PPC

Daphne Kany, Victor Spitzer

Janvier 2022

Table des matières

1 Introduction

L'objectif de ce projet est d'implémenter un solveur générique de CSP à contraintes binaires et variables entières. Le solveur que nous présentons est construit comme suit :

- Une classe objet pour la modélisation, avec un Wrapper pour faciliter cette modélisation, en particulier en intégrant une binarisation des contraintes
- Une classe objet pour la résolution par Backtrack, Graph Based Backjump ou Conflict Directed Backjump
- Dans cette même classe, les algorithmes de consistance Forward Check, AC3 et AC4.

Nous démontrons son efficacité sur les problèmes Reines, Colorabilité, en mettant en évidence les différences parmi les méthodes de résolution ainsi que l'impact sur la résolution des ordres dans lequel traiter les variables et leurs valeurs.

Nous avons choisi la programmation orientée objet pour l'implémentation du solveur. Cela permet :

- Une modélisation efficace du problème par une structure adaptée au CSP (variables, domaines, contraintes)
- La gestion intelligente des paramètres lors de la résolution par un passage en attribut de certains d'entre eux
- La création de plusieurs instances d'un même problème par lesquelles on peut comparer les méthodes de résolution et les heuristiques

2 Classe Problem : objet pour la modélisation

La modélisation de notre classe **Problem** a été pensée pour résoudre des problèmes binaires. Un objet de la classe **Problem** est défini par trois attributs :

- Un entier **Variables** qui représente le nombre de variables du problème. Chaque variable sera alors représentée par un entier commpris entre 0 et **Variables**-1
- Un tableau **Domain** de taille **Variables**, tel que **Domain**[i]=[valeurs possibles pour la variable i]
- Un dictionnaire **Constraint** tel que si deux variables i et j ont une ou plusieurs contraintes entre elles, **Constraints**[(i,j)]=[couples de valeurs qui respectent cette contraintes]

A cela, on ajoute une fonction de classe **all-diff** qui permet d'implémenter la contrainte fréquente exigeant que toutes les variables prennent des valeurs différentes, ainsi qu'une fonction **add-constraint** qui permet d'ajouter une nouvelle contrainte entre deux variable, possiblement de façon implicite (ex : i!=j).

Cette modélisation binaire permet une implémentation rapide et simple de nombreux problèmes courants tels que Reines ou Coloration. Néanmoins, on peut facilement rencontrer des problèmes lorsque les contraintes rencontrées ne sont initialement pas binaires. C'est le cas pour le problème de l'hexagone magique, considéré ici dans le cas d'un coté de taille 3. Les deux moyens souvent utilisés pour binariser un problème sont Hidden Variables Encoding et Dual Encoding.

Dans le cas de Hidden Variables Encoding, on a une variable par case de l'hexagone + une variable par contrainte, ie ici par ligne, soit un total de 19+15=34 variables. Mais là où la complexité spatiale explose, c'est pour traiter les domaines des variables liées aux contraintes. En effet, ces domaines ont pour taille le nombre de combinaison de trois, quatre ou cinq nombres deux à deux différents compris entre 1 et 19 et dont la somme vaut 38, soit respectivement 189, 3996 et 41006. A cela s'ajoute la taille du dictionnaire des contraintes, puisque chaque variable initiale est contrainte par la valeur des lignes sur lesquelles elle se trouve. L'instanciation du problème reste possible, mais sa résoution n'aboutissait pas en un temps raisonnable.

La technique de dual encoding comprend moins de variables : une variable par ligne, donc 15 dans notre cas. La définition des domaines reste la même que pour Hidden Variables Encoding. Le dictionnaire des contraintes, par contre, explose : si la ligne i et la ligne j ont un point commun, alors il faut que les valeurs prises sur ce point soient identiques. Considérant la taille des domaines, cette contrainte avait une taille bien trop importante, et la modélisation n'aboutissait pas.

Un axe d'amélioration possible pour l'implémentation binaire du problème de l'hexagone magique serait de travailler sur les symétries des variables. En effet, les rotations ne changent pas le problème rencontré.

3 Classe ToSolve : objet pour la résolution

On a présenté la classe d'objet **Problem** créé pour effectuer la modélisation d'un problème. On explique désormais le fonctionnement d'une seconde classe d'objet **ToSolve**, dont les attributs et méthodes sont directement liés aux choix de résolution du problème et à l'exécution de celle-ci. Ainsi, un unique objet **Problem** est créé pour modéliser un problème, et celui-ci n'est pas destiné à être modifié ; tandis qu'à chaque résolution un nouvel objet **ToSolve** est produit, et on pourra modifier ces attributs lors de la résolution. Ce fonctionnement à deux classes d'objet distinctes permet donc de modifier l'instance du problème à résoudre sans avoir à encoder le problème à chaque résolution.

