Qualité algorithmique (R5.A.04)
Partie I : backtracking

Marin Bougeret
LIRMM, IUT/Université de Montpellier

Outline

Admin

Les deux pages du module

- sur moodle : R5.1.04 qualité algorithmique
- sur gitlab : `https://gitlabinfo.iutmontp.univ-montp2.fr/r5.a.04-qualite-algorithmique`

Sur moodle

- salon BBB (et enregistrements de cours, disponibles environ 1H après la fin du cours)
- slides, sujets TP

Sur gitlab

- squelettes de code

Admin

Organisation :

- cours de 5 semaines ($\approx$ par semaine : 1h CM et 2*1h30)
- notation 25% interro cours, 25% contrôle continu, 50% exam final

Interro cours :

- après l'amphi de semaine n, vous aurez dans chaque groupe TD, en semaine n+1, une mini interro (facile !) sur le cours
- uniquement demi-feuille A4 manuscrite recto/verso autorisée
- ces minis sujets interros seront auto-suffisants, donc pas besoin de recopiez bêtement les définitions du cours (plutôt des exemples, vos remarques, ..)

Admin

Organisation :

- cours de 5 semaines ($\approx$ par semaine : 1h CM et 2*1h30)
- notation 25% interro cours, 25% contrôle continu, 50% exam final

Contrôle continu :

- à la responsabilité de l'enseignant TD (typiquement 1 interro, notes de TPs, ..)

Admin

Organisation :

- cours de 5 semaines ($\approx$ par semaine : 1h CM et 2*1h30)
- notation 25% interro cours, 25% contrôle continu, 50% exam final

Exam final :

- sur papier, environ 2H, feuille A4 manuscrite recto/verso autorisée

Plan du module

Thème

Résolution de problèmes combinatoires difficiles par des algorithmes de branchement.

I. Constraint Satisfaction Problem : Backtracking.
II. (Si on a le temps) Problèmes d'optimisation : Branch and Bound.
III. Un peu de magie : Programmation dynamique.

Outline

(dec)-n-Queen

- entrée : un entier $n > 0$
- solution : un placement de n reines sur un damier $n \times n$ de telle sorte qu'une reine ne puisse manger une autre
- but : décider si il existe une solution (et la retourner si oui)

schéma en direct pour n=4 (valide vs pas vailde)

Problèmes combinatoires

(count)-n-Queen

- entrée : un entier $n > 0$
- solution : un placement de n reines sur un damier $n \times n$ de telle sorte qu'une reine ne puisse manger une autre
- but : compter le nombre de solutions

schéma en direct pour n=4 (valide vs pas vailde)

Problèmes combinatoires

(list)-n-Queen

- entrée : un entier $n > 0$
- solution : un placement de n reines sur un damier $n \times n$ de telle sorte qu'une reine ne puisse manger une autre
- but : lister toutes les solutions

schéma en direct pour n=4 (valide vs pas vailde)

(dec)-Sudoku

- entrée : un tableau carré d'entiers t de taille $n \times n$ (avec $n = a^2$ où a est entier), partiellement rempli avec des entiers dans $[1, n]$
  (t est appelé une grille de sudoku)
- solution : un remplissage des cases vides avec des entiers dans $[1, n]$ tel qu'il n'y ait jamais deux fois le même entier
  - dans chacune des n lignes,
  - dans chacune des n colonnes,
  - dans chacun des n "sous carrés",
- but : décider si il existe une solution (et la retourner si oui)

(dec)-Sudoku

- entrée : un tableau carré d'entiers t de taille $n \times n$ (avec $n = a^2$ où a est entier), partiellement rempli avec des entiers dans $[1, n]$
  (t est appelé une grille de sudoku)
- solution : un remplissage des cases vides avec des entiers dans $[1, n]$ tel qu'il n'y ait jamais deux fois le même entier
  - dans chacune des n lignes,
  - dans chacune des n colonnes,
  - dans chacun des n "sous carrés",
- but : décider si il existe une solution (et la retourner si oui)

Exemple avec $n = 4$

2	1	0	4
0	3	0	0
0	2	0	0
0	0	0	0

(dec)-Sudoku

- entrée : un tableau carré d'entiers t de taille $n \times n$ (avec $n = a^2$ où a est entier), partiellement rempli avec des entiers dans $[1, n]$
  (t est appelé une grille de sudoku)
- solution : un remplissage des cases vides avec des entiers dans $[1, n]$ tel qu'il n'y ait jamais deux fois le même entier
  - dans chacune des n lignes,
  - dans chacune des n colonnes,
  - dans chacun des n "sous carrés",
- but : décider si il existe une solution (et la retourner si oui)

Une solution

2	1	3	4
4	3	1	2
1	2	4	3
3	4	2	1

(dec)-Sudoku

- entrée : un tableau carré d'entiers t de taille $n \times n$ (avec $n = a^2$ où a est entier), partiellement rempli avec des entiers dans $[1, n]$
  (t est appelé une grille de sudoku)
- solution : un remplissage des cases vides avec des entiers dans $[1, n]$ tel qu'il n'y ait jamais deux fois le même entier
  - dans chacune des n lignes,
  - dans chacune des n colonnes,
  - dans chacun des n "sous carrés",
- but : décider si il existe une solution (et la retourner si oui)

On peut aussi définir (count)-Sudoku et (list)-Sudoku.

(dec)-Vertex Cover

- entrée : un graphe $G = (V(G), E(G))$ (simple, non orienté), et un entier k
- solution : un ensemble de sommets X tel que $|X| \leq k$, et qui couvre toutes les arêtes : pour tout $e \in E(G)$, $e \cap X \neq \emptyset$
- but : décider si il existe une solution (et la retourner si oui)

schéma en direct

(dec)-Vertex Cover

- entrée : un graphe $G = (V(G), E(G))$ (simple, non orienté), et un entier k
- solution : un ensemble de sommets X tel que $|X| \leq k$, et qui couvre toutes les arêtes : pour tout $e \in E(G)$, $e \cap X \neq \emptyset$
- but : décider si il existe une solution (et la retourner si oui)

schéma en direct

On peut aussi définir (count)-Vertex Cover et (list)-Vertex Cover et ... (opt)-Vertex Cover.

(opt)-Vertex Cover

- entrée : un graphe $G = (V(G), E(G))$ (simple, non orienté), ~~et un entier k~~
- solution : un ensemble de sommets X ~~tel que $|X| \leq k$, et~~ qui couvre toutes les arêtes : pour tout $e \in E(G)$, $e \cap X \neq \emptyset$
- but : retourner une solution la plus petite possible

schéma en direct

Problèmes combinatoires : bilan

(dec)-Π

- entrée : I (ensemble d'entiers représentant l'input)
- solution : S (ensemble d'entiers représentant la solution) telle que $P_\Pi(I, S)$ vrai (S doit satisfaire une certaine propriété)
- but : décider si il existe une solution

Problèmes combinatoires : bilan

(dec)-Π

- entrée : I (ensemble d'entiers représentant l'input)
- solution : S (ensemble d'entiers représentant la solution) telle que $P_\Pi(I, S)$ vrai (S doit satisfaire une certaine propriété)
- but : décider si il existe une solution

- On a toujours 3 variantes possibles : (dec)-Π, (count)-Π, (list)-Π.
- Remarque : (dec)-Π "moins dur que" (count)-Π "moins dur que" (list)-Π.
- Pour certains problème, on peut définir une variante (opt)-Π.

