Algo 2 - Synthèse

Edgardo Cuellar-Sanchez
Attilio Discepoli

Mai 2021

Contents

1 Algorithme de recherche dichotomique

C'est un algorithme qui permet de trouver la position d'un élément dans un tableau trié. Il est utilisé dans pas mal d'algorithmes.
Principe: Il regarde l'élément du milieu du tableau et selon s'il est plus grand ou plus petit, il recherche dans la partie droite ou gauche du tableau, reprenant l'élément du milieu de ce sous-tableau, et ainsi de suite.
Complexité: $O(log(n))$ car il recherche chaque fois dans le sous-groupe.

```
int dicho(int[] a, int elem, int low, int high){
        if (low == high) return low;
        int m = ( low + h ) / 2;        // Prend l'élement au milieu du tableau
        if (a[m] < elem) return dicho(a, elem, m+1, high);
        else if (a[m] > elem) return dicho(a, elem, low, m-1);
        else return m;
}
```

2 Structure union-find

Structure de données permettant de maintenir une partition en classes, avec une complexité linéaire au pire cas. (Utilisé dans l'algorithme de Kruskal)
2 opérations

- `int find(int p)`
  Détermine la classe d'équivalence de l'élément. Elle est utilisée pour savoir si deux éléments appartiennent à la même classe d'équivalence.
  Complexité: $O(n)$, linéaire

  ```
  int find(int p) {
   while (p != parent[p]) p = parent[p];
   return p;
  }
  ```

- `void union(int p, int q)`
  Réuni deux classe d'équivalences en une seule. Elle utilise deux `find()`
  Complexité: $O(n)$, linéaire.

  ```
  void union(int p, int q) {
   int rootP = find(p);
   int rootQ = find(q);
   if (rootP == rootQ) return;
   parent[rootP] = rootQ;
   count--; // On décrémente le nombre de classes de l'ensemble
  }
  ```

2.1 Union rapide pondérée

Au lieu de remplacer la racine par p, on change q en la racine, on attache toujours l'arbre de hauteur minimum à la racine de celui de hauteur maximum. Tout cela en conservant la hauteur des arbres de chaque classe dans un tableau (cf. Programmation Dynamique hehe) La hauteur maximum de l'arbre via union rapide pondérée ne dépassera jamais $log(n)$.
Complexité: en $O(logn)$, les deux find se font en $O(logn)$ car la hauteur maximum via union rapide pondéré est de $O(logn)$, on a donc une structure de complexité $O(logn)$, ce qui permet une nette optimisation.

```
void union(int p, int q){
 int rootP = find(p);
 int rootQ = find(q);
```

```
 if (rootP == rootQ) return;

 // Choisis l'arbre de hauteur minimum
 if (height[rootP] < height[rootQ]) parent[rootP] = rootQ;
 else if (height[rootP] > height[rootQ]) parent[rootQ] = rootP;
 else {
        parent[rootQ] = rootP;
        height[rootP]++;
 }
 count--; // On décrémente le nombre de classes de l'ensemble
}
```

2.2 Compression de chemin

Lors de l'opération `find()`, après avoir identifié la racine de l'arbre, chaque pointeur que nous avons rencontré sur le chemin, sera remplacé par un pointeur vers la racine. Tous les chemins sont alors compressés.
Complexité: $O(n)$, linéaire (je sais pas trop). Mais la complexité de la strucutre sera de $O(m\alpha(m,n))$ et donc presque en temps constant ($\alpha(m,n)$ est la fonction d'Ackermann inverse qui croît extrêmement lentement (SASAGEYO))

```
int find(int p){
 int root = p;
 while (root != parent[root]) root = parent[root]; // On retrouve la racine depuis p
 while (p != root) {
         int new_p = parent[p];
         parent[p] = root; // On associe chaque p à la racine
         p = new_p;
 }
 return root;
}
```

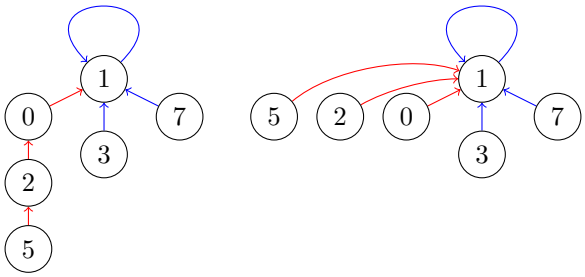

Figure 1: Compression de chemin

Playlist de vidéos: `https://www.youtube.com/playlist?list=PLDV1Zeh2NRsBI1C-mR6ZhHTyfoEJWlxvq`

3 Tris

3.1 MergeSort - Tri fusion

Algorithme de tri par comparaisons (seules opérations élémentaires permises en plus des déplacements). Il se fait en deux étapes:

- Trier récursivement les $n/2$ premiers éléments

- Trier récursivement les $n/2$ derniers suivants
- Fusionner les deux tableaux triès

Complexité: $O(nlog(n))$ (moyenne et pire cas)
Analyse: $D(n) = 2D(n/2) + n$

```
// Tri
void sort(Comparable[] a, Comparable[] aux, int lo, int hi){
        if (hi <= lo) return;
        int mid = lo + (hi - lo) / 2;
        sort(a, aux, lo, mid);
        sort(a, aux, mid+1, hi);
        merge(a, aux, lo, mid, hi);
}

// Fusion
void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi){
        for(int k = lo; k <= hi; k++) aux[k] = a[k];
        int i = lo, j = mid+1;
        for (int k = lo; k <= hi; k++){
                if (i > mid) a[k] = aux[j++];
                else if (j > hi) a[k] = aux[i++];
                else if (less(aux[j], aux[i])) aux[i] = aux[j++];
                else a[k] = aux[i++];
        }
}
```

