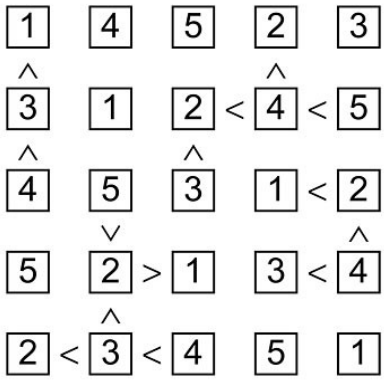
Projet d'intelligence artificielle : résolution de grilles de Futoshiki

Le but de ce projet est de résoudre des grilles de Futoshiki, un jeu qui consiste à remplir une grille n x n avec des chiffres compris entre 1 et n. Chaque chiffre doit être présent une seule fois par ligne et par colonne. Certaines cases peuvent être déjà remplies. Des indices sont fournies sur les valeurs des cases, par exemple, la valeur de la première case est inférieure à la valeur de la deuxième case.



Pour cela, nous allons utiliser les algotithmes de BackTrack (BT) et Forward-Checking (FC), de deux manières différentes : avec un CSP binaire et avec un CSP n-aire. Ce sera le choix de l'utilisateur. Un CSP binaire est un CSP dont toutes les contraintes sont unaires ou binaires. Un CSP n-aire est un CSP dont au moins une contrainte est n-aire.

BT étend progressivement une affectation consistante. En cas d'échec, on change la valeur de la variable courante. S'il n'y a plus de valeur, on revient sur la variable précédente. On réitère le procédé tant que l'on n'a pas trouvé une solution et que l'on n'a pas essayé toutes les possibilités.

Soit X la variable courante. FC filtre les domaines des variables voisines de X en retirant les valeurs incompatibles avec l'affectation de X. Si un domaine devient vide, on essaie une nouvelle valeur pour X. Si toutes les valeurs ont été essayées, on revient en arrière sur la variable précédente.

Mode d'emploi du programme :

Un makefile est disponible pour la compilation. Pour exécuter le programme, tapez :
./Futoshiki grilles/Nom de la grille.fut {--bBT, --bFC, --nBT, --nFC} {--none, --alea, --heuF, --dom_deg}

Pour plus d'informations sur la commande, tapez :
./Futoshiki -h

Partie I : CSP

a) Les énumérations

Nous avons deux énumérations : symbole et type.
Symbole représente le type des indices que peut contenir la grille : supérieur, inférieur, différent et « none » qui représente un non-indice car, par exemple, les deux recoins de la grille n'ont pas d'indices entre eux.

Type représente le CSP que l'on veut : binaire ou n-aire. Il servira lors de l'execution du programme, quand l'utilisateur devra choisir la méthode qu'il souhaite.

b) La structure CSP

Cette structure comprend 6 variables : le type de CSP que l'on souhaite, le nombre de cases de la grille, la taille de la grille (par exemple, si une grille est nxn, ce sera n), un tableau à deux dimensions pour les relations binaires qui stocke les symboles (supérieur, différent...) entre deux cases, un tableau de booléens à deux dimensions pour les relations n-aires (est-ce-que la case numéro j est dans la relation i ?), et un tableau à deux dimensions pour le domaine des variables.
Domaine[i][0] donne le nombre de valeur possible pour la variable i.
Domaine[i][j] peut prendre plusieurs valeurs :

- 0 : la variable i ne peut pas prendre la valeur j.
- -1 : la variable i peut prendre la valeur j.
- >0 : le domaine a été modifié, pour le moment la variable i ne peut pas prendre la valeur j. Ceci servira pour FC.

c) Les fonctions pour le CSP

La fonction creerCSP fait toutes les allocations dynamiques. Elle appelle alors la fonction lireFutoshiki qui se charge de remplir le CSP. Elle retourne le CSP créé.

