centre de recherche en informatique de lens

UNIVERSITÉ D'ARTOIS

OPC - Filtrage, recherche et optimisation

1 Efficacité/capacité de filtrage

Dans cette section, nous allons nous intéresser aux algorithmes de filtrage. Dans un premier temps, avec le problème `Domino`, nous allons constater que différents algorithmes de filtrage (ici, des algorithmes génériques pour établir AC) peuvent avoir une efficacité (temps d'exécution) très différente. Dans un second temps, avec le problème `Pigeons`, nous allons constater que certaines décompositions de contraintes (ici, une contrainte `allDifferent` décomposée comme une clique de contraintes binaires de différence) affaiblissaient sérieusement la capacité de filtrage.

1.1 Domino

```
class Domino extends Problem {
  int nDominos = askInt("Number of dominos");
  int nValues = askInt("Number of values");

  void model() {
    Var[] x = array("x", dom(0, nValues - 1), nDominos);
    Var first= x[0], last=x[x.length-1];

    allEqual(x);
    ctr(or(eq(add(first, 1), last), eq(first, last, nValues - 1)));
  }
}
```

Le code ci-dessus est pour un problème académique simple appelé `Domino`.

1. Décrire le CN modélisé par ce code.
2. Décrire le résultat de la propagation de contraintes (type AC) lorsque les paramètres sont `nDominos = 4` et `nValues = 4`.
3. Après avoir créée et compilée la classe Domino, essayer de résoudre différentes instances de ce problème avec une commande telle que :

```
java abscon.Resolution Domino
```

Rappelons qu'il est possible de placer directement les paramètres sur la ligne de commande comme par exemple :

```
java abscon.Resolution Domino 100 100
```

4. Par défaut, l'algorithme qui établit la propriété AC est appelé $AC3^{rm}$. Il est intéressant de noter que l'algorithme classique, de nom AC3, a un comportement pathologique sur ce problème. Essayer :

```
java abscon.Resolution Domino 200 200
```

puis :

```
java abscon.Resolution Domino 200 200 -res=false
```

5. Comparer les temps de résolution (la différence tient uniquement à l'algorithme de propagation/filtrage car les arbres construits sont les mêmes).

1.2 Pigeons

Le problème des pigeons consiste à placer n pigeons dans $n-1$ casiers avec la contrainte que deux pigeons ne puissent se trouver dans le même casier.

1. Que peut-on conclure de cet énoncé ?
2. Écrire le code pour ce problème dans une classe `Pigeons`.
3. Résoudre des instances de ce problème avec des valeurs croissantes de n (par exemple, $n = 8, n = 12, n = 20, n = 50, n = 100$)
4. Résoudre des instances de ce problème avec des valeurs croissantes de n (par exemple, $n = 8, n = 12, n = 20, n = 50, n = 100$) mais cette fois-ci en utilisant le paramétrage `-ugc=no` signifiant que les contraintes globales doivent être décomposées. Par exemple :

```
java abscon.Resolution Pigeons 10 -ugc=no
```

5. Comparer le nombre de contraintes, le niveau d'inférence obtenu au preprocessing (c'est-à-dire le nombre de valeurs éliminées) et les temps de résolution.

2 Techniques de recherche - illustration avec le problème RLFAP

RLFAP est un problème d'allocation de fréquences de liaisons radios, issu du Centre d'Electronique de l'Armement. Ci-dessous, nous allons essayer de résoudre les instances (CSP) de ce problème les plus difficiles (instances `scen11-fXX.xml.lzma` que l'on peut télé-charger à partir de la page `www.cril.fr/~lecoutre`) en employant différentes techniques.

1. Une heuristique de choix de variables classique est `dom/ddeg` qui sélectionne à chaque étape (noeud) une variable avec le plus petit ratio "taille courante du domaine" sur "degré dynamique". La classe qui implante cela sous AbsCon s'appelle `DDegOnDom`. Tester :

```
java abscon.Resolution scen11.xml.bz2 -varh=DDegOnDom
java abscon.Resolution scen11-f08.xml.bz2 -varh=DDegOnDom -t=30s
```

Un timeout de 30 secondes est utilisé car il n'est pas possible de résoudre l'instance `scen11-f08.xml` en moins de plusieurs heures.

