NOM : BERNARD   Prénom : Sophie   Jury :

~~Algèbre~~ ← Entourez l'épreuve → ~~Analyse~~ Informatique

Sujet choisi : 925 - Graphes ; représentations et algorithmes

Ref : Cormen
Beauquier

Autre sujet :

Motivations : modéliser différents systèmes : cartes, réseaux, dépendances, ..

I. DEFINITIONS ET REPRESENTATIONS

1. Définitions

Def 1 : Un **graphe (simple) orienté** est un couple (V, E) où V est un ensemble fini de **sommets** et $E \subset V^2$ un ensemble d'**arêtes**.

Rq 2 : Un graphe orienté peut contenir des boucles $(u,u) \in E$ et des arêtes en double sens : $(u,v) \in E$ et $(v,u) \in E$.

Def 3 : Un **graphe (simple) non orienté** est un couple (V,E) avec $E \subset V^2$ et $\forall u \in E, (u,u) \notin E$ et $\forall (u,v) \in E, (v,u) \in E$.

Ex 4 : $G_1 = (\{1,..,5\}, \{(1,2),(2,3),(3,3),(3,4)\})$ est un graphe orienté représenté en annexe.
$G_2 = (\{1,..,5\}, \{(1,2),(2,3),(3,4),(2,1),(3,2),(4,3)\})$ est un graphe non orienté représenté en annexe.

Rq 5 : En général, pour un graphe non orienté, on considère que dans E, les paires de sommets ne sont pas ordonnées, ce qui évite d'écrire deux fois une arête.

Rq 6 : On peut représenter un graphe en mémoire comme le couple de listes V et E, ce qui assure un espace minimal, mais aucune opération simple (trouver un sommet adjacent, tester l'existence d'une arête) n'est en $O(1)$.

2. Liste d'adjacence

Def 7 : Représentation sous la forme d'un tableau Adj de taille $|V|$ où $\forall u \in V$, $Adj(u)$ est la liste chaînée des $v \in V$, tels que $(u,v) \in E$.

Ex 8 : La liste d'adjacence du graphe G_1 est en annexe.

Prop 9 : Taille en mémoire : $O(|V|+|E|)$.
Test d'existence d'un sommet adjacent en $O(1)$.

Rq 10 : La taille en mémoire est quasiment minimale, donc très utile pour les graphes peu denses. Peu pratique pour tester l'existence d'une arête entre deux sommets donnés.

3. Matrice d'adjacence

Def 11 : Après numérotation quelconque des sommets de 1 à $|V|$, représentation sous la forme d'une matrice $A = (a_{i,j})_{1 \leq i \leq |V|, 1 \leq j \leq |V|}$ avec $a_{i,j} = 1$ si $(i,j) \in E$ et $a_{i,j} = 0$ sinon.

Ex 12 : Matrice d'adjacence du graphe G_1 :
$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Prop 13 : Taille en mémoire : $O(|V|^2)$.
Test d'existence d'une arête de sommets donnés : $O(1)$.

Rq 14 : Utile pour les graphes denses ($|E| \approx |V|^2$) car on ne perd presque plus de place en mémoire par rapport aux listes d'adjacence, ou pour les petits graphes. Peu pratique pour trouver un sommet adjacent.

II. PARCOURS DE GRAPHES

1. Parcours en largeur

Principe 15 : Pour $G = (V,E)$ et $s \in V$, on parcourt G, à partir de s, en explorant tous les sommets directement accessibles depuis s (distance 1), puis ceux à une distance 2.

