TP08-Branch and bound

12 et 19 janvier

1 Maxsat

On va décrire une formule sous forme CNF comme une liste de listes d'entiers où chaque liste correspond à une clause. Les entiers contenus dans une clause correspondent aux littéraux : x_i est représentée par i et $\neg x_i$ par $-i$.

```
type formule = int list list;;
```

1. Donner la représentation de $\varphi = (x_1 \vee x_2 \vee \neg x_3) \wedge (x_3 \vee x_4) \wedge (\neg x_4 \vee x_2)$.
   On va implémenter la méthode de Branch and bound décrite en classe.
   Une valuation sera représentée par un tableau de booléens : la case `i` contiendra la valeur de vérité attribuée à la variable x_i.

   ```
   type valuation = bool array;;
   ```

2. Ecrire une fonction `sat : formule -> valuation ->int` telle que `sat phi v` renvoie le nombre de clauses de `phi` satisfaites par la valuation `v`.
3. Ecrire une fonction `insat : formule -> valuation ->int->int` telle que `insat f valuation k` renvoie le nombre de clauses totales moins celles que l'on ne pourra pas satisfaire avec un prolongement de la valuation partielle définie par les k premières cases de `val` (les cases entre 0 et $k-1$ de `valuation` sont les seules considérées c'est-à-dire les variables entre 1 et k).
4. En déduire une implémentation de la résolution exacte du problème MAXSAT avec la méthode branch and bound utilisant l'heuristique `insat`. On écrira une fonction `maxsat : formule -> int ->int` qui renvoie le nombre optimal de clauses satisfaites par la formule. La fonction prendra en entrée le nombre de variables utilisées par la formule.

2 PVC

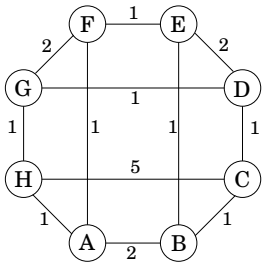
On va considèrer ici le problème de recherche d'un chemin hamiltonien de poids minimal dans un graphe pondéré dont les poids sont positifs ou nuls.

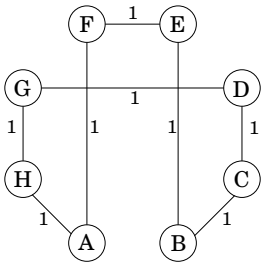
On va représenter un graphe par sa matrice d'adjacence et un chemin par une liste ordonnée de sommets :

```
type graphe = int array array;;
type chemin =int list;;
```

Quand une arête ne sera pas présente dans le graphe, on lui donnera un poids valant `max_int`.

1. Donner la matrice d'adjacence du graphe de gauche de la figure suivante (le graphe de droite est un cycle hamiltonien de poids minimal et permettra de tester votre fonction) :
2. Supposons qu'on dispose d'un chemin partiel $\tilde{c}$ dont la longueur est strictement inférieure au nombre total de sommets. Donner un minorant simple du poids d'un chemin hamiltonien c qui prolonge $\tilde{c}$.





3. Ecrire une fonction `supprimer : int->int list->int list` qui supprime un élément dont la valeur est passée en entrée de la liste.
4. Ecrire une fonction `poids_chemin : graphe->chemin->int` qui calcule le poids d'un chemin donné dans un graphe pondéré (attention à la gestion des arêtes de poids `max_int` et à ne pas dépasser la capacité mémoire).
5. En utilisant cette heuristique et la méthode de branch and bound, écrire une fonction `pvc : graphe->chemin` qui renvoie un chemin hamiltonien de poids minimal.

3 Sac à dos

La version du problème du sac à dos où chaque objet est limité est souvent appelée `SAC À DOS 0-1` :

* **Instance** : une liste de poids $p_1, \ldots, p_n$, une liste de valeurs $v_1, \ldots, v_n$ et un poids maximal P, ces valeurs étant des entiers naturels non nuls.
* **Solution** : une liste d'entiers naturels $(a_1, \ldots, a_n) \in \{0,1\}^n$ telle que $\sum_{i=1}^{n} a_i p_i \leqslant P$
* **Optimisation** : Maximiser $\sum_{i=1}^{n} a_i v_i$.

On admet que la version décisionnelle de ce problème est NP-complète. On considère une variante où les coefficients a_i ne sont pas limités aux entiers 0 et 1, mais à toute valeur entre 0 et 1 : `SAC À DOS FRACTIONNAIRE` :

* **Instance** : une liste de poids $p_1, \ldots, p_n$, une liste de valeurs $v_1, \ldots, v_n$ et un poids maximal P, ces valeurs étant des entiers naturels non nuls.
* **Solution** : une liste de rationnels $(a_1, \ldots, a_n) \in [0,1]^n$ telle que $\sum_{i=1}^{n} a_i p_i \leqslant P$
* **Optimisation** : Maximiser $\sum_{i=1}^{n} a_i v_i$.