3.1.1 Version bottom-top

Même chose mais sans les appels récursifs

- Fusionner les paires des sous-tableaux successif de taille 1
- Puis de taille 2, 4, ...
- À la i^{eme} étape, on partitionne donc le tableau en $n/2^{(i-1)}$ sous-tableaux qu'on fusionne 2 à 2.

```
// Version bottom-up du tri fusion
void sort(Comparable[] a){
        int n = a.length;
        Comparable[] aux = new Comparable[n];
        for (int sz = 1; sz < n; sz = sz + sz)
                for (int lo = 0; lo < n-sz; lo += sz+sz)
                        merge(a, aux, lo, lo+sz-1, Math.min(lo+sz+sz-1, n-1));
}
```

Bien que différents physiquement, ils sont identiques sur la façon de procéder.

3.2 QuickSort - Tri rapide

Algorithme de tri par comparaisons (seules les opérations élémentaires permises en plus des déplacements). Très performant et similaire au tri fusion (2 appels récursifs et une étape linéaire), il diffère de celui-ci car l'étape linéaire à lieu avant les appels récursifs. Il utilise un **pivot**.
Principe: On place un élément du tableau à sa place définitive, en permutant tous les éléments plus petits à gauche et tout ceux plus grands à droite.

- Le pivot est plaçé à la fin, en l'échangeant avec le dernier élément du sous-tableau.

- Tous les éléments inférieurs au pivot sont placés en début du sous-tableau
- Le pivot est déplacé à la fin des éléments déplacés

Complexité: $O(nlog(n))$ (moyenne) et $O(n^2)$ (pire cas)

```
// Partition
int partition(Comparable[] a, int lo, int hi){
        int i = lo, j = hi + 1;
        while (true){
                while (less(a[++i], a[lo])) {
                        if (i == hi) break;
                }
                while (less(a[lo], a[--j])) {
                        if (j == lo) break;
                }
                if (i >= j) break;
                exch(a, i, j);
        }
        exch(a, lo, j);
        return j;
}

// QuickSort
void sort(Comparable[] a, int lo, int hi){
        if (hi <= lo) return;
        int j = partition(a, lo, hi);
        sort(a, lo, j-1);
        sort(a, j+1, hi);
}
```

4 Heap - Tas

Structure de données d'arbre binaire, dans laquelle chaque noeud (élément) à une valeur de clé superieure (ou inférieur dans le cas d'un min heap) à celle de ses enfants.

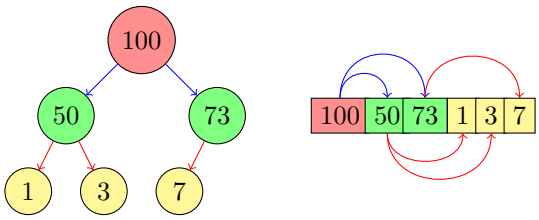

Figure 2: Heap

Complexité:

- On peut récupérer le maximum (ou le minimum) en temps constant (il suffit de prendre la racine de l'arbre)
- L'insertion se fait en $O(log(n))$, il suffit d'arriver à une clé de valeur inférieure et l'insérer.
- La suppresion du maximum (ou minimum) se fait également en $O(log(n))$

- La suppression et la recherche sont de complexité $O(n)$
- La construction se fait en $O(nlog(n))$ si l'on doit insérer n éléments.

```
// Permet de faire remonter un élément dans le heap
void swim(int k) {
        while (k > 1 && less(k/2, k)) exch(k, k/2);
}

// Permet de faire descendre un élément dans le heap
void sink(int k) {
        while (2*k <= N) {
                int j = 2*k;
                if (j < N && less(j, j+1)) j++;
                if (!less(k, j)) break;
                exch(k, j);
                k = j;
        }
}
```

À l'aide de ces deux méthodes nous pouvons implémenter efficacement les procédures d'insertion et de suppression du maximum.

```
void insert(Key x){
        pq[++n] = x;
        swim(n);
}

void delMax(){
        Key max = pq[1]; // On récupère le maximum
        exch(1, n--);         // On échange ça place avec le dernier élément du heap
        sink(1);         // On fait redescendre cet élément jusqu'à une position correcte
        pq[n+1] = null; // On supprime le dernier élément (le maximum donc)
        return max; // On retourne ce maximum
}
```

4.1 Construction en temps linéaire

Il est possible de créer un tas en temps linéaire ($O(n)$) avec un tableau trié, en utilisant les appelant la méthode `sink()` sur toutes les clés du tableau en allant de droite à gauche.

```
for (int k = n/2; k >= 1; k--) sink(a, k, n);
```

4.2 HeapSort - Tris par tas

Méthode de tri par comparaisons qui crée un tas à partir des données dans la liste à trier, et qui appelle n fois la méthode de suppression du maximum.

- On construit le heap avec les valeurs de départ
- La valeur maximum est alors en haut du heap, on le remplace avec le dernier élément (suppression du Maximum)
- On continue jusqu'à ce que toute la liste sois triée.

