

TER Stratégies parallèles de solveurs de contraintes

Adam FERREIRA, Clément TURCAT

Université de Nantes

1 Introduction

Ce TER s'intitule « Stratégies parallèles de solveurs de contraintes », au cours duquel nous avons étudié le comportement de différentes stratégies de résolution de contraintes à l'aide de solveurs parallèles. Afin de cerner les tenants et les aboutissants de ce TER, il est nécessaire de définir le cadre dans lequel il se trouve. Pour cela nous allons présenter l'idée générale à travers un état de l'art de la programmation par contraintes et les problèmes de satisfaction de contraintes. Nous allons également présenter *POSL*, un langage permettant une approche parallèle de la programmation par contrainte, ainsi que les fonctionnalités de celui-ci que nous avons exploité dans le cadre de ce TER. Enfin nous présenterons nos travaux ainsi que les différents résultats qu'ils ont pu produire.

2 Les problèmes de satisfaction de contraintes

Nous présentons dans cette partie les problèmes de satisfaction de contraintes, ou CSP. Les problèmes CSP consistent en trois composantes, X , D et C :

X est un ensemble de variables, $\{X_1, \dots, X_n\}$.

D est un ensemble de domaines, $\{D_1, \dots, D_n\}$, un pour chaque variables.

C est un ensemble de contraintes sur les variables.

Chaque domaine D_i contient un ensemble de valeurs possibles $\{v_1, \dots, v_n\}$. Pour résoudre un CSP, nous avons besoin de définir la notion d'espace de recherche et la notion de solution. Une état dans un espace de recherche est défini par l'affectation de valeurs à certaines ou toutes les variables. Une solution est une affectation qui respecte toutes les contraintes et où toutes les variables ont été affectées.

L'avantage d'une telle définition des problèmes est d'éliminer des portions de l'espace de recherche en retirant les combinaisons de valeurs et variables qui violent les contraintes.

Deux exemples de problèmes bien connus de la littérature et qui ont été



utilisées dans ce TER sont le problème des *N-QUEENS* et le problème des *Social Golfers*.

Le problème des *N-QUEENS* consiste à placer N reines sur un échiquier de dimension $N \times N$ de sorte qu'aucune reine ne puisse attaquer une autre. Une reine peut se déplacer sur toute sa ligne, colonne ou ses diagonales.

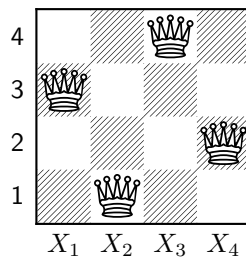


FIGURE 1: L'affectation $\{X_1 = 3, X_2 = 1, X_3 = 4, X_4 = 2\}$ est une solution du problème 4-QUEENS

Le problème des *Social Golfers* est généralisé au problème de décision suivant : est-il possible d'affecter $g \times p$ golfeurs dans g groupes de p golfeurs pendant w semaines de manière à ce que deux golfeurs quelconques ne se retrouvent dans le même groupe jamais plus d'une fois. Une instance du problème est désigné par le triplet $g - p - w$.

Formalement dans un espace de recherche, un algorithme ne fait que chercher. Avec les CSP, il est possible soit de chercher, soit de réaliser de la propagation de contraintes (ou filtrage) : en utilisant les contraintes il est possible de réduire le nombre de valeurs possibles pour une variable. Si on considère chaque variable comme un nœud dans un graphe et chaque contrainte binaire comme une arête, alors l'idée est d'appliquer le principe de consistance locale, c'est à dire d'éliminer dans le graphe les valeurs qui causent des inconsistances. Il y a principalement deux types de consistances : la consistance de nœud et la consistance par arc.

La consistance de nœud consiste à retirer d'une variable toutes les valeurs pour lesquelles au moins une contrainte est impossible à satisfaire.

La consistance par arcs consiste à supprimer les valeurs qui n'ont pas de support. Une valeur a_i d'un domaine D_i a un support a_j dans le domaine

D_j si le couple (a_i, a_j) est autorisé par la contrainte liant les variables i et j .

Une autre approche pour la résolution des CSP est la recherche locale qui donne de très bon résultats et qui est favorisée dans le cadre de ce TER. La recherche locale consiste à suivre un chemin depuis une affectation initiale complète jusqu'à atteindre une solution ou une impasse. Sur ce chemin on se déplace de voisins en voisins, c'est à dire en appliquant une opération aux affectations.



