Récapitulatif séances 12 à 19

Arbres couvrants

- Contexte : graphes orientés ou non
- *sous-graphe couvrant* d'un graphe G : graphe G' tq $S_{G'} = S_G$ et $A_{G'} \subseteq A_G$ (sous-graphe qui a les mêmes sommets que G).
- *arbre couvrant* d'un graphe G : sous-graphe couvrant qui est un arbre

Parcours

Deux façons basiques de parcourir un graphe (orienté ou non) en construisant un arbre couvrant (représenté par le tableau `pred`). Même complexité au pire cas : $O(|S| + |A|)$. NB: les couleurs n'ont pas la même signification dans BFS et DFS.

```
fonction BFS(G, s):
    créer file vide
    enfiler s
    colorer s en gris
    tant que la file n'est pas vide:
        défiler u
        pour chaque successeur v de u:
            si v est blanc:
                pred[v] = u
                colorer v en gris
                enfiler v
        colorer u en noir
```

```
fonction DFS(G, s):
    colorer s en gris
    pour chaque successeur v de s:
        si v est blanc:
            pred[v] = s
            DFS(G, v)
    colorer s en noir
```

Plus courts chemins

- Contexte: graphe orienté pondéré
- Ici on appelle *longueur* d'un chemin la somme des poids de ses arcs (et non pas son nombre d'arcs !)
  - on appelle *plus court chemin* (PCC) de s à s' un chemin de longueur minimum de s à s'
  - on appelle *distance* de s à s' la longueur d'un plus court chemin de s à s'
- Problème SSSP (*single source shortest paths*) : on cherche les plus courts chemins d'un sommet source à tous les sommets du graphe.
- Un *cycle absorbant* est un cycle de longueur strictement négative. S'il existe un chemin de s à s' qui contient un cycle absorbant, les notions de plus court chemin et de distance de s à s' ne sont pas définies.
- L'algorithme de Dijkstra construit un arbre de PCC pour un graphe dont tous les poids sont positifs. Complexité de l'implémentation standard (qui utilise une file à priorité implémentée avec un tas binaire) : $O((|S| + |A|) \cdot \log|S|)$
- L'algorithme de Bellman-Ford construit un arbre de PCC pour tout graphe qui ne contient pas de cycle absorbant ; s'il existe un cycle absorbant, il le détecte. Complexité : $O(|S| \cdot |A|)$

```
fonction initSSSP(G, source):
    pour chaque sommet s de G:
        dist[s] = +oo
        pred[s] = nil
    dist[source] = 0

fonction relâcherArc(s, s'):
    alt = dist[s] + poids[s,s']
    si alt < dist[s']:
        dist[s'] = alt
        pred[s'] = s

fonction Dijkstra(G, source):
    initSSSP(G, source)
    nonVisités = ensemble des sommets de G

    tant que nonVisités n'est pas vide:
        retirer le sommet s de nonVisités
                  tq dist[s] est minimale
        pour chaque successeur s' de s qui est
                                dans nonVisités:
            relâcherArc(s, s')

fonction BellmanFord(G, source):
    initSSSP(G, source)

    répéter |S| fois:
        pour chaque arc (s,s') de G:
            relâcherArc(s, s')
    si dist a changé lors de la dernière ité
     ration:
        afficher "trouvé un cycle absorbant"
```

- L'algorithme de Floyd-Warshall calcule les distances de tout sommet à tout sommet :

```
fonction FloydWarshall(G):
    dist = matrice (n,n)
    pour i de 1 à n:
        pour j de 1 à n:
            s'il y a un arc (i,j) dans G:
                dist[i,j] = poids[i,j]
            sinon:
                dist[i,j] = +oo
        dist[i,i] = 0

    pour k de 1 à n:
        pour i de 1 à n:
            pour j de 1 à n:
                dist[i,j] = min(
                        dist[i,j],
                        dist[i,k]+dist[k,j]
                    )
    renvoyer dist
```

