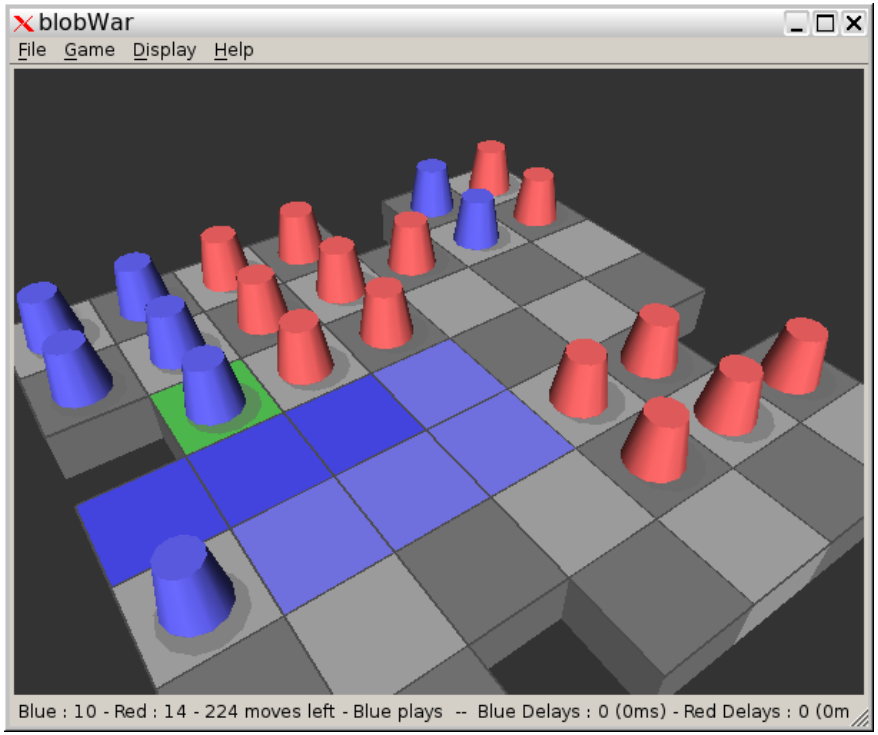
Projets, Algorithmique, C++, Théorie des Jeux

Guerre des blobs *



*. Jean-Guillaume Dumas

13 janvier 2014

Table des matières

1 Enoncé du mini-projet

1.1 Objectif

Le TP consiste en la mise en œuvre dans le langage C++ d'un jeu à deux et à l'évaluation de différentes stratégies d'optimisation. Outre les aspects algorithmiques et surtout efficacité, l'objectif pédagogique est aussi un apprentissage pratique du langage C++, notamment de trois de ces caractéristiques : programmation objet (classes et héritage) ; généricité (`template`) ; containers et itérateurs (`STL`, *Standard template Library*) ; contrats, tests, profilage.

1.2 Contraintes

- Le langage de programmation est le C++.
- Les structures de données sont libres mais celles de type liste, vecteur doivent suivre l'interface de la STL (par exemple utiliser d'abord les `std::vector` puis passer ensuite au conteneur développé en Programmation par contrats). Plus de détails en section 1.6.3.

N.B. : Le compilateur est g++ sur la machine `imag-mathappli-01.e.ujf-grenoble.fr`.

1.3 Documentations

Les questions peuvent être posées par mail à Jean-Guillaume.Dumas@imag.fr. Des documentations sont proposées[†] :

- Ce polycopié : `$JGD/BlobWar.pdf`
- Des fichiers sources (horloge, puissance 4, etc.) : `$JGD/Sources/`
- Le standard C++ : `$JGD/C++draft/`
- Documentations C++ online : `http://www.icce.rug.nl/documents/cplusplus`, `http://www.cs.cf.ac.uk/Dave/C/CE.html`
- Référence OpenMP : `http://www.plutospin.com/files/OpenMP_reference.pdf`
- une documentation sur la STL : `http://www.cs.rpi.edu/~musser/stl-book`, `http://www.sgi.com/Technology/STL/index.html` (ce dernier existe en version postcript : `http://www.cs.rpi.edu/~musser/doc.ps`)
- Généralités sur l'évaluation : `http://perso.wanadoo.fr/alemanni/page3.html`
- Un cours sur la théorie des jeux : `http://www.ics.uci.edu/~eppstein/180a/w99.html`

1.4 Sujet

1.4.1 Première partie : le jeu

1. Implémenter un plateau de Jeu $m \times n$ pour jouer à la Guerre des blobs à deux joueurs humains.
2. Implémenter un algorithme de jeu par ordinateur utilisant l'élagage en «alpha-bêta» d'arbre minimax. L'ordinateur doit être capable de jouer *un coup*, pour les blancs tout comme il doit être capable de jouer pour les noirs, et ce à partir de *n'importe quelle position de départ*. L'utilisateur pourra définir la stratégie de l'ordinateur en spéficiant au moins l'un des deux paramètres suivants : soit la profondeur maximale d'évaluation, soit le temps maximal de chaque coup.
3. Les humains comme l'ordinateur doivent respecter les règles proposées à la section 1.5.
4. L'interface doit respecter les spécifications de la section 1.6.1.

