Introduction
Le problème CSP
Interface des solveurs
Exploration naïve
Exploration efficace
L'algorithme MAC
Conclusion

Algorithmique & Programmation (INF 431)

Exploration

François Pottier   Benjamin Werner

2 avril 2014

1. Introduction
2. Un problème illustratif : CSP
3. Quelle interface pour un solveur ?
4. Exploration naïve, en 3 transparents
5. Exploration efficace : principes
6. Exploration efficace : l'algorithme MAC
7. Conclusion

Quelques techniques algorithmiques

Parfois, un problème se ramène à :

- un seul sous-problème un peu plus simple ;
- plusieurs sous-problèmes indépendants et nettement plus simples ;
- plusieurs sous-problèmes un peu plus simples et très corrélés.

Quelques techniques algorithmiques

Parfois, un problème se ramène à :

- un seul sous-problème un peu plus simple ;
  – *algorithmes gloutons*
- plusieurs sous-problèmes indépendants et nettement plus simples ;
- plusieurs sous-problèmes un peu plus simples et très corrélés.

Quelques techniques algorithmiques

Parfois, un problème se ramène à :

- un seul sous-problème un peu plus simple ;
  – *algorithmes gloutons*
- plusieurs sous-problèmes indépendants et nettement plus simples ;
  – *diviser pour régner*
- plusieurs sous-problèmes un peu plus simples et très corrélés.

Quelques techniques algorithmiques

Parfois, un problème se ramène à :

- un seul sous-problème un peu plus simple ;
  – *algorithmes gloutons*
- plusieurs sous-problèmes indépendants et nettement plus simples ;
  – *diviser pour régner*
- plusieurs sous-problèmes un peu plus simples et très corrélés.
  – *programmation dynamique*

Quelques techniques algorithmiques

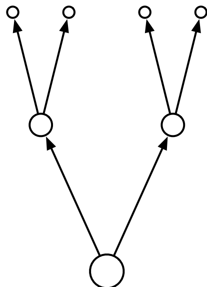
Visuellement, l'approche gloutonne peut être présentée comme ceci...

Exemple :

- problème de l'ordonnancement d'intervalles,
- algorithme « earliest deadline first ».

Quelques techniques algorithmiques

Diviser pour régner comme ceci...

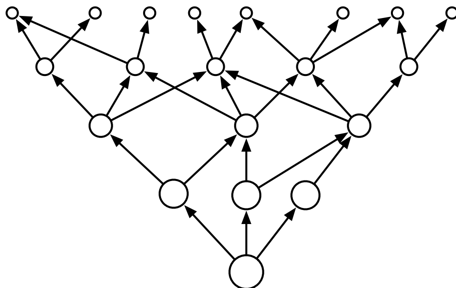


Exemple :

- problème du tri,
- algorithmes « merge sort » et « quicksort ».

Quelques techniques algorithmiques

La programmation dynamique comme ceci...



Exemple :

- le problème du sac-à-dos lorsque les poids sont entiers ;
- algorithme de complexité $O(nP)$ vu en PC.

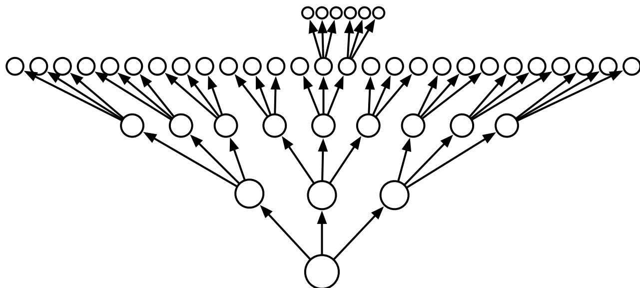
Et si ?

Que faire si le problème se ramène à :

- plusieurs sous-problèmes...
- peu ou pas corrélés...
- ... et seulement un peu plus simples ?

Quelques techniques algorithmiques

C'est-à-dire si la situation ressemble à ceci :



Exemples (transparents suivants) :

- satisfiabilité de circuit ;
- coloriage de graphes.

Exemple : satisfiabilité de circuit

Donnée : une formule F de la logique propositionnelle à n inconnues.

$$F ::= X \mid vrai \mid faux \mid \neg F \mid F \wedge F \mid F \vee F$$

Problème : F est-elle satisfiable ?