Problèmes combinatoires : bilan

(dec)-Π

- entrée : I (ensemble d'entiers représentant l'input)
- solution : S (ensemble d'entiers représentant la solution) telle que $P_\Pi(I, S)$ vrai (S doit satisfaire une certaine propriété)
- but : décider si il existe une solution

- Ce type de problème capture énormément de problèmes classique appelés problèmes (d'optimisation) combinatoires.
- Ne capture pas tout non plus (entrée arrivant en "flux", avec de l'incertitude, problèmes multiobjectifs, problème du type "créer une structure de donnée qui sache faire telles opérations en telle complexité, ..).
- Dans "la vraie vie", il faut parfois itérer longtemps avec un client pour l'aider à définir ce qu'il veut résoudre.

Problèmes combinatoire difficile

Il existe un théorie (NP-complétude) qui indique que pour “presque tous” ces problèmes, on n’a pas d’espoir d’avoir un algorithme rapide (complexité polynomiale de la forme $nbOpA(I) \leq c_1|I|^{c_2}$ avec c_i constantes).

- Les algorithmes "naturels" qui nous viennent à l’esprit sont en général polynomiaux, et donc faux ! (ex pour Vertex Cover : prendre le sommet de plus haut degré, recommencer)
- Approche suivie : on abandonne la rapidité (complexité polynomiale), et on va donc s’autoriser des approches "brute force" ($\approx$ tester toutes solutions)

But du module

Aborder des techniques (backtracking, dynamic programming) pour écrire des algorithmes type "brute force" les plus rapides possibles.

Outline

Modélisation

(dec)-n-Queen

- entrée : un entier $n > 0$
- solution : un placement de n reines sur un damier $n \times n$ de telle sorte qu'une reine ne puisse manger une autre
- but : décider si il existe une solution (et la retourner si oui)

But de la modélisation

- Traduire ce qu'est une solution en terme de variables x_i devant prendre des valeurs dans un certain domaine $dom(x_i)$ et vérifier certaines contraintes..
- .. afin que le brute force puisse tester toutes ces valeurs (ou mieux ..)

Modélisation

(dec)-n-Queen : modélisation $V1$

Solution :

- variables : pour $0 \leq i \leq n-1$, $p_i = (l_i, c_i)$ avec l_i,c_i dans $[0, n-1]$ (représente la position de la i-ème reine)
- contraintes : pour tout $i < j$, $C_{ij}(p_i, p_j)$ (les reines en p_i et p_j ne peuvent pas se manger)

On appelle tuple toute liste de p_i.
Exemple : tuple (p_0, p_5, p_7) avec $p_0 = .., p_5 = .., p_7 = ...$

- Bof : on autorise plein de tuples “stupides" : on laisse par exemple la possibilité de les mettre toute sur la première ligne.
- Autrement dit, le domaine des variables est trop grand ($dom(p_i) = [1, n] \times [1, n]$).
- Même si on a pas encore vu comment écrire des brute force, avoir des domaines grand n'aidera pas !

(dec)-n-Queen : modélisation $V2$

Solution :

- variables : pour $0 \leq i \leq n-1$, $c_i \in [0, n-1]$ (représente la colone de la i-ème reine, que l'on impose sur la ligne i)
- contraintes : pour tout $i < j$, $C_{ij}(c_i, c_j)$ (les reines en p_i et p_j ne peuvent pas se manger)

schéma

Résolution V0 : générer et tester

```
Solution generateAndTest(int n){
  candidat = (0,..,0);
  res = null;
  trouve = false;
  while(!trouve && hasNext(candidat)){
     trouve=test(candidat);//retourne vrai ssi
         candidat est une solution
     if(trouve)
       res=candidat;
     else
       candidat = next(candidat);
  }
  return res;
}
```

- next() peut ne pas être évident à écrire
- complexite $\geq$ # candidats $= n^n$

Résolution V0 : générer et tester

Pourquoi est-ce si mauvais : on s'acharne à essayer d'étendre des solutions partielles "non valides".

Exemple pour $n = 5$, toute la tranche de solutions suivante est inutile :

- $(0, 0, 0, 0, 0)$
- $\ldots$
- $\ldots$
- $(0, 0, n-1, n-1, n-1)$

Résolution V0 : générer et tester

Et même plus généralement pour tout n, toute tranche

- $(c_0, \ldots, c_k, 0, 0, 0, .., 0)$
- $\ldots$
- $\ldots$
- $(c_0, \ldots, c_k, n-1, n-1, n-1, .., n-1)$

est inutile si la "solution partielle" $(c_0, \ldots, c_k)$ viole déjà une contrainte.

Résolution V0 : générer et tester

Et même plus généralement pour tout n, toute tranche

- $(c_0, \ldots, c_k, 0, 0, 0, .., 0)$
- $\ldots$
- $\ldots$
- $(c_0, \ldots, c_k, n-1, n-1, n-1, .., n-1)$

est inutile si la "solution partielle" $(c_0, \ldots, c_k)$ viole déjà une contrainte.

Conclusion

- On voudrait se restreindre seulement aux solutions partielles valides.
- Une solution élégante pour faire cela (et bien plus) : la récursivité !

Résolution V0 : générer et tester : traduite en en récursif

Définitions (provisoires)

- Une solution partielle s est un tuple $(c_0, \ldots, c_k)$ avec $k < n$ et $c_i \in [0, n-1]$.

Pour $n = 4$, $s_1 = (0, 1)$ et $s_2 = (1, 3)$ sont des solutions partielles.

Définition clef

Etant donné une solution partielle $s = (c_0, \ldots, c_k)$, on dit qu'une solution s^* étend s si $s^* = (c_0, \ldots, c_k, \star, ., \star)$.

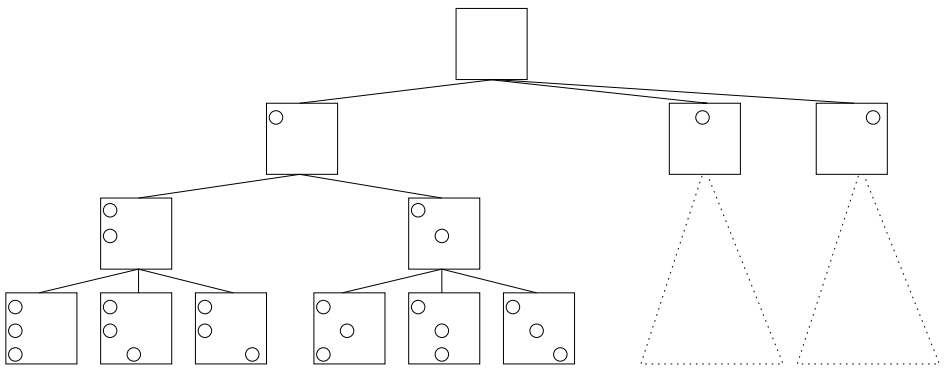
La solution $s^* = (1, 3, 0, 2)$ n'étend pas s_1, mais étend s_2.