Le respect de ces spécifications permettra d'organiser un **tournoi** où chaque coup sera en temps limité.

1.4.2 Deuxième Partie : optimisations

Il s'agit d'implémenter, puis d'évaluer par l'expérimentation ou l'analyse, différentes heuristiques stratégiques. L'évaluation se fera par le temps gagné, la profondeur maximale atteinte et éventuellement par une comparaison entre deux méthodes (une phase de jeu ordinateur contre ordinateur avec deux de vos versions, analyse comparée de l'élagage, etc.).

5. Les principales optimisations se font sur le parcours de l'arbre d'évaluation :
   - Gestion du temps et élagage efficace : l'alpha-bêta permet toujours un gain de temps plus important quand les solutions les meilleures sont examinées d'abord. Ainsi il est utile d'implémenter un alpha-bêta *itératif* : on commence par appeler l'algorithme sur une faible profondeur, pour *ordonner heuristiquement* les prochains coups à jouer. L'algorithme est ensuite relancé sur une profondeur plus grande. Cela permet d'une part d'élaguer plus efficacement et, d'autre part, de gérer le temps par coups : si, après un alpha-bêta de niveau k, le temps de recherche écoulé jusqu'à présent ne dépasse pas le temps maximal autorisé multiplié par le nombre moyen de coups possibles (le *facteur de branchement* peut être estimé autour de 10 pour le jeu de dames), on effectue un alpha-bêta de niveau $k+1$.
   - Parallélisme : pour permettre une exploration plus profonde, l'algorithme d'«alpha-bêta» sera parallélisé avec openMP et/ou MPI.
   - Tables de hachage : toute position peut être atteinte depuis de nombreuses autres par des cheminements différents. Il n'est pas nécessaire d'évaluer les positions plusieurs fois. Une position et son évaluation seront donc stockées pour être réutilisées dans l'évaluation. Si votre structure de donnée est trop imposante, la position peut être compressée : si par exemple le plateau comporte 50 cases possibles, 2 ou 4 entiers de 64 bits (`long long`) peuvent alors être suffisants pour les stocker. Ensuite, une table de hachage (par exemple, 1024 vecteurs de stockage) permet de retrouver une position rapidement en additionnant les 4 entiers, en prenant le reste modulo 1024 et en parcourant le «petit» vecteur de stockage associé.
6. Le deuxième point crucial réside dans la détermination, à chaque étape, de la liste des coups possibles. Cette procédure doit être la plus efficace. Une première approche consiste à parcourir l'ensemble du plateau et à reconstruire totalement cette liste à chaque étape. Une deuxième approche peut être de conserver la précédente liste et de la modifier à partir du coup qui vient d'être joué.
7. La dernière difficulté est la fonction d'évaluation. La différence en nombre de blobs est une fonction d'évaluation simple et relativement efficace. Dans ce projet nous nous

†. `$JGD` correspond à `/u/d/dumas1/BlobWar`, avec un miroir sur ma page personnelle : `http://www-ljk.imag.fr/membres/Jean-Guillaume.Dumas/Enseignements/Projet_M1MAI`

focalisons sur l'optimisation du parcours de l'arbre. Si les réalisations qui précèdent sont plutôt indépendantes du jeu, la fonction d'évaluation elle nécessite une connaissance profonde de la stratégie du jeu considéré, ce qui n'est pas notre propos ici. Une fois les optimisations structurelles réalisées il est possible d'améliorer la fonction d'évaluation, par exemple en considérant aussi la position des pions sur le plateau.

