Parcours de graphes

1	**Rappels sur les graphes**	**1**
2	**Parcours en largeur**	**2**
3	**Parcours en profondeur**	**3**
3.1	avec la récursivité	3
3.2	avec une pile	4
4	**Algorithme de Dijkstra**	**6**
5	**Algorithme A* : recherche informée**	**9**
5.1	Vers l'algorithme A*	9
5.2	L'algorithme A*	11
5.3	Bilan	14

1 Rappels sur les graphes

Définition 1 Un **graphe** est un couple $G = (S, A)$ où S est un ensemble fini de **sommets** et A est un ensemble :

- de **paires** $\{x, y\}$ (avec $x \neq y$) de sommets appelés **arêtes** si le graphe est **non orienté**
- de **couples** (x, y) (avec $x \neq y$) de sommets appelés **arcs** si le graphe est **orienté**

On exprimera les complexités des algorithmes sur les graphes en fonction de $n = \text{Card}(S)$ et $m = \text{Card}(A)$. On notera dans ce cours $xy \in A$ pour signifier indistinctement que l'arête ou l'arc appartient au graphe selon s'il est orienté ou non.

- Un graphe peut-être représenté au choix:
  - par une **matrice d'adjacence** M de taille $n \times n$ où $m_{ij} = 1$ si $x_i x_j \in A$ et $m_{ij} = 0$ sinon.
  - par **listes d'adjacence** c'est-à-dire à l'aide d'un **tableau** T de longueur n où chaque case $T[i]$ contient la liste des numéros de voisins (sortants) du sommet x_i.

  Dans les deux cas, cela nécessite une numérotation des sommets que l'on débutera par 0.

Définition 2 Soit $G = (S, A)$ un graphe. Un **chemin** dans G est une suite finie x_0, x_1, ..., x_{p-1} de sommets de G qui vérifient :

$$\forall i \in [|0, p|],\ x_i x_{i+1} \in A$$

L'entier $p \in \mathbb{N}$ s'appelle la **longueur** du chemin.

Remarquons qu'un sommet seul consitue toujours un chemin de longueur 0.

Généralités sur les parcours

- Un **parcours** de graphe consiste en une exploration de ses sommets en choisissant un sommet de départ et en explorant de proche en proche ses voisins.

- Dans un parcours on peut distinger 3 types de nœuds :

  - les nœuds blancs : non visités
  - les nœuds gris, aussi appelés **nœuds ouverts** : ils ont été découverts pendant le parcours mais sont en attente d'être explorés
  - les nœuds noirs, aussi appelés **nœuds fermés** : ils ont été explorés.

- À chaque itération du parcours un nœud ouvert est visité et devient alors fermé, on considère alors tous ses voisins qui peuvent devenir à leur tour ouverts.

- De plus, lors d'un parcours, chaque sommet $x \in S$ (sauf celui de départ) se voit attribuer un **prédécesseur** $p(x)$ qui est le sommet depuis lequel on a exploré x. L'ensemble des prédécesseurs définit une **arborescence du parcours** qui est un arbre de racine le sommet de départ et qui permet de reconstruire les chemins d'exploration depuis le nœud de départ.

2 Parcours en largeur

Commençons par le parcours le plus simple conceptuellement. Dans le **parcours en largeur**, les sommets sont explorés dans l'ordre dans lequel ils ont été découverts. On utilise donc une **file** (FIFO) pour stocker les sommets ouverts (gris).

Algorithme 1

Entrées : un graphe G sous forme de listes d'adjacence, un numéro x_d de sommet de départ

1 **Initialisation :**
2 $file \leftarrow$ nouvelle_file();
3 $p \leftarrow$ nouveau_tableau$(n, -1)$;
4 $c \leftarrow$ nouveau_tableau$(n, Blanc)$;
5 $c[x_d] \leftarrow Gris$;
6 inserer$(x_d, file)$;
7 **Boucle principale :**
8 **Tant que** non_vide$(file)$ **Faire**
9 $x \leftarrow$ extraire$(file)$;
10 $c[x] \leftarrow Noir$;
11 $v \leftarrow$ voisins(x);
12 **Pour chaque** $y \in v$ **Faire**
13 **Si** $c[y] = Blanc$ **Alors**
14 $c[y] \leftarrow Gris$;
15 $p[y] \leftarrow x$;
16 inserer$(y, file)$;
17 **Fin Si**
18 **Fin Pour**
19 **Fin Tant que**