Les attributs d'un objet instancié sont :

— Un objet **Problem** qui modélise le problème à résoudre.

— Un tableau de taille fixée, de valeurs initialisées à None, qui est mis à jour à chaque affectation de valeur à une variable et représente la solution construite par l'algorithme de résolution

— Deux listes représentant l'ordre dans lequel traiter les variables et les valeurs.

Les méthodes à retenir pour cette classe sont les algorithmes d'arc-consistance AC3 et AC4, ainsi que ceux de résolution Backtrack, Graph Based Backjump et Conflict Directed Backjump. Tous les arguments de ces algorithmes sont passés en attributs de l'objet instancié donc ils n'ont pas d'autres paramètres que l'objet lui-même. Un détail important sur l'implémentation est le traitement dans un ordre fixé des variables et des valeurs : cela signifie que pour deux variables i et j tel que $i < j$ selon l'ordre des variables, une valeur sera toujours affectée à i avant j. De même, pour deux valeurs v_1 et v_2 tel que $v_1 < v_2$ selon l'ordre des valeurs, v_1 sera toujours affectée avant v_2. Cela permet les simplifications décrites ci-dessous.

Imposer un ordre sur les variables permet une simplification de la résolution dès l'instanciation de l'objet. En effet, une symétrie commune à tous les CSP est d'imposer pour deux variables i et j le même ensemble de contraintes à permutation prêt pour les tuples (i, j) et (j, i). En réalité, seule la connaissance des contraintes pour un seul de ces tuples suffit. C'est pourquoi on supprime dès l'instanciantion les contraintes (i, j) si $i > j$ selon l'ordre imposé sur les variables. On a ainsi réduit la dimension de l'instance à résoudre.

Toujours concernant l'ordre sur les variables, on modifie les contraintes et les domaines du problème pour qu'ils correspondent à cet ordre : c'est à dire qu'on affecte à l'indice de chaque variable sa position dans l'ordre imposé, et on modifie les contraintes et les domaines en conséquence pour que le problème reste le même à une permutation près. On traite l'ordre sur les valeurs en triant les domaines de chaque variable selon l'ordre sur les valeurs imposé. De cette manière, il suffit d'implémenter les algorithmes d'arc-consistance et de résolution selon un ordre canonique des variables et des valeurs, et pour retrouver la solution il suffit d'appliquer la permutation inverse de l'ordre sur les variables. Ainsi il n'est pas nécessaire de se référer à ces ordres à chaque affectation de valeur à une variable.

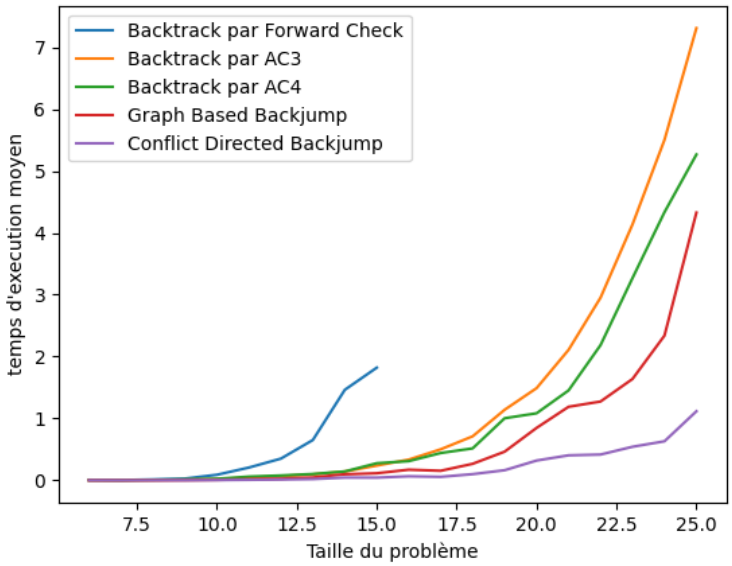
Nous présentons désormais une comparaison des méthodes implémentées, avec des expérimentations sur différents problèmes.