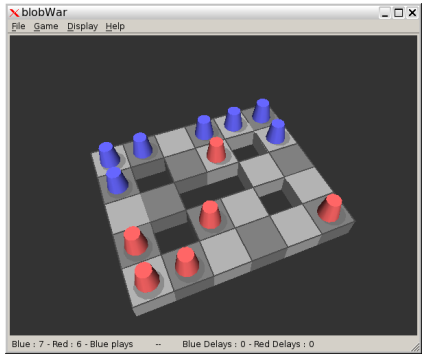
1.5 Règles de la guerre des blobs

1. Le plateau est une grille $m \times n$ sur laquelle se trouvent des cases et des trous. Les trous sont bien sûr inaccessibles.
2. Au début de la partie, chaque adversaire dispose de k pions (bleus et rouges) répartis de manière symétrique sur le plateau.
3. Les bleus commencent.
4. Un pion peut se cloner sur une case adjacente libre.
5. Un pion peut sauter sur une case libre située une case au delà de ses cases adjacentes (saut de deux cases ou déplacement de cavalier).
6. À la fin du mouvement (l'un des deux choix précédents) tous les blobs adverses adjacents au pion respectivement crée ou déplacé sont «capturés» (ils changent de couleur) pour devenir des votres.
7. Le jeu se termine quand toutes les cases sont remplies.
8. Si un joueur ne peut plus jouer toutes les cases libres restantes sont automatiquement remplies de blobs adverses.
9. Le gagnant est le joueur possédant le plus de blobs à la fin du jeu.

1.6 Spécifications

1.6.1 Interface

Les cases sont numérotées `[0..(m-1)][0..(n-1)]` en partant du côté des bleus (B). Le plateau peut donc ressembler à la figure de gauche, mais sera supposé compressé dans un fichier au format de droite (la première ligne contient les dimensions, puis chaque ligne est composé de n caractères : les pions bleus sont indiqués par un 'B', les pions rouges par un 'R', les autres cases par des '.' et les trous par des 'X').




```
5 6
BB.BBB
BX.RXB
..XX..
RXR.X.
RR...R
```

Les coups sont désignés par la suite des positions du pion. Par exemple, le coup `(0,1)>(1,2)` signifie que le pion bleu de la case `(0,1)` s'est cloné sur la case `(1,2)`.

1.6.2 Entrées sorties de votre programme

Votre programme prendra en entrée, sur la ligne de commande, trois paramètres : un fichier contenant un plateau au format ci-dessus, un temps maximal en secondes avant de jouer (positif pour jouer les bleus, négatif pour jouer les rouges) et un nombre maximal de demi-coups avant la fin du jeu.

Votre programme affichera sur la sortie standard (`std::cout`) un coup possible.

1.6.3 Structures de données

Si, dans son ensemble la structure de votre programme est laissée totalement libre, les points suivants sont obligatoires :

- Une classe `Coup` sera implémentée. Outre les constructeurs et accesseurs, elle comportera des opérateurs d'entrée/sortie (`operator<<` et `operator>>`) respectant l'interface.
- Une classe `Plateau` sera implémentée. Elle possédera une méthode `eval()` calculant l'évaluation du plateau courant. Deux méthodes `bouger(...)` et `enlever(...)` doivent permettre, respectivement, de jouer et de reprendre un coup. Une méthode de construction ou de mise à jour de la liste des coups possibles pourra être prévue.
- Là encore, les opérateurs d'entrée/sortie sont requis.
- Pour pouvoir paralléliser, l'algorithme d'«alpha-beta» sera implémenté à part dans une fonction classe et les variables globales seront évitées.

1.6.4 Contrôle du temps

U ne classe `Timer` est à votre disposition pour le contrôle du temps. Elle se trouve dans les fichiers `givtimer.[hC]` du répertoire `Sources`. L'interface est simple :

```
Timer Global; Global.clear();
// ...
Timer chrono; chrono.clear();
chrono.start();
plateau.alphabeta( /* ... */ );
chrono.stop();
double cpu_time = chrono.usertime();
double real_time = chrono.realtime();
double system_time = chrono.systime();
// ...
Global += chrono;
cerr << "Temps global : " << Global << endl;
```

1.7 Rapport

Les comptes rendus (≈ 10 pages) doivent être remis à la fin du **tournoi du mercredi 19 mars 2014** ou envoyés par courriel au plus tard le **lundi 23 mars 2014**. Ils doivent contenir : une **brève description** des structures de données choisies ; une **analyse synthétique** des expérimentations ainsi qu'un **comparatif** des heuristiques et optimisations employées ; plusieurs **analyses par gprof** successives doivent justifier des optimisations. **Une ou deux** situations précises (deux positions : initiale + quelques coups plus tard) seront détaillées et analysées. Des **listings partiels** abondamment commentés peuvent être joints pour éclairer un point précis. **Une annexe**, réalisée pendant le tournoi (manuscrite), présentera les résultats et commentaires des matches effectués.