Sorties : c, p

- **Complexité :** Grâce au marquage un sommet est ajouté au plus une seule fois dans la file. Ainsi la boucle **Tant que** s'exécute au pire n fois. La boucle **Pour** est effectuée $\mathrm{d}^+(x)$ fois, donc sur l'ensemble de l'exécution elle est effectuée au pire m fois en tout. Ainsi la complexité du parcours est $O(n+m)$ (linéaire).
- **Résultat :** À la fin de l'algorithme le tableau de couleurs permet de déterminer l'ensemble des sommets **accessibles** depuis x_d.Si le graphe est non orienté, ce la signifie qu'il permet de déterminer la composante connexe à laquelle x_d appartient. L'arborescence du parcours permet de construire les chemins pour atteindre chaque sommet *Noir* depuis x_d.

Proposition 1 Le parcours en largeur depuis x_d construit une arborescence de chemins optimaux, au sens des plus courts chemins en nombre d'arcs depuis x_d.

[Preuve : dans le parcours on pourrait ajouter la construction d'un tableau $d[n]$ notant la distance en nombre d'arcs depuis la source. On montre ensuite l'invariant que pour tout sommet n noir, $d[n]$ est bien la distance à la source]

Voir le fichier de présentation pour un exemple animé de parcours en largeur

3 Parcours en profondeur

Le **parcours en profondeur** est légèrement plus difficile que le parcours en largeur, mais il est le plus facile à programmer. En effet, ce parcours est essentiellement une exploration **récursive** du graphe.

Dans le parcours en profondeur, le dernier sommet ouvert rencontré est choisi. Cela peut s'obtenir de deux manières : soit par récursivité, soit à l'aide d'une pile (LIFO).

3.1 avec la récursivité

Le **parcours en profondeur** est un parcours de nature récursive : on explore le sommet de départ x_d en explorant ses voisins, de la même manière (d'où la récursivité). Pour éviter d'explorer plusieurs fois le même sommet on utilise un marquage :

- sommet vert : non découvert
- sommet orange : en cours d'exploration (on est en train d'explorer ses descendants)
- sommet rouge : exploration terminée

Attention, exceptionnellement le marquage en couleur est différent des autres algorithmes présentés dans ce cours car il ne correspondent pas aux notions de nœuds ouverts et fermés précédemment introduites.

Intuitivement, ce parcours tente d'avancer au maximum dans le graphe en faisant à chaque fois un choix d'arête et quand il ne trouve plus d'issue il revient sur ces pas pour essayer les arêtes non utilisées. Ainsi l'exploration du graphe se fait en profondeur.

Algorithme 2

Entrées : un graphe G sous forme de listes d'adjacence, un numéro x_d de sommet de départ

1 **Initialisation**
2 $p \leftarrow$ nouveau_tableau$(n, -1)$;
3 $c \leftarrow$ nouveau_tableau$(n, Vert)$;
4 **Fonction récursive**
5 FONCTION explore(x : numero de sommet, prec : sommet précédent)
6 **Si** $c[x] = Vert$ **Alors**
7 $c[x] \leftarrow Orange$;
8 $p[x] \leftarrow$ prec;
9 $v \leftarrow$ voisins(x);
10 **Pour chaque** $y \in v$ **Faire**
11 $explore(y, x)$;
12 **Fin Pour**
13 $c[x] \leftarrow Rouge$;
14 **Fin Si**
15 FIN FONCTION
16 **Premier appel**
17 $explore(x_d, -1)$;

On remarque que le marquage sert à ne pas redéclencher l'exploration d'un sommet qui a déjà été visité (Rouge) ou qui est en cours de visite (Orange).