Complexité: $O(nlog(n))$ en moyenne et $O(2nlog(n))$ au pire cas. La construction et l'opération de suppression (`sink()`) se font toutes deux en $O(nlog(n))$

```
void sort(Comparable[] a){
        int = a.length;
        // Construction du tas en parcourant le tableau de clés de droite à gauche
        for (int k = n/2; k >= 1; k --)
                sink(a, k, n);
        while (n > 1){
                exch(a, 1, n);
                sink(a, 1, --n);
        }
}
```

5 Arbres de recherche

5.1 Arbres binaires de recherche

Structure de données d'arbre binaire (chaque noeuds a au maximum 2 enfants) dans laquelle chaque noeuds possède une clé supérieure à son membre de gauche et inférieur à son membre de droite (En gros, un heap amélioré)
Complexité: Si l'arbre est balancé (Fig. 7), la recherche, l'insertion et la suppression se feront en $0(log(n))$, sinon la complexité dans le pire cas sera de $O(n)$ (Fig. 4).

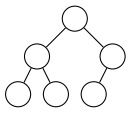

Figure 3: Arbre balancé

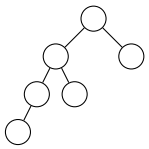

Figure 4: Arbre non-balancé

Recherche: On part de la racine, si l'élément que l'on recherche a une valeur plus grande que celle-ci, on part vers la gauche, sinon on part vers la droite. Et ainsi de suite jusqu'à trouver l'élément.
Insertion: On effectue une recherche sur base de l'élément à ajouter, lorsque l'on arrive sur une feuille, on ajoute l'élément à droite ou à gauche de celle-ci selon s'il est plus grand ou plus petit.
Sur cette base, l'on sait que la construction d'un arbre binaire de recherche peut se faire en $O(nlog(n))$.
Suppression: On effectue une recherche jusqu'au noeud que l'on veut supprimer, 3 scénarios sont alors possibles:

- Si l'élément est une feuille, on le supprime simplement
- Si l'élément est un noeud avec un enfant, on remplace l'élément son fils.
- Si l'élément est un noeud avec deux enfants, on recherche la feuille la plus à droite du fils gauche de notre élément, on le inverse, puis on supprime l'élément.

5.2 Arbre 2-3

Arbre de recherche dont chaque noeud pointe vers un, deux ou trois sous-arbres.
Un **2-noeud** a la même structure qu'un noeud dans un ABR (Arbre binaire de recherche)
Un **3-noeud** contient 5 attributs:

- 2 clés (2 valeurs)
- Un pointeur vers le sous-arbre inférieur à la 1ère clé
- Un pointeur vers le sous-arbre superieur à la 2ème clé

- Un pointeur vers le sous-arbre dont les valeurs sont entre la 1ère et la 2e clé

On demande que toutes les feuilles soient à la même profondeur.
Complexité: L'insertion et la recherche se font en $O(log(n))$

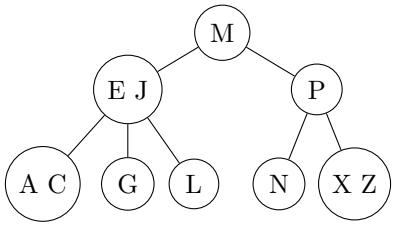

Figure 5: Arbre 2-3

Recherche: Se fait comme dans un ABR, avec une valeur en plus à comparer dans un 3-noeud. **Insertion**: Initialement, on parcourt l'arbre pour trouver un endroit om insérer la clé, on identifie le noeud dont la profondeur est maximale (feuille) dans laquelle on peut insérer la nouvelle clé:

- Si il s'agit d'un 2-noeud, on le transforme en 3-noeud.
- Si il s'agit d'un 3-noeud, on le transforme en 2-noeud et on envoie la clé médiane vers le haut, on réitère cette opération au niveau précédent. Si la racine est un 3-noeud, on la transforme elle aussi en 2-noeud et la hauteur de l'arbre augmente de 1.

5.3 Arbres rouges-noirs - Red-Black Tree

ABR équilibrés qui implémentent éfficacement les arbres 2-3. Il s'agit d'une bijection d'un arbre 2-3.
Même structure qu'un arbre binaire de recherche classique. On encode les 2-noeuds et les 3-noeuds en utilisant des arêtes de différentes couleurs. Avantage sur les BST: A une meilleure complexité au pire cas pour la recherche, l'insertion et la suppression car ils sont équilibrés.

- Deux noeuds dans un arbre rouge sont connectés par une arête rouge s'ils appartiennent à un même 3-noeud dans l'arbre 2-3 correspondant.
- Dans le cas contraire, ils sont connectés par une arête noire

Propriétés:

1. Les arêtes sont soit rouges, soit noires
2. La racine est les feuilles sont noires
3. Si un noeud est lié par une arête rouge alors les arêtes le liant à ses fils sont noires
4. Tous les chemins d'un noeud vers ses feuilles descendantes contiennent le même nombre d'arêtes noires
5. Seuls les fils gauches peuvent être rouge

Convention: Une arête rouge connecte toujours un noeud à son fils gauche. La hauteur d'un arbre rouge-noir est au plus $2log(n)$