Algo 16 : Entrée : $G = (V,E)$ et $s \in V$.
Sortie : liste l des sommets rencontrés, en ordre inverse.

```
PL(G,s):
  pour tout u ∈ V\{s},                  } Initialisation des
      couleur(u) = blanc                }  sommets de V\{s}
      d(u) = +∞
  couleur(s) = gris                     } Initialisation de s
  d(s) = 0
  l = [s].                              } Initialisation des
  F = ∅                                 }  variables
  Enfiler(F,s)
  Tant que F ≠ ∅
      u = Défiler(F)
      pour tout v ∈ Adj(u)              } Découverte des     } Etude du
          si couleur(v) = blanc         }  sommets           }  sommet
              couleur(v) = gris         }  adjacents à       }  u de la
              d(v) = d(u)+1             }  u                 }  file
              Ajouter(l,v)
              Enfiler(F,v)
      couleur(u) = noir
```

Ex 17 : Déroulement de l'algorithme sur un exemple en annexe.

Complexité 18 : avec liste d'adjacence : $O(|V|+|E|)$
avec matrice d'adjacence : $O(|V|^2)$.

Appl 19 : Arbre couvrant minimal, calcul de plus court chemin, ...

2. Parcours en profondeur

Principe 20 : Pour $G = (V,E)$ et $s \in V$, on parcourt G à partir de s, en suivant des sommets adjacents les uns aux autres jusqu'à ne plus pouvoir le faire ou rencontrer un sommet déjà vu. On remonte alors jusqu'à trouver un autre sommet à partir duquel on recommence le procédé.

Algo 21 : Entrée : $G=(V,E)$.
Sortie : tableaux d et f des dates de début et fin de visite de chaque sommet

```
PP(G):
  pour tout u ∈ V                          } Initialisation
    L couleur(u) = blanc
  date = 0

  pour tout u ∈ V                          } Etude de tous les
    si couleur(u) = blanc                    sommets blancs
      L PP-visiter(G,u)

PP-Visiter(G,u):
  couleur(u) = gris                        } Etude
  date = date + 1                            du
  d(u) = date                                sommet
  pour tout v ∈ Adj(u)                       u
    si couleur(v) = blanc                  } Etude des sommets
      L PP-visiter(G,v)                      adjacents à u
  couleur(u) = noir
  date = date + 1
  f(u) = date
```

Ex 22 : Déroulement de l'algorithme sur un exemple en annexe

Complexité 23 : avec liste d'adjacence : $O(|V|+|E|)$
avec matrice d'adjacence : $O(|V|^2)$

Appl 24 : Composantes fortement connexes, tri topologique, ...

DEV 1 : Tri topologique : modéliser, algorithme, correction, complexité et exemple

III_ PLUS COURT CHEMIN

Def 25 : Un graphe $G=(V,E)$ orienté est dit **pondéré** si on lui associe une fonction de poids $\omega : E \to \mathbb{R}$. Le **poids d'un chemin** est alors la somme des poids de ses arêtes.

1. Source unique

Pb 26 : Pour $G=(V,E)$ un graphe pondéré par ω, avec $s\in V$, on cherche à trouver pour tout $u\in V$, un chemin $s=s_0\to s_1\to \dots \to s_m = u$ (s'il existe) tel que $\sum_{i=0}^{m-1} \omega(s_i,s_{i+1})$ soit minimal.

Algo 27 (Dijkstra) : Il se décompose en 2 parties : Dijkstra qui fait la phase de calcul et CheminD qui renvoie le chemin de poids minimal pour une certaine destination.
Entrée : $G=(V,E)$ graphe pondéré par ω avec $\forall (u,v)\in E,\ \omega(u,v)\geq 0$ et $s\in V$.
Sortie : π, tableau des prédécesseurs à suivre pour retrouver les chemins et d tableau des distances minimales à s.

```
Dijkstra(G,ω,s):
  pour tout u ∈ V                          } Initialisation
    d(u) = +∞                                des sommets
    π(u) = NIL
  d(s) = 0                                 } Initialisation de s
  S = ∅                                    } Initialisation
  F = V                                      des variables
  tant que F ≠ ∅                           } Etude du sommet u
    u = extraire_min(F)
    S = S ∪ {u}
    pour tout v ∈ Adj(u)                   } Mise à jour
      si d(v) > d(u) + ω(u,v)                des distances
        d(v) = d(u) + ω(u,v)                 des sommets
        π(v) = u                             adjacents à u
```