Le parcours en profondeur, en plus d'être très facile à programmer, possède également de bonnes propriétés concernant l'ordre dans lequel les sommets deviennent Orange et Rouge. Ces propriétés sont utiles pour :

- calculer un ordre topologique dans un graphe orienté acyclique (DAG) (Revoir le cours de 1ere année)
- calculer les composantes fortement connexes d'un graphe (voir le chapitre correspondant)

3.2 avec une pile

Cette section a surtout pour but de montrer que le parcours en profondeur rentre dans le cadre général des parcours exprimé plus haut avec les notions de nœuds ouverts et fermés. On utilise cette fois-ci une **pile** LIFO afin de visiter le dernier sommet gris rencontré. On préférera toutefois programmer la version récursive lorsque c'est possible.

Algorithme 3

Entrées : un graphe G sous forme de listes d'adjacence, un numéro x_d de sommet de départ

1 **Initialisation**
2 $pile \leftarrow$ nouvelle_pile();
3 $p \leftarrow$ nouveau_tableau$(n, -1)$;
4 $c \leftarrow$ nouveau_tableau$(n, Blanc)$;
5 $c[x_d] \leftarrow Gris$;
6 inserer$(x_d, pile)$;
7 **Boucle principale :**
8 **Tant que** non_vide$(pile)$ **Faire**
9   $x \leftarrow$ extraire$(pile)$;
10   **Si** $couleur[x] = Gris$ **Alors**
11     $couleur[x] \leftarrow Noir$;
12     $v \leftarrow$ voisins(x);
13   **Pour chaque** $y \in v$ **Faire**
14   **Si** $couleur[y] = Blanc$ **Alors**
15         $c[y] \leftarrow Gris$;
16         $p[y] \leftarrow x$;
17   inserer$(y, pile)$;
18   **SinonSi** $couleur[y] = Gris$ **Alors**
19         $p[y] \leftarrow x$;
20   inserer$(y, pile)$;
21   **Fin Si**
22   **Fin Pour**
23   **Fin Si**
24 **Fin Tant que**

Sorties : c, p

Danger

Il est faux de dire que le parcours en profondeur s'obtient en prenant le parcours en largeur en remplaçant la file par une pile.

Il est utile de remarquer certaines différences :

- dans le parcours en largeur on n'insère dans la file que les sommets blancs. Dans le parcours en profondeur on insère dans la pile tous les voisins gris et blancs du sommet exploré.
- cela signifie que les sommets ouverts (gris) peuvent être revisités, et dans ce cas leur prédecesseur est mis à jour et ils sont de nouveau insérés dans la pile
- la pile peut donc contenir plusieurs fois le même sommet

- au moment de l'extraction de la pile il est alors nécessaire d'effectuer une vérification supplémentaire pour ne pas retraiter un sommet fermé (noir).
- malgré ces différences, il est important de comprendre que la complexité du parcours reste linéaire en la taille du graphe $O(n+m)$ car la boucle tant que est exécutée $n+m$ fois au pire.

Voir le fichier de présentation pour un exemple animé de parcours en profondeur à l'aide d'une pile

4 Algorithme de Dijkstra

L'**algorithme de Dijkstra** est une adaption du parcours en largeur dans le cas où les arêtes du graphe sont pondérées.

Définition 3 Un **graphe pondéré** est un triplet $G = (S, A, p)$ où (S, A) est un graphe et $p : A \to \mathbb{R}$ est une fonction qui à chaque arête/arc du graphe associe un poids.

Profitons pour rappeler la définition de coût d'un chemin :

Définition 4 Soit $G = (S, A, p)$ un graphe pondéré et $C = [x_0, \dots, x_k]$ un chemin dans G, on appelle **coût (ou poids) du chemin** la somme des poids des arêtes/arcs qui le composent :

$$\text{coût}(C) = \sum_{i=0}^{k-1} p(x_i x_{i+1})$$

Pour l'algorithme de Dijkstra, on fait l'hypothèse que les poids sont positifs

L'algorithme de Dijkstra explore les sommets par ordre de distance depuis le sommet source. Il est utile de définir ce que l'on appelle distance dans ce cas :

Définition 5 Soit $G = (S, A, p)$ un *graphe pondéré* par des poids positifs, soit deux sommets $x, y \in S$, on appelle **distance** de x à y :

$$\delta(x, y) = \min\{\text{coût}(c) \ / \ c \in \{\text{chemins menant de } x \text{ à } y\}\}$$

On remarquera que si les poids sont bien positifs, il suffit de considérer les chemins sans boucle menant de x à y qui sont en nombre finis, donc la définition est correcte et il existe au moins un chemin de x à y dont le coût est $\delta(x, y)$. On dira d'un tel chemin qu'il est optimal.