La fonction lireFutoshiki appelle pour chaque ligne la fonction lireLigne.
Celle-ci va lire deux lignes du fichier : la ligne qui correspond aux chiffres de la grille et celle qui correspond aux symboles (les indices) de la grille.
Tout d'abord elle lit la ligne des chiffres de gauche à droite. Elle appelle lireValeur et lireSymbole pour chaque couple (valeur, symbole). Le dernier chiffre n'a pas de symbole à sa droite, donc elle finit sur lireValeur.
Puis elle lit la ligne des symboles de gauche à droite en appelant la fonction lireSymbole.
La fonction lireValeur remplit le domaine des variables. Si une case contient déjà un chiffre, seul ce chiffre sera dans le domaine de cette case.
La fonction lireSymbole remplit les tableaux de relations avec les symboles inférieur et supérieur.
Les relations de différences ont déjà été faits à la création du CSP, car dans un Futoshiki les relations de différences sont les mêmes pour toutes les grilles (nous n'avons pas besoin d'un fichier pour les connaître).

La fonction differenceBinaire est appelée avant la fonction lireFutoshiki, uniquement pour un CSP binaire. Elle remplit le tableau de relation binaire avec la relation « différent ».
La fonction differenceNaire est appelée avant la fonction lireFutoshiki, uniquement pour un CSP n-aire. Elle remplit le tableau de relation n-aire.

La seule différence entre un CSP binaire et n-aire est l'appelle de ces deux fonctions. Un CSP n-aire aura son tableau de relation binaire remplit de « none », « inférieur », « supérieur ». A l'inverse, un CSP binaire aura un tableau de relation binaire remplit de « none », « inférieur », « supérieur » et « différent » et il n'aura pas de tableau de relation n-aire.

La fonction libererCSP se contente de libérer la mémoire.

Partie II : résolution

a) Structure de données

La structure ALGO contient 7 variables : une pile avec son sommet, un tableau de valeur, la case courante, le nombre de nœud contenu dans l'arbre produit par les algotithmes BT et FC, le nombre de contraintes testées et le temps que met l'algorithme à trouver une solution.
La pile permet de garder l'ordre du choix de variable. Elle peut-être utile selon les heuristiques.

Nous avons également une énumération qui nous permet de savoir quelle heuristique l'utilisateur veut choisir.

b) BackTrack

Pour l'algorithme BT, on choisit une case grâce à une heuristique, puis on lui donne une valeur. Tant que cette case peut obtenir une valeur possible mais qui ne respecte pas une contrainte (que ce soit un CSP binaire ou n-aire), on lui donne une autre valeur. On teste les valeurs dans l'ordre croissant (de 1 jusqu'à taille).
Deux cas sont alors possibles :

- Il n'y a plus de valeur possible pour la case. Dans ce cas, on récupère la dernière case implémentée en dépilant.
- On a trouvé une valeur qui respecte les contraintes. Dans ce cas, on l'empile et on recommence pour une autre case.

L'algorithme se termine si on a trouvé une solution ou si on tente de dépiler une pile vide (pas de solution).

Pour savoir si une valeur d'une case respecte les contraintes, nous avons la fonction contrainte. Pour les CSP binaires, celle-ci se contente de tester la case courante avec les autres cases implémentée (se trouvant dans la pile). Si c'est un CSP n-aire, en plus de cela, on regarde si la case courante est dans une relation. Si oui, on regarde si toutes les autres cases de la relation sont implémentées : si c'est le cas, on teste la contrainte.

c) Forward-Checking

FC est quasiment identique à BT, les différences sont :

- Avant d'empiler, on réduit le domaine des autres cases non implémentées en relation avec la case courante. Si on a un domaine vide pour une case, on restaure les domaines et on change de valeur pour la case courante (l'équivalent d'une contrainte non respectée).
- Quand on dépile une case, on restaure les domaines des cases en relation avec la case dépilée.

