Algorithmes gloutons

- Le type de problèmes : trouver une "solution" entre plusieurs possibles qui sont comparables selon un critère global;

- La méthode : à chaque étape choisir selon les informations locales;

- pas forcément évident si la méthode gloutonne trouve la meilleure;

Exemple 1: arbre recouvrant de poids minimum

Dans un graphe (peut-être complet) où les arêtes ont un poids, trouver l'arbre qui relie tous les sommets avec la somme des poids minimisée

Algorithme glouton non optimal:
commencer avec un arbre d'un seul sommet et choisir toujours l'arête du poids minimum pour rajouter le prochain sommet

Et un autre (optimal ou non?); Prim:
choisir toujours l'arête du poids minimum pour rajouter un nouveau sommet

Algorithme glouton et optimal; Kruskal:
commencer avec n arbres chacun d'un seul sommet
choisir toujours l'arête du poids minimum qui relie deux arbres;
raison d'optimalité;
détails de la recherche de cette arête omis ici.

Exemple 2: sac-a-dos continu

- le problème: Remplir un containeur avec la charge de la plus grande valeur possible étant donné plusieurs matériaux, chacun avec une valeur/kilo et une quantité disponible données

- l'algorithme glouton: choisir toujours le matériau avec la meilleure rapport valeur:poids;

- pourquoi optimale?

Des cas où la méthode gloutonne n'est pas optimale

- sac-a-dos discret: des objets qu'on ne peut pas découper;
  exemple de non optimalité;

- coloration de graphes: on veut colorier les sommets d'un graphe avec le plus petit nombre de couleurs possible en respectant à la règle que deux sommets reliés par une arête doivent recevoir des couleurs différentes;
  exemple de non optimalité;

- commis voyageur: Etant donné une carte avec les distances (coûts) entre des paires de villes, trouver le chemin le moins coûteux qui passe par chaque ville une seule fois (et retourne au point de départ)
  exemple de non optimalité;

commentaires sur non existence d'algorithmes efficaces (connus) pour ces 3 derniers

Ces 3 problèmes sont des variantes des problèmes *NP-complets*; Personne ne connait des algorithmes garantis de trouver une solution optimale en temps raisonnable (borné par un polynôme); mais on ne sait pas si c'est impossible ou non.

Donc, il est utile de considérer des algorithmes qui ont une garantie (ou même une bonne chance) de trouver une solution près de l'optimale.

Un premier essai à un tel algorithme pourrait être un algorithme glouton simple.

Un deuxième essai serait d'essayer d'améliorer le résultat du premier par des modifications locales gloutonnes.

Arbres binaires
(de recherche)

- Arbres binaires: une structure très utile;

- soit la structure représente la structure logique des données (arbre syntaxique, arbre généalogique)

- soit utilisée pour permettre les opérations nécessaires sur une structure abstraite; (arbre binaire de recherche qui permet des recherches, additions, suppressions d'éléments plus efficacement que tableau ou liste avec pointeurs)

- terminologie panachée d'arbres botaniques et généalogiques: racines, fils, feuilles, frères, etc.

- hauteur et profondeur: la racine est-elle en haut ou en bas?

Insertion, recherche, annulation dans un arbre binaire de recherche

- recherche: commencer à la racine; parcourir à la gauche ou la droite selon comparaison avec la valeur à chaque noeud; échec si sous-arbre vide

- insertion: pareille sauf remplacer sous-arbre vide par nouveau sous-arbre avec la valeur souhaitée

- effacer un élément: facile si à une feuille; facile si un seul fils; sinon remplacer par (par exemple) premier élément dans sous-arbre droit

- toujours nombre d'opérations linéaire dans la hauteur de l'arbre; quelques manipulations simples de pointeurs

Les ordres de parcours

- Trois ordres de parcourir tous les noeuds d'un arbre (binaire) sont souvent utiles

- ordre infixe: (par exemple afficher les éléments d'un arbre binaire de recherche en ordre): d'abord parcourir tout le sous-arbre gauche (en utilisant le même ordre), ensuite la racine; enfin le sous-arbre droite

- ordre préfixe: (calculer la profondeur de chaque noeud): d'abord la racine, ensuite à gauche, enfin à droite

- ordre postfixe: (par exemple calculer la valeur d'une expression à partir de son arbre syntaxique; calculer la hauteur): d'abord les deux sous-arbres (gauche, droite), enfin la racine

Arbres équilibrés

- Un arbre binaire de recherche peut avoir hauteur près de son nombre de noeuds (n); par contre la moyenne est $O(logn)$;

- Et c'est cette hauteur qui détermine le temps des opérations de base;

- Il existe un grand nombre possible d'arbres avec n noeuds; on en obtient un selon l'ordre des insertions;

- Donc, l'idée de faire un peu plus de travail (à chaque insertion) afin de garder une hauteur plus raisonnable
(de préférence $O(logn)$).

- Plusieurs méthodes peuvent être utilisées

1: Arbres AVL

- Principe: la différence entre les hauteurs des deux sous-arbres est au maximum 1 (à la racine et chaque autre sommet);

- nécessite le stockage d'une petite information cachée à chaque noeud (-1 ou 0 ou 1) la différence entre les deux hauteurs des sous-arbres;

- propriété préservée à une insertion par des *rotations*.

- manipulations locales des pointeurs.

- effacer un élément un peu plus compliqué

2: Arbres 2-3

- chaque noeud a soit une valeur et deux fils, soit deux valeurs et trois fils

- l'ordre est sous-arbre, valeur, sous-arbre [valeur, sous-arbre]

- toutes les feuilles sont à la même profondeur

- pour ajouter une valeur dans un noeud qui n'en contient qu'un: trivial

- s'il y en a déjà deux: insérer le deuxième dans le noeud père et diviser ce noeud en deux (avec peut-être une réaction en chaîne vers la racine)