Nous pouvons maintenant expliciter l'algorithme de Dijkstra qui, on le rappelle, est une adaptation du parcours en largeur qui tient compte des poids sur les arêtes. Pour cela, on choisit

à chaque itération du parcours, parmi les sommets ouverts, celui qui est atteint avec un coût minimal. On devra donc mémoriser l'information $g(x)$ qui est le coût du chemin actuellement établi depuis le sommet de départ x_d vers le sommet x. On se servira également d'une **file de priorité** pour récupérer facilement le sommet ouvert de valeur $g(x)$ minimale.

Algorithme 4

Entrées : un graphe G sous forme de listes d'adjacence, les poids des arcs qui doivent être positifs, un numéro x_d de sommet de départ

```
1   Initialisation :
2   fileprio ← nouvelle_fileprio(); //c'est une file de priorité min
3   p ← nouveau_tableau(n, −1);
4   c ← nouveau_tableau(n, Blanc);
5   g ← nouveau_tableau(n, +∞);
6   c[x_d] ← Gris;
7   g[x_d] ← 0;
8   inserer((x_d, 0), fileprio);
9   Boucle principale :
10  Tant que non_vide(fileprio) Faire
11      x ← extraire(fileprio);
12      c[x] ← Noir;
13      v ← voisins(x);
14      Pour chaque y ∈ v Faire
15          Si c[y] = Blanc Alors
16              c[y] ← Gris;
17              p[y] ← x;
18              g[y] ← g[x] + p(x, y);
19              inserer((y, g[y]), fileprio);
20          SinonSi c[y] = Gris ∧ g[x] + p(x, y) < g[y] Alors
21              p[y] ← x;
22              g[y] ← g[x] + p(x, y);
23              diminuer((y, g[x] + p(x, y)), fileprio);
24          Fin Si
25      Fin Pour
26  Fin Tant que
```

Sorties : c, p, g

Remarquons que lors de l'exploration d'un sommet x, contrairement au parcours en largeur, on étudie les voisins gris de x : il est en effet nécessaire de l'actualiser ce voisin lorsqu'on trouve un chemin de meilleur coût en passant par x.

Proposition 2 L'algorithme de Dijkstra construit une arborescence de chemins optimaux depuis x_d, c'est-à-dire que pour tout sommet x fermé, on a : $g[x] = \delta(x_d, x)$.

Preuve: on utilise une preuve par invariant

- Étape 0 : il n'y a aucun sommet noir
- Étape 1 : le premier sommet noir est nécessairement x_d, la valeur attribuée est alors $g(x_d) = 0$ et le meilleur chemin pour atteindre x_d est bien le chemin vide qui a pour coût 0
- Hérédité : On suppose que la propriété est vraie avant le début d'une itération de la boucle Tant Que et on veut montrer qu'elle reste vraie à la fin de l'itération. Lors de cette itération, un sommet x est choisi pour devenir noir, les autres sommets noirs sont inchangés. Il suffit donc de montrer que $g(x) = \delta(x_d, x)$. On procède par double inégalité :
  - $\delta(x_d,x) \leq g(x)$: c'est le sens facile, on sait qu'à tout moment de l'algorithme $g(x)$ contient le coût du chemin actuellement construit et menant vers lui, comme $\delta(x_d, x)$ est le coût d'un chemin minimal, on obtient tout de suite l'inégalité.
  - $g(x) \leq \delta(x_d, x)$: pour cela, considérons un chemin optimal C menant de x_d à x et notons y le premier sommet non noir rencontré dans ce chemin, enfin notons z le prédécesseur de y sur le chemin. Ainsi le chemin se décompose en $C = \underbrace{x_d \dots z}_{C_1}.\underbrace{y \dots x}_{C_2}$. On a alors :