Avec extraire_min(F) qui extrait un sommet u de F pour lequel d est minimal.
Entrée : π, tableau des prédécesseurs renvoyé par Dijkstra, s sommet source et t sommet destination.
Sortie : liste l des sommets d'un chemin de poids minimal de s à t.

```
CheminD(π,s,t):   (uniquement si d(t) ≠ +∞)
  l = []
  u = t
  tant que u ≠ s
    Ajouter(l,u)
    u = π(u)
  Ajouter(l,u)
```

Complexité 28 : avec les listes d'adjacence : $O(|V|^2+|E|)=O(|V|^2)$
Si le graphe est peu dense, on peut améliorer cette complexité.

Rq 29 : L'algorithme de Dijkstra peut être étendu à des poids négatifs mais uniquement si le graphe ne contient pas de cycle de poids négatif.

DEV 2 : Algorithme de Bellman-Ford : algorithme, correction, complexité et exemple

2_ Pour toutes les sources

Pb 30 : Pour $G=(V,E)$ un graphe pondéré par ω, on cherche à trouver pour tout couple (u,v) de sommets un chemin de u à v de poids minimal s'il existe.

Notation 31 : On numérote les sommets de 1 à $|V|$. On note W la matrice de poids : $W=(w_{ij})_{1\leq i\leq |V|,\,1\leq j\leq |V|}$ avec $w_{ij}=\omega(i,j)$ si $(i,j)\in E$ et $w_{ij}=+\infty$ sinon.

Principe 32 : Par programmation dynamique, on calcule $D^{(i)}$ la matrice des poids des chemins ne passant que par les sommets 1 à i.

Algo 33 (Floyd-Warshall) : Il se décompose en 2 parties : FW qui fait la phase de pré-traitement et CheminFW qui renvoie un chemin de poids minimal entre deux sommets donnés.
Entrée : $G=(V,E)$ un graphe pondéré par ω sans cycle de poids négatif et W la matrice de poids de ω.
Sortie : $D^{(|V|)}$ la matrice des poids minimals des chemins entre tout couple de sommets et $\Pi^{(|V|)}$ la matrice des prédécesseurs sur les chemins de poids minimals.

```
FW(G,W):
  D^(0) = W                                } Initialisation
  pour i = 1 à |V|                           des matrices
    pour j = 1 à |V|
      Si D_ij^(0) = +∞
        alors Π_ij^(0) = 0
        sinon Π_ij^(0) = D_ij^(0)
  pour k = 1 à |V|
    pour i = 1 à |V|
      pour j = 1 à |V|
        Si D_ij^(k-1) ≤ D_ik^(k-1) + D_kj^(k-1)
          Alors D_ij^(k) = D_ij^(k-1)                    } Chemin sans k
                Π_ij^(k) = Π_ij^(k-1)
          Sinon D_ij^(k) = D_ik^(k-1) + D_kj^(k-1)       } Chemin passant par k
                Π_ij^(k) = Π_kj^(k-1)
```

Entrées : $(s,t)\in V^2$, le couple de sommets entre lesquels on cherche un chemin de poids minimal, et $\Pi^{(|V|)}$
Sortie : liste l des sommets d'un chemin de poids minimal de s à t.

```
CheminFW(Π^(|V|),s,t):   (uniquement si D_st^(|V|) ≠ +∞)
  l = []
  u = t
  tant que u ≠ s
    Ajouter(l,u)
    u = Π_{s,u}^(|V|)
  Ajouter(l,u)
```

Complexité 34 : $O(|V|^3)$

Appl 35 : En appliquant l'algorithme de Floyd-Warshall à un graphe $G=(V,E)$ pour lequel les poids ω valent 1 pour chaque arête, on calcule sa clôture transitive (mêmes sommets, arête entre $(u,v)\in V^2$ s'il existe un chemin de u à v dans G). En effet, si $D_{ij}^{(|V|)}=+\infty$ alors $(i,j)\notin E^*$ et sinon $(i,j)\in E^*$ et $G^*=(V,E^*)$ est la clôture transitive de G, calculée en $O(|V|^3)$ temps.