Exemple : $(X \vee \neg Y) \wedge (\neg X \vee Y)$ est-elle satisfiable ?

Problème pertinent, par exemple, pour la vérification de microprocesseurs.

Exemple : satisfiabilité de circuit

Il existe un algorithme très simple :

- si $n = 0$, évaluer la formule ;
- sinon, choisir une inconnue X et considérer les deux sous-problèmes obtenus en remplaçant X respectivement par *vrai* et par *faux*.

Un problème à n inconnues donne deux sous-problèmes à $n - 1$ inconnues.

Complexité : $\Omega(2^n)$.

Exemple : coloriage de graphes

Donnée : un graphe non orienté à n sommets ; un entier k.

Problème : peut-on colorier chaque sommet, de sorte que deux voisins n'aient jamais la même couleur, et en employant au plus k couleurs ?

Problème d'allocation de ressources en présence de conflits.

Exemple : coloriage de graphes

Donnée : un graphe non orienté à n sommets ; un entier k.

Problème : peut-on colorier chaque sommet, de sorte que deux voisins n'aient jamais la même couleur, et en employant au plus k couleurs ?

Problème d'allocation de ressources en présence de conflits.

Il existe un algorithme très simple :

- si tous les sommets sont coloriés, vérifier si le but est atteint ;
- sinon, choisir un sommet x et considérer les k sous-problèmes obtenus en attribuant à x chacune des k couleurs possibles.

Un problème à n sommets non coloriés donne k sous-problèmes à $n-1$ sommets non coloriés.

Complexité : $\Omega(k^n)$.

Deux problèmes similaires

Ces deux problèmes présentent des similitudes :

- vérifier une solution est facile ;
- trouver une solution est difficile.

Vérifier une solution est facile

« Satisfiabilité de circuit » appartient à la classe *NP*.

> *Si on sait quelle valeur doit prendre chaque variable, on vérifie facilement que la formule est satisfaite.*

« Coloriage de graphes » appartient à la classe *NP*.

> *Si on sait quelle couleur doit recevoir chaque sommet, on vérifie facilement que deux voisins n'ont jamais la même couleur.*

Un problème appartient à *NP* (« Nondeterministic Polynomial time ») si, dans les cas où la réponse est « oui », on peut fournir un certificat vérifiable en temps polynomial (voir INF423 ou Kleinberg & Tardos, §8).

Trouver une solution est difficile

« Satisfiabilité de circuit » est *NP*-difficile.

C'est le théorème de Cook-Levin (1971).

« Coloriage de graphes » est *NP*-difficile lorsque $k > 2$.

Voir Kleinberg et Tardos (§8.7).

Un problème *Y* est *NP*-difficile si tout problème *X* de *NP* se ramène à *Y* en temps polynomial.

Introduction

Le problème CSP

Interface des solveurs

Exploration naïve

Exploration efficace

L'algorithme MAC

Conclusion

Que faire ?

Ces problèmes sont donc *NP*-complets : dans *NP* et *NP*-difficiles.

On ne connaît aucun algorithme de complexité polynomiale pour les résoudre.

S'il en existait un, on aurait $P = NP$, ce qui est peu vraisemblable.

Que faire ?

Que faire en pratique pour résoudre ces problèmes ?

Nous allons explorer – énumérer toutes les solutions candidates...

... en essayant de ne pas faire preuve de trop de naïveté.

Que faire ?

La complexité sera toujours exponentielle dans le cas le pire.

Des techniques algorithmiques permettent néanmoins de gagner beaucoup de temps.

Je discuterai au passage plusieurs techniques de programmation.

1. Introduction
2. Un problème illustratif : CSP
3. Quelle interface pour un solveur ?
4. Exploration naïve, en 3 transparents
5. Exploration efficace : principes
6. Exploration efficace : l'algorithme MAC
7. Conclusion

1 Introduction

2 Un problème illustratif : CSP

3 Quelle interface pour un solveur ?

4 Exploration naïve, en 3 transparents

5 Exploration efficace : principes

6 Exploration efficace : l'algorithme MAC

7 Conclusion

Un problème général

Le problème CSP (Constraint Satisfaction Problem) généralise « satisfiabilité de circuit » et « coloriage de graphes ».