Résolution V0 : générer et tester : traduite en en récursif

```
Solution backTrackV0(int n, PartialSol s){
  //prérequis : s=(c0,..,ck) est une solutions
     partielle (pas forcément valide)
  //action : si il existe une solution s* qui
  //  étend s, alors en retourne une, sinon
     retourne null
  //rmk : ne modifie pas s

  ..




}
```

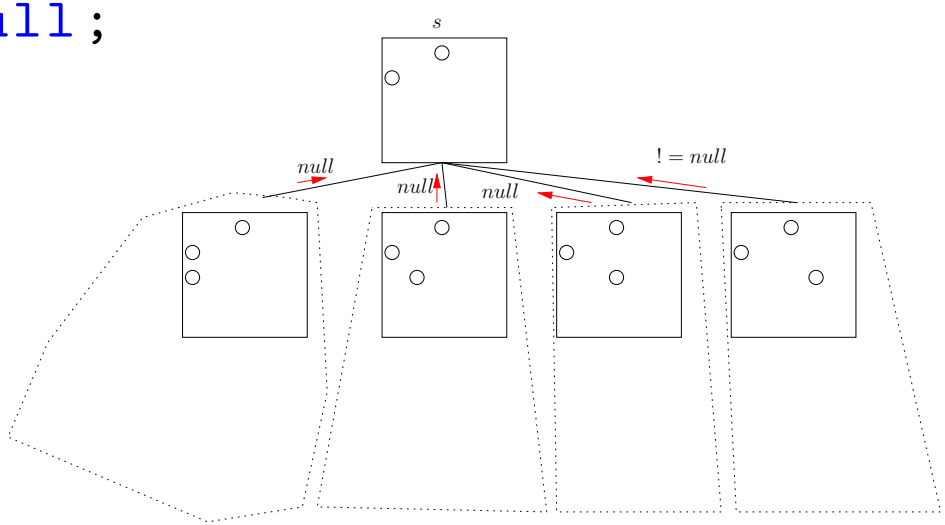
Résolution V0 : générer et tester : traduite en en récursif

```
Solution backTrackV0(int n, PartialSol s){
  if(s.isFullSolution()){ //si k==n-1
    if(test(s)) return s else return null;
  }
  int i = s.getUnAffectedVar(); //i ==k+1
  for(int c=0;c<n;c++){
    s =  s + (ci <- c); //s=(c0,..,ck,c)
    res = backTrackV0(n,s);
    if(res!=null) return res;
    s = s - (ci <- c); //s=(c0,..,ck)
  }
  return null;
}
```



Résolution V0 : générer et tester : traduite en en récursif

```
Solution backTrackV0(int n, PartialSol s){
  if(s.isFullSolution()){ //si k==n-1
    if(test(s)) return s else return null;
  }
  int i = s.getUnAffectedVar(); //i ==k+1
  for(int c=0;c<n;c++){
    s =  s + (ci <- c); //s=(c0,..,ck,c)
    res = backTrackV0(n,s);
    if(res!=null) return res;
    s = s - (ci <- c); //s=(c0,..,ck)
  }
  return null;
}
```



Résolution V1 : restrictions aux solutions partielles valides

Définition

Une solution partielle s est valide ssi $(c_0, \ldots, c_k)$ ne viole aucune contrainte.

- Si une solution partielle s est valide, et que $s.isFullSolution()$, alors s est bien une solution à notre problème.
- Pour $n = 4$, $s_1 = (0, 1)$ est une solution partielle non valide, et $s_2 = (1, 3)$ est une solution partielle valide.

On va ré-écrire l'algorithme récursif précédent en se restreignant aux solutions partielles valides.

Résolution V1 : restrictions aux solutions partielles valides

```
Solution backTrackV1(int n, PartialSol s){
  //prérequis : s=(c0,..,ck) est une solutions
     partielle *valide*
  //action : si il existe une solution s* qui
  //  étend s, alors en retourne une, sinon
     retourne null
  //rmk : ne modifie pas s

  ..

}
```

Résolution V1 : restrictions aux solutions partielles valides

```
Solution backTrackV1(int n, PartialSol s){
  if(s.isFullSolution()){ //si k==n-1
    return s;
  int i = s.getUnAffectedVar(); //i ==k+1
  for(int c=0;c<n;c++){
    if(s.checkNewVal(ci,c)){//s est toujours
       valide
      s =  s + (ci <- c); //s=(c0,..,ck,c)
      res = backTrackV1(n,s);
      if(res!=null) return res;
      s = s - (ci <- c); //s=(c0,..,ck)
    }
  }
  return null;
}
```

Reprendre schéma précédent en pensant à s valide ou non.

Yeah !

- C'est déjà beaucoup mieux : expérience en direct (running time, et nombre d'appels pour version compter le nombre de solutions)
- La récurrence aide beaucoup (imaginer le "next()" si on devait se restreindre aux solutions partielles valides en itératif)
- Et ça n'est pas fini :)

On va encore améliorer en introduisant en paramètres des domaines restreints pour les variables.

Motivation pour ajouter les domaines

Idée

- Au début, le domaine de chaque variable est $dom(c_i) = [0, n-1]$.
- Cependant, on pourrait au fur et à mesure éliminer des valeurs inutiles des domaines.
- Par exemple, si je place ma première reine en 0, je peux propager les contraintes pour réduire mes domaines :
  - enlever 0 des $dom(c_j)$ pour $j > 0$..
  - et même mieux, enlever $\{0, j\}$ pour $j > 0$

Conséquences

- plus besoin de faire s.checkNewVal(ci,c) (on sait que toute valeur $c \in dom(c_i)$ redonne bien une solution partielle valide)
- et même beaucoup plus : on peut découvrir en avance que certains domaines deviennent vides (et donc qu'il n'y a pas de solution)

Motivation pour ajouter les domaines

Exemple pour $n = 6$, backTrackV1 va :

- affécter $c_0, .., c_4$ à $(0, 2, 4, 1, 3)$
- puis retourner null car aucune valeur possible pour c_5

schéma branche gauche de l'arbre qui correspond à $(0, 2, 4, 1, 3)$
schéma de la même branche, avec les domaines à chaque étape

Motivation pour ajouter les domaines

Si l'on avait propagé les contraintes à chaque fois : backTrackV2 pourrait :

- affecter $c_0 \leftarrow 0$, puis propager et réduire D
- affecter $c_1 \leftarrow 2$, puis propager et réduire D
- affecter $c_2 \leftarrow 4$, puis propager et réduire D
- affecter $c_3 \leftarrow 1$, puis propager et réduire D
- on réalise que $dom(c_5) = \emptyset$: STOP, retourner null

schéma de la même branche, avec les domaines à chaque étape

Motivation pour ajouter les domaines

Nous allons donc écrire backTrackV2(n,s,D) qui prend en paramètre des domaines.. ayant une certaine propriété.