4 Benchmarks

Resolution de Reines et comparaison des résolutions

On modélise le problème Reines comme vu en cours, c'est à dire avec une variable par colonnes de l'échiquier et des valeurs représentant les cases sur chaque colonne. On ne propose pas d'heuristique pour la résolution de ce problème : on essaie cependant des ordres aléatoires pour diversifier les simulations. On remarque cependant que l'ordre canonique des valeurs et variables est sous-optimal, et que certains ordres sont bien plus performants que d'autres.

Le problème est suffisamment simple pour être lancé plusieurs fois, et accroître la taille du problème permet de différencier les performances des solveurs. On lance pour chaque taille $n \in [|6, 25|]$ une résolution pour 30 ordres aléatoires choisis de manière aléatoire et on compare la moyenne des temps de résolution.



Comparaison du temps de résolution moyen pour le problème Reines

Une telle comparaison confirme ce qu'on pouvait déjà supposer : les techniques de lookback sont plus efficaces qu'un simple Backtrack. Parmi les Backtrack, utiliser des algorithmes d'arc-consistance est bien plus intéressant car on exclus plus rapidement certains choix en regardant toute la propagation à chaque branchement (un simple Forward Check provoque un explosion du temps de calcul). En comparaison, l'algorithme Conflict Directed Backjump et Graph Based Backjump semblent plus performants, et le premier plus encore que le second, ce qui est attendu car il cherche de plus grands sauts à chaque branchement.

Pour chacun de ces algorithmes, on observe une corrélation quasi linéaire entre le temps d'exécution et le nombre de branchements : Backtrack revient à l'état précédent, Graph Based revient

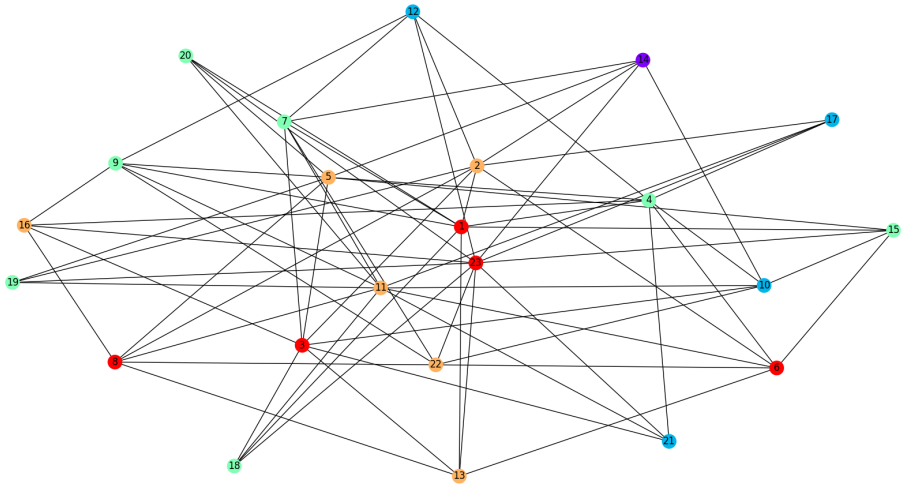
à un des ancêtres sans identifier si c'est celui qui a provoqué la contradiction, contrairement à Conflict Directed. Par la suite, on ne travaillera plus qu'avec Conflict Directed qui est la méthode qui fonctionne le mieux en moyenne.

Resolution de problèmes de colorabilité

On modélise le problème de colorabilité comme un problème de décision : Soit un graphe $G = (V, E)$ et un entier k, existe-t-il une coloration de G à k couleurs ? En représentant chaque sommet par des variables, à valeurs dans $\{1, ..., k\}$ les couleurs possibles, et les contraintes telles que pour une arête (i, j) du graphe, la variable i doit être à valeur différente de la variable j. On revient au problème d'optimisation en incrémentant k si on ne trouve pas de solution, jusqu'à en obtenir une.

On impose une heuristique naïve pour la résolution : on développera cet aspect dans la section d'approfondissement. L'ordre sur les valeurs n'a pas d'importance, car les permutations de couleurs sont équivalentes. On propose d'ordonner les variables dans l'ordre décroissant de leur nombre de voisin : en effet, plus une variable a de voisins, plus grand devient la possibilité qu'il existe un conflit. On cherche ainsi à éviter des sauts inutilement trop grand lors des branchements.