En gros, le problème est :

> *Étant données n variables sur lesquelles porte une contrainte, peut-on attribuer à chaque variable une valeur, de façon à satisfaire la contrainte ?*

Le problème CSP

Donnée :

- un entier n ;
- pour chaque variable $x \in [0, n)$, un domaine fini D_x ;
- une contrainte C,
  - que nous pouvons considérer abstraitement comme un élément de $\Pi_{x=0}^{x<n} D_x \to \{vrai, faux\}$,
  - même si elle devra en fait être présentée sous une forme syntaxique de taille polynomiale vis-à-vis de n.

Problème : existe-t-il une valuation φ, c'est-à-dire un élément de $\Pi_{x=0}^{x<n} D_x$, qui satisfait C ?

Variantes

On peut également demander :

- Quelles sont toutes les solutions ?
- Quelles sont les solutions optimales selon un certain critère ?

1 Introduction

2 Un problème illustratif : CSP

3 Quelle interface pour un solveur ?

4 Exploration naïve, en 3 transparents

5 Exploration efficace : principes

6 Exploration efficace : l'algorithme MAC

7 Conclusion

Introduction

Le problème CSP

Interface des solveurs

Exploration naïve

Exploration efficace

L'algorithme MAC

Conclusion

Qu'est-ce qu'un solveur ?

Posons :

- un solveur est un objet auquel on soumet un problème et qui permet d'en énumérer les solutions.

Qu'est-ce qu'un solveur ?

Posons :

- un solveur est un objet auquel on soumet un problème et qui permet d'en énumérer les solutions.

Deux points restent à préciser :

- par quel mécanisme s'effectue l'énumération des solutions ?
- sous quelle forme est représenté le problème ?

Énumération via un dialogue

Il ne faut pas construire gloutonnement une liste de toutes les solutions, qui pourrait être de longueur exponentielle.

Un dialogue entre solveur et client est préférable, car il permet de travailler en espace $O(n)$, et permet au client d'arrêter l'énumération à tout moment.

Dialogue via des appels de méthodes

Qui contrôle ce dialogue ? Qui en a l'initiative ? Qui appelle qui ?

Premier scénario :

- le client : « s'il-te-plaît, donne-moi la solution suivante » ;
- le solveur : « je l'ai trouvée, tiens, la voici ».

Second scénario :

- le solveur : « tiens, voici une solution, fais-en ce que tu veux » ;
- le client : « je l'ai traitée, merci, continue la recherche ».

Introduction

Le problème CSP

Interface des solveurs

Exploration naïve

Exploration efficace

L'algorithme MAC

Conclusion

Mécanismes de dialogue

Le premier scénario facilite l'écriture du client.

Le solveur est alors un « itérateur » sur l'ensemble des solutions.

Voir Enumeration ou son cousin Iterator.

Introduction

Le problème CSP

Interface des solveurs

Exploration naïve

Exploration efficace

L'algorithme MAC

Conclusion

Mécanismes de dialogue

Le premier scénario facilite l'écriture du client.

Le solveur est alors un « itérateur » sur l'ensemble des solutions.

Voir Enumeration ou son cousin Iterator.

Le second scénario facilite au contraire l'écriture du solveur.

Je choisis celui-ci.

Dans ce scénario, il faut que le solveur puisse appeler le client.

Mécanismes de dialogue

Le premier scénario facilite l'écriture du client.

Le solveur est alors un « itérateur » sur l'ensemble des solutions.

Voir Enumeration ou son cousin Iterator.

Le second scénario facilite au contraire l'écriture du solveur.

Je choisis celui-ci.

Dans ce scénario, il faut que le solveur puisse appeler le client.

Posons donc :

- un solveur est un objet auquel on soumet un problème et un client ;
- un client est un objet auquel on peut soumettre une solution.

Interfaces– première version

Un solveur pourrait donc satisfaire l'interface suivante :

```
interface Solver {

  // Enumerates all solutions of the problem and passes
  // them (one after the other) to the client.

  void enumerate (Problem problem, Client client) ;
}
```

Le client pourrait de son côté satisfaire cette interface :

```
interface Client {

  // The client receives a solution.

  void act (int[] solution) ;
}
```

Mettre fin à l'énumération

Il manque un aspect.

Nous souhaitons que le client puisse arrêter l'énumération à tout moment.