3 Un langage orienté parallèle pour modéliser des solveurs de contraintes

POSL (prononcé « puzzle ») pour Parallel-Oriented Solver Language est un langage pour la construction de méta-heuristiques interconnectées travaillant en parallèle[1]. Avec POSL on s'intéresse principalement aux CSP (problèmes de satisfaction de contraintes), où il s'agit de trouver (si elle existe) une solution à un problème donné. Bien qu'il existe de nombreuses techniques permettant de résoudre efficacement ces problèmes (méta-heuristiques, arbres de recherches, etc...), la croissance exponentielle de leur espace de recherche pose des limites sur la taille des instances traitables en un temps raisonnable. L'avancée technologique et l'apparition de machines massivement multi-cœurs et de supers calculateurs a ouvert la porte à de nouvelles manières d'approcher les problèmes combinatoires. Notamment d'abord par la parallélisation de méta-heuristiques existante (lancement en parallèle de stratégies de résolution indépendante), puis ensuite par l'introduction de communications d'informations entre les différents solveurs aussi appelées *coopérations*.

Différents outils proposent des approches pour la modélisation et la résolution de problèmes parallèles. L'une des principales motivations derrière POSL a été de fournir un outil innovant permettant de regrouper et de manipuler facilement des modèles à la manière de systèmes existants. On pensera notamment à *Hyperion*[2] un système Java pour méta et hyper-heuristiques fournissant des patrons génériques pour une variété d'algorithmes de recherche locale permettant la possibilité de réutiliser le code source. POSL vise à exploiter ces possibilités à travers l'agencement de composants indépendants (*operation module*, *open channel*, *computation strategy* et les *communication channels* ou *subscription*) pour la construction de prototypes réutilisables de solveurs et de leur protocoles de

communications. POSL est conçu pour obtenir simplement des ensembles de solveurs différents à exécuter en parallèle.

Le cœur de POSL réside dans la possibilité de modéliser facilement des stratégies de communication indépendantes de l'implémentation des solveurs, donnant la possibilité d'étudier facilement les processus de résolution et leurs résultats. Les solveurs peuvent être connectés à d'autres solveurs, en fonction de la structure de leurs *open channels* qui va jouer le rôle de port. Templar[3] propose un système pour générer des algorithmes changeant de composants prédéfinis via une hyper-heuristique, de la même manière POSL offre la possibilité d'échanger des comportements entre les différents solveurs en plus de simple informations. En recevant un *operation module* (via communication), les solveurs vont être capable d'évoluer (remplacer leur opération module par celui reçu) dynamiquement pendant leur exécution en fonction d'un contexte parallèle.

4 Construction de solveurs parallèles avec POSL

POSL permet de d'implémenter des schémas de stratégies génériques à l'aide de composants pré-construits et de *modules*. Une *computation strategy* est construite comme étant un assemblage de composants élémentaires : *operation module*, *open channel*. Un *operation module* est l'élément le plus basique d'un code POSL, il reçoit une entrée, exécute un algorithme et produit une sortie. On peut trouver différentes nature d'*operation module* comme par exemple une configuration, un ensemble de configurations (typiquement un voisinage), un ensemble de valeurs (vecteurs de coûts) etc...



POSL permet de regrouper les associations (opérations) entre les *operation modules* à l'intérieur d'autres modules : les *compound modules*. La syntaxe de création d'un *compound module* dans POSL s'exprime sous forme des balises : `<S> ... </S>` entre lesquelles nous allons trouver d'autres *compound modules* comme par exemple une opération entre plusieurs *operation modules* comme illustré en figure 2.