On propose par exemple la résolution obtenue en moins d'une seconde du graphe de Mycielski 4 (myciel4), de solution optimale 5 couleurs :



Coloration du graphe de 4-Mycielski à 5 couleurs

5 Approfondissements

On a cherché à améliorer les heuristiques employées pour la résolution des problèmes de colorabilité car l'accroissement de la taille des instances provoquaient une explosion du temps de calcul. On a mentionné précédemment que le temps de calcul était directement lié au nombre de saut requis pour déterminer l'existence ou non d'une solution. On cherche donc à minimiser ce nombre de saut, ainsi que le temps nécessaire pour revenir à la variable depuis laquelle on a opéré ce saut. On réduit le domaine de chaque variable pour limiter les redondances des tests d'affectation liées à l'équivalence des colorations à permutation près.

Notons N_i le nombre de voisin du sommet associé à la variable i. On opère les changements suivants afin de réduire la dimension du problème.

Réduction du domaine de chaque variable

Par défaut, le domaine de chaque variable est $\{1, ..., k\}$. Cependant, comme le sommet associé à la variable a N_i voisins, il est certain qu'on peut lui associer une couleur dans $\{1, ..., N_i + 1\}$ car au moins une couleur parmi elles sera différente de celle de ses N_i voisins. On peut modifier les domaines tel que :

$$\forall iV \quad D_i = \{1, ... \min(k, N_i + 1)\}$$

Réduction des contraintes

Puisqu'on a éliminé la possibilité de certaines valeurs pour certaines variables, on supprime les contraintes correspondantes pour ces valeurs à ne plus considérer :

$$\forall i \in V,\, v_i \notin D_i \Rightarrow (v_i, q) \notin C(i, j) \vee (q, v_i) \notin C(j, i), \quad \forall j \in V, \forall q \in \{1, ..., k\}$$

Ordre sur les valeurs

Avec cette réduction de domaine, seuls les sommets avec un grand nombre de voisins ont accès aux couleurs proche de k. D'une certaine manière, on a supprimé la symétrie sur les valeurs car il n'y a plus d'équivalence des colorations à permutation près puisque tous les sommets n'ont pas accès à toutes les valeurs. On impose donc un ordre décroissant des valeurs en affectant en priorité les valeurs les plus grandes car la probabilité est plus faible qu'elle soit dans le domaine d'un sommet voisin, donc on minimise les conflits :

$$k <_{val} k - 1 <_{val} ... <_{val} 1$$

Ordre sur les variables

On ordonne les variables par un algorithme glouton, initialisé à l'état disposant du plus grand nombre de voisin, et qui à chaque itération ajoute dans une liste ordonnée le sommet voisin des sommets déjà choisis de plus grand N_i si $N_i \geq k$. Si on ne trouve pas de tel sommet, on ajoute à la liste celui, voisin ou non, avec le plus grand N_i :

Algorithm 1 Ordre sur les variables

Input: Graphe $G = (V, E)$, nombre de voisins N_i pour chaque sommet, nombre de couleurs k.

```
X = {}
X ← arg max{N_i; i ∈ V}
while |X| < |V| do
    Y = {y; (y, x) ∈ E, y ∉ X, x ∈ X, N_y ≥ k}
    if Y = {} then
        X ← arg max{N_i; i ∈ V, i ∉ X}
    else
        X ← arg max{N_i; i ∈ Y}
    end if
end while
return X
```

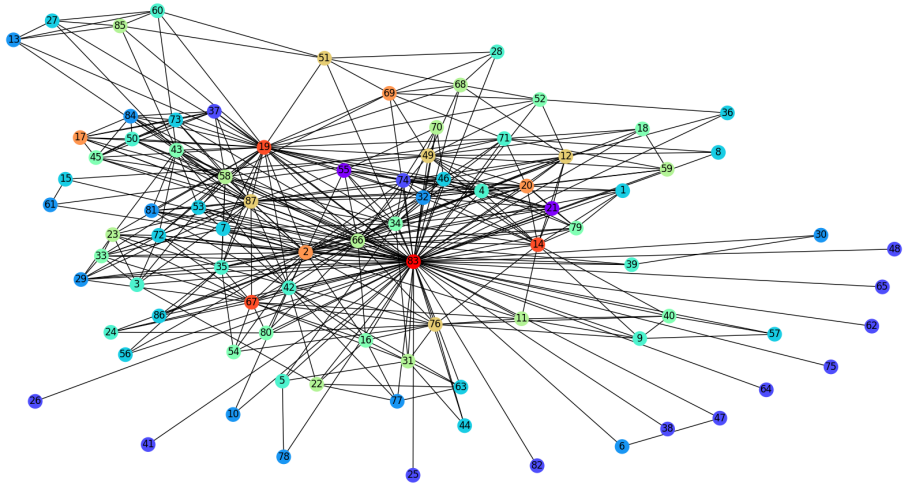
L'argument pour un ordre selon le nombre de voisin a déjà été donné : les variables les plus susceptibles d'entrer en conflit sont celles de plus grande densité. Ici on exige également que le sommet choisi soit voisin : de cette manière, en cas de conflit, on minimise les chances que le saut soit trop grand et nécessite de chercher de nouveau une couleur pour des sommets qui n'ont rien à voir avec ce conflit : un exemple extrême serait un graphe non connexe où on colore plusieurs composantes connexes à la fois, et à chaque saut dans une composante on recommence la même coloration pour les autres. La condition $N_i \geq k$ impose que les sommets avec plus de couleurs disponibles que de sommets soient traités en dernier : en effet de tels sommets ne produisent pas de conflit puisqu'il existera toujours une couleur qu'on peut leur affecter, on minimise donc encore les opérations inutiles après chaque saut.