Il peut signaler cela via une exception.

L'exception `Stop`

Définissons donc une exception dédiée :

```
class Stop extends Exception {

  // The solution that was found.
  final int[] solution ;

  Stop (int[] solution) { this.solution = solution ; }

}
```

Le champ `solution` permet de mémoriser quelle solution on avait sous les yeux au moment où on a stoppé l'énumération.

L'interface du client – version définitive

Le client peut lancer cette exception :

```
interface Client {

  // The client receives a solution. It may throw an
  // exception so as to stop the enumeration.

  void act (int[] solution) throws Stop ;

}
```

L'interface du solveur – deuxième version

Le solveur n'attrape pas cette exception, qui peut donc s'échapper :

```
interface Solver {

  void enumerate (Problem problem, Client client)
    throws Stop ;

}
```

Représentation du problème

Reste à préciser :

- sous quelle forme est représenté le problème ?

Représentation du problème

La contrainte C peut a priori être représentée sous forme :

- abstraite : fonction des valuations dans les booléens ;
- syntaxique : conjonction de contraintes élémentaires ($x \neq y$, etc.).

La première offre moins d'information au solveur.

Un solveur naïf peut se contenter de la première, mais un solveur efficace aura besoin de la seconde.

Introduction

Le problème CSP

Interface des solveurs

Exploration naïve

Exploration efficace

L'algorithme MAC

Conclusion

L'interface du solveur – version définitive

Une interface paramétrée permet de décrire un solveur indépendamment de la manière dont les problèmes sont représentés en mémoire :

```
interface Solver<P> {

  void enumerate (P problem, Client client)
    throws Stop ;

}
```

Un peu de code ré-utilisable

Pour déterminer s'il existe une solution, il suffit d'arrêter l'énumération dès que la première solution est trouvée.

C'est vrai quel que soit le solveur et quelle que soit la représentation `P` utilisée par ce solveur...

Introduction

Le problème CSP

Interface des solveurs

Exploration naïve

Exploration efficace

L'algorithme MAC

Conclusion

Un peu de code ré-utilisable

Voici un client qui s'arrête à la première solution obtenue :

```
static <P> int[] solve (Solver<P> solver, P problem)
{
  Client client =
    new Client () {
      public void act (int[] solution) throws Stop {
        throw new Stop (solution) ;
      }
    } ;
  try {
    solver.enumerate(problem, client) ;
    return null ;
  } catch (Stop stop) {
    return stop.solution ;
  }
}
```

Ce code est générique vis-à-vis de `P`. Il emploie une classe anonyme.

1. Introduction
2. Un problème illustratif : CSP
3. Quelle interface pour un solveur ?
4. Exploration naïve, en 3 transparents
5. Exploration efficace : principes
6. Exploration efficace : l'algorithme MAC
7. Conclusion

Représentation abstraite du problème

Le solveur naïf utilise une contrainte abstraite, ou opaque.

```
interface NaiveConstraint {

  // Maps total valuations to truth values.
  boolean satisfied (int[] valuation) ;

}
```

Une contrainte est un objet auquel on soumet une valuation totale et qui répond « solution » ou « pas solution ».

C'est une fonction des valuations vers les valeurs de vérité.

Introduction

Le problème CSP

Interface des solveurs

Exploration naïve

Exploration efficace

L'algorithme MAC

Conclusion

Représentation abstraite du problème

Un problème est alors un triplet :

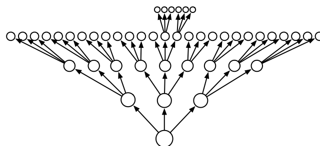
```
class NaiveProblem {
  // The number of variables.
  public final int n ;
  // The domain of each variable.
  public final int[] domain ;
  // The constraint.
  public final NaiveConstraint constraint ;
}
```

On suit la description théorique du problème CSP.

L'algorithme naïf

Il existe un algorithme très simple :

- parcourir un arbre de valuations partielles :



- aux feuilles, qui correspondent aux valuations totales, évaluer la contrainte C ; si elle est satisfaite, la soumettre au client.

Complexité en temps : $\Omega(d^n)$.

Complexité en espace : $O(n)$. Parcours en profondeur d'abord.