$$\begin{aligned}
\delta(x_d, x) &= \text{coût}(C) \\
&= \text{coût}(C_1) + p(z, y) + \text{coût}(C_2) \\
&= \delta(x_d, z) + p(z, y) + \text{coût}(C_2) && \text{car } C \text{ est optimal donc } C_1 \text{ aussi} \\
&\geq \delta(x_d, z) + p(z, y) && \text{car les poids sont positifs} \\
&\geq g(z) + p(z, y) && \text{car } z \text{ est noir et qu'on a supposé l'invariant vrai} \\
&\geq g(y) && \text{par exploration de l'arc zy + opérations diminuer} \\
&\geq g(x) && \text{car } x \text{ est choisi minimal parmi les sommets gris}
\end{aligned}$$

- La preuve par invariant est établie : en sortie de boucle Tant que la propriété sera vérifiée.
- Comme l'arborescence construit des chemins vers x de poids $g(x)$, ces chemins sont optimaux.

La complexité de l'algorithme de Dijkstra dépend de l'implémentation de la file de priorité. On remarque qu'on effectue n fois l'opération d'extraction et que pour chaque arc rencontré on effectue une insertion (ou diminution) :

Implémentation de la file de priorité	extraire	inserer/diminuer	Dijkstra
tableau	$O(n)$	$O(1)$	$O(n^2 + m)$
liste triée	$O(1)$	$O(n)$	$O(mn)$
tas binaire	$O(\log(n))$	$O(\log(n))$	$\mathbf{O((n+m)\log(n))}$

La meilleure complexité est atteinte avec l'utilisation du **tas binaire**.

Voir le fichier de présentation pour un exemple animé de l'algorithme de Dijkstra

5 Algorithme A* : recherche informée

Si on s'intéresse au problème de trouver un chemin, si possible optimal, entre un sommet de départ x_d et un sommet cible x_t on peut utiliser l'algorithme de Dijkstra. Toutefois, l'algorithme de Dijkstra va conduire à l'exploration d'un grand nombre de sommets qui sont éloignés de x_t. Sur des problèmes réels cela peut rapidement devenir prohibitf.

5.1 Vers l'algorithme A*

L'algorithme A^* se propose de construire une arborescence de chemins optimaux en essayant de se rapprocher à chaque étape du sommet x_t visé. Il est donc nécessaire de savoir si on se rapproche ou pas du but. On doit donc disposer d'information supplémentaire sur la distance restante au but, qui prend la forme d'une fonction heuristique :

Définition 6 Une **heuristique** est une fonction $h : S \to \mathbb{R}$ où $h(x)$ représente une *estimation* de la distance au but $\delta(x, x_t)$. On supposera que $h(x_t) = 0$. De plus l'heuristique sera dite :

- **admissible** : si elle ne sur-évalue pas la distance réelle, c'est-à-dire lorsque $\forall x \in S, h(x) \leq \delta(x, x_t)$.
- **monotone** (on dit aussi parfois *cohérente*) : si elle vérifie l'**inégalité triangulaire** suivante

$$\forall (x, y) \in A,\ h(x) \leq h(y) + p(x, y)$$

Proposition 3 Si une heuristique est monotone alors elle est admissible.

[Preuve : somme téléscopique]

L'algorithme que nous allons proposer est une généralisation de l'algorithme de Dijkstra. La différence réside dans le choix des sommets ouverts (gris). Pour chaque sommet ouvert on va calculer une valeur $f(n) = g(n) + h(n)$ = coût du chemin construit pour atteindre n + coût estimé restant à parcourir. On choisit alors le sommet ouvert de $f(n)$ minimum.