IV. COMPOSANTES FORTEMENT CONNEXES ET ARBRE COUVRANT MINIMAL

Def 36: Un graphe $G=(V,E)$ est **connexe** si $\forall (u,v)\in V^2$, il existe un chemin de u à v. Il est **fortement connexe** si $\forall (u,v)\in V^2$, il existe un chemin de u à v et de v à u.

Rq 37: Les deux notions coïncident pour un graphe non orienté.

Def 38: Pour $G=(V,E)$ un graphe non orienté connexe, un **arbre couvrant** A est un sous-ensemble de E tel que (V,A) est acyclique et $\forall u\in V, \exists v\in V, (u,v)\in A$. Il est dit **minimal** si G est pondéré par w et pour tout arbre couvrant B, $\sum_{e\in A} w(e) = w(A)\leq w(B)$. $w(A)$ est son **poids**.

1. Composantes fortement connexes

Def 39: Pour $G=(V,E)$ un graphe, le **graphe transposé** ${}^tG$ est $(V,{}^tE)$ avec ${}^tE=\{(v,u), (u,v)\in E\}$.

Rq 40: On le calcule en temps $O(|V|+|E|)$ avec des listes d'adjacence et $O(|V|^2)$ par des matrices d'adjacence.

Algo 41 (Kosaraju): Entrée : $G=(V,E)$ un graphe orienté
Sortie : liste l de listes des sommets des composantes fortement connexes de G.

```
Kosaraju(G):
| P = PP(G)
| tG = Transpose(G)
L PP_modif(tG, P)
```

Entrée : $G=(V,E)$ un graphe orienté et P une liste des sommets de G.
Sortie : liste l de listes des sommets de G.

```
PP_modif(G, P):
| pour tout u ∈ V                      } Initialisation
|    couleur(u) = blanc                }
| l = [] _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ }
| tant que P ≠ ∅                       } Calcul de la
| | u = tête(P)                        } composante
| | cc = []                            } connexe dans
| | PP-modif-visiter(G, u)             } laquelle u est.
L L Ajouter(l, cc)                     }

PP_modif_Visiter(G, u):
| couleur(u) = gris                    } parcours en
| pour tout v ∈ Adj(u)                 } profondeur
| | si couleur(v) = blanc              } classique
| L L PP-modif-Visiter(G, v)           }
| couleur(u) = noir _ _ _ _ _ _ _ _ _ _}
| ajouter(cc, u)                       } Mise à jour de la
L supprimer u de P                     } composante connexe
                                       } et de la liste à étudier
```

Complexité 42: avec listes d'adjacence : $O(|V|+|E|)$
avec matrice d'adjacence : $O(|V|^2)$

Ex 43: Déroulement de l'algorithme sur un exemple en annexe

2. Algorithme de Kruskal

Principe 44: L'algorithme maintient d'un côté un ensemble A d'arêtes et de l'autre les composantes connexes de (V,A), pour un graphe $G=(V,E)$ non orienté. Créer_ensemble initialise les composantes connexes avec A vide, Trouver_ensemble(u) renvoie un représentant de la composante connexe de u et Union fusionne deux composantes connexes. L'algorithme commence donc par une forêt sans arêtes, et ajoute une arête de plus petit poids possible qui n'ajoute pas de cycle dans A.