1.8 Interface graphique

Le but de ce TP est de programmer efficacement en C++. L'interface graphique est donc fournie. Son utilisation n'est pas obligatoire. L'exécutable est disponible sur `imag-mathappli-01.e.ujf-grenoble.fr` : `Viewer/blobWar`.

Il est également possible d'utiliser une interface textuelle : `$JGD/blobTerm`.

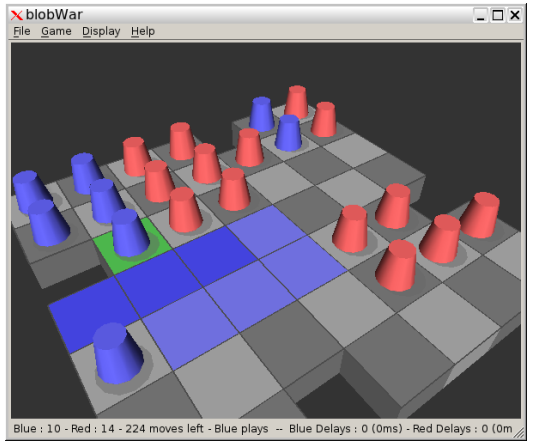
1.8.1 Paramètres de blobWar et blobTerm

blobWar est une interface graphique. Il est possible de lancer l'exécution de plusieurs manières : `blobWar [plateau] [joueur1 temps1] [joueur2 temps2] [coups]`

> `blobWar` : sans paramètre, l'interface suppose deux joueurs à la souris.

> `blobWar blob_JGD -2.4` : l'interface suppose un joueur à la souris et un joueur par le programme blob_JGD, avec un intervalle de temps par demi-coup de 2.4 secondes. Comme le temps est négatif le programme est supposé jouer avec les rouges.

> `blobWar blob_JGD 1 blob_JGD -2.4` : l'interface suppose deux joueurs par programme. Ici blob_JGD est schizophrène et joue contre lui-même, avec 1 seconde pour les bleus, 2.4 secondes pour les rouges.



> `blobWar plateau.bwb` : un paramètre, l'interface suppose deux joueurs à la souris sur le plateau de jeu "plateau.bwb".

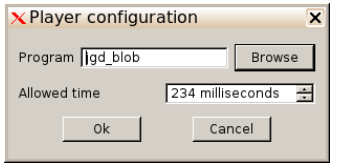
> `blobWar plateau.bwb blob_JGD -2.4`

> `blobWar plateau.bwb blob_JGD 1 blob_JGD -2.4`

> etc.

1.8.2 Modification des paramètres en cours de jeu

Il est possible de changer les paramètres de agora en cours d'exécution. Il suffit de cliquer sur File –> Load Game pour changer le plateau ; sur Game –> Configure, pour faire apparaître une fenêtre de choix :



ATTENTION : le point ('.') est requis pour les temps de `blobTerm` (par exemple 1 seconde et 3 dixièmes : `1.3`), alors que c'est la virgule qui est utilisée par `blobWar` (par exemple 1 seconde et 3 dixièmes : `1,3`).

1.9 Barême indicatif

Programmation (structures de données, alpha-beta, gestion des entrées/sorties, de l'évaluation, de la profondeur, du temps, etc.)	7 points
Puissance du programme (profondeur maximale moyenne pour un temps donné, classement au tournoi, etc.)	5 points
Une optimisation (Parallélisation ou tables de hachage ou ...)	3 points
Qualité du rapport (cf. 1.7)	5 points

La note finale du mini-projet ne sera pas une note de groupe. Les sources devront êtres séparées en plusieurs fichiers (une dizaine maximum) comportant chacun *un seul nom*, celui du responsable de ce fichier. Les 7 points de programmation seront attribués par personne au vu de ces fichiers.