L'algorithme naïf

Au cœur de ce solveur, une méthode récursive très simple :

```
class NaiveSolver implements Solver<NaiveProblem> {
  ...
  private void search (int x) throws Stop
  {
    if (x == p.n) { // the current valuation is total
      if (p.constraint.satisfied(current))
        client.act(current) ;
    }
    else              // the current valuation is partial
      for (int v = 0 ; v < p.domain[x] ; v++) {
        current[x] = v ;
        search(x + 1) ;
      }
  }
}
```

Un algorithme analogue sera étudié en TD.

1. Introduction
2. Un problème illustratif : CSP
3. Quelle interface pour un solveur ?
4. Exploration naïve, en 3 transparents
5. Exploration efficace : principes
6. Exploration efficace : l'algorithme MAC
7. Conclusion

En quoi ce solveur est-il naïf ?

Il construit une valuation totale avant de tester si la contrainte est satisfaite.

Il parcourt donc tout l'arbre.

Or certaines valuations partielles violent déjà la contrainte.

On pourrait donc élaguer.

Introduction

Le problème CSP

Interface des solveurs

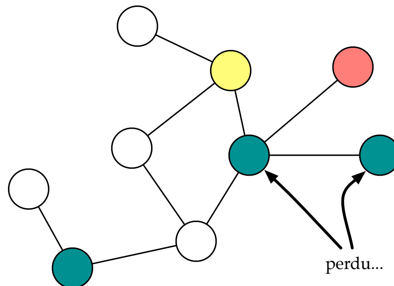
Exploration naïve

Exploration efficace

L'algorithme MAC

Conclusion

Exemple



Cette valuation partielle viole déjà la contrainte et ne pourra donc pas être étendue en une solution.

Une première idée

À chaque sommet, tester si la valuation partielle semble « viable ». Si ce n'est pas le cas, élaguer.

Il y aura compromis entre :

- le temps passé à chaque sommet (qui va augmenter) et
- le nombre de sommets étudiés (qui va diminuer).

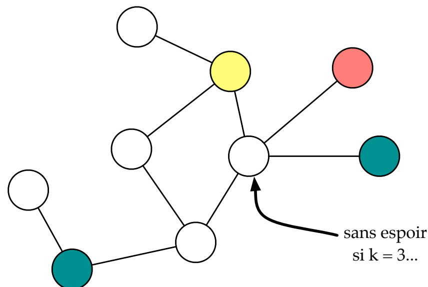
Test nécessairement incomplet : la réponse sera « non viable » ou « peut-être viable ».

Plusieurs tests de « viabilité » sont possibles, plus ou moins coûteux, plus ou moins précis.

Introduction
Le problème CSP
Interface des solveurs
Exploration naïve
Exploration efficace
L'algorithme MAC
Conclusion

Exemple

Cette valuation semble-t-elle « viable » ?



Oui, si le test de viabilité vérifie seulement les contraintes entre sommets déjà coloriés.

Pourtant, si $k = 3$, il n'y a « évidemment » aucune solution dans cette voie.

Une seconde idée

Un test de viabilité plus ambitieux risque d'être trop coûteux.

Au lieu de cela, on peut jouer sur le choix du prochain sommet à colorier :

- si nous choisissons le sommet « problématique », nous verrons très vite que chacune des trois couleurs mène à une situation non viable.
- si au contraire nous « partions colorier ailleurs », nous explorerions un grand sous-arbre sans y découvrir aucune solution.

Un bon ordonnancement des variables permet un élagage plus important !

L'heuristique MRV

L'heuristique « minimum remaining values » consiste à choisir la variable à qui il semble rester le moins de valeurs « possibles ».

Pour l'implémenter efficacement, il semble naturel de maintenir pour chaque variable x un sous-ensemble $possible(x)$ de D_x.

À nouveau, ce sera une approximation : $possible(x)$ sera un sur-ensemble des valeurs que peut prendre x dans une solution qui étend la valuation courante.

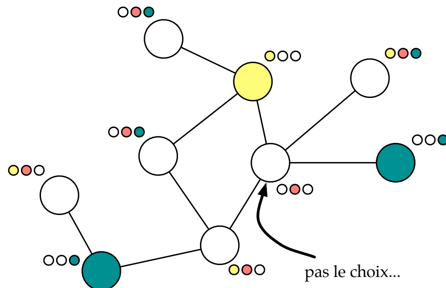
Introduction
Le problème CSP
Interface des solveurs
Exploration naïve
Exploration efficace
L'algorithme MAC
Conclusion

Exemple

Si l'un des *possible*(x) devient vide, l'échec est immédiat : on « élague ».