Algo 45: Entrée : $G=(V,E)$ graphe orienté pondéré par w.
Sortie : A un arbre couvrant minimal de G.

```
Kruskal(G, w):
| A = ∅
| pour tout u ∈ V                                   } Initialisation
| L Créer_ensemble(u)                               }
| trier les arêtes de E par ordre croissant de w _ _}
| pour tout (u,v) ∈ E
| | si Trouver_ensemble(u) ≠ Trouver_ensemble(v)    } Traitement de
| | | A = A ∪ {(u,v)}                               } chaque arête
L L L Union(u, v) _ _ _ _                           }
```

Ex 46: Déroulement de l'algorithme sur un exemple en annexe.

Complexité 47. avec les listes d'adjacence : $O(|E|\log|E|)$
$= O(|E|\log|V|)$.

3. Algorithme de Prim

Principe 48: Pour $G=(V,E)$ un graphe non orienté, pondéré par w, l'algorithme ajoute, à partir d'un sommet $s\in V$, l'arête de plus petit poids à A, un arbre ancré en s. On utilise pour cela une file de priorité (minimale).

Algo 49: Entrée : $G=(V,E)$ graphe orienté pondéré par w, $s\in V$.
Sortie : π, tableau des prédécesseurs permettant de reconstruire $A=\{(u,\pi(u)), u\in V\setminus\{s\}\}$.

```
Prim(G, w, s):
| pour tout u ∈ V
| | prio(u) = +∞
| L π(u) = NIL
| prio(s) = 0
| Q = V
| tant que Q ≠ ∅
| | u = Extraire_min(Q)
| | pour tout v ∈ Adj(u)
| | | si v ∈ Q et w(u,v) < prio(v)
| | | | π(v) = u
L L L L prio(v) = w(u,v)
```

Ex 50: Déroulement de l'algorithme sur un exemple en annexe

Complexité 51: avec des matrices d'adjacence : $O(|V|^2)$
avec des listes d'adjacence et tas binaire : $O(|E|\log|V|)$.

V. FLOT MAXIMUM

Def 52: Un **réseau de transport** est un graphe $G=(V,E)$, orienté pondéré par $c : E\to\mathbb{R}_+$. Si $(u,v)\notin E$, on note $c(u,v)=0$, ce qui étend c à V^2. Un **flot** sur un réseau de transport $G=(V,E)$ avec c est une fonction $f : V^2\to\mathbb{R}$ entre une source $s\in V$ et un puits $t\in V$, avec :
(i) contrainte de capacité : $\forall (u,v)\in V^2$, $f(u,v)\leq c(u,v)$.
(ii) symétrie : $\forall (u,v)\in V^2$, $f(u,v)=-f(v,u)$
(iii) conservation du flot : $\forall u\in V\setminus\{s,t\}$, $\sum_{v\in V} f(u,v)=0$.
La **valeur** d'un flot f est $|f|=\sum_{v\in V} f(s,v)$.

Pb 53: Etant donné un réseau $G=(V,E)$ de capacité c, et $(s,t)\in V^2$, $s\neq t$, une source et un puits, on cherche un flot dont la valeur sera la plus grande possible.

Rq 54: On peut aussi considérer plusieurs sources et puits. Il est alors facile de le modéliser par un réseau avec une seule source et un seul puits.

Ex 55: En annexe, un exemple de réseau à plusieurs sources/puits modélisé par un réseau avec source et puits uniques.

Def 56: Pour un réseau $G=(V,E)$ de capacité c, et f un flux sur celui-ci avec $(s,t)\in V^2$ la source et le puits, la **capacité résiduelle** de $(u,v)\in V^2$ est $c_f(u,v)=c(u,v)-f(u,v)$. Le **réseau résiduel** de G, induit par f est $G_f=(V,E_f)$ avec $E_f=\{(u,v)\in V^2,\ c_f(u,v)>0\}$.