Dans le cadre d'un algorithme de recherche locale, POSL définit les types d'*operation modules* suivants :

- `OM_S` : Génère une configuration S .
- `OM_V` : Définit le voisinage $V(S)$ (produit un ensemble de configurations).
- `OM_SS` : Sélectionne $s^* \in V(S)$.

s^* : l'étoile est utilisé dans l'optimisation pour appeler une SOLUTION

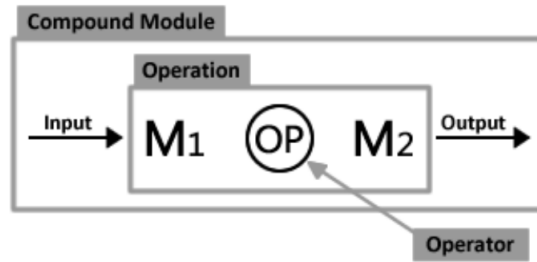


FIGURE 2: Illustration du principe de compound module. (extrait de l'article[1])

— **OM_D** : Évalue le critère d'acceptation pour S^*

C'est avec ces types de l'on va pouvoir définir le « squelette » d'une stratégie, indépendamment du fonctionnement interne des *operation modules* (mais pas indépendamment de leur types d'entrée - sortie!). La figure 3 illustre le code de la *computation strategy* « ma stratégie ». Il s'agit ici de la description d'un algorithme simple de recherche locale. L'opérateur séquentiel **OP.**|-> prend la sortie d'un module (*operation module*, *compound module* ou *open channel*) pour la donner en entrée à un autre. Cette relation est représentée par l'opération préfixée **OP.**|-> **M1 M2** où **M1** et **M2** sont des modules.

```
ma_strategie := cStrategy
  oModule: OM_S, OM_V, OM_SS, OM_D;
{
<S> OP.|-> OM_S
<S> OP.Cyc ( BE.AND ( BE.LoopBnd 1000 , BE.SCI 200 ) )
<S> OP.|->
  <S> OP.|->
    <S> OP.|-> OM_V OM_SS </S>
    OM_D
  </S> OMS.IterCounter
  </S> OMS.TimeCounter
</S>
</S>
</S>
};
```

FIGURE 3: Stratégie basique avec POSL

- Cette stratégie de recherche locale s'interprète de la façon suivante :
- S produit une configuration qui est donnée à V .
 - V génère le voisinage de son entrée sous forme d'un ensemble de configurations qui est passé à l'entrée de SS .
 - SS y sélectionne une configuration s^* qui va être donnée à D .
 - D évalue s^* et réitère la boucle **Cyc**.

Ici la l'algorithme boucle tant que l'on a pas fait 1000 itérations ou tant que la solution n'a pas été amélioré depuis 200 itérations. La figure 5 illustre graphiquement le principe de cette stratégie.

Évidement il existe une multitude d'opérateurs POSL afin de créer des stratégies complètes. On peut notamment penser à l'opérateur probabiliste **OP.Rho(p)** qui en fonction d'un nombre n tiré aléatoirement exécute **M1** si $n \leq p$ et **M2** sinon. La figure 6 illustre graphiquement la même stratégie, où **OM_SS** est remplacé de la sorte :

```
<S>
  OP.Rho (0.5)
  <S> OP.|-> OM_SS1 OM_D </S>
  <S> OP.|-> OM_SS2 OM_D </S>
</S>
```

FIGURE 4: Code pour l'opérateur Rho

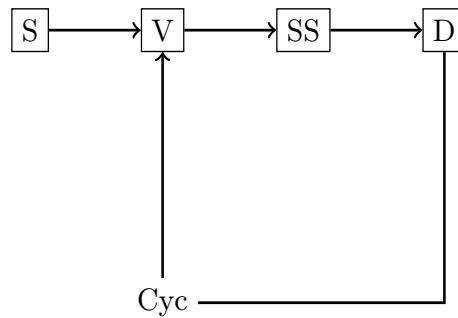


FIGURE 5: Représentation graphique de « ma stratégie »

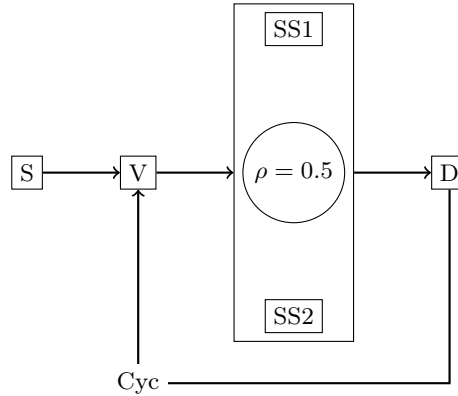


FIGURE 6: Représentation graphique de « ma stratégie » avec le *compound module* de la figure 4.