Pour FC n-aire, nous avons choisi l'algorithme FC1 : toutes les cases doivent êtres instanciées sauf la case courante.

Partie III : les heuristiques

Nous avons implémenté 4 heuristiques : aucune, aléatoire, heuristique futoshiki, domaine/degré.

a) Aucune

Même si le nom porte à confusion, aucune est une heuristique basique qui prend les cases dans l'ordre, c'est-à-dire de la première case en haut à gauche jusqu'à la dernière en bas à droite. La première case est la case 0, la dernière est la case « nombre de cases - 1 ».

Pseudo-code :

SI pile est vide, RETOURNER l'indice de la première case
SI plus de case possible, RETOURNER -1
SINON, RETOURNER l'indice de la dernière case implémentée +1

b) Aléatoire

Aléatoire est une heuristique qui choisit des cases au hasard.

Pourquoi avoir choisi cette heuristique ?
Cette heuristique est facile à implémenter et facile à trouver. Nous n'effectuerons pas de tests avec cette heuristique.

Pseudo-code :

SI plus de case possible, RETOURNER -1
SINON, RETOURNER une variable non-implémentée.

c) Heuristique Futoshiki

Heuristique Futoshiki est notre heuristique ! Elle regarde le nombre de valeur qu'il y a dans les domaines de chaque cases. Ceci est instantané car la valeur domaine[i][0] donne le nombre de valeur dans le domaine de la case i. Elle prend la case qui a le plus petit domaine. Si une case a un domaine égal à 1, cela montre que la case a déjà une valeur donc on ne recherche pas une case qui a un domaine plus petit car on ne peut pas faire mieux que domaine[i][0]=1.

Pourquoi avoir choisi cette heuristique ?
Nous avons pensé que c'était logique de prendre les cases qui ont le domaine le plus petit. De cette façon, on aura moins de nœud.
Le forward-checking amélioré demandé dans le TP sera le FC initial avec cette heuristique.

Pseudo-code :

```
INT imin := infini
SI plus de case possible, RETOURNER -1
SINON TANT QUE i < nombre de case
        SI valeur possible de i = 1 ET i pas dans la pile RETOURNER i
        SI valeur possible de i < imin ET i pas dans la pile imin = i
FIN TANT QUE
RETOURNER imin
```

d) Heuristique dom/deg

Cette heuristique prend la case qui a le plus petit domaine/nombre de contrainte.

Pseudo-code:

```
INT imin := infini
SI plus de case possible, RETOURNER -1
SINON TANT QUE i < nombre de case
        SI valeur possible de i/nombreContrainte de i < imin ET i pas dans la pile
        imin = i
FIN TANT QUE
RETOURNER imin
```

Partie IV : comparaisons

Nous allons montrer les différences de résultats selon les méthodes, les heuristiques et les grilles utilisées. Certaines lignes n'y figurent pas pour les heuristiques "aucune" (par exemple : BT n-aire grille1 9x9) car la résolution prenait trop de temps. De ce fait, nous n'avons pas testé les grilles manquantes avec les autres heuristiques car nous ne pouvons pas comparer avec l'heuristique "aucune". De plus, pour les autres heuristiques, nous les avons testées uniquement en binaire car le binaire est plus performant que le n-aire. On a choisi de les faire avec FC car nos heuristiques travaillent sur les domaines.

Configuration de l'ordinateur :
Ubuntu 14.04 LTS
Mémoire : 4 Gio
Processeur : Intel core i3 CPU 530 @ 2.93 GHz x 4
Type d'OS : 32 bits

