Algorithmes de parcours de graphe

Ce document résume l'ensemble des algorithmes de parcours de graphe au programme en MP2I-MPI(*). Ces algorithmes doivent être compris et appris par cœur. Il faut savoir programmer de A à Z un parcours de graphe sans aide.

1 Parcours générique

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

```
sac ← nouveau_sac();
inserer(x_d, sac);
Tant que non_vide(sac) Faire
    x ← extraire(sac);
    Si x n'est pas marqué Alors
        marquer x
        Pour chaque y ∈ voisins(x) Faire
            inserer(y, sac);
        Fin Pour
    Fin Si
Fin Tant que
```

Algorithme 1 Parcours générique

Marquer x correspond au moment où on traite/explore le sommet x. On réalise en général des actions de traitement du sommet à ce moment.

2 Parcours en profondeur (DFS = *depth-first search*)

On utilise une pile LIFO (*last in first out*) pour stocker les sommets découverts. La conséquence est que le dernier sommet qui a été découvert sera le prochain à être exploré.

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

```
pile ← nouvelle_pile();
inserer(x_d, pile);
Tant que non_vide(pile) Faire
    x ← extraire(pile);
    Si x n'est pas marqué Alors
        marquer x
        Pour chaque y ∈ voisins(x) Faire
            inserer(y, pile);
        Fin Pour
    Fin Si
Fin Tant que
```

Algorithme 2 Parcours DFS

En général, on préfère utiliser la récursivité et la pile d'appels récursifs pour gérer la pile. C'est la manière la plus simple de programmer un parcours de graphe :

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

```
FONCTION PARCOURS(x : numero de sommet)
    Si x n'est pas marqué Alors
        marquer x
        Pour chaque y ∈ voisins(x) Faire
            PARCOURS(y);
        Fin Pour
    Fin Si
FIN FONCTION
PARCOURS(x_d);
```

Algorithme 3 Parcours DFS (version récursive)

3 Parcours en largeur (BFS = *breadth-first search*)

On utilise cette fois comme structure de données une file FIFO (*first in first out*) Cela a pour conséquence que les sommets sont explorés dans l'ordre dans lequel ils sont découverts.

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

$file \leftarrow$ nouvelle_file();
inserer(x_d, $file$);
Tant que non_vide($file$) **Faire**
    $x \leftarrow$ extraire($file$);
    Si x n'est pas marqué **Alors**
        marquer x
        Pour chaque $y \in$ voisins(x) **Faire**
            inserer(y, $file$);
        Fin Pour
    Fin Si
Fin Tant que

Algorithme 4 Parcours BFS

On peut remarquer que dans ce parcours basé sur une file, il ne sert à rien de réinsérer un seconde fois un élément dans la file car seule la première occurrence sera traitée. Cela conduit à l'alternative simplifiée suivante dans laquelle on marque les sommets quand on les place dans la file et on n'enfile que les sommets non marqués :

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

$file \leftarrow$ nouvelle_file();
inserer(x_d, $file$);
marquer x_d;
Tant que non_vide($file$) **Faire**
    $x \leftarrow$ extraire($file$);
    Pour chaque $y \in$ voisins(x) **Faire**
        Si y n'est pas marqué **Alors**
            inserer(y, $file$)
            marquer y
        Fin Si
    Fin Pour
Fin Tant que

Algorithme 5 Parcours BFS simplifié

4 Versions avec construction de l'arborescence

Lorsqu'on réalise un parcours, on peut mémoriser pour chaque sommet exploré x, le sommet $pred(x)$ qui a découvert x et conduit à son exploration. L'ensemble des arcs $(x, pred(x))$ forme alors un arbre dont la racine est le sommet de départ x_d et appelé **arborescence du parcours** (les arcs sont orientés des feuilles vers la racine).

L'arborescence du parcours est utile pour reconstruire les chemins qui mènent de x_d aux sommets explorés : il suffit de partir sommet d'arrivée et remonter les arcs jusqu'au sommet de départ.

On donne ici une ré-écriture des trois algorithmes de parcours (parcours générique, parcours DFS et parcours BFS) qui permettent d'obtenir en sortie l'arborescence du parcours.

4.1 Parcours générique

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

$sac \leftarrow$ nouveau_sac();
inserer($(x_d, \varnothing), sac$);
Tant que non_vide(sac) **Faire**
  $(x, p) \leftarrow$ extraire(sac);
  Si x n'est pas marqué **Alors**
    marquer x
    $pred[x] \leftarrow p$
    Pour chaque $y \in$ voisins(x) **Faire**
      inserer($(y, x), sac$);
    Fin Pour
  Fin Si
Fin Tant que

Algorithme 6 Parcours générique avec arborescence

Nous voyons que la différence consiste à stocker dans le sac des couples (sommet decouvert, sommet responsable de la découverte).