1.10 Calendrier prévisionnel

Semaines	Contenu théorique	Avancement du TP
3	Théorie des jeux	Présentation du TP.
4	Programmation efficace : gestion de la mémoire (localité, contiguïté, réutilisation). Compilateurs et optimisations.	Structures de données : Plateau, Coups, mouvements, évaluation.
5	Une première évaluation des structures de données doit avoir eu lieu. Minimax, «alpha-beta», «alpha-beta» itératif. Gestion du temps. Tables de hachage. Heuristiques pour le jeu à deux.	La liste des coups possibles à chaque position doit être établie.
6	Il doit être possible de jouer avec une première vérification des coups et le calcul de la liste des coups possibles. La structure de jeu humain-ordinateur doit être en place. *Un rapport à mi-parcours ainsi que les sources et un exécutable (après* `strip`*) seront copiés dans le répertoire*† `$JGD/Executables/Trinome_##` *(ou remis par courriel à Jean-Guillaume.Dumas@imag.fr) pour le* **mardi 11 février 2014, 23h59**.	
7	L'«alpha-beta» doit fonctionner.	
8	L'«alpha-beta» itératif doit fonctionner pour la gestion du temps. Après une étude des performances par `gprof`, une analyse des optimisations possibles doit être effectuée : des tables de hachage ou encore une parallélisation avec openMP et/ou MPI doivent augmenter la profondeur moyenne d'évaluation ou le nombre de situations (`eval()`) explorées.	
9	Le jeu séquentiel doit être terminé. L'«alpha-beta» doit être mis sous forme de fonction classe indépendante des structures de données du jeu. Ne pas oublier de conserver à part la version séquentielle la plus efficace.	
11	L'«alpha-beta» doit être parallèle avec openMP et/ou MPI ou fonctionnner avec des tables de hachage.	
12 : **Tournoi**	Au plus tard le **mardi 18 mars 2014 à 23h59** vous aurez créé un répertoire† `Trinome_##` dans le répertoire `$JGD/Executables`. À cette heure je lancerai `make` dans ce répertoire. Un exécutable† `blob_##` devra alors être produit et servira pour le tournoi.	

†. où ## est le numéro du trinôme.

2 Théorie des Jeux à deux

2.1 Arbres d'évaluation

2.1.1 Minimax

L'algorithme Minimax permet de choisir un "meilleur" successeur dans un arbre où chaque niveau est successivement à maximiser et à minimiser. L'algorithme parcourt l'arbre en profondeur d'abord.

```
function MINIMAX(N) is
begin
   if N is a leaf then
        return the estimated score of this leaf
   else
        Let N1, N2, .., Nm be the successors of N;
        if N is a Min node then
           return min{MINIMAX(N1), .., MINIMAX(Nm)}
        else
           return max{MINIMAX(N1), .., MINIMAX(Nm)}
end MINIMAX;
```

Cet algorithme s'adapte aux jeux en considérant que la maximisation est le choix du meilleur coup possible et que la minimisation représente le choix de l'adversaire en supposant que celui-ci choisit aussi la meilleure position pour lui (d'après nous).

C'est cette dernière supposition qui est très forte.

```
double ami (Plateau& Jeu, int profondeur ) {
   if ( profondeur <= 0 ) return Jeu.eval();
   // Attention : éviter la copie suivante
   Plateau : :possibles Liste = Jeu.liste_coup();
   double eval = -INFINI;
   for( Plateau : :possibles : :const_iterator iter = Liste.begin();
       iter != Liste.end();
       ++iter ) {
       Jeu.bouger( *iter ); // je joue
       eval = MAX ( eval, ennemi ( Jeu, profondeur-1 ) );
       Jeu.remettre ( *iter );
   }
   return eval;
}
```



```
double ennemi (Plateau& Jeu, int profondeur ) {
   if ( profondeur <= 0 ) return Jeu.eval();
   Plateau : :possibles Liste = Jeu.liste_coup();
   double eval = +INFINI;
   for( Plateau : :possibles : :const_iterator iter = Liste.begin();
       iter != Liste.end();
       ++iter ) {
       Jeu.bouger( *iter ); // l'autre joue
       eval = MIN ( eval, ami ( Jeu, profondeur-1 ) );
       Jeu.remettre ( *iter );
   }
   return eval;
}
```

```
int main(int argc, char ** argv) {
   ...
   ami ( Jeu, profondeur_max )
   ...
}
```

2.1.2 α-β

ALPHA-BETA est une méthode de "Branch and Cut" réduisant le nombre de noeuds explorés par la stratégie "Minimax" : pour chaque noeud, elle calcule la valeur de la position ainsi que deux valeurs, alpha et beta.

Alpha : une valeur **toujours plus petite** que la vraie évaluation. Au début du parcours Alpha vaut -INFINI pour un noeud quelconque et la valeur de l'évaluation pour une feuille. Ensuite, pour un noeud à maximiser, alpha vaut la plus grande valeur de ses successeurs et, pour un noeud à minimiser, alpha est celui du prédécesseur.

Beta : une valeur **toujours plus grande** que la vraie évaluation. Au début du parcours beta vaut INFINI pour un noeud quelconque et la valeur de l'évaluation pour une feuille. Ensuite, pour un noeud à maximiser, beta est celui du prédécesseur et, pour un noeud à minimiser, beta vaut la plus petite valeur de ses successeurs.

