Tout ceci (codes sources, algorithmes, l'idée géniale d'indenter le code, etc) est entièrement libre de droits, peut être distribué sur un site et tout et tout. Voila, c'est dit.

1 Compilation

Le programme a été testé avec Microsoft Visual C++ versions 8 et 9, ainsi qu'avec MinGW. Il doit pouvoir être compilé sans souci avec MSVC 8 ou 9, pour lesquels un fichier .vcproj est fourni.

Pour GCC (version 3.X), un makefile est fourni.

Le makefile est volontairement basique, de faon à pouvoir être utilisé avec un peu n'importe quelle version de make (typiquement, je visais gcc-make et Nmake). Et puis ce n'est pas un concours de makefile ;)

2 Implémentation

2.1 Les structures de données

Il existe 3 types d'emplacements : les cases vides, les cases occupées par un pion, et les cases hors du plateau (mais qui ont des coordonnées). Il m'a semblé préférable de gérer des cases à 2 états. Un plateau de jeu contient donc 2 description :

- l'ensemble des cases disponibles
- l'ensemble des cases occupées par un pion

Les mouvements possibles sur le plateau sont pré-calculés et mémorisés. Cela présente 2 avantages :

- On isole facilement la routine de calcul des mouvements possibles. La lisibilité est meilleure, en particulier si on veut autoriser ou pas d'autres types de mouvements (en diagonale par exemple).
- ces mouvements sont calculés à partir de la description des cases, mais sont utilisés sur la description des pions. On utilise bien la séparation cases/pions.

J'ai choisi de ne pas gérer les symétries du plateau. La principale raison à cela est qu'un plateau "générique" n'en présentera pas (mais en même temps, tous les exemples proposés en proposent au moins une). D'autre part, j'ai estimer que cela n'améliorera pas tant que cela les performances : on y gagne au maximum un facteur 8 sur les configurations à étudier, et on doit tester tous les symétriques d'une configuration donnée pour savoir si elle a déjà été rencontrée.

2.2 Algorithmes utilisés

2 algorithmes ont été implémentés. Le premier est récursif : partant d'une configuration, il essaye succéssivement tous les mouvements possibles. Les mouvements valides lancent l'exploration de la configuration obtenue (donc à nouveau le test de tous les mouvements possibles). Si on atteint la configuration finale, on remonte en mémorisant le chemin parcouru et c'est fini. Quand aucune exploration n'a abouti, on prévient que c'est un cul de sac.

Il est extrêmement efficace de mémoriser les configurations déjà atteintes. On peut ainsi immédiatement identifier une situation qui n'aboutira pas. En contrepartie, cela consomme de la mémoire en grande quantité. Dans le cas peu probable d'une machine très rapide mais avec peu de mémoire, on pourra désactiver la mémorisation des positions parcourues. La mémorisation des configurations atteintes est implémentés très facilement avec un "set" de la bibliothèque standard

(ce qui demande de définir un ordre entre les configurations). Ainsi la mémoire est gérée automatiquement. Plus précisemment, il y a un "set" par nombre de pions sur le plateau, pour accélérer la recherche.

Le second algorithme explore toutes les configurations atteignables. Il n'est pas capable de reconstituer une solution (un chemin allant de la configuration initiale à celle finale), mais donne des informations sur le nombre de chemins possibles, de chemins menant à une position bloquée, et de chemins arrivant à une solution. Il part de l'ensemble des positions à N pions atteignables (ainsi que du nombre de chemins arrivant à ces solutions), et en appliquant tous les mouvements possibles à toutes ces configurations, on obtient l'ensemble des positions atteignables à N-1 pions et le nombre de chemins y conduisant.

En pratique, la consommation de mémoire fait que l'algorithme bloquera avant la fin (en fait avant la moitié, puisque le nombre de configuration est maximal lorsqu'une case sur 2 est vide, et que cela ne doit pas être très différent pour les configurations atteignables).

2.3 Architecture

J'ai rapidement décidé qu'on pourrait utiliser différentes implémentations pour décrire les cases du plateau. Le plus naturel est un champ de bit (puisque je me suis ramené à une description à 2 états), mais je voulais pouvoir gérer des plateaux plus larges qu'un entier (surtout que la taille d'un entier n'est pas garantie par le langage). J'ai hésité entre utiliser la programmation générique (les "templates") et de l'héritage, j'ai choisi la programmation générique car je trouvais son utilisation plus estétique. Au final, c'est un peu regrettable dans le cadre d'un concours mixte C et C++ : on peut faire de l'objet en C (avec des pointeurs de fonction par exemple), alors que les "templates" sont vraiment spécifiques au C++. J'ai pensé trop tard à cet argument, je me console en me disant que j'ai beaucoup plus appris ainsi que je n'aurais appris sur les hiérarchies de classes.