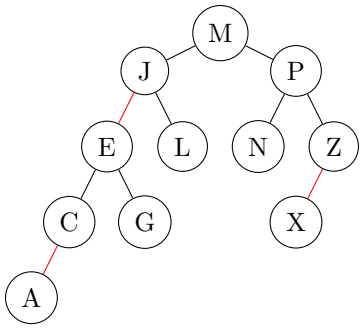

Figure 6: Arbre Rouge-Noir de l'arbre 2-3 de la fig. 5

5.3.1 Rotations

Pour effectuer les opératoins d'insertion et de suppression il faut utiliser des **rotations** qui permettent de maintenir l'équilibre de l'arbre en baissant la hauteur de l'arbre sans changer l'ordre des éléments (et tout ça en $O(1)$, mais que demande le peuple ?)

```
Node rotateLeft(Node h) {
        Node x = h.right;
        h.right = x.left;
        x.left = h;
        return x;
}
```

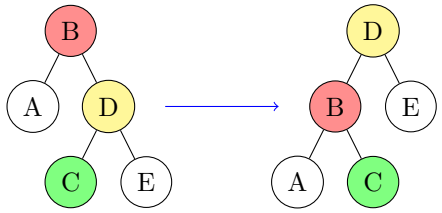

Figure 7: Rotation gauche de l'élément B

Recherche: Identique à celle dans un ABR ($O(log(n))$)
Insertion: On insère le nouveau noeud au fond de l'arbre avec une arête rouge et l'on effectue des opérations supplémentaires pour maintenir les propritétés de l'arbre (avec des rotation et recolorations)

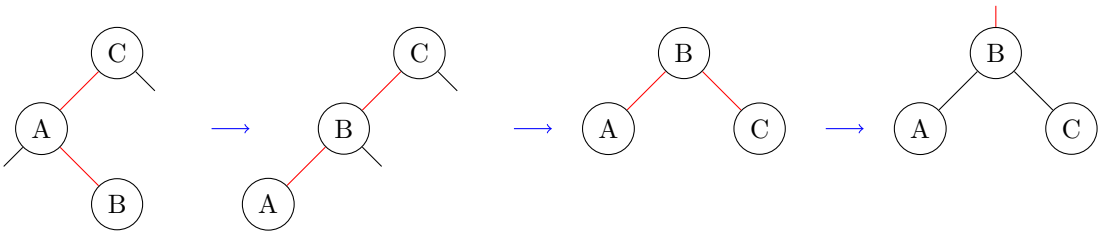

Figure 8: Insertion de l'élément et opérations de ré-équilibrage

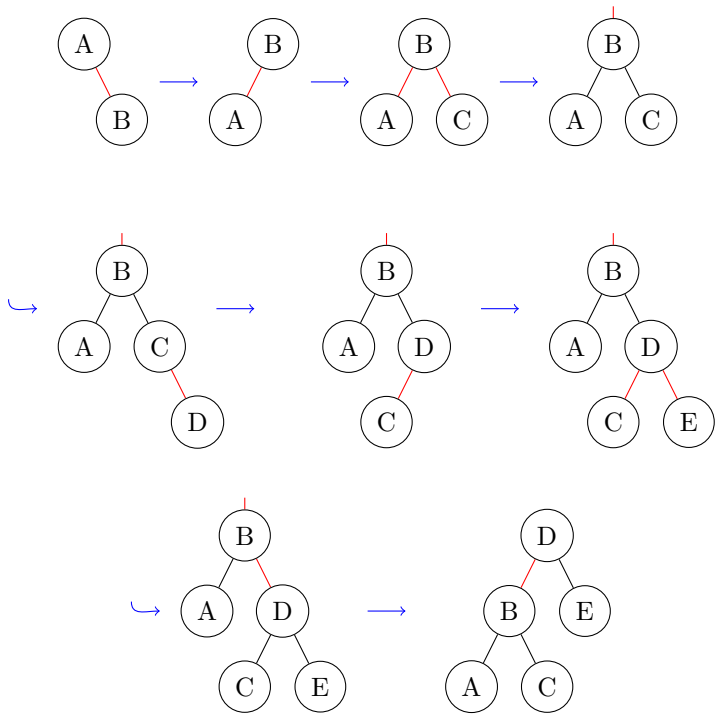

Figure 9: Insertion des éléments B, C, D et E dans l'arbre A (avec les opérations de ré-équilibrage)

```
Node insert(Node h, Key key, Value val){
        if (h == null) return new Node(key, val, RED);
        if      (key < h.key)  h.left = insert(h.left, key, val);
        else if (key > h.key)  h.right = insert(h.right, key, val);
        else if (key == h.key) h.val = val;

        if (isRed(h.right) && !isRed(h.left))     h = rotateLeft(h);
        if (isRed(h.left)  && isRed(h.left.left)) h = rotateRight(h);
        if (isRed(h.right) && isRed(h.left))      flipColors(h);
}
```

Après l'insertion il faut s'assurer que la racine reste noire:

```
root = insert(root, key, value):
root.color = BLACK;
```

Suppression: On retire le noeud voulu et on ré-équilibre l'arbre.
Complexité: L'insertion et la suppression prenent un temps $O(log(n))$ au pire cas et la rotation un temps $O(1)$
`https://courses.cs.washington.edu/courses/cse373/21wi/dna-indexing/left-leaning-red-black-trees/#llrb-tree-demos`

6 Graphes

6.1 Types de graphe

Un graphe est une paire composée d'un ensemble de sommets V et d'arêtes E.