POSL permet également de définir facilement des stratégies de communication. Un solveur peut recevoir les informations venant d'un autre solveur par le biais d'un *open channel*. Les solveurs peuvent envoyer ou recevoir de simples informations ou bien des composants (typiquement des *operation modules*). Si on reprend le principe la stratégie « ma stratégie », on peut définir deux variantes :

```
OP.Rho (0.5) OM_SS <S> OP.OSend (send_1) OM_SS </S>
```

```
OP.Rho (0.5) OM_SS <S> OP.Min OM_SS OCh (Ch1) </S>
```

FIGURE 7: Variantes pour des stratégies d'envoi et de réception de configuration

Le solveur implémentant la première variante aura 50% de chance d'envoyer la configuration s^* sélectionnée, tandis que le solveur implémentant la seconde variante aura 50% de chance de choisir le minimum parmi son s^* calculé et celui reçu sur son *open channel*. On peut alors instancier ces solveurs et définir le schéma de communication qui les lie, dans la figure 14 la connexion `[solver1] OP.--> [solver2]n` définit n fois une connexion entre un unique solver1 et un unique solver2, soit l'instanciation de $2n$ solveurs.

Notre travail dans le cadre de ce TER a été principalement de faire varier le squelette de cette stratégie à travers l'introduction de divers opérateurs et la variation de paramètres afin d'étudier l'évolution du comportement des solveurs.

5 Expériences sur les fonctionnalités de POSL

Nous avons mené des expériences sur différents modules et opérateurs de POSL pour comparer différentes stratégies génériques sur différents problèmes CSP.


Nous avons exécuté une première stratégie sur les problèmes N-QUEENS et Social Golfers sur les instances respectives $n = 1500$ et $5 - 3 - 7$. Cette stratégie consiste à répéter un nombre maximum $maxrep$ une recherche locale tant que l'on a pas obtenu une solution satisfaisant les contraintes. La recherche elle même consiste à itérer un nombre maximum $itermax$ l'opération suivante : nous choisissons entre deux modules si une condition est respectée ou non. La condition est vérifiée si la configuration courante a un coût inférieur à un nombre $coutmax$ et dans ce cas nous choisissons le premier voisin améliorant le coût, sinon nous prenons une configuration voisine aléatoire. Le but de l'expérience est de faire varier les paramètres $maxrep$, $itermax$ et $coutmax$ pour obtenir les meilleurs résultats possibles en terme de temps d'exécution et du nombre d'itérations réalisées.

Nous avons réalisé une deuxième expérience sur les mêmes problèmes mais sur les instances respectives $n = 300$ et $5 - 3 - 7$. Ici la stratégie consiste également à répéter un nombre maximum de répétitions de la recherche locale. La recherche dispose de deux critères d'arrêt : un nombre maximum d'itérations et un nombre maximum d'itérations sur un plateau, c'est à dire un nombre d'itérations pendant lesquelles nous n'améliorons pas le coût de la configuration. Ce qui nous intéresse ici, c'est la sélection du voisin. Nous avons expérimenté deux façons de choisir le voisin. La première consiste à ne pas choisir le meilleur voisin mais le K^{ieme} meilleur voisin, avec $K \in \{2, 3, 4\}$. La deuxième consiste à ajouter une stratégie de diversification en appliquant une condition probabiliste : nous avons une probabilité ρ de choisir le K^{ieme} voisin et une probabilité $1 - \rho$ de choisir le meilleur voisin. Le but était de voir si ces stratégies permettaient un gain de performance ou de robustesse grâce à de la diversification.

6 Extension de l'existant

Création de nouveaux voisinages



POSL étant un travail en cours, nous avons contribué à son implémentation en développant des voisinages. Nous disposons du voisinage *Adaptive Search*, utilisé abondamment, qui travaille sur les plus mauvaises variables, en utilisant un principe de projection du coût des contraintes sur les variables. C'est un voisinage très efficace qui nous apparaissait être basé sur une stratégie d'intensification en gardant les variables posant le moins de problème. C'est pourquoi nous nous sommes plutôt orienté sur des voisinages plutôt orienté diversification en ajoutant notamment un côté probabiliste et des affectations ne prenant pas en compte la notion de coût. Puisque l'objectif est de travailler sur des stratégies pour problèmes de grandes tailles, nous nous sommes imposés des voisinages de taille linéaire pour maximiser le nombre d'itérations. 