Algorithme 5

Entrées : un graphe G sous forme de listes d'adjacence, les poids des arcs, une fonction heuristique h, un numéro x_d de sommet de départ, un numéro x_t de sommet cible

```
1   Initialisation
2   fileprio ← nouvelle_fileprio(); (c'est une file de priorité min)
3   p ← nouveau_tableau(n, −1);
4   c ← nouveau_tableau(n, Blanc);
5   g ← nouveau_tableau(n, +∞);
6   f ← nouveau_tableau(n, +∞);
7   c[x_d] ← Gris;
8   g[x_d] ← 0;
9   f[x_d] ← g[x_d] + h(x_d);
10  inserer((x_d, f[x_d]), fileprio);
11  Boucle principale :
12  Tant que non_vide(fileprio) ∧ c[x_t] ≠ Noir Faire
13      x ← extraire(fileprio);
14      c[x] ← Noir;
15      v ← voisins(x);
16      Pour chaque y ∈ v Faire
17          Si c[y] = Blanc Alors
18              c[y] ← Gris;
19              p[y] ← x;
20              g[y] ← g[x] + p(x, y);
21              f[y] ← g[y] + h(y);
22              inserer((y, f[y]), fileprio);
23          SinonSi c[y] = Gris ∧ g[x] + p(x, y) < g[y] Alors
24              p[y] ← x;
25              g[y] ← g[x] + p(x, y);
26              f[y] ← g[y] + h(y);
27              diminuer((y, f[y]), fileprio);
28          Fin Si
29      Fin Pour
30  Fin Tant que
```

Sorties : c, p, g

Proposition 4 Si l'heuristique est **monotone** alors pour tout sommet fermé (noir) x, $g[x] = \delta(x_d, x)$, c'est-à-dire que l'algorithme construit une une arborescence de chemin optimaux.

Preuve :

- On considère des poids alternatifs $\tilde{p}(x, y) = p(x, y) + h(y) - h(x)$. Comme l'heuristique est monotone, ces poids sont **positifs**.

- On remarque que lorsqu'on met à jour une valeur la valeur $g(y)$ depuis un sommet x on lui affecte :

$$\begin{aligned} g(y) &= g(x) + p(x, y) \\ g(y) &= g(x) + \tilde{p}(x, y) - h(y) + h(x) \\ g(y) + h(y) &= g(x) + h(x) + \tilde{p}(x, y) \\ f(y) &= f(x) + \tilde{p}(x, y) \end{aligned}$$

  Ainsi, l'algorithme proposé qui choisit f minimal effectue les mêmes actions que l'algorithme de Dijkstra mais pour les poids $\tilde{p}$.

- D'après l'optimalité de Dijkstra, les chemins construits sont optimaux au sens des poids $\tilde{p}$.

- Notons $\widetilde{\text{coût}}$ le coût d'un chemin pour les poids $\tilde{p}$, on vérifie facilement par téléscopage que pour tout chemin C menant de a à b, on a:

$$\widetilde{\text{coût}}(C) = \text{coût}(C) + \underbrace{h(b) - h(a)}_{\text{constant pour tous les chemins de } a \text{ à } b}$$

  Ainsi, un chemin de a et b est optimal au sens de $\tilde{p}$ si et seulement s'il est optimal au sens de p.

- Conclusion : les chemins construits sont bien optimaux.

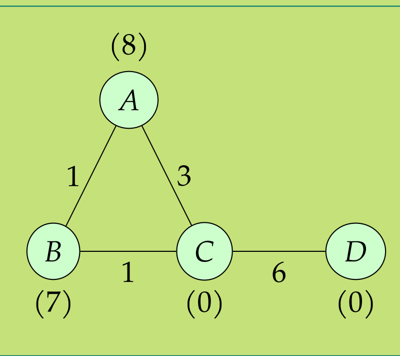
Remarque : on remarque que si $h = 0$ et que les poids sont positifs alors l'algorithme équivaut à celui de Dijkstra, il s'agit donc bien d'une généralisation de l'algorithme de Dijkstra.