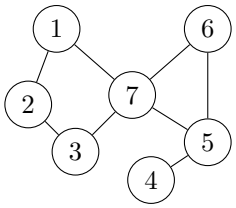

Figure 10: Un graphe simple ($V = 7$, $E = 8$)

Un chemin est une suite de sommets distincts adjacents deux à deux qui relient deux sommets u et v.
Un cycle est un chemin sont le premier et le dernier sommet sont également adjacent (une boucle en gros)
Un graphe dirigé est un graphe dont les arêtes ont un sens d'un sommet vers un autre, dans ce type de graphe il se peut donc qu'un chemin $u - v$ soit possible mais pas un chemin $v - u$
Un graphe dirigé acyclique (ou DAG) est un graphe dont les arêtes ont un sens et où il n'existe pas de cycles.

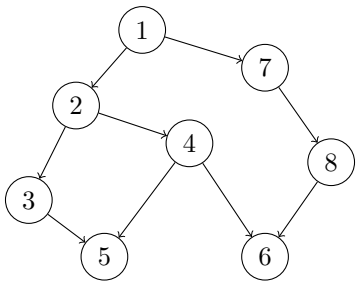

Figure 11: Un graphe dirigé acyclique

Graphe Eulérien: Graphe qui contient un cycle eulérien, càd un chemin qui passe une seule fois par chaque arête (un dessin sans lever la main)
Graphe Hamiltonien: Graphe qui contient un cycle hamiltonien, càd un chemin reliant tous les sommets du graphe sans passer deux fois par un même sommet. (Fait partie des problèmes NP-Complet)

6.2 Parcours de graphe

6.2.1 Parcours en profondeur

Marque tous les sommets accessibles à un sommet de départ en un temps proportionnel à la somme de leur degrés ($O(V + E)$)

```
void dfs(Graph G, int v) {
 marked[v] = true;
 preorder.enqueu(v); // Récupérer le préordre
 for (int w: G.adj(v)) {
  if (!marked[w]) {
   dfs(G, w);
   edgeTo[w] = v;
  }
 }
 postorder.enqueu(v); // Récupérer le postordre
}
```

L'ordre dans lequel la procédure de parcours en profondeur (`dfs`) est appelé le **préordre**. Dans le graphe de la fig. 10, ce préordre vaut {1, 2, 3, 7, 6, 5, 4}

L'ordre dans lequel la procédure de récursion d'un parcours en profondeur se termine est appelé le **postordre**. Il est utilisé dans l'algorithme de Kosaraju-Sharir (7.1), et aussi pour trouver le tri topologique d'un DAG. Dans le graphe de la fig. 11, ce postordre vaut {5, 3, 6, 4, 2, 8, 7, 1}

6.2.2 Parcours en largeur

L'algorithme de parcours en largeur calcule un plus court chemin entre le sommet de départ et tous les sommets accessibles, en $O(E + V)$ au pire cas.
Étapes:

1. Mettre le noeud source dans une file de priorité
2. Retirer le noeud du début de la file pour le traiter
3. Mettre tous ses voisins non explorés dans la file (à la fin)
4. Si la file n'est pas vide reprendre à l'étape 2

```
void bfs(Graph G, int s){
        Queue<Integer> q = new Queue<Integer>();
         q.enqueue(s);
        marked[s] = true;
        distTo[s] = 0;
         while (!q.isEmpty()){
                  int v = q.dequeue();
                  for (int w : G.adj(v)) {
                           if (!marked[w]) {
                            q.enqueue(w);
                            marked[w] = true;
                            edgeTo[w] = v;
                            distTo[w] = distTo[v] + 1;
                           }
                  }
         }
}
```

6.2.3 Tri topologique

Ne concerne que les graphes dirigés acycliques (DAG).
Principe: Ordre total sur l'ensemble des sommets, dans lequel s précède t pour tout arc d'un sommet s à un sommet t. En gros, chaque sommet doit apparaître avant ses successeurs. Aussi il n'y a pas d'unicité de l'ordre.
Il suffit de faire un parcours en profondeur du graphe (6.2.1) et récupérer son postordre.
Algorithmes basés sur le tri topologique: La recherche de plus court chemin dans un graphe acyclique, avec l'algo de Dijkstra.

7 Composantes fortement connexes

Il existe une composante fortement connexe s'il existe un chemin dirigé de u vers v et également de v vers u. La relation est réflexive, symétrique et transitive (relation d'équivalence). Les classes d'équivalences de cette relation sont le composantes fortement connexes.
`https://www.youtube.com/watch?v=ohObUJ9Q6wQ`

7.1 Algorithme de Kosaraju-Sharir

Pour obtenir les composantes fortement connexes d'un graphe, on peut utiliser l'algorithme de Kosaraju-Sharir (qui est assez épatant)
Principe:

1. Faire un `dfs` du graphe G et récupère son postordre inverse (6.2.1)
2. Prendre la transposée du graphe G noté G^T, il suffit d'inverser le sens des arcs.
3. Effectuer un `dfs` de G^T en choisissant les sommets non marqués dans l'ordre de la liste du postordre inverse de G
4. Magie ! Chaque arbre est une composante fortement connexe !