Nous avons quatre voisinages, séparés en deux principes avec chacun deux variations. Les deux principes sont *shift* et *inverse* travaillant tous les deux sur un sous-ensemble de variables $X' \subset X$. Avec *inverse* nous échangeons les valeurs des variables situées aux positions i et $|X'| - i$, $i \in \{1, \dots, \frac{|X'|}{2}\}$. Avec *shift* nous décalons toutes les affectations d'un cran vers la droite de manière circulaire, c'est à dire que la première variable prend la valeur de la dernière. Nous avons choisis le sens et le nombre de décalages de manière arbitraire car nous ne supposons rien sur une éventuelle différence d'efficacité.

Par exemple sur un ensemble $X' = \{X_1, X_2, X_3, X_4, X_5\}$ ayant les affectations $\{X_1 = 1, X_2 = 2, X_3 = 3, X_4 = 4, X_5 = 5\}$ on obtient avec *shift* $\{X_1 = 5, X_2 = 1, X_3 = 2, X_4 = 3, X_5 = 4\}$ et avec *inverse* $\{X_1 = 5, X_2 = 4, X_3 = 4, X_4 = 2, X_5 = 1\}$.

La construction de X' a deux variantes :

- dans la première on tire aléatoirement i et j tel que $1 \leq i < j \leq |X|$, puis on construit $X' = \{x_k \mid x_k \in X \text{ et } i \leq k \leq j\}$.
- dans la deuxième on considère chaque $x_i \in X$ et nous avons une probabilité ρ de l'ajouter dans X' .

Nous nommons nos modules de voisinages `OM_V.2BndsInv`, `OM_V.2BndsShift`, `OM_V.MaskInv` et `OM_V.MaskShift`. Le préfixe `2Bnds` correspond à la variante où l'on tire i et j et le préfixe `Mask` à l'autre variante (comme si on appliquait un masque binaire).

Stratégie de communication

Pour continuer à développer les modules, en particulier ceux de communication (*open channel*), il faut savoir de quelle manière ils vont être utilisés. Nous proposons à cet effet deux stratégies de communication entre solveurs.

Dans la première, l'information que l'on cherche à transmettre est une configuration (affectation). Nous supposons qu'une mauvaise configuration apporte plus d'informations qu'une bonne configuration. En effet une mauvaise configuration pourrait indiquer une zone de l'espace de recherche à éviter en utilisant une notion de distance, alors que l'on ne peut pas réellement conclure pour une bonne configuration. Notre intuition est qu'une mauvaise configuration est une configuration qui ne bat pas une configuration aléatoire. Dans cette stratégie, nous générons plusieurs configurations aléatoires pour déterminer le coût moyen sur les contraintes, puis différents solveurs réalisent un *multi-walk*. À intervalles réguliers (temps d'exécution, nombre d'itérations...) les solveurs vérifient qu'ils sont sur une configuration inférieure au coût moyen, si ce n'est pas le cas ils réalisent un *restart*.

Dans la deuxième nous pensons répartir la partie recherche et filtrage de la résolution entre solveurs. L'information transmise serait alors un nouvel espace de recherche (domaines). On aurait un groupe de solveurs réalisant un *multi-walk* et un autre se concentrant sur des algorithmes de propagation de contraintes parfois coûteux en complexité. Notre hypothèse est que réduire de plus en plus l'espace de recherche au fur et à mesure que l'on avance dans la recherche locale pourrait améliorer la convergence, critère essentiel d'un bon solveur CSP.

7 Présentation des résultats

La première expérience que l'on va présenter ici s'appuie sur la façon dont un voisin va être choisis. Il s'agit d'utiliser le module POSL *KBestImpr*, $K \in \{2, 3, 4\}$ de façon à déterminer la valeur de k donnant les meilleurs résultats pour une stratégie donnée. Ici la stratégie utilisée est celle présentée en figure 3. Les modules utilisés sont présentés dans l'instanciation du solveur en figure 8.

Ce solveur utilise des composants POSL déjà implémentés :

- `OM_S.PermRand` : Génère une configuration aléatoire S .
- `OM_V.AS` : Définit le voisinage $V(S)$ sous la forme du voisinage de l'algorithme *AdaptiveSearch*.

```

solver0 := solver{
  cStrategy: ma_strategie;
  oModule: OM_S.PermRand, OM_V.AS, OM_SS.<K>BestImpr, OM_D.
    AlwImpr;
};

```

FIGURE 8: Solveur POSL pour déterminer le meilleur K

- `OM_SS.<K>BestImpr` : Sélectionne le k ème meilleur voisin s^k dans $V(S)$.
- `OM_D.AlwImpr` : Évalue le critère d'acceptation pour s^k .

