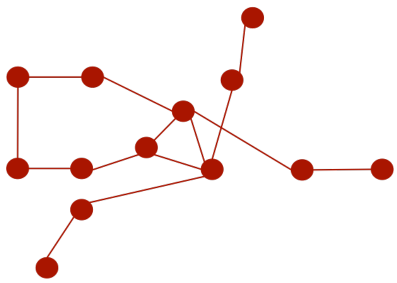
Algorithmes et Structures de Données 2



Pseudo-code pour les algorithmes de traitement de graphes

Version 1.1

14 Novembre 2014

Olivier Cuisenaire

Parcours en profondeur

Entrée : un graphe `G` et un sommet `v` de `G.` Il peut être orienté ou non. On note `e(v,w)` ses arêtes `(v–w)` ou arcs `(v⟶w)`. `G.adjacentEdges(v)` nous donne la liste des arêtes/arcs adjacents à un sommet `v.`

Sortie : tous les sommets atteignable depuis v sont marqués `discovered`.

On peut par ailleurs faire (ou pas) une action spécifique en pré-ordre ou en post-ordre, c.-à-d. en début ou en fin de récursion.

```
procedure DFS(G,v):
  do pre-order action
  label v as discovered
  for all edges e(v,w) in G.adjacentEdges(v) do
     if vertex w is not labeled as discovered then
            recursively call DFS(G,w)
        end if
     end for
  do post-order action
```

Source : http://en.wikipedia.org/wiki/Depth-first_search ,
modifié avec actions en pré et post-ordre

Parcours en profondeur itératif

Entrée : un graphe G et un sommet u de G. Il peut être orienté ou non. On note e(v,w) ses arêtes (v-w) ou arcs (v⟶w). G.adjacentEdges(v) nous donne la liste des arêtes/arcs adjacents à un sommet v.

Sortie : tous les sommets atteignable depuis v sont marqués discovered.

On peut par ailleurs faire (ou pas) une action spécifique en pré-ordre ou en post-ordre, comme pour la mise en œuvre récursive. Alors que dans la version récursive, on avait deux états possibles (no label/discovered), on en a ici trois (no label/pre-discovered/discovered), et on renvoie les sommets pre-discovered dans la pile pour être traités une seconde fois.

On utilise une pile LIFO (stack) qui défini les opérations push(v) et pop(). Cette dernière est équivalent à top() suivi de pop() en c++.

```
procedure DFS-iterative(G,u):
  let S be a stack
  S.push(u)
  while S is not empty do
    v ← S.pop()
    if v is not labeled then
      do pre-order action
      label v as pre-discovered
      S.push(v)
      for all edge e(v,w) in G.adjacentEdges(v) do
        if w is not labeled then
          S.push(w)
        end if
      end for
    else if v is labeled as pre-discovered then
      do post-order action
      label v as discovered
    end if
  end while
```

Source : http://en.wikipedia.org/wiki/Depth-first_search ,
modifié avec actions en pré et post-ordre

Parcours en largeur

Entrée : un graphe `G` et un sommet u de `G`. Il peut être orienté ou non. On note `e(v,w)` ses arêtes `(v–w)` ou arcs `(v⟶w)`. `G.adjacentEdges(v)` nous donne la liste des arêtes/arcs adjacents à un sommet `v`.

Sortie : tous les sommets atteignable depuis v sont marqués `discovered`.

On peut par ailleurs faire une action spécifique en ordre BFS.

On utilise une file FIFO (queue) qui défini les opérations `enqueue(v)` et `dequeue()`.

```
procedure BFS(G,u) :
  let Q be a queue
  Q.enqueue(u)
  mark u as discovered
  while Q is not empty do
    v ← Q.dequeue()
    do BFS action
    for all edges e(v,w) in G.adjacentEdges(t) do
      if w is not discovered then
        Q.enqueue(w)
        mark w as discovered
      end if
    end for
  end while
```

Source : http://en.wikipedia.org/wiki/Breadth-first_search ,
Modifié pour être plus cohérent avec les parcours en profondeur ci-dessus

Composantes connexes

Entrée : un graphe G non orienté

Sortie : tous les sommets sont marqués avec le numéro de la composante connexe à laquelle ils appartiennent. n contient le nombre de composantes connexes.

```
procedure CC(G) :
  n ← 0
  for all vertices v in G do
    if v is not labeled then
      call DFS(G,v), label all discovered
      vertices with value n
      n ← n+1
    end if
  end for
```

Cette procédure est optimale quand le graphe est donné. Si on veut connaître les composantes connexes d'un graphe en cours de construction dans lequel on ajoute des arêtes au fur et à mesure, il convient d'utiliser la structure de donnée Union-Find (disjoint sets)

Tri topologique

Le tri topologique est simplement le tri des sommets d'un graphe orienté **acyclique** selon l'ordre inverse du post-ordre DFS.