DEV 3: **Algorithme de Ford-Fulkerson** : méthode, algorithme de Edmonds-Karp, exemple

Appl 57: couplage maximum dans un graphe biparti,

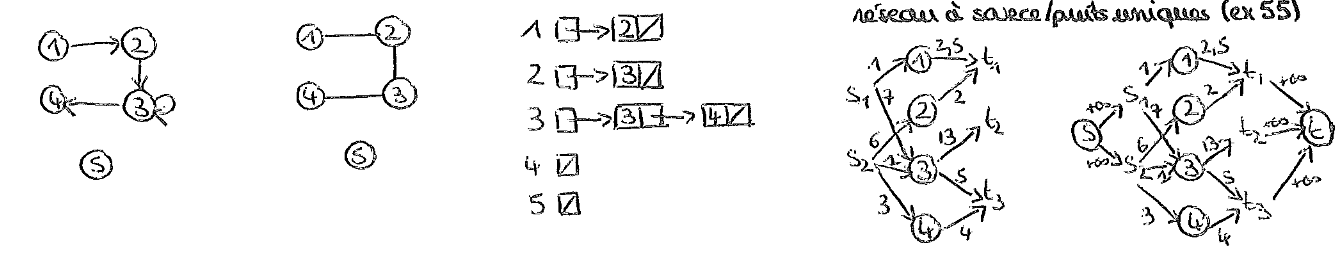
Def 58: Une **coupe** d'un graphe $G=(V,E)$ est une partition de V. Si G est pondéré par w, le **poids** d'une coupe (C_1,C_2) est $\sum_{u\in C_1, v\in C_2} w(u,v)$ (avec $w(u,v)=0$ si $(u,v)\notin E$). $=|C_1,C_2|$

Th 59: Soit $G=(V,E)$ un réseau de capacité c. Si f est un flot et C est une coupe, alors $|f|\leq|C|$. De plus, si f est maximum et C de poids minimum, alors $|f|=|C|$.

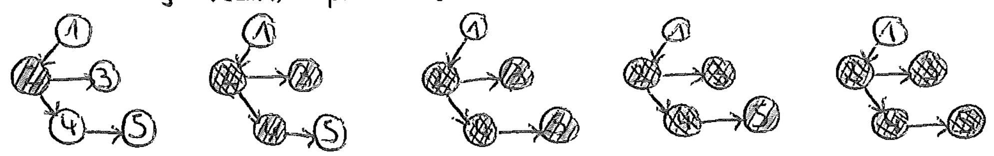
Appl 60: Si seule la valeur du flot est intéressante, on peut utiliser les algorithmes servant à déterminer une coupe minimale plutôt qu'un flot maximal, et inversement.

925 - Graphes: représentations et algorithmes.

Graphe G_1 (ex 4) — Graphe G_2 (ex 4) — Liste adjacence G_1 (ex 8) — Réseau à plusieurs sources/puits converti en réseau à source/puits uniques (ex 55)



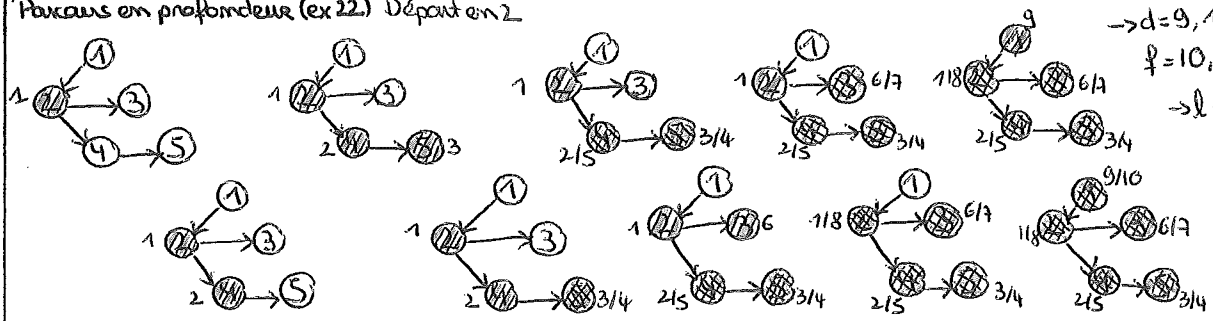
Parcours en largeur (ex17) Départ en 2 et F={2}



1 -> 2
2 -> 4 -> 3
3 /
4 -> 5
5 /

F={2} F={3,4} F={5,3} F={5} F=ϕ -> l = 5,3,4,1.