Résolution V2 : FCC

Définition (provisoire)

Etant donné une solution partielle $s = (c_0, \ldots, c_k)$, et $D = \{D_{k+1}, \ldots, D_{n-1}\}$, on dit que (s, D) est

- Forward Checking Consistant (FCC) si s valide et pour tout $\ell \in [k+1, n-1]$, pour $c \in D_\ell$, placer reine en (l, c) ne viole aucune contrainte avec s
- non trivial si pour tout $\ell \in [k+1, n-1]$, $D_\ell \neq \emptyset$

Etant donné une solution $s^* = (c_1, \ldots, c_k, c_{k+1}, \ldots, c_{n-1})$ qui étend s, on dit que s^* respecte D si pour tout $\ell \in [k+1, n-1]$, $c_\ell \in D_\ell$.

Résolution V2 : FCC

Définition (provisoire)

Etant donné une solution partielle $s = (c_0, \ldots, c_k)$, et $D = \{D_{k+1}, \ldots, D_{n-1}\}$, on dit que (s, D) est

- Forward Checking Consistant (FCC) si s valide et pour tout $\ell \in [k+1, n-1]$, pour $c \in D_\ell$, placer reine en (l, c) ne viole aucune contrainte avec s
- non trivial si pour tout $\ell \in [k+1, n-1]$, $D_\ell \neq \emptyset$

Etant donné une solution $s^* = (c_1, \ldots, c_k, c_{k+1}, \ldots, c_{n-1})$ qui étend s, on dit que s^* respecte D si pour tout $\ell \in [k+1, n-1]$, $c_\ell \in D_\ell$.

Pour $n = 4$ et $s = (0, 3)$:

- le domaine $D_2 = \{1\}, D_3 = \{2\}$ est FCC pour s
- le domaine $D_2 = \{1\}, D_3 = \{1, 2\}$ n'est pas FCC pour s

Résolution V2 : FCC

- Quand (s, D) est FCC, plus besoin de faire s.checkNewVal(ci,c).
- Par contre, quand on choisit une valeur c pour une nouvelle variable c_{k+1}, il faut maintenant faire appel à s.propagateConstraints(D,k+1,c) qui fait du forward checking pour enlever de D les valeurs nécessaires :
- toute valeur c' pour une variable c_ℓ (avec $\ell > k+1$) qui violerait la contrainte avec (c_{k+1}, c) doit être enlevée

Résolution V2 : FCC

- Quand (s, D) est FCC, plus besoin de faire s.checkNewVal(ci,c).
- Par contre, quand on choisit une valeur c pour une nouvelle variable c_{k+1}, il faut maintenant faire appel à s.propagateConstraints(D,k+1,c) qui fait du forward checking pour enlever de D les valeurs nécessaires :
- toute valeur c' pour une variable c_ℓ (avec $\ell > k+1$) qui violerait la contrainte avec (c_{k+1}, c) doit être enlevée

Remarque sur FCC et foward checking

Finalement, l'idée naturelle de

- "quand je choisis une colonne, je propage l'information en enlevant les bonnes colonnes pour les variables suivantes".

Se traduit par

- Je fais du forward checking (FC) pour maintenir mon entrée (s, D) Forward Checking Consistant (FCC)

Résolution V2 avec FCC

```
Solution backTrackV2(int n, PartialSol s, D){
  //prérequis : s=(c0,..,ck) est une solutions
     partielle (s,D) est FCC et non trivial
  //action : si il existe une solution s* qui
  //  étend s et respecte D, alors en retourne
     une, sinon retourne null
  //rmk : ne modifie pas s

  ..



}
```

Résolution V2 avec FCC

```
Solution backTrackV2(int n, PartialSol s, D){
  //prérequis : s=(c0,..,ck) est une solutions
     partielle (s,D) est FCC et non trivial
  //action : si il existe une solution s* qui
  //  étend s et respecte D, alors en retourne
     une, sinon retourne null
  //rmk : ne modifie pas s

  ..



}
```

Pour des raisons techniques, on retournera toujours une solution indépendante (en mémoire) de s.

Résolution V2 avec FCC

```
Solution backTrackV2(int n, PartialSol s, D)
 if(s.isFullSolution()){ //si k==n-1
    return new Solpartielle(s); // et pas ret s
 int i = s.getUnAffectedVar(); //i ==k+1
 for(c in D.get(i)){
   Dmodif = deepCopy(D);
   ok = s.propagateConstraints(Dmodif,i,c);
   // vrai si aucun D_l n'est devenu vide
   if(ok){
     s =  s + (ci <- c); //s=(c0,..,ck,c)
     res = backTrackV2(n,s,Dmodif);
        //on est content que backTrackV2 ne
            modifie pas s, et que res soit indep
            en mémoire de s
     s = s - (ci <- c); //s=(c0,..,ck);
     if(res!=null) return res;
   }
 }
 return null;
```

Conclusion n-queens

Le backTrackV2 est une base saine d'algorithme de backtrack.

Conclusion n-queens

Le backTrackV2 est une base saine d'algorithme de backtrack.

Aller plus loin

- On peut choisir dans quel ordre on instancie les variables : (pas forcement $c_0, c_1, ..$)
  - pose la question du critère pour choisir sa prochaine variable (cf exemples suivants)
  - implique qu'une solution partielle n'est pas $(c_0, \ldots, c_k)$ mais $(c_{i_0}, \ldots, c_{i_k})$ (cf définitions générales à suivre)
- Une fois la variable choisie, on peut choisir dans quel ordre parcourir les valeurs possibles (ne change rien si il n'existe pas de solution)

Conclusion n-queens

Aller plus loin

- On peut renforcer encore les propriétés du domaine :
  - Exemple : pour $n = 4$ et $s = (0, 4)$ $D_2 = \{2\}, D_3 = \{3\}$ on pourrait réaliser qu'il faut couper la branche car contrainte entre ligne 2 et ligne 3 sera forcément violée.
  - Il existe de très nombreuses propriétés ("arc-consistency", ..).
  - Plus la propriété demandée est forte, plus on coupe de branches, mais plus on passe de temps à faire de la propagation (mesurer avec un "profiler").
  - Tout cela est discuté en profondeur dans des cours de "Constraint Satisfaction Problems (CSP)".