Arbres couvrants de poids minimum

- Contexte : graphes simples non orientés valués.
- On appelle *poids d'un graphe* G la somme des poids de ses arêtes : $\text{poids}(G) = \sum_{a \in A_G} \text{poids}(a)$
- Arbre couvrant de poids minimum (ACPM) d'un graphe G: arbre couvrant T de G tel qu'il n'existe pas d'arbre couvrant de G de poids strictement inférieur ; autrement dit, tel que pour tout arbre couvrant T' de G, $\text{poids}(T) \leq \text{poids}(T')$.
- Une *coupe* d'un graphe G est une partition de S_G en deux ensembles, E et $S_G \setminus E$.
  - une coupe définit des *arêtes traversantes* : les arêtes de G dont les extrémités ne sont pas dans le même ensemble
  - on appelle aussi « coupe » l'ensemble des arêtes traversantes
- Propriété des coupes : pour toute coupe de G, une arête traversante a de poids strictement inférieur à toutes les autres arêtes traversantes de la coupe appartient forcément à tout ACPM de G.
- Deux algos gloutons classiques, Prim et Kruskal, permettent de trouver un ACPM. La propriété des coupes permet de montrer que l'arbre couvrant construit par les deux algos est de poids minimal

```
fonction Prim(G, source):
    priorité, pred = dictionnaires vides
    nonVisités = {}
    pour chaque sommet s de G:
        priorité[s] = +oo
        pred[s] = nil
        ajouter s à nonVisités
    priorité[source] = 0

    tant que nonVisités n'est pas vide:
        prendre le sommet s de nonVisités tq
     priorité[s] est minimale
        retirer s de nonVisités

        pour chaque successeur s' de s:
            si s' est dans nonVisités et poids[s,
    s'] < priorité[s']:
                pred[s'] = s
                priorité[s'] = poids[s,s']
    renvoyer pred
```

```
fonction Kruskal(G):
    composantes = {} (ensemble d'ensembles de
    sommets)
    T = {} (ensemble d'arêtes)
    pour chaque sommet s de G:
        ajouter {s} à composantes
    pour chaque arête (s,s') de G, dans l'ordre
     croissant des poids:
        si s et s' ne sont pas dans la même
     composante:
            ajouter (s,s') à T
            fusionner les composantes de s et s'
    renvoyer T
```

Ordre topologique

- Contexte: graphes simples orientés.
- Un ordre total $<$ sur S_G est un *ordre topologique* de G s'il respecte la condition suivante : pour tout couple de sommets (s, s'), s'il y a un arc de s à s' dans G, alors $s < s'$.
- Prop. : G est un DAG si et seulement si G admet un ordre topologique.
- Trouver un ordre topologique peut se faire très simplement avec un DFS :

```
fonction triTopologique(G):
    créer une pile p vide
    pour chaque sommet s de G:
        si s est blanc:
            DFS(G, s)
    tant que p n'est pas vide:
        dépiler s de p
        afficher s

fonction DFS(G, s):
    colorer s en gris
    pour tout successeur s' de s:
        si s' est blanc:
            DFS(G, s')
    colorer s en noir
    empiler s dans p
```

Coloration de graphes

- Contexte : graphes simples non orientés.
- Une k-coloration d'un graphe $G = (S, A)$ est une partition de S en k stables (non vides). Formellement : un ensemble $\mathcal{C} = \{C_1, \dots C_k\}$ avec
  - $\forall i$: $\emptyset \subsetneq C_i \subseteq S$
  - $\forall i, \forall s, s' \in C_i$: s et s' ne sont pas voisins
  - $\forall i, j$ tq $i \neq j$: $C_i \cap C_j = \emptyset$
  - $\bigcup_{1 \leq i \leq k} C_i = S$
- Un graphe est dit k-colorable s'il admet une k-coloration.
- On appelle *nombre chromatique* de G, noté $\chi(G)$, le plus petit entier k pour lequel G admet une k-coloration.
- Prop. : $\chi(G) \leq \Delta(G) + 1$
  (avec $\Delta(G) = \max_{s \in S_G} \mathrm{d}(s)$)
- Prop. : $\chi(G) \leq 1 + |S_G| - \alpha(G)$
  (avec $\alpha(G)$ la taille d'un stable maximum de G)
- Prop. : $\chi(G) \geq \omega(G)$
  (avec $\omega(G)$ la taille d'une clique maximum de G)
- Un graphe est dit *planaire* si on peut le dessiner sur une surface plane sans que les arêtes se croisent.
- Thm des quatre couleurs: si G est un graphe planaire, alors $\chi(G) \leq 4$.
- Trouver une coloration : on connaît des algorithmes gloutons, comme l'algorithme de Welsh-Powell, efficaces mais qui ne donnent pas forcément une coloration minimale

```
fonction WelshPowell(G):
trier les sommets de G par ordre décroissant de
                                        leur degré
tant que tous les sommets ne sont pas colorés:
    choisir une nouvelle couleur K
    pour chaque sommet s non coloré (dans l'ordre
                                        choisi)
    :
        si aucun voisin de s n'est de couleur K:
            colorer s avec la couleur K
```