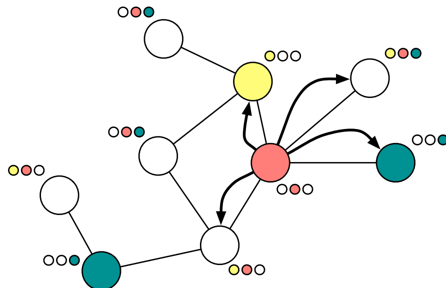
Ceci remplace le test de viabilité.

Exemple



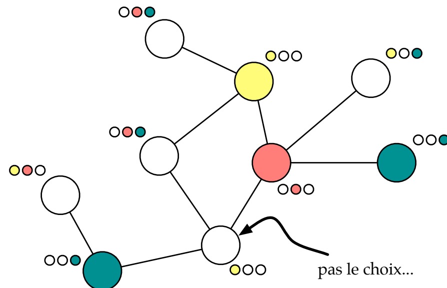
Si *possible*(x) est un singleton, on attribue à x la seule valeur possible...

Exemple



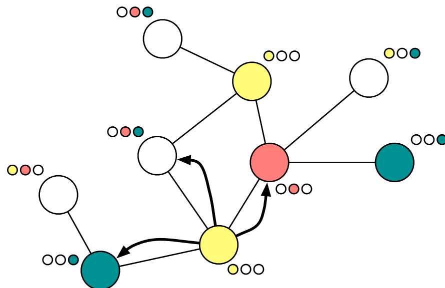
... puis on met à jour *possible*(y) pour chaque voisin y de x...

Exemple



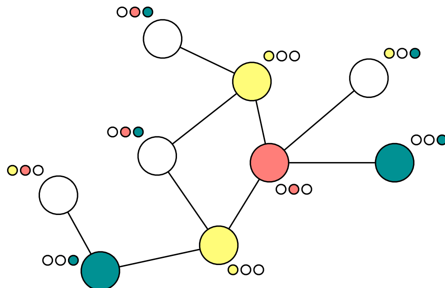
... ce qui peut faire apparaître de nouveaux singletons...

Exemple



... et on continue ainsi...

Exemple



... jusqu'à stabilisation.

Une fois cette phase de propagation terminée, il faut à nouveau choisir une variable (suivant l'heuristique MRV) et essayer ses différentes valeurs possibles.

Propagation de contraintes

Cette phase de propagation de contraintes :

- exploite la structure de graphe des contraintes ;
- ne nécessite pas l'exploration de différents choix.

Elle découvre les « conséquences immédiates » de nos décisions précédentes.

Graphe des contraintes

Les exemples précédents concernent le coloriage de graphes, mais ces idées s'appliquent au problème CSP en général.

Pour simplifier, supposons que la contrainte C est une conjonction de contraintes élémentaires $c(x, y)$ portant chacune sur deux variables.

Variables et contraintes élémentaires forment alors un graphe orienté le long des arêtes duquel se propage l'information.

Dans le cas du coloriage de graphes, les contraintes élémentaires sont de la forme $x \neq y$, et chaque contrainte élémentaire donne lieu à deux arêtes, de x vers y et de y vers x.

Cohérence d'arc

Une arête c d'une variable x vers une variable y est dite cohérente si :

> pour toute valeur $w \in possible(y)$,
> il existe une valeur $v \in possible(x)$
> telle que la contrainte $c(v, w)$ est satisfaite.

Si elle n'est pas cohérente, cela signifie qu'il faut propager de l'information de x vers y en retirant une ou plusieurs valeurs w de l'ensemble $possible(y)$.

Introduction

Le problème CSP

Interface des solveurs

Exploration naïve

Exploration efficace

L'algorithme MAC

Conclusion

Propagation de contraintes

L'algorithme de propagation de contraintes se résume alors à :

> tant qu'il existe une arête c de x vers y non cohérente,
> réduire l'ensemble $possible(y)$ pour la rendre cohérente.

Cette boucle termine en $O(nd)$ itérations.