Conclusion n-queens

Aller plus loin

- On peut renforcer encore les propriétés du domaine :
  - Exemple : pour $n = 4$ et $s = (0, 4)$ $D_2 = \{2\}, D_3 = \{3\}$ on pourrait réaliser qu'il faut couper la branche car contrainte entre ligne 2 et ligne 3 sera forcément violée.
  - Il existe de très nombreuses propriétés ("arc-consistency", ..).
  - Plus la propriété demandée est forte, plus on coupe de branches, mais plus on passe de temps à faire de la propagation (mesurer avec un "profiler").
  - Tout cela est discuté en profondeur dans des cours de "Constraint Satisfaction Problems (CSP)".

What's next

- Définitions générales au format CSP
- Etude des problèmes du Sudoku et Vertex Cover.

Outline

CSP (Constraint Satisfaction Problem)

(dec)-CSP

- Entrée : un ensemble de variables $X = (x_1, \ldots, x_n)$, un domaine $D = \{dom(x_i)\}$ (entiers) pour chaque variable, et un ensemble $\mathcal{C}$ de contraintes.
- Solution : ..
- But : décider si il existe une solution.

Contrainte

- Une contrainte C est une fonction d'un sous ensemble de variable dans $\{true, false\}$.
  - Exemple $C_0(x_3, x_5, x_6) =$ "retourner vrai ssi $x_3 + x_5 < x_6$".
- Un tuple de valeurs v satisfait C ssi $C(v) = true$.
  - Exemple $(5, 5, 11)$ satisfait C_0.
- On note $var(C_0) = \{x_3, x_5, x_6\}$.

CSP (Constraint Satisfaction Problem)

(dec)-CSP

- Entrée : un ensemble de variables $X = (x_1, \ldots, x_n)$, un domaine $D = \{dom(x_i)\}$ (entiers) pour chaque variable, et un ensemble $\mathcal{C}$ de contraintes.
- Solution : ..
- But : décider si il existe une solution.

Solution

- Une solution partielle s est un ensemble $\{(x_{i_1}, v_{i_1}), \ldots, (x_{i_k}, v_{i_k})\}$ avec x_{i_ℓ} distincts et $v_{i_\ell} \in dom(x_{i_\ell})$.
- On note $var(s) = \{x_{i_1}, \ldots, x_{i_k}\}$ les variables de s et $val(x_{i_\ell}, s) = v_{i_\ell}$.
- Pour toute contrainte C telle que $var(C) \subseteq var(s)$, on dit que s satisfait C si le tuple v de valeurs de s correspondant aux variables de C satisfait C.

CSP (Constraint Satisfaction Problem)

(dec)-CSP

- Entrée : un ensemble de variables $X = (x_1, \ldots, x_n)$, un domaine $D = \{dom(x_i)\}$ (entiers) pour chaque variable, et un ensemble $\mathcal{C}$ de contraintes.
- Solution : ..
- But : décider si il existe une solution.

Solution

- Une solution partielle s est un ensemble $\{(x_{i_1}, v_{i_1}), \ldots, (x_{i_k}, v_{i_k})\}$ avec x_{i_ℓ} distincts et $v_{i_\ell} \in dom(x_{i_\ell})$.
- On note $var(s) = \{x_{i_1}, \ldots, x_{i_k}\}$ les variables de s et $val(x_{i_\ell}, s) = v_{i_\ell}$.
- Pour toute contrainte C telle que $var(C) \subseteq var(s)$, on dit que s satisfait C si le tuple v de valeurs de s correspondant aux variables de C satisfait C.
- Ex : $s = \{(x_2, 2), (x_3, 5), (x_5, 5), (x_6, 11), (x_8, 4)\}$ satisfait C_0.

CSP (Constraint Satisfaction Problem)

(dec)-CSP

- Entrée : un ensemble de variables $X = (x_1, \ldots, x_n)$, un domaine $D = \{dom(x_i)\}$ (entiers) pour chaque variable, et un ensemble $\mathcal{C}$ de contraintes.
- Solution : ..
- But : décider si il existe une solution.

Solution

- Une solution partielle est valide si pour tout C telle que $var(C) \subseteq var(s)$, s satisfait C.
- Une solution s^* est une solution partielle valide s^* avec $var(s^*) = X$.

CSP (Constraint Satisfaction Problem)

Exemple 1

- Entrée : $X = \{x_1, x_2, x_3, x_4\}$, $\mathcal{C} = \{C_1, C_2, C_3\}$ et D avec
  - $C_1(x_1, x_2) : x_1 <= x_2$
  - $C_2(x_1, x_2, x_3) : allDifferent(x_1, x_2, x_3)$ //vrai ssi ils sont tous différents
  - $C_3(x_1, x_3, x_4) : x_3 + x_4 <= x_1$
  - $D(x_1) = \{4, 8\}$, $D(x_2) = \{4, 5\}$, $D(x_3) = \{1, 4, 9\}$, $D(x_4) = \{2, 6\}$

- $s = \{(x_1, 8), (x_2, 4)\}$ est une solution partielle non valide.
- $s = \{(x_1, 8), (x_3, 8)\}$ est une solution partielle valide :
  - il faut "pour tout C telle que $var(C) \subseteq var(s)$, s satisfait C" : ok pour C_2 car $var(C_2) \nsubseteq var(s)$
  - donc valide même si "aucun espoir d'être étendue à une solution"

Ex2 : n-queens modélisation $V2$

- Entrée : $X = \{x_1, \ldots, x_n\}$, $\mathcal{C} = \{C_{ij}\}$ et D avec
  - $C_{ij}(x_i, x_j)$: vrai ssi reine en (i, x_i) ne peut pas manger reine en (j, x_j)
  - $D(x_i) = [0, n-1]$

Graphe d'un CSP

A tout CSP on peut associer un hypergraphe $G = (V, \mathcal{H})$:

- on crée un sommet par variable
- pour chaque contrainte C on ajoute une hyperarête $var(C)$

Schéma avec ex1

- Ce graphe est une bonne façon de se représenter un CSP et de s'imaginer les algorithmes de propagation.
- Si toutes les contraintes sont d'arité 2 ($|var(C)| = 2$), alors on obtient un graphe classique (et quel est le graphe associé aux n-queen ?).
- Il existe de nombreuses techniques avancées selon la structure de ce graphe (pas abordées ici).

Modèle de base de backtracking pour un CSP

```
Solution backTrack(partialSol s, domain D){
//prérequis :
 -s solution partielle
 -D domaine pour les variables restantes
 -s et D vérifient une certaine propriété P (ex
    FCC .. cf après)
 //action :
si il existe une solution s* qui étend s et
   respecte D la retourne, sinon ret null
```

Il faut définir dans le cas général "s^* qui étend s et respecte D".

Etant donné une solution partielle s, des domaines D pour les variables hors de s, une solution s^* :

- étend s si pour tout $x \in var(s)$, $val(x, s^*) = val(x, s)$.
- respecte D si pour tout $x \notin var(s)$, $val(x, s^*) \in dom(x)$.