POSL permet de facilement analyser les performances d'un solveur ou d'une stratégie en émettant à chaque lancement une sortie de la forme : `<temps (ms)> <nb_iter> <strategie_gagnante> <solveur_gagnant>`. Tout les solveurs s'arrêtent lorsque l'un deux a trouvé une solution, POSL affiche donc le temps mis par ce solveur pour trouver la solution, le nombre d'itérations de l'algorithme dont il a eu besoin ainsi que sa stratégie utilisée et son identifiant.

A l'aide de scripts python nous avons pu récupérer ces informations sur plusieurs centaines de *runs* POSL afin d'établir des statistiques (temps moyens pour trouver une solution, meilleure stratégie, meilleurs solveurs, etc...) sur celle-ci. La figure 9 donne ces résultats en fonction de K sous la forme d'un tableau pour le problème des 300Queens avec un minimum de 100 *runs* pour chaque valeurs de K .

K	temps moyens (ms)	nombre d'itérations moyen
1 (référence)	249	153
2	1405	106
3	1632	128
4	1956	156

FIGURE 9: Résultats POSL en fonction de K , 300Queens



On serait tenté de dire que *2BestImpr* est la meilleur solution puisque c'est lui qui trouve une solution avec le plus petit nombre d'itérations en moyenne, mais il est clair que pour tout $K > 1$ le ratio temps/itérations est bien plus mauvais que le temps de référence. On peu alors en conclure

que sur cette stratégie, toujours choisir le premier voisin, ou au moins les deux premiers améliorant la solution courante est la meilleur décision.

Afin de tester la robustesse de *2BestImpr* qui est le meilleur pour $K \in \{2, 3, 4\}$, nous avons construit des stratégie POSL permettant de le mettre en concurrence avec la référence de l'expérience précédente. Cette fois-ci nous ne travaillons non plus avec un seul composant de sélection (*KBestImpr* auparavant) mais avec une paire de composants de sélection liés par un opérateur. La stratégie utilisée est donc ici exactement la même que présentée sur les figures 4 et 6. Ainsi on définit un solveur à la manière de la figure 10, où *FirstImpr* fonctionne comme *KBestImpr* pour $K = 1$. Dans l'expérience qui va suivre nous avons fait varier la valeur de ρ de 0.1 à 0.9. Cela signifie que si $\rho = 0.1$ alors l'algorithme à 10% de chance de choisir le deuxième meilleur voisin, et 90% de choisir seulement le meilleur, à chaque itération. De la même manière que l'expérience précédente, on représente nos résultats sur la forme d'une matrice en fonction des valeurs de ρ .

```
solver2 := solver{
  cStrategy: St_experience2;
  oModule: OM_S.PermRand, OM_V.AS, OM_SS.2BestImpr, OM_SS.
    FirstImpr, OM_D.AlwImpr;
};
```

FIGURE 10: Solveur POSL pour comparer les deux modules de sélection

ρ	temps moyens (ms)	nombre d'itérations moyen
0.1	1397	73
0.2	1445	77
0.3	1539	79
0.4	1675	81
0.5	1662	82
0.6	1766	87
0.7	1853	89
0.8	1999	90
0.9	2054	93

FIGURE 11: Résultats POSL en fonction de ρ , 300Queens

Pour l'expérience représentée dans la figure 11, nous avons également lancé 100 fois POSL pour chaque valeurs de ρ . La valeur de référence ($\rho = 0.0$) correspond à la première ligne du tableau en figure 9 (100% de chance d'utiliser *FirstImpr*). Ici il est clair de plus la probabilité d'utiliser *2BestImpr* augmente, plus les performances diminuent tant en terme de temps d'exécution qu'en nombre d'itérations. Il est donc indiscutable que pour notre problème des 300Queens et avec notre recherche locale, le module *FirstImpr* sera toujours préférable. On notera cependant que l'utilisation de *2BestImpr* et de *FirstImpr* dans une même stratégie semble diviser par deux le nombre d'itérations nécessaires pour trouver une solution. En effet, choisir tantôt le meilleur voisin puis le deuxième meilleur permet de diversifier l'espace de recherche et offre une convergence moins « radicale » que le simple glouton qui se base uniquement sur la meilleure configuration et qui par conséquent à plus de chance de rester coincé dans un minimum local.