2. L'heuristique de choix de variables adaptative `dom/wdeg` qui sélectionne à chaque étape (noeud) une variable avec le plus petit ratio "taille courante du domaine" sur "degré pondéré" surclasse les heuristiques classiques telle que `dom/ddeg`. C'est l'heuristique par défaut. On écrira alors par exemple :

```
java abscon.Resolution scen11.xml.bz2
java abscon.Resolution scen11-f08.xml.bz2
```

3. Redémarrer la recherche permet une plus grande diversification et de corriger d'éventuels mauvais choix initiaux. On utilisera le paramétrage `-rc=10` pour signifier qu'après 10 échecs, la recherche doit redémarrer (rc = restarts cutoff). Tester :

```
java abscon.Resolution scen11-f08.xml.bz2 -rc=10
java abscon.Resolution scen11-f06.xml.bz2 -rc=10
```

4. Il est parfois intéressant de concentrer la recherche sur la dernière variable entrée en conflit. On utilisera le paramétrage `-lc=10` pour signifier qu'on utilise la technique de last-conflict permettant une forme de retour-arrière intelligent. Comparer :

```
java abscon.Resolution scen11-f05.xml.bz2
java abscon.Resolution scen11-f05.xml.bz2 -lc=10
```

3 Optimisation

Dans cette section, nous allons nous intéresser aux problèmes d'optimisation sous contraintes (COP pour Constraint Optimization Problem). Il s'agit d'intégrer aux réseaux, en plus des variables et des contraintes, une fonction d'objectif qu'il faut soit minimiser soit maximiser. Dans certains cas, cette fonction d'objectif peut se réduire à une simple variable (qui est alors contrainte avec les autres variables du problème). Dans d'autres cas, il peut s'agir d'une somme.

3.1 Modéliser un problème d'optimisation avec AbsCon

```
class Optim extends Problem {
  void model() {
    Var x = var("x", dom(1, 10));
    Var y = var("y", dom(1, 10));
    Var z = var("z", dom(1, 10));

    ctr(lt(x, y));
    ctr(lt(y, z));

    maximize(SUM, vars(x, y, z));
  }
}
```

Le code ci-dessus est pour un problème d'optimisation simple appelé `Optim`.

1. Décrire le réseau de contraintes (de type COP) modélisé par ce code. Il est à noter que la constante SUM est définie dans la classe TypeObjective (import static nécessaire).
2. Après avoir créée et compilée la classe Optim, essayer de résoudre ce problème avec une commande telle que :

```
java abscon.Resolution Optim -f=cop
```

Nous indiquons ici que le type de problème à résoudre est COP. Noter que par défaut, pour un COP, toutes les solutions sont recherchées (plus précisément, toute solution permettant d'améliorer la valeur courante de la fonction d'objectif est systématiquement recherchée). C'est équivalent à utiliser `-s=all`.

3. Pour finir, sur ce petit exemple, il est préférable de chercher à instantier les valeurs plus grandes dans chaque domaine. Puisque l'heuristique de choix de valeurs est par défaut `-valh=First` pour indiquer que la valeur à instantier est toujours la première du domaine, on indiquera explicitement `-valh=Last` sur la ligne de commande, ce qui donne :

```
java abscon.Resolution Optim -f=cop -valh=Last
```

3.2 Problème du sac à dos

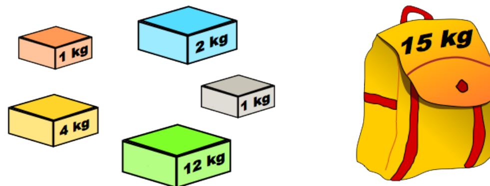
On nous donne un ensemble de n objets que nous devons sélectionner et placer dans un sac à dos d'un certain volume. Chaque objet à un volume (ou poids) et apporte un certain bénéfice. L'objectif est de choisir un sous-ensemble d'objets qui puisse être placés dans le sac tout en maximisant le bénéfice.