Au final, il n'y a que 2 implémentations (même si l'une est générique, indexée par un type d'entier), et c'est un peu compliqué pour rien. Surtout que sur un plateau trop large, la recherche sera probablement trop longue pour aboutir (mais l'honneur est sauf, ce n'est pas mon implémentation qui ne gère pas le cas).

2.4 Cases hexagonales

Le texte du défi présentait un plateau à cases hexagonales, il était tentant d'y adapter le programme. Cela demande un codage pour le fichier texte codant le plateau, une routine d'affichage modifiée, ainsi qu'une prise en compte des mouvements permis (d'où l'intérêt d'avoir externalisé la routine de pré-calcul). Un fichier décrivant un plateau à cases hexagonales commencera par la ligne "HEXA". Dans ce cas, chaque ligne dera décalée d'une demi-case vers la droite (ce qui demande d'avoir orienté le plateau préalablement avec des lignes horizontales et pas verticales). Le défi demandant qu'une telle ligne soit ignorée, on doit activer une option (-x) pour la prendre en compte. Cela permet d'avoir un comportement par défaut respectant le cahier des charges.

3 Développement

J'ai principalement développé avec GCC (MinGW sous Windows XP), et sur la fin j'ai testé la compilation avec Visual Studio (le 8 du bureau et le 9 à la maison), ce qui est toujours intéressant puisque les compilateurs ne donnent pas toujours les mêmes erreurs.

Une première version de ce programme a été développée en environ 2 heures. Et pour être honnête, c'était avant ce défi. Le plateau et ses mouvements étaient alors implémentés dans le code, il n'y avait pas de lecture de fichier. Par contre, à ce moment je gérais les symétries, et c'était même la description du pleteau qui s'en chargeait (j'encodais les couronnes concentriques).

Implémenter un codage générique et la lecture robuste des fichiers m'a demandé plus de travail. Probablement entre 5 et 10 heures.

Permettre différents encodages internes de la description du plateau de jeu a été le plus long. Bon, c'est là que j'ai vraiment appris des choses, mais j'y ai passé la majeure partie du temps de développement (entre 10 et 20 heures ?). Mais au final, j'aurais préféré implémenter d'autres fonctionnalités.

4 Performances

Je suis impatient de voir les comparaisons avec d'autres implémentations (voir d'autres algorithmes). La lecture des commentaires me laisse penser que mes performances sont un peu décevantes. Je manque de pistes pour les améliorer, on pourrait penser à :

- recoder les conteneurs de la bibliothèque standards (en particulier les "set") ne sont peut être pas optimaux. Je n'y crois pas vraiment.
- gérer les symétries.
- trouver un moyen de décider qu'une configuration n'est pas soluble (pion isolé par exemple).
- essayer de tester plus efficacement les mouvements possibles. Par exemple, éviter de tester 12 fois qu'une case est vide. Mais je n'ai aucune idée sur comment faire.

5 TODO

J'aurais aimé implémenter :

- La possibilité de laisser libre la dernière case, ou de l'imposer ailleurs que sur la première case libre.
- Permettre la recherche de solutions avec plusieurs pions à la fin (par exemple dans les coins du diamant).
- La gestion des symétries.

Tant pis !

6 Remarques

- Je tiens à préciser que le programme (hors commentaire) est entièrement en ASCII, mais que tous les affichages sont en franais correct (en principe !). Je me suis parfoit cassé la tête pour éviter les mots avec accent.
- le type long long int ne passe le l'option -pedantic de GCC. C'est à mon goût regrètable
- Cetains operateurs (par exemple le constructeur par défaut) sont demandé par MSVC uniquement. j'ai donc dû rajouter des constructeur par défaut, des constructeurs par copie et des méthodes "swap". Mais j'ai refusé d'avoir des portions de code spécifiques à MSVC ou à GCC.
- Merci aux organisateurs du défi, je me suis quand même bien amusé.
- Vivement la correction !