À la sortie, toutes les arêtes sont cohérentes.

À tout moment, si l'un des $possible(x)$ devient vide, on abandonne.

Si $possible(x)$ devient un singleton $\{v\}$, on considère qu'on a étendu la valuation courante avec $x \mapsto v$. Ceci réalise déjà une partie de l'heuristique MRV.

1. Introduction
2. Un problème illustratif : CSP
3. Quelle interface pour un solveur ?
4. Exploration naïve, en 3 transparents
5. Exploration efficace : principes
6. Exploration efficace : l'algorithme MAC
7. Conclusion

Le solveur MAC

L'algorithme MAC (« maintain arc consistency ») :

- explore un arbre de valuations partielles, comme le solveur naïf ;
- rétablit la cohérence d'arc après chaque décision.

Le solveur MAC

En voici le cœur :

```
// requires : the current state is arc-consistent
private void search () throws Stop
{
  if (state.isTotal())
    client.act(state.currentValuation()) ;
  else {
    int x = state.selectUndefinedVariable() ;
    int version = state.getVersion() ;
    for (int v = 0 ; v < p.domain[x] ; v++)
      if (state.isPossible(x, v)) {
        state.set(x, v) ;
        if (restoreArcConsistency(x))
          search() ;
        state.goBackTo(version) ;
      }
  }
}
```

La méthode récursive `search` exige que chaque arête soit cohérente.

Représentation du problème

La méthode `restoreArcConsistency` doit :

- avoir accès au graphe des contraintes ;
- pouvoir propager de l'information le long d'une arête.

Introduction
Le problème CSP
Interface des solveurs
Exploration naïve
Exploration efficace
L'algorithme MAC
Conclusion

Représentation du problème

On ne représente donc plus la contrainte sous forme abstraite mais sous forme d'un graphe dont les arêtes sont des contraintes élémentaires :

```
class MACProblem {
  // The number of variables.
  public final int n ;
  // The domain of each variable.
  public final int[] domain ;
  // The outgoing edges of each variable.
  public final ArrayList<List<MACEdge>> outgoing ;
}
```

Le solveur MAC sera de type `Solver<MACProblem>`.

Représentation du problème

Chaque arête doit permettre la propagation d'information :

```
abstract class MACEdge {
  // The source and destination variables of this edge.
  public final int x, y ;
  // Exploit possible[x] to narrow down possible[y].
  abstract Prop propagate (MACSolverState state) ;
}

// propagate returns one of these three values.
enum Prop { UNCHANGED, DECREASED, EMPTIED }
```

C'est à nouveau une représentation abstraite et non pas syntaxique.

Représentation du problème

Voici par exemple un arc de x vers y correspondant à la contrainte $x \neq y$:

```
class DistinctEdge extends MACEdge {
  Prop propagate (MACSolverState state)
  { return state.isDefined(x)  ?
      state.forbid(y, state.currentValuation()[x])   :
      Prop.UNCHANGED ;
  }
}
```

Si x a pris la valeur v, alors on impose $v \notin possible(y)$.

C'est clairement nécessaire pour rendre cohérent cet arc.

C'est également suffisant pour cela :

- si x a pris la valeur v, toute autre valeur de y reste possible ;
- si x peut encore prendre plusieurs valeurs, alors toute valeur de y reste possible.

Propagation de contraintes

La propagation de contraintes s'effectue comme annoncé précédemment :

```
boolean restoreArcConsistency (IntegerQueue queue)
{
  while ( !queue.isEmpty()) {
    int x = queue.remove() ;
    for (MACEdge edge  : p.outgoing.get(x))
      switch (edge.propagate(state)) {
      case UNCHANGED :
        break ;
      case DECREASED :
        queue.add(edge.y) ; break ;
      case EMPTIED :
        return false ; // failure : no solution exists
      }
  }
  return true ; // success
}
```

La file `queue` contient les sommets où la cohérence est peut-être brisée.

L'état interne du solveur MAC

Mais qu'est-ce donc que l'objet `state` ?

C'est l'état interne du solveur.

La séparation entre algorithme (transparents précédents) et structures de données (transparents suivants) clarifie la structure du code.

Le code du solveur est aussi élégant que du « pseudo-code » !

L'état interne du solveur MAC

L'objet `state` représente les ensembles *possible*(x).