Modèle de base de backtracking pour un CSP

```
Solution backTrack(partialSol s, domain D){
//prérequis :
 -s solution partielle
 -D domaine pour les variables restantes
 -s et D vérifient une certaine propriété P (ex
    FCC .. cf après)
 //action :
si il existe une solution s* qui étend s et
   respecte D la retourne, sinon ret null
```

- On va choisir P comme on veut selon les problèmes.
- Au moment d'essayer une valeur : $s = s + (x \leftarrow v)$, on fera
- boolean ok = s.propagateConstraints(D,x,v) ;
- "s.propagateConstraints(D,x,v))" va modifier D en propageant "plus ou moins fort" :
  - si la propagation mène à nouveau D qui satisfait encore la propriété P, parfait ! (retourne true)

Modèle de base de backtracking pour un CSP

```
Solution backTrack(partialSol s, domain D){
//prérequis :
 -s solution partielle
 -D domaine pour les variables restantes
 -s et D vérifient une certaine propriété P (ex
    FCC .. cf après)
 //action :
si il existe une solution s* qui étend s et
   respecte D la retourne, sinon ret null
```

- On va choisir P comme on veut selon les problèmes.
- Au moment d'essayer une valeur : $s = s + (x \leftarrow v)$, on fera
- boolean ok = s.propagateConstraints(D,x,v) ;
- "s.propagateConstraints(D,x,v))" va modifier D en propageant "plus ou moins fort" :
  - sinon, on a un problème (retourne faux), inutile d'essayer la valeur v pour x

Modèle de base de backtracking pour un CSP

```
Solution backTrack(partialSol s, domain D){
  if(s.isFullSolution())
    return new partialSol(s);
  int x = s.getUnAffectedVar(); //be smart
  for(v in D.get(i)){ //be smart
    Dmodif = deepCopy(D);
    ok = s.propagateConstraints(D,x,v);
    if(ok){
      s =  s + ( x  <-  v);
      res = backTrack(s,Dmodif);
      s = s - ( x  <- v); //on annule
      if(res!=null) return res;
    }
  }
  return null;
}
```

Une propriété classique (et la propagation qui va avec)

Etant donné une solution partielle s, un domaine D pour les variables restantes, on dit que (s, D) est

- Forward Checking Consistant (FCC) si s est valide, et pour toute contrainte C telle que C n'a plus qu'une variable (x) pas dans s, toutes les valeurs de $dom(x)$ sont utilisables pour rendre C vraie.
- Arc Consistant (AC) si .. (on oublie pour ce cours)

Une propriété classique (et la propagation qui va avec)

Exemple 1

- $C_1(x_1, x_2) : x_1 <= x_2$
- $C_2(x_1, x_2, x_3) : allDifferent(x_1, x_2, x_3)$ //vrai ssi ils sont tous différents
- $C_3(x_1, x_3, x_4) : x_3 + x_4 <= x_1$
- $D(x_1) = \{4, 8\}$, $D(x_2) = \{4, 5\}$, $D(x_3) = \{1, 4, 9\}$, $D(x_4) = \{2, 6\}$

Avec $s = \{(x_1, 4), (x_2, 5)\}$:

- D' avec $D'(x_3) = \{1, 4, 7\}$ et $D'(x_4) = \{2, 6\}$ n'est pas FCC
- D'' avec $D''(x_3) = \{1, 7\}$ et $D''(x_4) = \{2, 6\}$ est FCC (même si on sait que 6 inutilisable pour x_4)

Une propriété classique (et la propagation qui va avec)

Propriété agréable de FCC

Si l'on a un (s, D) FCC, alors pour toute $x \notin var(s)$ et tout $v \in dom(x)$, $s' = s + x \leftarrow v$ est encore une solution partielle valide (mais $(s', D - x)$ n'est plus forcément FCC, c'est pour cela qu'on refait un forward checking).

Une propriété classique (et la propagation qui va avec)

Propriété agréable de FCC

Si l'on a un (s, D) FCC, alors pour toute $x \notin var(s)$ et tout $v \in dom(x)$, $s' = s + x \leftarrow v$ est encore une solution partielle valide (mais $(s', D - x)$ n'est plus forcément FCC, c'est pour cela qu'on refait un forward checking).

Remarque sur FCC et foward checking : toujours valide

Finalement, l'idée naturelle de

- "quand je choisis une colonne, je propage l'information en enlevant les bonnes colonnes pour les variables suivantes".

Se traduit par

- Je fais du forward checking (FC) pour maintenir mon entrée (s, D) Forward Checking Consistant (FCC)

What's next ?

- On va maintenant étudier deux autres problèmes (Sudoku et Vertex Cover) pour voir ce qu'on peut faire d'intelligent.
- En TP vous coderez ces algorithmes de backtracking.

Outline

Modélisation CSP

(dec)-Sudoku : modélisation

- entrée : un tableau carré d'entiers t de taille $n \times n$ (avec $n = a^2$ où a est entier), partiellement rempli avec des entiers dans $[1, n]$
  (t est appelé une grille de sudoku)
- variables : x_{ij} pour avec i, j dans $[0, n-1]$ et $dom(x_{ij}) = [0, n-1]$
- contraintes :
  - pour toute ligne i, $AllDiff(x_{i0}, \ldots, x_{i(n-1)})$ (tous les entiers sur ligne i différents)
  - pour toute colonne j, $AllDiff(x_{0j}, \ldots, x_{(n-1)j})$ (tous les entiers sur colonne j différents)
  - pour chacun des n sous-carrés, $AllDiff(\ldots)$ (tous les entiers du sous-carré différents)

Brancher sur les valeurs d'une variable va donc correspondre ici à essayer toutes les valeurs du domaine d'une case.

Solutions partielles intéressantes

- Pour les n-queen, la première version satisfaisante était backTrackV1(n,s) où s était une solution partielle valide.
- Ici, se restreindre aussi aux solutions partielles valides :
  - est déjà bien : cela évite de prendre en paramètre un s qui aurait par exemple une ligne complète avec une répétition
  - mais pas si bien : un s avec ligne incomplète mais comportant déjà une répétition peut être partielle valide !

- On aurait le même problème avec toute contrainte "globale" (qui utilise toutes les (ou beaucoup de) variables.
- La nature des contraintes "allDiff" va permettre de corriger ce problème : on re-modélise le problème avec selon des contraintes binaires.
- (En vrai, garder les contraintes globales, mais en faisant de l'arc consistance..)

Modélisation CSP

(dec)-Sudoku : modélisation V2

- entrée : un tableau carré d'entiers t de taille $n \times n$ (avec $n = a^2$ où a est entier), partiellement rempli avec des entiers dans $[1, n]$
  (t est appelé une grille de sudoku)
- variables : x_{ij} pour avec i, j dans $[0, n-1]$ et $dom(x_{ij}) = [0, n-1]$
- contraintes :
  - pour toute ligne i, pour tout $j_1 < j_2$, $x_{ij_1} \neq x_{ij_2}$ (tous les entiers sur ligne i différents)
  - pour toute colonne j, pour tout $i_1 < i_2$, $x_{i_1 j} \neq x_{i_2 j}$ (tous les entiers sur colonne j différents)
  - pour chacun des n sous-carrés, .. (tous les entiers du sous-carré différents)

Brancher sur les valeurs d'une variable va donc correspondre ici à essayer toutes les valeurs du domaine d'une case.