Soit v_i le volume de l'objet i, p_i son bénéfice (profit), et C la capacité (volume) du sac. Soit x_i une variable dont la valeur peut être 0 ou 1 : la variable x_i a la valeur 1 lorsque l'objet i est placé dans le sac. Notre objectif est de maximiser :

$\sum_{i=1}^{n} p_i x_i$

soumis à la contrainte :

$\sum_{i=1}^{n} v_i x_i \leq C$



1. Dans une première étape, définir l'instance de problème, de nom SacDos, correspondant aux données suivantes (un sac de capacité $C = 50$, et 50 objets) :

```
class SacDos extends Problem {
  int[] volumes = { 1, 5, 2, 12, 12, 6, 12, 10, 4, 3, 6, 6, 6, 11, 12, 9, 5, 4, 6, 9, 8,
      1, 9, 12, 9, 5, 4, 11, 1, 12, 6, 6, 6, 12, 3, 3, 10, 6, 11, 4, 9, 8, 12, 3, 11, 4,
      11, 5, 5, 7 };
  int[] profits = { 20, 78, 17, 47, 8, 17, 9, 38, 44, 48, 88, 89, 6, 59, 9, 18, 52, 19,
      59, 19, 39, 73, 58, 52, 27, 81, 98, 63, 36, 21, 68, 74, 18, 70, 6, 53, 90, 44, 2,
      42, 24, 81, 69, 15, 17, 24, 58, 23, 6, 72 };
  int nObjects = volumes.length;
  int capacity = 50;

  void model() {
    // a compléter
  }
}
```

Vous intégrerez un tableau de variables 0-1 pour représenter la présence ou non des objets dans le sac. Vous intégrerez une contrainte `sum` et une fonction d'objectif `maximize`. Vérifier que vous obtenez 834 comme bénéfice de votre solution optimale lorsque vous tapez la commande :

```
java abscon.Resolution SacDos -f=cop
```

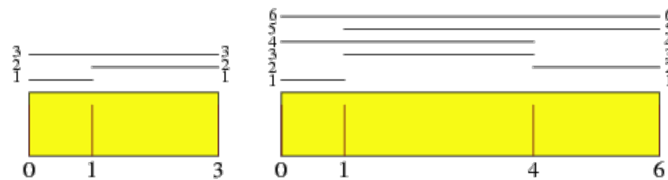
2. Comme ce n'est pas si simple, on peut essayer différentes stratégies d'optimisation. Rechercher une meilleure borne de manière croissante, décroissante ou dichotomique. Essayer :

```
java abscon.Resolution SacDos -f=cop -os=increasing
java abscon.Resolution SacDos -f=cop -os=decreasing
java abscon.Resolution SacDos -f=cop -os=dichotomic
```

3.3 Golomb Ruler

Ce problème (et ses variantes) a de nombreuses applications pratiques incluant le placement de capteurs pour la x-ray crystallography et radio astronomy. Une règle de Golomb peut être définie comme m entiers $0 = a_1 < a_2 < ... < a_m$ tels que les $m(m-1)/2$ différences $a_j - a_i$, $1 \leq i < j \leq m$ soient distinctes. Une telle règle est dite contenir m marques et de longueur a_m. L'objectif est de trouver des règles optimales (c'est-à-dire de longueur minimale) ou proches de l'optimum.

Des règles optimales pour $m = 3$ et $m = 4$ sont illustrées ci-dessous :



1. Construire un premier modèle avec des contraintes binaires et quaternaires.
2. Construire un second modèle avec des contraintes binaires et ternaires (après avoir ajouté des variables auxiliaires).
3. Donnons nous la possibilité de convertir ce problème de décision en un problème d'optimisation. Que devons-nous faire ?
4. Expérimentez avec différentes heuristiques.