Il représente aussi la valuation partielle courante : c'est la fonction φ telle que $\varphi(x) = v$ ssi *possible*$(x) = \{v\}$.

On peut le consulter, le modifier, et revenir à une version antérieure.

Introduction
Le problème CSP
Interface des solveurs
Exploration naïve
Exploration efficace
L'algorithme MAC
Conclusion

L'état interne du solveur MAC

Voici les méthodes qu'il fournit :

```
class MACSolverState {
  // Consulting the current state.
  boolean isTotal () ;
  boolean isDefined (int x) ;
  int[] currentValuation () ;
  boolean isPossible (int x, int v) ;
  int selectUndefinedVariable () ;
  // Modifying the current state.
  void set (int x, int v) ;
  Prop forbid (int x, int v) ;
  // Versioning.
  int getVersion () ;
  void goBackTo (int version) ;
}
```

L'état interne du solveur MAC

On peut représenter *possible* par une matrice de booléens.

```
class MACSolverState {

  boolean[][] possible ;

  Prop forbid (int x, int v) {
    if ( !possible[x][v])
      return Prop.UNCHANGED ;
    possible[x][v] = false ; // x cannot take the value v
    if (isEverywhereFalse(possible[x]))
      return Prop.EMPTIED ;
    else
      return Prop.DECREASED ;
  }

}
```

La méthode `forbid` modifie l'état. (La méthode `set` également.)

Gestion du retour en arrière

Comment implémenter `getVersion` et `goBackTo` ?

Ces méthodes sont nécessaires au retour en arrière (backtracking).

Une technique consiste à défaire les modifications...

Gestion du retour en arrière

Posons qu'une action d'annulation est un objet capable de défaire un changement d'état.

```
abstract class MACSolverUndoAction {

  abstract void undo () ;

}
```

Introduction
Le problème CSP
Interface des solveurs
Exploration naïve
Exploration efficace
L'algorithme MAC
Conclusion

Gestion du retour en arrière

L'état interne contient une pile d'actions d'annulation : un « trail ».

```
class MACSolverState {
  private Stack<MACSolverUndoAction> stack ; // The trail.

  Prop forbid (int x, int v) {
    ...
    possible[x][v] = false ;
    stack.push(new MACSolverUndoAction () {
      void undo () {
        possible[x][v] = true ;
      }
    }) ;
    ...
  }
}
```

À chaque changement d'état, on crée une action d'annulation.

Gestion du retour en arrière

Le mécanisme est alors simple :

- `getVersion` permet de mémoriser la hauteur actuelle de la pile ;
- `goBackTo` dépile et exécute une série d'actions d'annulation.

Voir le poly (et ses exercices) pour les détails.

État persistant

Si `set` et `forbid` renvoyaient un nouvel état au lieu de modifier l'état courant, ce mécanisme d'annulation serait inutile.

On a le choix entre

- état non persistant et retour en arrière explicite ;
- état persistant et retour en arrière gratuit.

1 Introduction

2 Un problème illustratif : CSP

3 Quelle interface pour un solveur ?

4 Exploration naïve, en 3 transparents

5 Exploration efficace : principes

6 Exploration efficace : l'algorithme MAC

7 Conclusion

Qelques idées algorithmiques

Quelques idées à retenir :

- l'exploration se fait en profondeur d'abord ;
- il faut élaguer les branches qui n'ont aucun espoir de réussite ;
- un ordonnancement des variables bien choisi permet d'élaguer plus ;
- la propagation de contraintes est une technique d'ordonnancement.

On trouve ces idées dans la plupart des solveurs SAT et CSP modernes.

Quelques notions de programmation

Quelques idées à retenir :

- le dialogue entre solveur et client ;
- l'existence de différentes représentations des problèmes ;
- la séparation entre l'algorithme du solveur et son état ;

  Wirth : "Algorithms + Data Structures = Programs"
- la technique de retour en arrière à l'aide d'un « trail ».

Introduction

Le problème CSP

Interface des solveurs

Exploration naïve

Exploration efficace

L'algorithme MAC

Conclusion

Et bien sûr...

Lisez le poly !

*Pale le **lundi 28 avril**.*

N'oubliez pas le projet info !

Cette après-midi, TD : skyscrapers.