Solutions partielles intéressantes

- Maintenant, une solution partielle valide ne comporte aucune répétition, parfait !
- A nouveau deux versions possibles : sans les domaines, ou avec.. mieux car :
  - on va s'imposer un (s, D) FCC et faire du forward checking (permet de réaliser en avance que certaines domaines deviennent vides)
  - et on va voir un nouvel avantage d'avoir des domaines !

Résolution V1

```
Solution backTrackSudoV1(int n, PartialSol s){
  //s solution partielle valide
  if(s.isFullSolution()){
    return new PartialSol(s);
  int numCase = s.getUnAffectedVar();
  for(int c=0;c<n;c++){
    if(s.checkNewVal(numcase,c)){//si s reste
       valide
      s =  s + (numcase <- c);
      res = backTrackSudoV1(n,s);
      s = s - (numcase <- c);
      if(res!=null) return res;
  }
  return null;
}
```

Utilisation de domaines

Forward checking

Quand je choisis la valeur v pour la case (i, j), enlever v des domaines de toutes les variables :

- de la ligne i
- de la colonne j
- du sous-carré de (i, j)

Utilisation de domaines

Forward checking

Quand je choisis la valeur v pour la case (i, j), enlever v des domaines de toutes les variables :

- de la ligne i
- de la colonne j
- du sous-carré de (i, j)

Rappel : Propriété agréable de FCC

Si l'on a un (s, D) FCC, alors pour toute $x \notin var(s)$ et tout $v \in dom(x)$, $s' = s + x \leftarrow v$ est encore une solution partielle valide (mais $(s', D - x)$ n'est plus forcément FCC, c'est pour cela qu'on refait un forward checking).

Utilisation de domaines

Forward checking

Quand je choisis la valeur v pour la case (i, j), enlever v des domaines de toutes les variables :

- de la ligne i
- de la colonne j
- du sous-carré de (i, j)

Remarque sur FCC et foward checking : toujours valide

A nouveau, le FC correspond à l'idée naturelle de "minimum" de propagation d'information suite à une prise de décision.

Backtracking sudoku

```
Solution backTrackSudoV2(partialSol s, domain D)
    {
//prérequis :
 -(s,D) est FCC et non trivial
 //action :
 si il existe une solution s* qui étend s et
    respecte D la retourne, sinon ret null

 .. il y a juste à appliquer le backTrack
 général vu avant !
```

Séléction de la prochaine variable : MRV

On avait vu que l'on pouvait améliorer le schéma général à plusieurs endroit, dont la sélection de la prochaine variable.

Quand vous faites un sudoku à la main (beurk), comment choisissez vous votre prochaine case ?

- si il reste une case avec un seul choix restant, on la remplit (puis à la main aussi on propage !)
- sinon, on choisit une case avec le minimum de choix restants, et on les essaye (à la main aussi on branche !)

MRV (Minimum Remaining Value)

Une heuristique classique de choix de prochaine variable : choisir $x \notin var(s)$ telle que $dom(x)$ est le plus petit.

Voilà encore un avantage à avoir introduit des domaines.

Backtracking avec MRV

```
Solution backTrackSudoV3(partialSol s, domain D)
//prérequis :
 -(s,D) est FCC et non trivial
 //action :
 si il existe une solution s* qui étend s et
     respecte D la retourne, sinon ret null
 //stratégie :
 utilise MRV pour choisir sa prochaine case
 .. il y a juste à appliquer le backTrack
 général vu avant, et à recoder getUnAffectedVar
     ():

 int i = s.getUnAffectedVar(); //be smart : use
     MRV
```

Remarques d'implémentation

Pour éviter de recalculer à chaque tour une variable de domaine minimum, on peut

- niveau 1 : maintenir un tableau *tailleDom* des tailles des domaines
- niveau 2 : utiliser une priorityQueue dont les éléments sont les cases restantes, et les priorités les tailles de domaines

Plus généralement, c'est souvent une bonne idée d'ajouter dans s des attributs de "décoration" (comme la priorityqueue) que l'on maintient à jour et qui accélèrent les calculs.

Conclusions sudoku

Bilan

- On a re-modélisé le problème pour que la notion de "solution partielle valide" soit satisfaisante.
- N'oubliez pas que, dans la vraie vie, on peut garder les contraintes globales type "allDiff" (mais on fait alors de l'arc consistance qui permet non seulement de détecter les solutions partielles qui violeront à coup sûr des contraintes, mais même plus).
- On a vu un nouvel avantage des domaines : selection MRV.

What's next ?

On va étudier le problème Vertex Cover pour découvrir une nouvelle heuristique de choix de variable : MRV+Highest Degree.

Outline

Vertex Cover

(dec)-Vertex Cover (rappel)

- entrée : un graphe $G = (V(G), E(G))$ (simple, non orienté), entier k
- solution : un ensemble de sommets X tel que $|X| \leq k$, et qui couvre toutes les arêtes : pour tout $e \in E(G)$, $e \cap X \neq \emptyset$
- but : décider si il existe une solution (et la retourner si oui)

dessin

Vertex Cover

(dec)-Vertex Cover : modélisation

- entrée : un graphe $G = (V(G), E(G))$ (simple, non orienté), entier k
- variables : pour tout $i \in V(G)$, x_i avec $dom(x_i) = \{0, 1\}$ (1 quand on prend le sommet i)
- contraintes :
  - pour tout $e = \{i, j\} \in E(G)$, $x_i + x_j \geq 1$
  - $\sum_{i \in V(G)} x_i \leq k$
- but : décider si il existe une solution (et la retourner si oui)

Brancher sur les valeurs d'une variable va donc correspondre ici à essayer de "prendre ou ne pas prendre" un sommet.

Solutions partielles intéressantes

Comme pour Sudoku, posons nous la question : quelles sont les solutions partielles s intéressantes (pas vouées à l'échec) ?

Ici, une solution partielle $s = \{(x_{i_1}, v_{i_1}), \ldots, (x_{i_k}, v_{i_k})\}$ correspond donc à un choix (prendre ou non) déjà fait pour le sous ensemble de sommets $var(s)$. Conditions naturelles pour être intéressant :

1. s ne contient pas une arête non couverte (il n'existe pas $e = \{i, j\} \subseteq var(s)$ telle que $val(s, i) + val(s, j) = 0$)
2. s contient au plus k sommets ($nbSommets(s) = \sum_{i \in var(s)} val(s, i) \leq k$)

- La première condition correspond à la notion habituelle de "solution partielle valide", mais pas la deuxième (on peut donc avoir une solution partielle valide qui viole la deuxième).
- Cette fois-ci, pas de re-modélisation efficace.