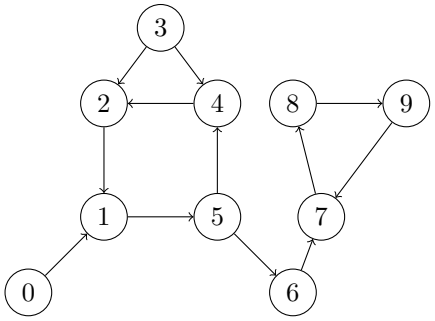

Figure 12: Graphe G

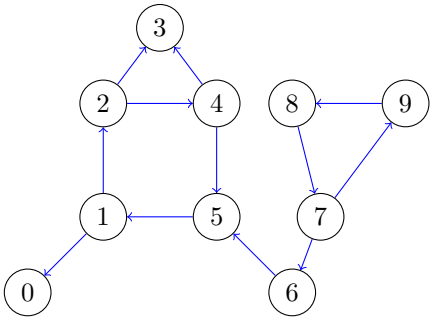

Figure 13: Graphe G^T (transposée de G)

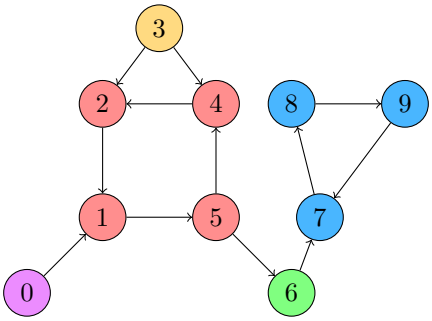

Figure 14: Composantes fortement connexes d'un graphe (les sommets appartenant à une même classe d'équivalence partagent la même couleur)

Postordre de G {9, 8, 7, 6, 2, 4, 5, 1, 0, 3}
Postordre inverse de G {3, 0, 1, 5, 4, 2, 6, 7, 8, 9}
Si l'on effectue un `dfs` de G en partant des sommets indiqués dans le postordre inverse, l'on obtient 5 arbres: {3}, {0}, {1, 2, 4, 5}, {6} et {7, 8, 9} qui sont bien les différents composantes connexes illustrées sur le graphe de la fig. 14 !

```
void KosarajuSharir(DiGraph G) {
        DepthFirstOrder G_inv = new DepthFirstOrder(G.reverse());
        for (int v : G_inv.reversePostorder())
                if (!marked[v]) {
                        dfs(G, v);
                        count++;
```

```
                }
}

void dfs(DiGraph G, int v) {
        marked[v] = true;
        id[v] = count;
        for (int w : G.adj(v))
                if (!marked[w]) dfs(G, w);
}
```

Complexité: linéaire, $O(V + E)$ car seulement 2 procédure de parcours en profondeur.
`https://www.youtube.com/watch?v=m2mdGfxs_5E`
`https://youtu.be/Rs6DXyWpWrI` (thx Gregory)

8 Arbres couvrants

Un arbre couvrant (spanning tree) est un sous-graphe d'un graphe G contenant tous les sommets de G (le sous-graphe est connexe et acyclique). Un arbre couvrant dans un graphe à V sommets contient $V-1$ arêtes.

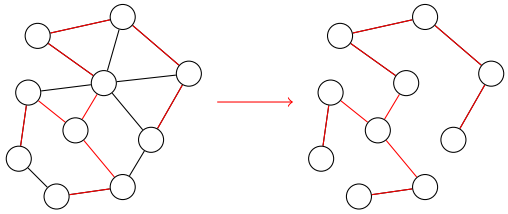

Figure 15: Arbre couvrant (en rouge) d'un graphe G

8.1 Coupes

Un coupe est une arête qui a ses deux extremités dans deux ensembles différents.
Propriétés: Dans un graphe pondéré sur les arêtes, où toutes les arêtes ont un poids (ω) positif distinct, pour toute coupe C de G, l'arête de poids minimum de C fait partie de l'arbre couvrant de poids minimum de G.

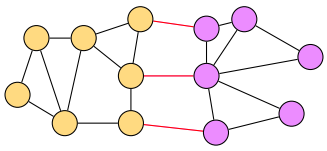

Figure 16: Coupe d'un graphe G (en rouge) séparant le graphe en deux ensembles distincts

8.2 Algorithme de Kruskal

Cet algorithme permet de trouver un arbre couvrant de poids minimum dont le poids des arêtes est minimum.
Principe:

1. On trie toutes les arêtes du graphe selon leur poids
2. Chaque arête est choisie si elle ne forme pas de cycle avec les arêtes déjà choisies
3. Lorsque toutes les arêtes ont été éxaminées, on a notre arbre couvrant de poids minimum !

Pour l'implémenter, on peut utiliser une file de priorité (pour trouver l'arête suivante minimum dans la liste) et une structure union-find sur les sommets du graphe (pour tester si une arête est un cycle)

```
while (!pq.isEmpty() && mst.size() < G.V() - 1) {
        Edge e = pq.delMin();
        int v = e.either(), w = e.other(v);
        if (!uf.connected(v, w)) {
                uf.union(v, w);
                mst.enqueue(e);
        }
}
```

Complexité: $O(Elog(E))$ car utilisation d'un algorithme de tri sur les arêtes. Les opérations d'union-find se fond en temps presque constant.
https://www.youtube.com/watch?v=U_itW96NnQ0

8.3 Algorithme de Prim

Comme Kruskal, il cherche à trouver l'arbre couvrant de poids minimum. C'est également un algorithme glouton.
Principe:

1. Initialiser l'arbre actuel T à un seul sommet initial 0.
2. Ajouter T à l'arête de poids minimum parmis toutes les arêtes ayant exactement une extremité dans T.
3. Répéter l'opération précédente jusqu'à ce que T contienne $E - 1$ arêtes.