Ce qui est garanti :

- La valeur d'un noeud ne sera jamais plus petite que alpha et jamais plus grande que beta.
- alpha ne décroît jamais, beta ne croît jamais.
- Quand un noeud est visité en dernier, sa valeur est celle de alpha si ce noeud est à maximiser, et celle de beta sinon.

```
double ami_ab (Plateau& Jeu, int profondeur, double A, double B ) {
    if ( profondeur <= 0 ) return Jeu.eval();
    double alpha = A, beta = B;
    Plateau : :possibles Liste = Jeu.liste_coup();
    for( Plateau : :possibles : :const_iterator iter = Liste.begin();
         iter != Liste.end();
         ++iter ) {
         Jeu.bouger( *iter ); // je joue
         alpha = MAX( alpha, ennemi_ab ( Jeu, profondeur-1, alpha, beta ) );
         Jeu.remettre ( *iter ) ;
         if (alpha >= beta) return beta;
    }
    return alpha;
}

double ennemi_ab (Plateau& Jeu, int profondeur, double A, double B ) {
    if ( profondeur <= 0 ) return Jeu.eval();
    double alpha = A, beta = B;
    Plateau : :possibles Liste = Jeu.liste_coup();
    for( Plateau : :possibles : :const_iterator iter = Liste.begin();
         iter != Liste.end();
         ++iter ) {
         Jeu.bouger( *iter ); // l'autre joue
         beta = MIN( beta, ami_ab ( Jeu, profondeur-1, alpha, beta ) );
         Jeu.remettre ( *iter ) ;
         if (alpha >= beta) return alpha;
     }
    return beta;
}

int main(int argc, char ** argv) {
    Plateau Jeu;
    // ...
    ami_ab ( Jeu, profondeur_max, -INFINI, INFINI );
    ;
    // ...
}
```


2.2 Variantes et optimisations d'Alpha-Beta

En général, les améliorations de l'algorithme alpha–beta sont de trois types :

Trier : pour améliorer l'ordre d'examen des noeuds. Cela permet de faire beaucoup plus de coupes.

Réduire : plus la fenêtre de recherche $(\beta - \alpha)$ est petite, plus il y a de coupes.

Réutiliser : sauvegarde des résultats pour le cas où ils ré-apparaîtraient (parce qu'une position est accessible par différents mouvements, parce que l'on a relancé alpha–beta, avec des paramètres différents, etc.).

Dans la suite, le **facteur de branchement** désigne la taille moyenne de la liste des coups à chaque étape (environ 35 aux échecs, de 8 à 10 aux dames, etc.).

La **profondeur** d'un algorithme de recherche, est le nombre de demi-coups que celui-ci a exploré en avance (aux échecs les meilleurs programmes regardent actuellement 8/9 demi-coups, cela monte à 11/12 pour le jeu othello, entre 9 et 12 pour les dames internationales 10×10).

2.3 Aspects de programmation efficace

Outre l'algorithmique (méthode alpha-beta et les optimisations que nous verrons sections suivantes), la programmation en tant que telle est fondamentale pour les performances du jeu. Tout d'abord, si l'aspect graphique est important pour l'esthétique, comme celui-ci n'est utilisé qu'une fois par coup, son optimisation temporelle n'est pas essentielle. On se concentrera plutôt sur les structures de données et quelques fonctions :

1. La fonction la plus appelée est la **fonction d'évaluation**. Pour une descente très profonde et un jeu systématique, on choisira une fonction extrêmement simple (même si elle est beaucoup trop simpliste). Pour une descente moins profonde et un jeu plus "intuitif", il sera alors possible de compliquer cette évaluation.
2. La quasi totalité du temps restant se passe dans la **génération des coups possibles**. Il faut donc impérativement *choisir ses structures de données (cases du plateau, pièces, coups) de manière à minimiser le temps de la génération des coups possibles.*
3. Enfin, les fonctions **bouger** (simuler un coup) et **remettre** (annuler cette simulation) sont les autres "gourmands". Elles sont appelées exactement autant de fois l'une que l'autre, il faut donc les optimiser équitablement.

2.4 Méthodes de tri

2.4.1 Méthode de l'ordonnancement quantitatif

On explore en premier les situations dont le nombre de coups légaux issus est le plus faible.

2.4.2 Descente itérative

Avec un facteur de branchement proche de 8, le coût d'un alpha–beta de profondeur $d+1$ est environ de 3 à 4 fois plus élevé que le coût de l'algorithme de profondeur d. Ainsi, il ne coûte pas tellement plus cher d'exécuter l'algorithme à une profondeur d, puis à une profondeur $d+1$.