Entrée : un graphe G orienté acyclique

Sortie : une pile *S* dont on peut extraire les numéros des sommets en ordre topologique.

```
procedure reversePostOrder(G) :
  let S be a stack
  for all vertices v in G do
    if v is not labeled then
      call DFS(G,v) and do post-order action
      on all discovered vertices w
        S.push(w)
        label w as discovered
      end BFS
    end if
  end for
  return S
```

Composantes fortement connexes

Entrée : un graphe G orienté

Sortie : tous les sommets sont marqués avec le numéro de la composante fortement connexe à laquelle ils appartiennent. n contient le nombre de composantes connexes

```
procedure SCC(G) :
  GR ← G dont tous les arcs sont inversés
  S ← reversePostOrder(GR)
  n ← 0
  while S is not empty do
    v ← S.pop()
    if v is not labeled then
      call DFS(G,v), label all discovered
      vertices with value n
      n ← n+1
    end if
  end while
```

Algorithme de Kruskal (Arbre couvrant de poids minimum)

Entrée : un graphe G pondéré non orienté

Sortie : l'ensemble A des arêtes formant l'arbre couvrant de poids minimum

Cet algorithme utilise la structure Union-Find avec les fonctions MAKE-SET, FIND-SET et UNION. Voir http://en.wikipedia.org/wiki/Disjoint-set_data_structure

```
procedure Kruskal(G) :
  A ← ∅
  for all vertices v in G do
    MAKE-SET(v)
    for all edges (v-w) ordered by increasing
    weight do
      if FIND-SET(v) ≠ FIND-SET(w) then
        A ← A ∪ {(v-w)}
        UNION(v,w)
      end if
    end for
  end for
  return A
```

Source : http://en.wikipedia.org/wiki/Kruskal%27s_algorithm , 14 novembre 2014

Algorithme de Prim (Arbre couvrant de poids minimum)

Entrée : un graphe G pondéré non orienté

Sortie : l'ensemble A des arêtes formant l'arbre couvrant de poids minimum

L'algorithme suppose l'existence d'une queue de priorité supportant les fonctions

- `add_with_priority(index, priority)`
- `contains(index)`
- `index = extract_min()`
- `decrease_priority(index, priority)`

```
procedure Prim(G):
  A ← ∅
  let Q be a priority queue
  for all edges e:0-w in G.adjacentEdges(0) do
    Q.add_with_priority(w,length(e))
    edge[w] ← e
  end for
  while Q is not empty do
    v ← Q.extract_min()
    mark v as scanned
    A ← A ∪ {edge[v]}
    for all edges e:v-w in G.adjacentEdges(v) do
      if w is not yet scanned then
        if Q.contains(w) and e < edge[w] then
          Q.decrease_priority(w,length(e))
          edge[w] ← e
        else
          Q.add_with_priority(w,length(e))
          edge[w] ← e
        end if
      end if
    end for
  end while
return A
```

Algorithme de Dijkstra (Arbre des plus courts chemins)

Entrée : un graphe orienté pondéré G. un sommet u de ce graphe

Sortie : le tableau `dist` qui stocke la longueur des plus courts chemins et le tableau `previous` qui contient le sommet précédent dans le plus court chemin.

Le code `v ← vertex in Q with min dist[v]` cherche le sommet *v* dans l'ensemble de sommets *Q* qui a la plus petite valeur de `dist[v]`. `length(e)` donne le poids de l'arc e. La variable `alt` à la ligne 15 est le poids du chemin depuis le sommet source jusqu'à w si celui-ci passe par l'arc v→ w. Si ce chemin est plus court que le plus court chemin actuellement enregistré, on remplace ce dernier.

Cette version de l'algorithme utilise une simple liste pour stocker les sommets à traiter, ce qui n'est vraiment pas optimal.

```
1.      procedure Dijkstra(G,u):
2.        let Q be a list
3.        dist[u] ← 0

4.        for all vertices v in G do
5.          if v ≠ u then
6.            dist[v] ← ∞
7.            previous[v] ← undefined
8.          end if
9.          add v to Q
10.       end for

11.       while Q is not empty do
12.          v ← vertex in Q with min dist[v]
13.          remove v from Q
14.          for all edges e:v→w in G.adjacentEdges(v) do
15.            alt ← dist[v] + length(e)
16.            if alt < dist[w] then
17.              dist[w] ← alt
18.              previous[w]  ← v
19.            end if
20.          end for
21.       end while
22.       return dist[], previous[]
```

Source : http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm , 14 novembre 2014

Algorithme de Dijkstra avec queue de priorité

Entrées et sorties identiques au précédent.

Cet algorithme nécessite une queue de priorité mettant en œuvre les opérations suivantes: `add_with_priority()`, `decrease_priority()` et `extract_min()`. Mis en œuvre avec un tas et des tableaux auxiliaires (pour `decrease_priority()`), la complexité de ces opérations est de $O(\log_2 n)$ pour n éléments.

```
procedure Dijkstra(G,u) :
  let Q be a priority queue
  dist[u] ← 0
  for all vertices v in G do
    if v ≠ u then
      dist[v] ← ∞
      previous[v] ← undefined
    end if
    Q.add_with_priority(v,dist[v])
  end for

  while Q is not empty do
    v ← Q.extract_min()
    mark v as scanned
    for all edges e:v→w in G.adjacentEdges(v) do
      if w is not yet scanned then
        alt ← dist[v] + length(e)
        if alt < dist[w] then
          dist[w] ← alt
          previous[w] ← v
          Q.decrease_priority(w,alt)
        end if
      end if
    end for
  end while
  return dist[], previous[]
```

Source : http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm , 14 novembre 2014

Algorithme de Bellman-Ford (Arbre des plus courts chemins)

Entrées et sorties identiques au précédent. Le graphe G a `G.V()` sommets

L'algorithme passe V fois sur tous les arcs du graphe. On peut faire mieux en stockant dans une queue la liste des sommets ayant été modifié à l'itération i et en ne traitant que ceux-ci à l'itération i+1.

```
procedure Bellman-Ford(G,u):
  dist[u] ← 0
  for all vertices v in G do
    if v ≠ u then
      dist[v] ← ∞
      previous[v] ← undefined
    end if
  end for

  for i from 0 to G.V() do
    for all edges e:v→w in G do
      alt ← dist[v] + length(e)
      if alt < dist[w] then
        dist[w] ← alt
        previous[w] ← v
      end if
    end for
  end for
  return dist[], previous[]
```