Solutions partielles intéressantes

Que faire ?

- Solution 1 : se définir une notion à nous :
  - (s, D) est $\underline{\text{FCC}^+}$ si (s, D) est FCC, et que $nbSommets(s) \leq k$
- Solution 2 : L'Arc Consistance (AC) implique les deux propriétés
  - "dans la vraie vie", on choisirait cela, mais ce que l'on fait ici est plus simple à décrire et quasi équivalent pour Vertex Cover

Solutions partielles intéressantes

Que faire ?

- Solution 1 : se définir une notion à nous :
  - (s, D) est $\underline{\text{FCC}^+}$ si (s, D) est FCC, et que $nbSommets(s) \leq k$
- Solution 2 : L'Arc Consistance (AC) implique les deux propriétés
  - "dans la vraie vie", on choisirait cela, mais ce que l'on fait ici est plus simple à décrire et quasi équivalent pour Vertex Cover

Comment implémenter la Solution 1

- D'habitude :
  - on s'impose en entrée du bactrack un (s, D) qui est FCC, et
  - quand on choisit une valeur pour une nouvelle variable, on fait du "forward checking" comme propagation (pour assurer que le nouveau (s', D') passé au backtrack est FCC)

Solutions partielles intéressantes

Que faire ?

- Solution 1 : se définir une notion à nous :
  - (s, D) est $\underline{\text{FCC}^+}$ si (s, D) est FCC, et que $nbSommets(s) \leq k$
- Solution 2 : L'Arc Consistance (AC) implique les deux propriétés
  - "dans la vraie vie", on choisirait cela, mais ce que l'on fait ici est plus simple à décrire et quasi équivalent pour Vertex Cover

Comment implémenter la Solution 1

- Ici :
  - on s'impose en entrée du backtrack un (s, D) qui est FCC^+ , et
  - quand on choisit une valeur pour une nouvelle variable, il faudra donc faire un peu plus que du "forward checking" comme propagation (pour assurer que le nouveau (s', D') passé au backtrack est FCC^+)

Propagation

Comment assurer FCC : foward checking

Quand je choisis la valeur v pour un sommet i :

- si $v = 0$ (ne pas prendre), forcer à prendre les voisins de i :
  - pour tout $j \in N(i) \setminus var(s)$, enlever 0 de $dom(x_j)$, retourner false si un des $dom(x_j)$ devient vide
- si $v = 1$ (prendre), rien à faire pour le forward checking

schéma

Propagation

Comment assurer FCC$^+$

Quand je choisis la valeur v pour un sommet i :

- si $v = 0$ (ne pas prendre), forcer à prendre les voisins de i :
  - pour tout $j \in N(i)$, enlever 0 de $dom(x_j)$, retourner false si un des $dom(x_j)$ devient vide
  - vérifier que AC2 est toujours vraie (car on vient de forcer plein de sommets..)
- si $v = 1$ (prendre)
- vérifier que AC2 est toujours vraie (car on vient de prendre un sommet de plus..)

schéma

Propagation

Comment assurer FCC$^+$

Quand je choisis la valeur v pour un sommet i :

- si $v = 0$ (ne pas prendre), forcer à prendre les voisins de i :
  - pour tout $j \in N(i) \setminus var(s)$, enlever 0 de $dom(x_j)$, retourner false si un des $dom(x_j)$ devient vide
  - vérifier que AC2 est toujours vraie (car on vient de forcer plein de sommets..)
- si $v = 1$ (prendre)
- vérifier que AC2 est toujours vraie (car on vient de prendre un sommet de plus..)

schéma

Remarque sur FCC et foward checking : toujours valide

A nouveau, le FC correspond à l'idée naturelle de "minimum" de propagation d'information suite à une prise de décision (quand je ne prends pas un sommet, prendre tous ses voisins).

Backtracking Vertex Cover V1

```
Solution backTrackVC-V1(partialSol s, domain D){
//prérequis :
 -(s,D) FCC+ et non trivial
 //action :
 si il existe une solution s* qui étend s et
    respecte D la retourne, sinon ret null

 .. il y a juste à appliquer le backTrack
 général vu avant !
```

Séléction de la prochaine variable : MRV

Que donne le MRV ici ?

- si il y a des variables avec un domaine de taille 1, on va commencer par celle là (c'est bien !)
- sinon, pas très intéressant, tous les domaines sont de taille 2

Highest-Degree

Une heuristique classique de choix de prochaine variable : choisir $x \notin var(s)$ telle que x apparaît dans le plus de contraintes pas encore satisfaites.

Remarques

- Le Highest-Degree correspond ici à choisir un $i \in V(G(s))$ dont le degré (dans $G(s)$) est le plus grand possible.
- Pourquoi est-ce prometteur : dans la branche où l'on ne choisit pas i, cela force beaucoup de sommets à 1.
- On va faire MRV+highest-degree

Backtracking avec MRV+highest-degree

```
Solution backTrackVC-V2(partialSol s, domain D)
//prérequis :
 -(s,D) FCC+ et non trivial
 //action :
 si il existe une solution s* qui étend s et
     respecte D la retourne, sinon ret null
 //stratégie :
 utilise MRV+highest-degree pour choisir sa
     prochaine case
 .. il y a juste à appliquer le backTrack
 général vu avant, et à recoder getUnAffectedVar
     ()

 int i = s.getUnAffectedVar(); //be smart : use
     MRV+highest-degree
```

Conclusions

Bilan de la démarche face à un nouveau problème

- Trouver une modélisation CSP (voir la solution comme "une succession de décisions" peut aider à trouver une première modélisation).
- Réfléchir à quelles propriétés on impose :
  - à ses solutions partielles
  - son domaine (en pensant à la propagation qui va avec)

Faites des tests !

- Pour comparer différentes modélisations ou versions de backtracking :
  - par exemple lancer sur 500 instances random et sommer les temps de résolution
  - éventuellement utiliser la version (count) du problème pour gommer l'effet "il y a une solution facile que même les algorithmes peu efficaces trouvent".
- Utiliser un profiler pr mesurer le tps passé dans la propagation.

Conclusions

Pour une résolution ultra efficace :

- savoir que selon le type de CSP, il existe des techniques spécialisées, par exemple :
  - CSP avec seulement des (in)égalités : Integer Linear Programming.
  - CSP binaire (toutes contraintes à 2 variables).
  - CSP dont domaines des variables sont binaires.
  - CSP dont l'hypergraphe à une structure particulière (arbre, ..)
- se renseigner (cours CSP) sur les techniques de propagation avancées
- pensez à utiliser des bibliothèques CSP

Des références

- Constraint propagation. Christian Bessiere.
  `https://www.lirmm.fr/~bessiere/Site/stock/TR06020.pdf`
- Page de Christophe Lecoutre.
  `http://www.cril.univ-artois.fr/~lecoutre/`
- Notes de cours de Christophe Dürr.
  `http://www-desir.lip6.fr/~durrc/Iut/Notes580.pdf`