L'**alpha–beta itératif** est donc en au moins deux phases : une première passe permet d'ordonner les coups du meilleur au plus mauvais à une profondeur k, l'exécution à une profondeur $k+j$ produit alors plus de coupes que si elle avait été lancée directement.

2.4.3 Descente itérative et gestion du temps

Un autre avantage est pour une utilisation à temps limité : on effectue un premier alpha–beta à une profondeur faible k, puis à toutes les profondeurs suivantes, jusqu'à avoir dépassé la limite de temps (ou jusqu'à ce que l'estimation prédise un dépassement au prochain appel).

2.4.4 Gestion globale du jeu

La gestion du jeu peut-être de deux sortes au moins : une gestion globale, où l'ordinateur va jouer ses coups et attendre à chaque fois le coup extérieur, ou une gestion à un coup, où un plateau représentant le jeu à un instant donné est fourni et l'algorithme doit déterminer uniquement le meilleur prochain coup.

La gestion à un coup est la plus simple, il faut optimiser en priorité les **structures de données** pour faire une descente la plus profonde possible.

Pour la gestion globale, les ressources sont en fait beaucoup plus importantes. Tout d'abord, il est possible de stocker des informations à partir des coups précédents dans des tables. Nous verrons comment dans les sections suivantes, mais en particulier :

- La liste des coups possibles peut être seulement mise à jour, et non plus reconstruite à chaque étape. Les tris précédents étant également réutilisés.
- La fonction d'évaluation peut être stockée pour une position donnée et n'a plus besoin d'être systématiquement calculée.
- Pendant que le joueur extérieur "réfléchit", l'ordinateur peut commencer à explorer l'arbre des coups suivants.

2.5 Tables

2.5.1 Tables de transposition

Il est possible d'atteindre une même position par plusieurs chemins différents. On stocke alors les positions, leur évaluation et la profondeur associée dans des tables. Retrouver un noeud déjà évalué dans les tables permet alors les optimisations suivantes :

- Si le noeud a déjà été évalué à une profondeur au moins aussi grande que celle désirée, sa valeur devient celle stockée.
- Si il a été évalué à une profondeur moins grande, cette valeur sert pour ordonner les noeuds du meilleur au moins bon.

Pour permettre une recherche rapide dans les tables, celle-ci sont en général implémentées par des tables de hachage (cf section 2.5.3). Il est en outre possible de stocker une table par profondeur évaluée.

2.5.2 Tables de réfutation

Le problème des tables précédentes est leur taille importante.

Avec un shéma itératif, il est possible de stocker plutôt des tables de réfutation. C'est-à-dire que pour les mauvais coups, on garde en mémoire le chemin qui indique que c'est un mauvais coup, puis on en explore la suite à une profondeur plus grande. Si le coup reste mauvais, cela suffit pour le rejeter.

2.5.3 Utilisation des tables de hachage de la STL

```
struct Plateau {
  long long _pb, _pn, _rb, _rn;
  double eval();
};

struct Hasher {
#define _MUL 950706376UL
#define _MOD 2147483647UL
  size_t operator()( const Plateau& _p ) const {
    size_t tmp = (_p._pb * _MUL) % _MOD;
    tmp = ((tmp + _p._pn) * _MUL) % _MOD;
    tmp = ((tmp + _p._rb) * _MUL) % _MOD;
    return  ((tmp + _p._rn) * _MUL) % _MOD;
  }
};
```

```
typedef hash_map< Plateau, double, Hasher > Hash_t;

void Memory_ab(Plateau& Jeu, Hash_t& Table, int profondeur ) {
  ...
  Hash_t : :const_iterator trouve = Table.find( Jeu );
  if (trouve != Table.end() )
      return (*trouve).second;
  else
      return Table[ Jeu ] = alpha_beta(Jeu, profondeur);
}
```

Des références sur les tables de hachage sur le net :

- http ://www.sgi.com/tech/stl/AssociativeContainer.html
- http ://www.sgi.com/tech/stl/HashedAssociativeContainer.html
- http ://www.sgi.com/tech/stl/hash_map.html

2.6 Fenêtres

2.6.1 PVS

L'algorithme PVS - Principal Variant Search - est un exemple typique d'heuristique. Il fait l'hypothèse d'une certaine "continuité" des valeurs des successeurs d'une position ; en effet, en général les successeurs d'une position ont des valeurs proches. On se sert donc de la valeur du premier successeur pour définir les bornes d'une fenêtre de recherche serrée, ce qui permettra beaucoup de coupures. Cette heuristique peut ne pas marcher : si la valeur d'un successeur tombe en dehors de la fenêtre prévue, il faut refaire le calcul.

