Number Match Solver

Éxecution:

Pour utiliser le programme, executez la commande:
`#use "NUMATCH_PATH"` en remplaceant NUMATCH_PATH avec l'ubication du fichier OCaml. (numatch.ml). Cette commande va évaluer toutes les fonctions du programme.

Pour génerer une matrice `m` x `n` aléatoire (Qui sera l'état initiale du number match) la fonction `rm m n` peut être utilisé.

Pour lancer le programme qui jouera number match avec une matrice, ici de type `list of list`, la fonction `fullgame mat` est utilisé.

Pour lancer `fullgame, k` fois, chacune avec une matrice aléatoire distincte de dimensions `mxn`, la fonction `avg m n k` est utilisé.

Choix d'implementation:

Répresentation Matricielle.

On utilise des matrices, de taille `m x n` representés avec des `list of lists.`, où chaque liste est une ligne. Toutes les lignes ont la même longeur (n), sauf la dernière. Ceci est cohérent avec les grilles du jeu. Le choix des listes vient de pouvoir les étudier par récurrence, ce qui est plus convénient en OCaml. Cépendant, ce choix est au péril de l'accessibilité, et souvent, complexité: Lire la case (i,j) à comme complexité O(i*j). Chaque élèment de la matrice représente le chiffre qu'il contient, et 0 s'il est grisé. Si on essaye de lire un élement dans une case plus grande ou plus petite que les dimensions de la matrice, la fonction `element mat i j` renvoie -1.

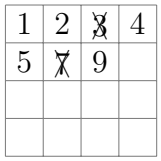

Figure 1: Exemple de matrice et sa répresentation au programme.

Recherche de voisins et matching:

La recherche des voisins est faite par récurrence. À partir de la matrice et les coordonnées d'un point, le programme cherche les voisins directs dans les 8 points cardinaux, en avanceant si le voisin direct est 0. Le résultat est une liste de `int * (int*int)`, où le premier int est le chiffre voisin et le couple est ces coordonnées.
(i.e: `neighbours mat 0 1` *pour la matrice ci-dessus renvoie* `[(4,(0,3)), (9, (1,2)), (5,(1,0)), (1,(0,0))]`*)*
Pour les voisins à droite/gauche, si toutes les chiffres à droite/gauche sont des zéros, le programme cherche la ligne suivante/précedante dès le début/la fin jusqu'en trouver un. (où -1.)
La fonction `posmatches` parcours chaque élement et évalue la compatibilité avec ses voisins (Que leur somme soit 10 ou ils soient égaux), en ométtant tout couple 'rédondant' (i.e: ((a,b) , (c,d)) et ((c,d) , (a,b))). L'output est une liste de couples de coordonnées (`(int * int) * (int * int)`). La fonction `removeall` elimine tout couple compatible, remplacéant la place des deux dans la matrice par un 0. L'élimination suit donc un ordre par indice (et la recherche de voisins, l'ordre horaire, partant du nord-ouest.).

Ajout de chiffres.

L'ajout des nombres à été la partie la plus complexe en utilisant des listes de listes. L'algorithme fonctionne comme suit:

1. `remaining mat` renvoie une `int list` des chiffres pas encore matchés en `mat`.
2. Selon la différence de la longeur de la dernière ligne par rapport à le nombre de colonnes de la matrice, la liste `remaining` est separé en deux listes.
3. La première "complète" la dernière ligne de la matrice.

4. La deuxième est, elle même, partitionné par le nombre de colonnes de la matrice, pour être directement concatenés à la matrice une à une.

Algorithme finale

Enfin, la fonction `fullgame mat` joue au number match avec la matrice presenté. L'algorithme fonctionne comme suit:

1. c = 0 et a = (m x n) / 4. a est la quantité de 'ajouts' possibles, et c est un compteur.
2. S'il y a des lignes entières avec uniquement des 0s, les extraire de la matrice.
3. Afficher la matrice actuelle.
4. Si `remaining mat = []` (Tout les nombres sont matchés), renvoyer `c`, le nombre de fois qu'on a bouclé (Nombre de ajouts et éliminations faites. Toute chaîne d'eliminations consecutives compte comme un seul "boucle".)
5. Si a == 0, on a perdu. Re-lancer dès l'étape 1, mais en éliminant au hazard les matches trouvés. La valeur de c est conservé.
6. S'il n'y-a pas de chiffres à matcher (`posmatches = []`), alors ajouter des chiffres. a = a-1, c = c+1 et on rétourne à l'étape 2.
7. S'il y a des chiffres à matcher, on les élimine. c=c+1 et on rétourne à l'étape 2.

La fonction `avg m n k` renvoie le nombre de coups (c) moyen pour k matrices mxn au hazard.

Exemples de Résultats:

Les moyennes qu'on obtient sont les suivantes:

- avg 8 3 50 → 6.5 coups.
- avg 8 4 50 → 6.58 coups.
- avg 9 4 10000 → 6.816 coups.
- avg 9 5 50 → 8.26 coups.

- avg 10 5 25 → 6.96 coups.
- avg 15 4 25 → 12.68 coups.
- avg 4 15 25 → 13.8 coups.

Améliorations possibles:

Ce programme donne une solution pour presque n'importe quelle matrice initiale, mais des optimisations pourraient être faites, spécialement au moment d'executer `fullgame` un grand nombre de fois et pour dimensions plus grandes. Un étude de graphe permettrait d'arriver à la victoire du jeu de la forme plus efficace, et sans devoir "relancer" le programme en cas d'echec. Une fonction interactive pour jouer number match avec une simple intelligence artificiale est aussi possible, si on optimise la vitesse de plusieurs fonctions. On observe que, pour des tableaux grands, la majorité du temps d'execution est pris à matcher la matrice initiale. Lors du premier coup, la matrice est largement plus petite (la majorité des lignes sont entièrement matchés), et prends un temps négligable à la matcher. On pourrait donc optimiser cet algorithme en divisant le problème en plusieurs sous-matrices, ou agiliser la recherche de voisins.