1. Montrer que `SAC À DOS FRACTIONNAIRE` $\in P$ en proposant un algorithme glouton qui le résout en complexité $O(n \log n)$.

On cherche à résoudre le problème `SAC À DOS 0-1` par un algorithme de type Branch and Bound. Pour $i \in [\![0, n]\!]$, une solution partielle du problème sera un i-uplet $(a_1, a_2, \ldots, a_i) \in \{0,1\}^i$ (vide si $i = 0$).

2. Dans quel ordre semble-t-il pertinent de ranger les objets avant de lancer l'algorithme BnB résolvant ce problème ? Avec cet ordre, quelle est l'heuristique de branchement à considérer ?

On cherche à déterminer une heuristique d'évaluation pour une solution partielle. On propose d'utiliser le problème de `SAC À DOS FRACTIONNAIRE` pour cela. Pour $\tilde{y} = (a_1, \ldots, a_i)$ une solution partielle, on pose $h(\tilde{y})$ comme la valeur maximale d'une solution à ce problème pour l'instance $((p_j)_{j>i}, (v_j)_{j>i}, P_{\tilde{y}})$ où $P_{\tilde{y}} = P - \sum_{j=1}^{i} a_j p_j$.

3. Montrer que h est admissible, c'est-à-dire que $h(\tilde{y})$ est toujours supérieur ou égal à la valeur totale d'une solution complétée à partir de $\tilde{y}$.

On suppose disposer de tableaux en C `int* p` et `int* v` de même taille `int n` contenant les poids et les valeurs des objets, déjà triés dans l'ordre de la question 2.

4. Écrire une fonction `double h(int* p, int* v, int* a, int n, int i, int P)` qui prend en argument les tableaux de poids $(p_j)_j$, de valeurs $(v_j)_j$ et des coefficients $(a_j)_j$, ainsi que la taille n identique pour ces trois tableaux, un entier i et un poids maximal P et calcule la valeur $h(\tilde{y})$ où $\tilde{y} = (a_1, \ldots, a_i)$. On ne modifiera aucun des trois tableaux.
   Attention : les indices dans un tableau commencent à 0 *contrairement à ceux des n-uplets considérés dans l'exercice.*
5. En déduire une fonction `int* sac_a_dos_01(int* p, int* v, int n, int P)` qui applique l'algorithme BnB permettant de résoudre ce problème. La fonction renverra le tableau des a_i. On pourra commencer par écrire une fonction récursive
   `void sac_aux(int* p, int* v, int* a, int* amax, int n, int i, int P, int* Vmax)`.

4 BinPacking

Le problème dit BinPacking est défini ainsi :

Instance : un entier naturel C et une famille $X = x_0, \ldots, x_{n-1}$ d'entiers naturels

Solution admissible : une partition de X en $B_0 \sqcup \cdots \sqcup B_{k-1}$ telle que $\sum_{x \in B_i} x \leqslant C$ pour tout i

Optimisation : minimiser k

Pour rappelle le principe de la stratégie gloutonne `first-fit-decreasing` :

- La stratégie **first-fit** considère aussi les objets dans l'ordre d'arrivée, mais maintient une liste (initialement vide) de boîtes $B_0, \ldots, B_{k-1}$. À chaque fois que l'on considère un objet, on cherche le premier i tel que l'objet rentre dans la boîte B_i :
  * s'il en existe un, on ajoute l'objet dans cette boîte ;
  * sinon, on crée une nouvelle boîte B_k dans laquelle on place l'objet.
- La stratégie **first-fit-decreasing** procède comme **first-fit** mais commence par trier les objets par ordre décroissant de volume.

On représente une instance de BinPacking par un entier `capacity` (la capacité C des boîtes) et une liste d'entiers correspondant aux poids des objets ; on supposera que tous les poids sont inférieurs ou égaux à la capacité.

```
type instance = int * int list
```

On représente une boîte par le stype suivant :

```
type box = {capacity : int ;content = int list};;
```

1. Écrire une fonction `first_fit_decreasing`.
2. Écrire une fonction `solve` calculant une solution optimale pour une instance de BinPacking. On utilisera (de manière assez basique) la technique dite *Branch and Bound* pour obtenir une fonction raisonnablement efficace.
   On initialisera notre recherche avec le résultat de la fonction `first_fit_decreasing` afin d'optimiser l'élagage.
   On doit par exemple trouver une solution utilisant 6 boîtes pour l'instance suivante, de manière presque instantanée.

```
let test_exact =
  (101, [27; 11; 41; 43; 42; 54; 34; 11; 2; 1; 17; 56;
   42; 24; 31; 17; 18; 19; 24; 35; 13; 17; 25])

val solve : instance -> box list
```