En pratique : supposons que l'on ait à maximiser et que $\alpha = 3$, $\beta = +\infty$. On effectue le prochain appel, pour le coup à évaluer suivant, par ennemi_ab avec 3 et $3 + 1$. Si le résultat vaut 3 cela veut dire que la vraie valeur est 3 ou moins, elle ne sert donc pas. Si le résultat vaut 4, cela veut dire que la vraie valeur est 4 ou plus, il faut donc relancer ennemi_ab avec $\alpha = 4$ et le précédent β.

2.6.2 Limitation a priori de la fenêtre de recherche

Par exemple autour de la valeur de la profondeur précédente, ou bien encore autour de la valeur de la fonction d'évaluation appliquée à la situation à étudier, plutôt qu'autour de la valeur de la situation précédente (PVS).

2.6.3 Recherche par aspiration

L'alpha-beta commence avec les valeurs $-\infty$ et $+\infty$. Une connaissance du jeu (nombre de pions restants, etc.) permet de restreindre la taille de la fenêtre de départ.

Si la valeur tombe dans l'intervalle choisi, alors cet intervalle était correct. Sinon, il faut relancer l'alpha-beta.

2.7 MTD(f)

2.7.1 A. Plaat

La méthode MTD(f) [A. Plaat, http://plaat.nl/mtdf.html] combine les idées précédentes : réduire la fenêtre au maximum (de taille 1), même si il faut relancer l'alpha-beta plusieurs fois, et utiliser des tables pour stocker les valeurs déjà calculées (si plusieurs alpha-beta ré-évaluent les mêmes positions).

```
double MTDf(Plateau& Jeu, int depth, double f) {
  double g = (f>MINFTY) ?f :0;
  double lowerbound = MINFTY;
  double upperbound = INFTY;
  double beta; int count = 0;
  for( ; lowerbound < upperbound; ++count)  {
    if (count > 5) {
      g = Memory_ennemi_ab(Jeu, depth, lowerbound, upperbound);
      break;
    }
    if (g == lowerbound)
      beta = g+1;
    else
      beta = g;
    g = Memory_ennemi_ab(Jeu, depth, beta - 1, beta);
    if (g < beta)
      upperbound = g;
    else
      lowerbound = g;
  }
  return g;
}
```

2.7.2 Variante à deux pas

```
double MTDFdouble(Plateau& Jeu, int depth, double f) {
  double g = (f>MINFTY) ?f :0;
  double lowerbound = MINFTY;
  double upperbound = INFTY;
  double beta; int count = 0;
  for(  double diffbound = DINFTY;
        diffbound > 1 ;
        ++count, diffbound = upperbound-lowerbound)  {

    if (count > 4) {
      g = Memory_ennemi_ab(Jeu, depth, lowerbound, upperbound);
      break;
    }
    if (g == lowerbound)
      beta = g+1;
    else if (g == upperbound)
      beta = g-1;
    else
      beta = g;
    g = Memory_ennemi_ab(Jeu, depth, beta - 1, beta + 1);
    if (g < beta)
      upperbound = g;
    else if (g > beta)
      lowerbound = g;
    else
      break;

  }
  return g;
}
```

2.8 Accélérer la victoire

Pour terminer la partie, il est possible de donner une évaluation différente aux victoires suivant le nombre de pions restants, le nombre de coups qu'il faut pour atteindre cette position, etc.

2.9 Heuristiques

2.9.1 Profondeur sélective

Dans un cadre itératif, chaque coup peut être examiné avec une profondeur qui lui est propre, suivant différents critères : sa précédente évaluation était très rapide donc on incrémente la

profondeur en peu plus, et vice-versa ; les coups avec une forte probabilité d'être bons sont examinés avec une profondeur faible tandis que les autres sont examinés avec une plus grande profondeur

Cette heuristique rend, en général, l'ordinateur plus humain !

2.9.2 La recherche focalisée

Il s'agit ici d'un shéma itératif dans lequel à partir des derniers niveaux, seules les meilleures possibilités (80/90 %) sont explorées, les autres sont abandonnées. Attention, il s'agit d'une heuristique à manipuler avec précaution.

2.9.3 Passe

Si une position semble suffisamment bonne à une profondeur k, l'ordinateur peut voir ce qui se passerait si il ne jouait pas (il passe) à la profondeur $k+1$. L'idée est qu'en général ce choix est mauvais. Ainsi, si le résultat obtenu reste bon, la position doit être vraiment bonne, dans le cas contraire, il faut faire une recherche normale.