4.2 Parcours DFS

On modifie la version récursive en ajoutant un paramètre pour le sommet qui a provoqué l'appel à la fonction PARCOURS :

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

FONCTION PARCOURS(x : numero de sommet, p: sommet d'origine)
  Si x n'est pas marqué **Alors**
    marquer x
    $pred[x] \leftarrow p$
    Pour chaque $y \in$ voisins(x) **Faire**
      PARCOURS(y, x);
    Fin Pour
  Fin Si
FIN FONCTION
PARCOURS($x_d, \varnothing$);

Algorithme 7 Parcours DFS récursif avec arborescence

Remarque : on aurait aussi pu reprendre la version générique avec arborescence et utiliser une pile.

4.3 Parcours BFS

On reprend ici la version simplifiée de l'algorithme. Lorsqu'on insère un sommet dans la file, on le marque et on note immédiatement son prédécesseur dans l'arborescence :

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

$file \leftarrow$ nouvelle_file();
inserer(x_d, $file$);
marquer x_d;
$pred[x_d] \leftarrow \varnothing$;
Tant que non_vide($file$) **Faire**
    $x \leftarrow$ extraire($file$);
    Pour chaque $y \in$ voisins(x) **Faire**
        Si y n'est pas marqué **Alors**
            inserer(y, $file$)
            marquer y
            $pred[y] \leftarrow x$
        Fin Si
    Fin Pour
Fin Tant que

Algorithme 8 Parcours BFS simplifié avec arborescence

Remarque : on aurait aussi pu reprendre la version générique avec arborescence et utiliser une file.

On rappelle qu'une bonne propriété du parcours en largeur (BFS) est que l'arborescence obtenue est une arborescence de chemins de longueur optimale (en nombre d'arcs) de x_d vers chacun des sommets atteints par le parcours.

5 Algorithme de Dijkstra

L'algorithme de Dijkstra reprend le principe du parcours en largeur qui consiste à explorer les sommets par ordre de distance au sommet de départ. La différence est qu'on considère cette fois un graphe pondéré par des poids positifs et que les distances sont calculées par rapport au poids des arêtes et non en nombre d'arêtes.

Nous allons également étoffer le système de marquage avec trois types de sommets :

- Les sommets *blancs* qui n'ont pas été découverts
- Les sommets *gris* aussi appelés *sommets ouverts* qui ont été découverts et sont en attente d'être traités.
- Les sommets *noirs* aussi appelés *sommets fermés* qui ont été traités.

L'algorithme de Dijkstra choisit le prochain sommet à explorer comme étant le sommet gris le plus proche de x_d. On va donc maintenir pour chaque sommet x, une valeur $g[x]$ qui représente le poids du chemin de x_d à x dans l'arborescence en cours de construction.

Entrées : un graphe $G = (S, A)$ sous forme de listes d'adjacence, une fonction $p : A \to \mathbb{R}^+$ de pondération des arêtes (poids positifs), un numéro x_d de sommet de départ

$fileprio \leftarrow$ nouvelle_fileprio(); (c'est une file de priorité min)
$pred \leftarrow$ nouveau_tableau(n, -1);
$c \leftarrow$ nouveau_tableau(n, $Blanc$);
$g \leftarrow$ nouveau_tableau(n, $+\infty$);
$c[x_d] \leftarrow Gris$;
$g[x_d] \leftarrow 0$;
inserer($(x_d, 0)$, $fileprio$);
Tant que non_vide($fileprio$) **Faire**
    $x \leftarrow$ extraire($fileprio$);
    $c[x] \leftarrow Noir$;
    Pour chaque $y \in$ voisins(x) **Faire**
        Si $c[y] = Blanc$ **Alors**
            $c[y] \leftarrow Gris$;
            $pred[y] \leftarrow x$;
            $g[y] \leftarrow g[x] + p(x, y)$;
            inserer($(y, g[y])$, $fileprio$);
        SinonSi $c[y] = Gris \wedge g[x] + p(x, y) < g[y]$ **Alors**
            $pred[y] \leftarrow x$;
            $g[y] \leftarrow g[x] + p(x, y)$;
            diminuer($(y, g[x] + p(x, y))$, $fileprio$);
        Fin Si
    Fin Pour
Fin Tant que
Sorties : c, $pred$, g

Algorithme 9 Algorithme de Dijkstra

Dans cet algorithme lorsqu'on examine les voisins du sommet x traité, on ignore les sommets déjà traités (sommets fermés), on ajoute les sommets non traités (sommets blancs), et on actualise les sommets gris (sommets ouverts) seulement lorsqu'on a amélioré le chemin qui mène à ce sommet.

On rappelle que l'algorithme de Dijkstra construit une arborescence de chemins optimaux depuis x_d.