Ordonnancement

- Problème d'ordonnancement (sans durées) :
  - ensemble de tâches à effectuer, l'une après l'autre, sans interruption ni parallélisation
  - avec des contraintes de précédence de la forme « avant de faire la tâche C, il faut avoir fini les tâches A et B. »
- Le problème peut être représenté par un *graphe d'ordonnancement* : graphe simple orienté dont les sommets sont des tâches et les arcs sont les contraintes.
- Solution du problème = liste ordonnée des tâches qui respecte les contraintes = ordre topologique du graphe
  - le problème n'a de solution que si le graphe est un DAG
- On considère maintenant que **chaque tâche a une durée**, et qu'on peut **les effectuer en parallèle**. On souhaite connaître la durée minimale nécessaire pour effectuer l'ensemble des tâches.
- On modifie le graphe d'ordonnancement comme suit :
  - on ajoute un sommet spécial Start, prédécesseur de tous les sommets sources du graphe précédent
  - on ajoute un sommet spécial End, successeur de tous les sommets puits du graphe précédent
  - on met comme poids sur chaque arc (s, s') la durée de la tâche s.
- Le graphe résultant est le « PERT à potentiels tâches » du problème, mais on l'appellera simplement « graphe d'ordonnancement ». Il faut que ce soit un DAG.
- Trouver la durée minimale du projet revient à calculer le poids d'un plus long chemin dans le graphe d'ordonnancement
  - un tel chemin est appelé *chemin critique*, les sommets sur un chemin critique sont les *tâches critiques*, celles qui ne peuvent pas prendre de retard sans allonger la durée du projet
- On peut calculer la *marge totale* de chaque tâche en calculant leurs dates de début au plus tôt (EST, *earliest starting time*) et au plus tard (LST, *latest starting time*) : la marge totale est LST $-$ EST. Les tâches critiques sont celles dont la marge totale est nulle.
  - Soit un sommet t avec comme prédécesseurs $p_1, \dots, p_k$: la date de début au plus tôt de t est $\max_{1 \leq i \leq k}(\mathrm{EST}(p_i) + \mathrm{poids}(p_i, t))$
    - En posant EST(Start) $= 0$ et en parcourant les sommets selon un ordre topologique, on peut donc calculer la EST de tous les sommets. NB: EST(End) est en fait la date de fin au plus tôt du projet (donc sa durée minimale)
  - Soit un sommet t avec comme successeurs $s_1, \dots, s_m$: la date de début au plus tard de t est $\min_{1 \leq i \leq m}(\mathrm{LST}(s_i) - \mathrm{poids}(t, s_i))$
    - En posant LST(End) = EST(End) (on veut que le projet soit fini dès que possible), et en parcourant les sommets selon un ordre topologique inverse, on peut donc calculer la LST de tous les sommets, et donc leur marge totale (LST $-$ EST).

Cycles hamiltoniens

- Un *cycle hamiltonien* d'un MGNO est un cycle (donc élémentaire) qui passe par tous les sommets du graphe.
- Un graphe est dit *hamiltonien* s'il a un cycle hamiltonien
- On ne connaît pas de caractérisation des graphes hamiltoniens. En fait déterminer si un graphe quelconque est hamiltonien est un problème NP-difficile.
- Pour montrer qu'un graphe n'est pas hamiltonien :
  - Règle 1 : si un sommet est de degré 2, tout cycle hamiltonien doit passer par ses deux arêtes incidentes
  - Règle 2 : quand on construit un cycle hamiltonien, on ne peut pas former de cycle avant que tous les sommets n'aient été visités
  - Règle 3 : si on sait que 2 des arêtes incidentes d'un sommet font partie de tout cycle hamiltonien, alors on peut supprimer ses autres arêtes incidentes
- Conditions suffisantes pour qu'un graphe soit hamiltonien :
  - Th. d'Ore : soit G un GNO *simple* d'ordre $n \geq 3$. Si pour toute paire de sommets $\{s, s'\}$ non voisins, on a $\mathrm{d}(s) + \mathrm{d}(s') \geq n$, alors G est hamiltonien.
  - Corollaire, th. de Dirac : soit G un GNO *simple* d'ordre $n \geq 3$. Si pour tout sommet s de G, on a $\mathrm{d}(s) \geq n/2$, alors G est hamiltonien.
- Problème du voyageur de commerce : étant donné un GSNO pondéré, trouver un cycle hamiltonien de poids minimal (poids d'un cycle = somme des poids des arêtes).
- Pour simplifier, on considérera toujours que le problème porte sur un graphe complet (on peut remplacer les non-arêtes par des arêtes de très grand poids)
- Le PVC est également NP-difficile. En pratique, on peut utiliser des *heuristiques* comme celle du plus proche voisin : on choisit un premier sommet, puis on prend toujours comme sommet suivant le sommet le plus proche du sommet courant. Ça ne marche pas trop mal en pratique, mais pas du tout sur certains graphes.