On peut utiliser une file de priorité pour contenir l'ensemble des arêtes ayant au moins une extremité dans T. La clé associé à une arête e dans la file est son poids $\omega(e)$. Si une arête extraite de la file) deux extremités dans T, on l'ignore et on passe à la suivante.
Lorsqu'on ajoute une arête dans T, on ajoute dans la file toutes les arêtes incidentes à celle-ci.

```
while (!pq.isEmpty() && mst.size() < G.V() - 1){
        Edge e = pq.delMin();
        int v = e.either(), w = e.other(v);
        if (marked[v] && marked[w]) continue;
        mst.enqueue(e);
        if (!marked[v]) visit(G, v); // visit() màj la file de priorité
        if (!marked[w]) visit(G, w);

void visit(EdgeWeightedGraph G, int v){
        marked[v] = true;
        for (Edge e: G.adj(v)) {
                int w = e.other(v);
                if (marked[w]) continue;
                // L'arête E est la meilleure connection de l'arbre vers w
                if (e.weight() < distTo[w]){
                        edgeTo[w] = e;
                        distTo[w] = e.weight();
                        if (pq.contains(w)) pq.change(w, distTo[w]):
                        esle                                pq.insert(w, distTo[w]);
                }
        }
}
```

Complexité: $O(Elog(E))$

Pour améliorer l'algo, au lieu de maintenir les arêtes et leur poids dans la file de priorité on peut maintenir les sommets; la clé associée sera l'arête de poids minimum connectant v à T. L'algorithme devient alors

1. Trouver le sommet v de clé minimum dans la file de priorité et son arête associée
2. Ajouter l'arête à T
3. Mettre à jour la file de priorité:
   - Si x est déjà dans T, on ignore
   - Ajouter x dans la file de priorité si il n'y est pas déjà
   - Décroitre la clé de x, si l'arête $v - x$ devient l'arête de poids minimum connectant x à T.

```
void PrimMST(EdgeWeightedGraph G){
        for (int v = 0; v < G.V(); v++)
                distTo[v] = Double.POSITIVE_INFINITY;
        distTo[0] = 0;
        pq.insert(0, 0); // Initialise la file de priorité avec la source de distance 0
        while (!pq.isEmpty()) visit(G, pq.delMin()); // Ajoute le sommet le plus proche à l'arbre
}
```

Complexité: grâce à l'amélioration: $O(Elog(V))$

https://www.youtube.com/watch?v=IOuiQyAs5G4

9 Plus court chemin

Algorithme fondamentaux pour tout ce qui est réseaux et GPS. Il trouve le plus court chemin entre deux sommets d'un graphe pondéré sur les arêtes.

Conditions d'optimalités: Soit $G(V, E)$ un graphe dirigé pondéré sur les arêtes, $s \in V$ un sommet particulier et une fonction $d : V \to \mathbf{R}$. La valeur de $d(v)$ est la longueur du plus court chemin de s vers v si et seulement si:

1. $d(s) = 0$
2. $\forall_v \in V, d(v)$ est la longueur d'un chemin $s - v$
3. Pour tout arc $e = (u, v) \to d(v) \leqslant d(u) + \omega(e)$

N'importe quel sommet du graphe peut être choisi comme source.

9.1 Algorithme de Dijkstra

Algorithme qui utilise une file de priorité qui correspondra à la distance entre le sommet s (source) et un autre sommet du graphe.
L'on considére les sommets dans l'ordre de leur distance au sommet source s. Á chaque étape, un nouveau sommet v est traité: celui pour lequel la plus petite distance connue est minimale. On effectue ensuite l'opération de **relaxation** sur tous les arcs incident à v. Puisque le sommet choisi est toujours celui dont la distance est minimale, elle ne pourra jamais être améliorée par une relaxation. On obtient donc la distance définitive.

1. On crée une liste avec tous les sommets du graphe, qu'on initialisent tous avec une distance infinie, sauf le sommet source qui prend une distance de 0.
2. L'opération de relaxation consiste à regarder nos voisins, on remplace la valeur infinie de leur distance par le poids de l'arête, et on marque notre sommet courant comme "relaché".

3. On prend le sommet ayant la distance la plus petite, on le marque et on effectue une nouvelle étape de relaxation, les sommets voisins prennent comme distance la somme de la distance du sommet courant et du poids de l'arête entre celui-ci et son voisin.
4. Et ainsi de suite, si l'on repasse sur un sommet qui a déjà un poids mais qu'une relaxation lui donne un nouveau poids plus petit, on lui attribue cette valeur plus petite