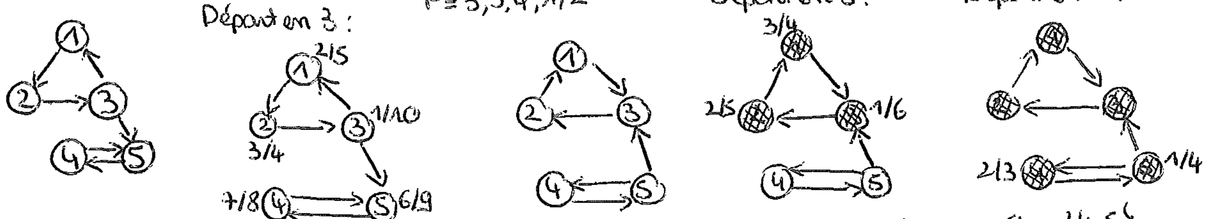
Parcours en profondeur (ex 22) Départ en 2



->d=9,1,6,2,3
f=10,8,7,5,4
->l=1,2,3,4,5

Kosaraju (ex43)

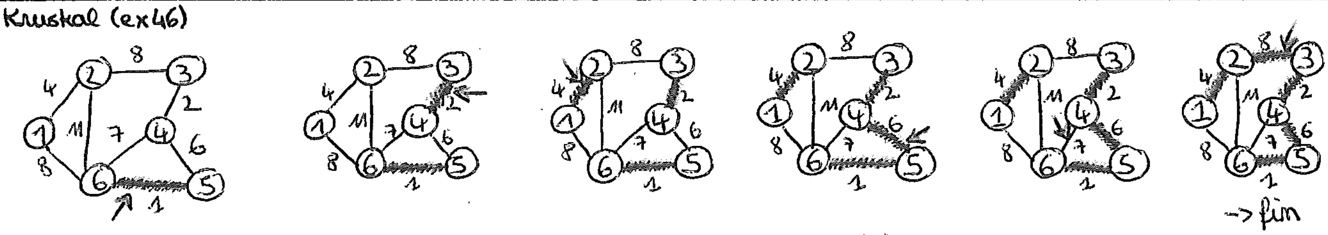
Départ en 3 : P=3,5,4,1,2 Départ en 3 : Départ en 5 :



-> Cc: {1,2,3} P=5,4

-> Cc: {4,5} P=ϕ

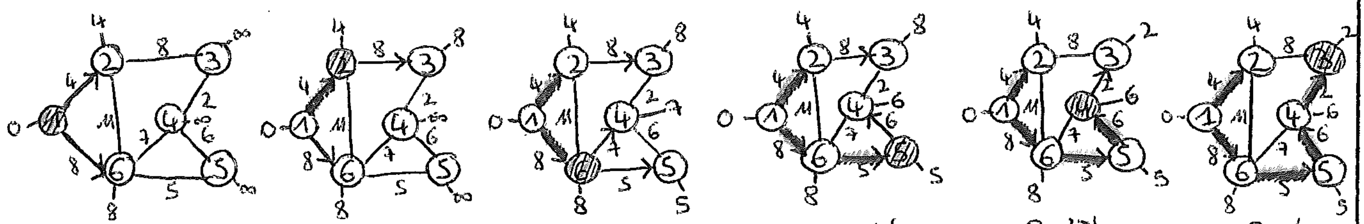
Kruskal (ex46)



-> fin

(On regarde encore 1->6 et 2->6).

Prim (ex50) Départ: 1 et Q={1,2,3,4,5,6}



Q={2,3,4,5,6} Q={3,4,5,6} Q={3,4,5} Q={3,4} Q={3} Q=ϕ