Tableau de comparaisons

Méthode	Heuristique	Grille	Nombre de noeuds	Nombre de contraintes testées	Temps en seconde
BT binaire	aucune	Grille1 5x5	22195	17746	0
FC binaire	aucune	Grille1 5x5	8636	4189	0
FC binaire	Futoshiki	Grille1 5x5	95166	27384	0
FC binaire	Dom/deg	Grille1 5x5	61284	22646	0
BT n-aire	aucune	Grille1 5x5	182660	149511	0
FC n-aire	aucune	Grille1 5x5	165597	132448	0
BT binaire	aucune	Grille1 6x6	289851	234393	0
FC binaire	aucune	Grille1 6x6	39339	17661	0
FC binaire	Futoshiki	Grille1 6x6	673089	154294	1
FC binaire	Dom/deg	Grille1 6x6	462295	74794	1
BT n-aire	aucune	Grille1 6x6	829720861	693220750	256
FC n-aire	aucune	Grille1 6x6	795490839	658990728	509
BT binaire	aucune	Grille1 7x7	2842348	2317527	0
FC binaire	aucune	Grille1 7x7	191104	109026	0
FC binaire	Futoshiki	Grille1 7x7	32459710	5221386	103
FC binaire	Dom/deg	Grille1 7x7	86762723	15648296	353
BT binaire	aucune	Grille1 8x8	2504997326	2127692682	239
FC binaire	aucune	Grille1 8x8	3433492	1654137	7
FC binaire	Futoshiki	Grille1 8x8	3250507	1418929	11
FC binaire	Dom/deg	Grille1 8x8	899234	399371	4
BT binaire	aucune	Grille2 5x5	391500	313190	0
FC binaire	aucune	Grille2 5x5	136109	59169	0
FC binaire	Futoshiki	Grille2 5x5	22760	8053	0
FC binaire	Dom/deg	Grille2 5x5	5099	2436	0
BT n-aire	aucune	Grille2 5x5	61911285	51094498	12
FC n-aire	aucune	Grille2 5x5	61494849	50678062	24
BT binaire	aucune	Grille2 6x6	690351	571480	0
FC binaire	aucune	Grille2 6x6	30225	10544	0

FC binaire	Futoshiki	Grille2 6x6	100711	24067	0
FC binaire	Dom/deg	Grille2 6x6	61069	16113	0
BT binaire	aucune	Grille2 7x7	7648908526	6534931609	555
FC binaire	aucune	Grille2 7x7	107118672	31866552	204
FC binaire	Futoshiki	Grille2 7x7	20409133	8455197	44
FC binaire	Dom/deg	Grille2 7x7	6025885	2495781	17
BT binaire	aucune	Grille2 8x8	85078548	73572998	7
FC binaire	aucune	Grille2 8x8	1671364	758464	3

a) Comparaison entre les méthodes binaire et n-aire, avec l'heuristique "none"

Dans une même méthode de résolution, les résolutions de type binaire écrasent les résolutions de type n-aire en terme de temps, de nombre de nœud et de nombre de contrainte . En effet, les méthodes n-aires mettent plus de temps à détecter les contraintes de différence.
Il vaut mieux éviter les méthodes n-aires.

b) Comparaison entre BT et FC

Quand on compare BT binaire et FC binaire pour le Futoshiki, FC est supérieur à BT. Le nombre de nœud et le nombre de contrainte pour FC sont toujours inférieurs à ceux de BT.

Parfois, le travail sur le domaine rend le temps de résolution plus long, comme pour la grille1_6x6 pour FC n-aire et BT n-aire.

c) Comparaison des heuristiques

Une heuristique peut faire une grande différence, que ce soit positif ou négatif. En effet, dans certains cas l'heuristique « none » est meilleure comme pour la grille1_7x7.

Partie V : conclusion

Pour les types de résolution, binaire est forcément meilleur que n-aire. Mais dans certains cas nous devons utiliser le n-aire, comme pour une résolution d'un carré magique.

Forward-Checking binaire est meilleur que BackTrack binaire.

Quant aux heuristiques, cela dépend des cas. Nous ne pouvons pas vraiment dire si une heuristique est meilleure qu'une autre. Nous pensions que notre heuristique « Futoshiki » allait être meilleure que l'heuristique « none » mais cela dépend des grilles.