Puisque *AdaptiveSearch* projette les coûts associés aux contraintes sur les variables, il est possible de mesurer le coût d'une configuration, et donc, d'utiliser ce coût afin de prendre des décisions au sein d'une stratégie de résolution. C'est donc dans le même esprit que l'expérience précédente que nous avons conduit une expérience basée sur le coût d'une configuration. De la même façon d'avec ρ , *2BestImpr* et *FirstImpr* vont être choisis en fonction de l'opérateur **RC** (**cost**). Celui ci va exécuter *2BestImpr* si *cost* est en dessous du coût (typiquement le nombre de contraintes violées) de la configuration courante ou *FirstImpr* autrement.

<i>cost</i>	temps moyens (ms)	nombre d'itération moyen
50	890	102
100	657	111
120	471	105
130	468	108
150	481	121
180	352	123
200	357	132

FIGURE 12: Résultats POSL en fonction de *cost*, 300Queens

Cette expérience a été particulièrement difficile à mener puisque nous avons dû travailler dans un intervalle de valeurs pour *cost* qui n'est pas du tout trivial de définir. POSL calcul lui-même les coûts des contraintes

et leur projection sur les variables (configuration), il n'est donc pas évident de déterminer l'ordre de grandeur de ceux-ci sans réaliser plusieurs fois l'expérience illustrée en figure 12. Dans tout les cas ici il est également plus intéressant de n'utiliser que *FirstImpr* tout seul. Cependant on remarque que lorsque le coût approche de la moitié du nombre de queens, il semble y avoir un minimum en terme de temps d'exécution et de nombre d'itérations. Si on estime qu'une solution est de bonne qualité si son coût est faible, et de mauvaise qualité si son coût est élevé, alors ici on dira qu'une solution est de qualité moyenne si son coût approche de la moitié du nombre de queens du problème. Il est logique qu'il ne soit pas intéressant de diversifier une solution lorsque celle-ci est de bonne qualité puisque on a toute les chance de s'éloigner de l'optimum en la perturbant. Cependant cette expérience montre qu'il n'est également pas judicieux de diversifier une mauvaise solution, il semblerait que pour le problème des NQueens cela a pour effet d'éloigner encore plus la solution d'un minimum. Cependant pour une solution de qualité moyenne on retrouve des résultats (bien qu'un peu moins bons) proche de la référence (figure 9 ligne 1).

Une autre expérience que l'on peut présenter ici concerne nos propres voisinages. Comme nous déjà mentionné plus tôt, nous avons codé des *operation modules* de voisinage dans le cœur même du code C++ de POSL. Dans la même idée que les expériences précédentes, on va ici la performance simultanée de deux modules à l'aide de l'opérateur ρ . La différence ici est que nous n'allons plus comparer deux modules de sélection, mais deux modules de voisinages. Dans cette expérience la valeur de ρ est fixée à 0.5. Cela signifie qu'à chaque itération de l'algorithme de résolution, celui-ci à 50% de chance de choisir une configuration parmi un voisinage et 50% de chance d'en choisir parmi un autre. Ici on va donc mettre nos voisinages en concurrence avec le voisinage *AdaptiveSearch* comme pour les précédentes expériences. Le tableau en figure 13 illustre les résultats comparatifs de nos voisinages, la run de référence correspond au solveur en figure 8 ($K = 1$) et à la stratégie représentée par la figure 5.