6 Algorithme A*

Dans l'algorithme A* on se fixe un sommet de départ x_d et un sommet d'arrivée x_t et on cherche un chemin optimal de x_d vers x_t. L'algorithme A* est similaire à l'algorithme de Dijkstra, avec quelques différences importantes :

- On choisit le sommet gris x qui minimise $f[x] = g[x] + h(x)$ c'est-à-dire le poids du chemin construit pour atteindre x plus l'estimation de la longueur du chemin restant pour atteindre le sommet de destination.
- Lorsqu'on examine les voisins du sommet traité x, on considère aussi les sommets fermés (noirs) : si on améliore le chemin alors ce sommet est ré-ouvert (il redevient gris) et doit de nouveau être exploré.
- On s'arrête dès que le sommet de destination x_t devient fermé (noir).

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

$fileprio \leftarrow$ nouvelle_fileprio(); (c'est une file de priorité min)
$pred \leftarrow$ nouveau_tableau$(n, -1)$;
$c \leftarrow$ nouveau_tableau$(n, Blanc)$;
$g \leftarrow$ nouveau_tableau$(n, +\infty)$;
$f \leftarrow$ nouveau_tableau$(n, +\infty)$;
$c[x_d] \leftarrow Gris$;
$g[x_d] \leftarrow 0$;
$f[x_d] \leftarrow g[x_d] + h(x_d)$;
inserer$((x_d, f[x_d]), fileprio)$;
Tant que non_vide$(fileprio) \wedge c[x_t] \neq Noir$ **Faire**
  $x \leftarrow$ extraire$(fileprio)$;
  $c[x] \leftarrow Noir$;
  Pour chaque $y \in$ voisins(x) **Faire**
    Si $c[y] = Blanc$ **Alors**
      $c[y] \leftarrow Gris$;
      $pred[y] \leftarrow x$;
      $g[y] \leftarrow g[x] + p(x, y)$;
      $f[y] \leftarrow g[y] + h(y)$;
      inserer$((y, f[y]), fileprio)$;
    SinonSi $c[y] = Gris \wedge g[x] + p(x, y) < g[y]$ **Alors**
      $pred[y] \leftarrow x$;
      $g[y] \leftarrow g[x] + p(x, y)$;
      $f[y] \leftarrow g[y] + h(y)$;
      diminuer$((y, f[y]), fileprio)$;
    SinonSi $c[y] = Noir \wedge g[x] + p(x, y) < g[y]$ **Alors**
      $c[y] \leftarrow Gris$;
      $pred[y] \leftarrow x$;
      $g[y] \leftarrow g[x] + p(x, y)$;
      $f[y] \leftarrow g[y] + h(y)$;
      inserer$((y, f[y]), fileprio)$;
    Fin Si
  Fin Pour
Fin Tant que
Sorties : $c, pred, g$

Algorithme 10 Algorithme A*

Lorsqu'on sait que l'heuritique est **monotone** il n'est pas nécessaire de vérifier si on améliore un sommet déjà fermé, autrement dit, la ré-ouverture de sommet n'arrive jamais. On peut donc simplifier dans ce cas l'algorithme A* par :

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

$fileprio \leftarrow$ nouvelle_fileprio(); (c'est une file de priorité min)
$pred \leftarrow$ nouveau_tableau$(n, -1)$;
$c \leftarrow$ nouveau_tableau$(n, Blanc)$;
$g \leftarrow$ nouveau_tableau$(n, +\infty)$;
$f \leftarrow$ nouveau_tableau$(n, +\infty)$;
$c[x_d] \leftarrow Gris$;
$g[x_d] \leftarrow 0$;
$f[x_d] \leftarrow g[x_d] + h(x_d)$;
inserer$((x_d, f[x_d]), fileprio)$;
Tant que non_vide$(fileprio) \wedge c[x_t] \neq Noir$ **Faire**
    $x \leftarrow$ extraire$(fileprio)$;
    $c[x] \leftarrow Noir$;
    Pour chaque $y \in$ voisins(x) **Faire**
        Si $c[y] = Blanc$ **Alors**
            $c[y] \leftarrow Gris$;
            $pred[y] \leftarrow x$;
            $g[y] \leftarrow g[x] + p(x, y)$;
            $f[y] \leftarrow g[y] + h(y)$;
            inserer$((y, f[y]), fileprio)$;
        SinonSi $c[y] = Gris \wedge g[x] + p(x, y) < g[y]$ **Alors**
            $pred[y] \leftarrow x$;
            $g[y] \leftarrow g[x] + p(x, y)$;
            $f[y] \leftarrow g[y] + h(y)$;
            diminuer$((y, f[y]), fileprio)$;
        Fin Si
    Fin Pour
Fin Tant que
Sorties : $c, pred, g$

Algorithme 11 Algorithme A* avec une heuristique monotone