Introduction :

Dans le cadre de ce projet, nous avons du réaliser un solveur.
Dans un premier temps, l'algorithme utilisé était un simple « backtracking », algorithme qui consiste a revenir en arrière lorsque l'on tombe sur une proposition qui n'est pas une solution.
Par la suite, nous avons du implémenter deux algorithmes différents de « pruning », qui est une amélioration du backtracking dans la mesure ou l'algorithme traite les propositions de manière a éliminer plus rapidement une proposition qui ne mènerait qu'à un blocage.
Dans ce rapport, nous vous détaillerons comment utiliser le solveur dans un premier temps, puis nous reviendrons sur nos choix de structure de données, avant de parler des différents algorithmes implémentés.

I) Utilisation du Solveur

Problèmes :

Nous avons implémentés deux problèmes pour le solveur.
Le premier est le problème dit des « N-Queens » qui consiste a placer N reines sur un jeu de dame de dimensions N*N, sans qu'aucune Reine ne puisse s'attaquer (les reines peuvent s'attaquer en diagonale, ligne ou colonne).
Le deuxième est le problème du carré latin (« Latin Square »), qui consiste a placer une suite de nombres allant de 1 à N dans un tableau de dimension N*N sans que des chiffres identiques se retrouvent sur la même ligne ou la même colonne.

Variation des règles :

Dans l'archive rendue, il y aura donc 2 fichiers : latinSquare_BaP.cpp et queens_BaP.cpp, qui exécuteront le solveur pour les deux types de problème. Afin de choisir le nombre de reines dans le premier problème, et l'entier final de la suite dans le deuxième, il faut modifier respectivement la valeurs des variables globales QUEENS_NUMBER et SIZE.
Les domaines des problèmes seront ainsi créés et initialisés dynamiquement. Nous avons tenu à améliorer cet aspect la du solveur, car dans la première version, il était nécessaire de commenter et/ou dé-commenter beaucoup de lignes de codes pour changer simplement le nombre de reines.

SCREENS #defines

Exécution du solveur :

Lorsque le solveur se lance, il est affiché sur le terminal le nombre de domaines (nombre qui dépends de la règle choisie) ainsi que leur initialisation. Le terminal vous propose ensuite les 3 algorithmes implémentés pour résoudre le problème. Il est donc de possible de choisir en cours d'exécution quel algorithme utiliser. Une fois l'algorithme terminé, le terminal affiche le nombre de solutions ainsi que la première. Il est possible de continuer a voir des solutions en appuyant sur une touche puis « Entrée ». Tant que la saisie n'est pas la lettre « q », le terminal vous montrera la ou les prochaines solutions.

SCREENS DU TERMINAL

II) Choix de structures

Nous avons déjà détaillé nos choix de structures dans rapport de mi-rendu. Cette partie sera donc quasiment identique.

Les domaines sont représentés par des List. Il est nécessaire de pouvoir accéder à n'importe quelle valeur du domaine et de pouvoir la copier et dans la structure List ces deux opérations se font en O(n). De plus, lorsque nous améliorerons l'algorithme pour qu'il effectue du pruning, il sera nécessaire de pouvoir supprimer grâce a une valeur passée en paramètre, au lieu d'un indice, ce qui est pratique avec List.
Les nœuds sont représentés par des vector. La seule opération effectuée sur les nœuds est l'accès à ses valeurs. La structure vector permet cela en O(1).
L'ensemble des variables n'est pas représenté par une structure. L'ensemble des variables étant supposé présenter des entiers triés par ordre croissant et correspondant toujours au nombre de domaines présents au sein d'un nœud, nous avons remarqué qu'il était plus simple de se contenter de prendre les indices des domaines contenus dans les nœuds. Ainsi la variable X 1 correspondra à l'indice du premier domaine contenu dans nœuds, autrement dit à l'indice de son domaine.
L'ensemble de nœuds est représenté par une Pile. En effet on remarque qu'au sein de l'algorithme, on effectue uniquement des opérations d'ajout et de suppression sur cet ensemble, et ce toujours en tête. Une Pile est donc la structure toute indiquée pour cet ensemble.
Les contraintes sont représentées directement par des fonctions au sein du code.

III) Différents algorithmes

1) Simple backtracking

Le backtracking implémenté ici permet de vérifier que les domaines réduits a une cardinalité de 1 respectent les contraintes entre eux. L'algorithme stocke donc ces domaine, et les envoie a une autre fonction, « constraintsQueens » ou « constraintsLatinSquare » qui retourne un booléen en fonction du respect des contraintes respectives aux problèmes. Si les contraintes ne sont pas respectées, le backtracking retourne un nœud vide, ce qui permet a l'algorithme principal « branchAndPrune » de « discard » la proposition. C'est donc une approche simpliste, mais couteuse car on ne fais aucun prétraitement et si on veut retourner toutes les solutions, il est nécessaire de parcourir toutes les possibilités.

2) Look Ahead

L'algorithme « Look Ahead » est un alorithme de pruning. Le fonctionnement est simple, lorsqu'un domaine est réduit a une cardinalité de 1, l'algorithme va parcourir les autres domaines afin de supprimer les valeurs qui ne respecteraient pas les contraintes.
Par exemple, prenons le problème des reines. Si l'un des domaine est réduit à la valeur « 3 », la fonction « pruningLookAhead » va supprimer la valeur « 3 » de tous les autres domaines afin de respecter la contrainte « alldifferent », ainsi que les valeurs « 2 » et « 4 » des domaines adjacents, et la valeur « 1 » du domaine éloigné pour respecter la contrainte sur les diagonales. Si ce pruning a réduit d'autre domaines à la cardinalité « 1 », alors la fonction recommence le pruning.

3) Arc consistency based pruning

Nous avons implémenté un dernier algorithme de pruning, basé sur la consistance (ou cohérence) d'arc (« Arc consistency »). Cette fois ci, l'algorithme « ACBasedPruning » vérifie pour tous les domaines si chaque valeur du domaine a « un support » dans chaque autre domaine du

nœud. On a donc une autre fonction « HasSupport » qui permet de vérifier la consistance d'arc d'une valeur sur un domaine, et renvoie un booléen. Si la valeur n'a pas de support dans l'un des domaines, elle est supprimé du domaine d'origine.

4) Comparatif des temps sur les 2 pb

Relevé de temps : Queens 4 : 2 solutions – 0,001 secondes – 0,0005 secondes – 0,0006 secondes
6 : 4 solutions – 0,01 secondes – 0,006 secondes – 0,007 secondes
8 : 92 solutions – 0,17 secondes – 0,03 secondes – 0,04 secondes
11 : 2680 solutions – 27.5 secondes – 1,9 secondes – 6,7 secondes
12 : 14200 solutions – Trop long – 10 secondes – 42,9 secondes

LatinSquare N=3 : 12 solutions – 0,01 secondes – 0,002 secondes – 0,007 secondes
4 : 576 solutions – 0,37 secondes – 0,06 secondes – 0,2 secondes
5 : 161 280 solutions – Trop long – 31,9 secondes – Trop long

Commentaires :

IV) Améliorations possibles (Conclusion ?)

Possibilité de préciser a l'execution le nombre de reines, ou la taille, etc..
+ de Généricité, pattern Strategy
Optimisation des algorithmes
Ajout de problèmes
Ajout d'algorithmes de pruning