Ici on va également s'intéresser à T_max et $Iter_max$ respectivement le temps maximum et le nombre d'itérations maximum rencontré sur 100 runs pour trouver une solution. On remarque qu'il est beaucoup moins intéressant de coupler nos voisinages avec *AdaptiveSearch* plutôt que de se contenter de celui-ci. En revanche une chose intéressante intervient à la dernière ligne de ce tableau. Cette dernière ligne correspond au lance-

Comparatif	temps moyens (ms)	nombre d'itération moyen	T_max	Iter_max
AS seul (référence)	195	146	2816	495
2BndsInv ρ AS	439	315	2146	757
2BndsShft ρ AS	477	321	2662	804
MaskInv ρ AS	393	309	2237	822
parallèle*	288	268	463	320

FIGURE 13: Résultats POSL en fonction des voisinages utilisés, $\rho = 0.5$, 300Queens

ment parallèle de quatre solveurs. Chaque solveur utilise la même stratégie (voisinage 1 ρ AS) mais avec un voisinage différents pour chacun. Typiquement pour les trois premiers il s'agit des lignes 2,3 et 4 du tableau lancées en simultané. Le quatrième solveur utilise un plus grand voisinage (n^2 où n est le nombre de variables) : génère toutes permutations de variables deux à deux. Il semble qu'en utilisant cette méthode de résolution (4 voisinages différents en parallèle), le temps et le nombre d'itérations moyen soit meilleur que lorsque ces solveurs sont lancés seuls. On observe également que T_max et $Iter_max$ sont bien plus faible ici, nous estimons que cela est dû à la présence du quatrième voisinage de taille n^2 offrant un choix de configuration bien plus important à chaque itérations, et semble empêcher l'algorithme de résolution de trop diverger. Nous sommes venus à cette conclusion car nous exécutons les lignes 2,3 et 4 du tableau en simultanées nous conservons les mêmes ordres de grandeurs pour T_max et $Iter_max$ que lorsque les lignes sont lancées toutes seules (comme présenté dans le tableau donc) soit environ 2000 et 800 respectivement.

Les expériences présentées ici ne constituent pas l'intégralité des expériences menées au cours de ce TER. Pour ne pas noyer le contenu de ce rapport dans un flot de données, nous avons choisis de présenter un échantillon représentatif (en terme de résultats) de notre travail. Les mêmes expériences (ici seulement pour le problèmes des 300Queens) ont été menées sur les problèmes des 1000 et 1500 Queens. Nous avons également mené une bonne partie de ces expériences sur le problèmes des Golfeurs pour les instances 5 – 3 – 7 et 5 – 3 – 5. A chaque fois les analyses des résultats pour ces problèmes menaient aux même conclusion que celles présentées ici.

8 Conclusion

A travers ce papier, nous avons pu découvrir à quel point il est facile de concevoir une stratégie générique de résolution de CSP à l'aide de POSL. De manière générale, il semble bien que les stratégies les plus efficaces soient celles ayant une heuristique gloutonne, c'est à dire qui se déplacent vite vers les voisins ayant le moins de conflits. Cependant nous avons remarqué quelques méthodes de diversification offrant des résultats intéressants. Dans le cas de ce TER nous n'avons pas eu l'occasion d'expérimenter sur nos stratégies de communication, nous ne pouvons donc pas conclure sur le bon fondement de nos hypothèses. POSL offre de bonne perspectives futures, en particulier lorsque les bibliothèques d'*operation modules* et d'*open channels* seront bien fournis, il sera intéressant d'expérimenter sur l'envoi de composants POSL entre solveurs en cours de résolution d'un problème.

A Exemple d’instanciation de solveur

```
mon_solveur := solver {  
  cStrategy: ma_strategie;  
  oModule: OM_S.PermRand, OM_V.AS, OM_SS.FirstImpr, OM_D.  
    AlwImpr;  
};  
  
mon_solveur2 := solver {  
  cStrategy: ma_strategie;  
  oModule: OM_S.PermRand, OM_V.AS, OM_SS.FirstImpr, OM_D.  
    AlwImpr;  
  oChannel: OCh.DP_Last;  
};  
connections:  
[mon_solveur.send_1] OP.--> [mon_solveur2.Ch1]5;
```

FIGURE 14: Instanciation de solveurs pour la communication

Références

1. Alejandro REYES AMARO, Eric MONFROY, Florian RICHOUX. Un langage orienté parallèle pour modéliser des solveurs de contraintes. *Actes JFPC*, 2015.
2. Alexander E.I. Brownlee, Jerry Swan, Ender Özcan, and Andrew J. Parkes. Hyperion 2. a toolkit for meta-, hyper- heuristic research. *Technical report*, 2014.
3. Jerry Swan and Nathan Burles. Templar - a framework for template-method hyper-heuristics. *Lecture Notes in Computer Science*, 9025 :205–216, 2014.