.create () in
  let add v d =
    dist.(v) <- d; Pqueue.insert pqueue (d +. h v, v) in
  add src 0.;
  let relax v (w, dvw) =
    let d = dist.(v) +. dvw in
    if d < dist.(w) then add w d in
  let rec loop () =
    if Pqueue.is_empty pqueue then raise Not_found;
    let _, v = Pqueue.extract_min pqueue in
    if v = dst then
      dist.(dst)
    else (
      List.iter (relax v) (succ g v);
      loop ()
    ) in
  loop ()
```

La fonction `relax` détermine si on a trouvé un meilleur chemin jusqu'à $w$ en passant par $v$. Le cas échéant, on insère $w$ dans la file de priorité avec la fonction `add`. C'est là que l'heuristique $h$ est utilisée.

La figure illustre la différence entre l'algorithme de Dijkstra et l'algorithme A* sur la recherche d'un plus court chemin entre la Corrèze (19) et le Bas-Rhin (67) sur le graphe des départements français. Dans les deux cas, on obtient le même plus court chemin, pour une distance totale de 1061,75. (On rappelle que ce ne sont pas des kilomètres mais seulement des distances dans le plan 2D calculées à partir des coordonnées $(x, y)$ où sont dessinées les préfectures.)

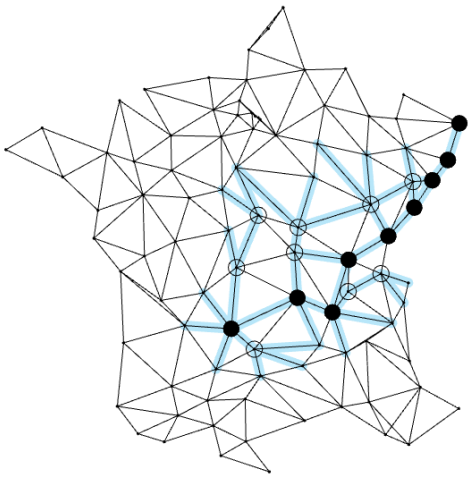


**Algorithme de Dijkstra**

On fait au total 117 insertions dans la file de priorité.

On constate que tous les départements sont visités par l'algorithme.

Et on a déterminé un plus court chemin depuis la source (la Corrèze) pour tous les départements à l'exclusion d'un seul (la Moselle).



**Algorithme A***

Seuls 41 départements ont été visités par l'algorithme.

Cette fois, on détermine un plus court chemin pour seulement 18 sommets, dont les 9 qui constituent le résultat.