Première variable fixée

Une fois obtenu cet ordre sur les variables, on connaît celle qui sera la première traitée par l'algorithme de résolution. Si il n'existe pas de solution pour un k donné, on ne souhaite pas que l'algorithme affecte toutes les couleurs possibles à cette première variable : si il existe une solution, il en existera toujours un telle que ce premier sommet ait la couleur k par permutation, donc on impose :

$$D_{0_{var}} = \{k\}$$

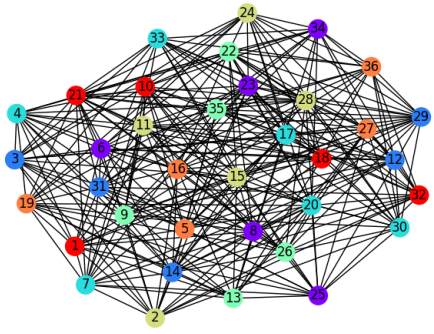
On présente la résolution d'un problème de grande taille par l'application de cette réduction de problème et des heuristiques sur l'ordre des variables et des valeurs : une coloration optimale en 11 couleurs du graphe de connexion des protagoniste du David Copperfield de Charles Dickens (David), à 87 sommets et 406 arêtes :



Coloration du graphe de David à 11 couleurs

On obtient cette solution en 10 minutes et deux millions de sauts (!) mais il est à noter que c'est la vérification qu'il n'existe pas de solution à 10 couleurs qui prend autant de temps : autrement dit le problème d'optimisation est difficile à résoudre, mais vérifier qu'il existe bien une 11-coloration ne l'est pas. On observera d'ailleurs que pour tous les problème de coloration rencontrés, il est plus long de vérifier qu'une couleur n'est pas réalisable plutôt que le contraire.

Afin de mieux identifier la contribution de chacun des changements précédents, on compare le temps de résolution de la coloration du graphe Reine pour $n = 6$, de solution optimale 7, à 36 sommets et 290 arêtes :



Coloration du graphe de 6-Reines à 7 couleurs

On obtient les temps d'exécution suivants :

Couleurs testées	$k = 6$	$k = 7$
Tous changements	13.41 s	28.48 s
Seulement l'ordre des variables	86.37 s	29.98 s
Tous changements sauf l'ordre des variables	107.08 s	8.34 s
Sans aucun changement	619.49 s	10.96 s

On observe donc que ces heuristiques offrent une meilleure performance pour déterminer qu'il n'existe pas de solution, mais il est intéressant de voir que changer l'ordre des variables augmente le temps de calcul lorsqu'on teste une coloration pour laquelle il en existe une. On peut l'expliquer ainsi : on compare une heuristique naïve qui traite les variables dans l'ordre décroissant de leur densité à une heuristique qui impose que le sommet suivant soit par ailleurs voisin. Ainsi pour la seconde, on traite plus tardivement des sommets de grande densité et sources de conflits : cette faiblesse de l'heuristique est largement compensée par sa détection précoce de conflit dans les cas non réalisables (voir $k = 6$).

Conclusion

On a proposé une solveur PPC qui intègre toutes les connaissances qui nous ont été enseignées lors de ce cours, et établi des comparaisons entre différentes méthodes. Plusieurs modélisation de problème ont été proposé, ainsi que des solutions à des difficultés pratiques comme la recherche d'heuristiques ou la suppression de certaines symétries.