Voir le fichier de présentation pour un exemple animé de cet algorithme

5.2 L'algorithme A*

On peut se demander si l'algorithme proposé est toujours valide si l'heuristique est choisie est seulement **admissible** (qui est, on l'a vu, une hypothèse plus faible que monotone). La réponse est non. Pour s'en convaincre on peut regarder le contre-exemple suivant :

Exercice



1. Appliquer l'algorithme 5 sur ce graphe. Les valeurs entre parenthèses sont les valeurs de l'heuristique.
2. L'heuristique proposée est-elle admissible ?
3. A-t-on obtenu une arborescence de chemins optimaux ?

C'est ici que le véritable algorithme A* intervient : pour obtenir réellement des chemins optimaux il faut aussi permetre la réouverture des sommets fermés lorsqu'on détecte une amélioration. Autrement dit l'algorithme A* diffère très peu de ce qui a été présenté auparavant. Le voici :

Algorithme 6

Entrées : un graphe G sous forme de listes d'adjacence, les poids des arcs, une fonction heuristique h, un numéro x_d de sommet de départ, un numéro x_t de sommet cible

```
1   Initialisation
2   fileprio ← nouvelle_fileprio(); (c'est une file de priorité min)
3   p ← nouveau_tableau(n, −1);
4   c ← nouveau_tableau(n, Blanc);
5   g ← nouveau_tableau(n, +∞);
6   f ← nouveau_tableau(n, +∞);
7   c[x_d] ← Gris;
8   g[x_d] ← 0;
9   f[x_d] ← g[x_d] + h(x_d);
10  inserer((x_d, f[x_d]), fileprio);
11  Boucle principale :
12  Tant que non_vide(fileprio) ∧ c[x_t] ≠ Noir Faire
13      x ← extraire(fileprio);
14      c[x] ← Noir;
15      v ← voisins(x);
16      Pour chaque y ∈ v Faire
17          Si c[y] = Blanc Alors
18              c[y] ← Gris;
19              p[y] ← x;
20              g[y] ← g[x] + p(x, y);
21              f[y] ← g[y] + h(y);
22              inserer((y, f[y]), fileprio);
23          SinonSi c[y] = Gris ∧ g[x] + p(x, y) < g[y] Alors
24              p[y] ← x;
25              g[y] ← g[x] + p(x, y);
26              f[y] ← g[y] + h(y);
27              diminuer((y, f[y]), fileprio);
28          SinonSi c[y] = Noir ∧ g[x] + p(x, y) < g[y] Alors
29              c[y] ← Gris;
30              p[y] ← x;
31              g[y] ← g[x] + p(x, y);
32              f[y] ← g[y] + h(y);
33              inserer((y, f[y]), fileprio);
34          Fin Si
35      Fin Pour
36  Fin Tant que
```

Sorties : c, p, g

La seule différence est que lors de l'exploration d'un sommet x, on vérifie même pour les sommets déjà fermés si on les améliore. Quand c'est le cas, le sommet redevient gris et rentre de nouveau dans la file de priorité... Evidemment, cela complexefie nettement la preuve de l'algorithme, à commencer par sa terminaison. On admettra la proposition suivante :

Proposition 5 Si l'heuristique est **admissible** alors l'algorithme A* termine et au moment où le sommet cible devient noir, le chemin obtenu est bien un chemin de coût minimal menant de x_d à x_t.

On peut aussi remarquer qu'il est difficile d'établir une complexité précise pour cet algorithme étant donné que les sommets fermés peuvent être ré-ouverts.

5.3 Bilan

Pour résumer :

- L'algorithme A* sert à trouver un chemin optimal menant de x_d à x_t
- Il utilise une fonction heuristique h pour estimer la distance restante au but
- Il procède comme pour l'algorithme de Dijkstra excepté :
  - Il choisit le sommet gris qui minimise $f(x) = g(x) + h(x)$ (distance trouvée + distance restante estimée)
  - Il réouvre les sommets fermés si nécessaire
- Si l'heuristique est **admissible** alors l'algorithme termine et trouve une solution optimale
- Si l'heuristique est **monotone** alors de plus, il n'est pas nécessaire de réouvrir les sommets fermés, l'algorithme est donc simplifié. De plus, on peut dire dans ce cas que la complexité est la même que pour l'algorioirthme de Dijkstra ($O((n+m)\log(n))$ avec une tas binaire).