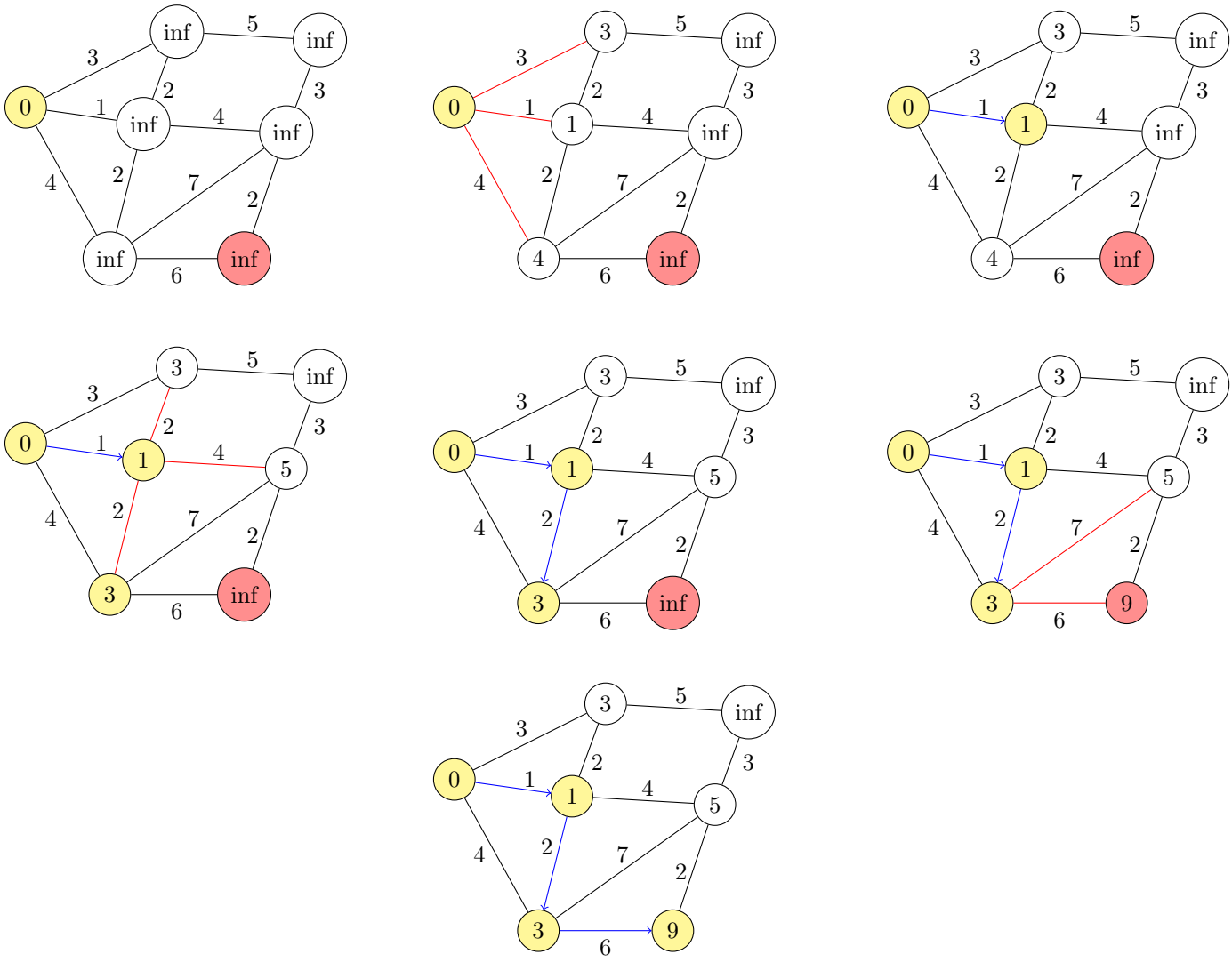

Figure 17: Début de l'exécution de l'algorithme de Dijkstra, le sommet source est en jaune et le sommet destination est en rouge. La distance de chaque sommet à la source est inscrite dessus.

```
for (int v = 0; v < G.V(); v++) distTo[v] = Double.POSITIVE_INFINITY;
distTo[s] = 0;
pq.insert(s, distTo[s]);
while (!pq.isEmpty()) {
        int v = pq.delMin();
        for (DirectedEdge e: G.adj(v)) relax(e);
}

void relax(DirectedEdge e){
        int v = e.from(), w = e.to();
```

```
        if (distTo[w] > distTo[v] + e.weight()) {
                distTo[w] = distTo[v] + e.weight(); // Màj des distances
                edgeTo[w] = e;
                // Modification des priorité dans la file pq
                if   (pq.contains(w)) pq.decrease(w, distTo[w]);
                else pq.insert(w, distTo[w]);
        }
}
```

Complexité: $O(Elog(V))$, on fait le tour de toutes les arêtes du graphe et on effectue la méthode de relaxation à chaque fois (pour tous les sommets dans le pire cas $\Rightarrow O(log(v))$ grâce à la file de priorité).
`https://www.youtube.com/watch?v=JPeCmKFrKio`

9.2 Dans un DAG

Grâce à l'absence de cycles, l'on peut éviter d'utiliser une structure de file de priorité. On peut trouver les plus courts chemins d'une source vers tous les autres sommets en les considérant dans un ordre topologique (6.2.3) ensuite on pourra effectuer les étapes de relaxation sur les arêtes en temps constant.

```
void relax(EdgeWeightedDiGraph G, int v) {
        for (DirectedEdge e : G.adj(v))
                relax(e);
}
```

Complexité: $O(V + E)$ il s'agit de faire un bête `dfs` pour obtenir l'ordre topologique et ensuite d'effectuer une étape de relaxation sur les arêtes

10 Programmation dynamique

Méthode de programmation, pour simplifier/diminuer la complexité d'un problème. Cela permet d'optimiser les algorithmes avec des relations de récurrence.
Principe: On résout un problème en décomposant en sous problème, et en résolvant les plus petits problèmes un à un, en stockant les résultats intermédiaires.
Et donc en gros, pour mettre en œuvre la programmation dynamique, il faut une **relation de récurrence**, et **retenir les solutions** des sous problèmes, pour les utiliser plus loin.
Beaucoup d'algorithmes utilisent cette technique pour réduire leur complexité en temps
Exemples:

- Plus longue sous-séquence commune
- Plus longue sous-séquence commune avec le plus court chemin
- Produits de matrices
- Algorithme pseudo-polynomiaux (sac à dos)

`FIN`
Merci aux étudiants d'Info2024 pour leurs corrections !
Lien vers la page Github du repo: `https://github.com/adiscepo/INFO-F-203-Synthese/`