Jeu de GO - Loïc Bine/Laurent Freyss

Documentation Structures & Algorithmes

Gestion des tours

Les tours de jeu sont gérés à chaque "click" et le programme donne ainsi la main à l'un ou l'autre joueur. C'est également lors du click que toute la mécanique du jeu s'enclenche. Un tour comprend:

- Le dessin des pions des joueurs
- Le repositionnement du click sur un des points du goban
- La vérification de coup : Répétition ainsi que les intersections vides
- La vérification de la capture de pions adverses
- L'incrémentation du compteur de tours
- Sauvegarde du plateau de jeu

Durant son tour le joueur peut également :

- Sauvegarder la partie
- Passer son tour

Algorithme de repositionnement

```
int i,j;
int largeurBordure = getLargeurBordure();
int taillePierre = getTaillePierre();
for(i = -1; i<getNbPierres()-1; i++) {
    if((x >= largeurBordure + i*taillePierre + taillePierre/2) && (x <
largeurBordure + (i+1)*taillePierre + taillePierre/2)) {
        for(j = -1; j<getNbPierres()-1; j++) {
            if((y >= largeurBordure + j*taillePierre + taillePierre/2) &&  (y <
largeurBordure + (j+1)*taillePierre + taillePierre/2)) {
                int xp = largeurBordure + (i+1)*taillePierre;
                int yp = largeurBordure + (j+1)*taillePierre;
            }
        }
    }
}
```

Gestion des pions et des Joueurs

Dans le jeu de Go, pions et joueurs son liés. Nous avons donc jugés intéressant d'affecter à une structure "Joueur" un tableau de listes doublement chaînées de pions. La taille de ce tableau est ensuite gérée via un entier "nbLliste" qui nous permet de parcourir toutes les listes d'un joueur.

```
typedef enum PlayerType{
Blanc,Noir
} PlayerType;
```

```
typedef struct Player{
    //Nom du Joueur
    char* nom;
    //Blanc ou Noir
    PlayerType type;
    //Tableaux de listes de pions
    Liste** pierres;
    //Compteur de listes
    int nbListe;
    //Booléen pour savoir si le jour souhaite terminer la partie
    bool aPasser;
}Player;
```

Ainsi nous pouvons réaliser plusieurs opérations sur ces listes, telle que :

- La fusion de chaînes
- Le calcul des degrés de libertés
- L'ajout ou la suppression de pions dans la chaîne

Ayant déjà développer la classe liste_double contenant des int, nous avons choisi de stocker les coordonnées d'un pion dans un entier de la liste :

```
int getX(int c){
    return c >> 16;
}

int getY(int c){
    return c & 0xFFFF;
}

int getCoord(int x,int y){
    return x << 16 | y;
}
```

Gestion de la capture et de la fusion

Une chaîne de pions est considérée comme capturée si elle ne possède plus de degrés de libertés. Nous avons donc pu, en modifiant quelques peu la fonction apply() qui applique une fonction à chaque élément de la liste, calculer les degrés de libertés :

```
int getDegreLiberte(Liste* l) {
    Liste* copyListe = liste_vide();
    int cpt = 0;
    applyCompteur(l,&cpt,getCptDegre);
    return cpt;
}
```

Cette fonction fait appel à une fonction qui calcule les degrés de libertés du pion actuel.

```
void getCptDegre(int* val,int* cpt){
    int tab[4];
    getAdjacent(getX(*val),getY(*val), tab);
    Adjacent adj = isAnyPierreAdjacentInEveryTable(tab);
    if(tab[2] == -1) {
        *cpt += 2-adj.nbAdj;
    } else if(tab[3] == -1) {
        *cpt += 3-adj.nbAdj;
    } else {
        *cpt += 4-adj.nbAdj;
    }

}
```

Cette dernière utilise une structure que l'on nomme "Adjacent", qui permet de stocker temporairement les adjacents du pions.

```
typedef struct Adjacent{
    int* adjs;
    int nbAdj;
}Adjacent;
```

Grâce à cette structure, nous pouvons faire la somme des degrés de la chaîne et ainsi déterminer si elle est capturée. Si cette dernière l'est, nous supprimons la chaîne des chaînes du joueur.

La fusion fonctionne également sur le principe des adjacents. Ainsi si la pierre que le joueur veut poser se trouve proche de l'une de ses chaînes, la pierre sera ajoutée à cette chaîne.

Gestion de la répétition

La vérification de la répétition consiste à empêcher de recréer une situation s'étant produite au tour précédent.

Pour ce faire nous sauvegardons à chaque tour une copie du plateau dans un tableau d'entier à une dimension. Ayant déjà développé cette librairie en cours, il nous a été simple de l'implémenter. Les valeurs possibles sont :

- 0 : Aucun pion
- 1 : Joueur Blanc
- 2 : Joueur Noir

Pour sauvegarder la couleur du joueur à l'aide de la coordonnée nous utilisons :

```
void fill_plateau(int x, int y, int playerValue) {
    plateau[y*getNbPierres() + x] = playerValue;
}
```

Ensuite il nous suffit de comparer le tableau du tour avec le tableau du tour contenant le nouveau du coup du joueur et dire si il y a répétition :

```
bool repetition(int x, int y,int playerValue) {
    int size = getNbPierres()*getNbPierres();

    //Copie des plateaux à t-1 et t+1
    tourSuiv = copyPlateau(plateau,size);
    tourSuiv[y*getNbPierres() + x] = playerValue;
    //Parcours des deux tableaux
    for (int i = 0; i < size; i++) {
        if(tourSuiv[i] == playerValue){
            if(tourSuiv[i] != plateauT_1[i]){
                return false;
            }
        }
    }
    return true;
}
```

Gestion de la sauvegarde

Le format standard de sauvegarde du jeu de go est le standard smart game format. Celui dispose de nombreuses options, nous avons donc choisi de n'en conserver que quelques unes :

- AW [aa][ba]... : Les lettres entre crochets indique les coordonnées des pions blancs
- AB [aa][ba]... : Les lettres entre crochets indique les coordonnées des pions noir
- SZ[19] : Indique la taille du plateau

Aves ces options nous pouvons recréer la partie et la sauvegarder.

Pour se faire nous nous servons du tableau créé durant chaque tour et en enregistrons les coordonnées sous formes de lettres à l'aide d'un tableau de conversion.

```
SZ[19]
AW[ca][ea][ga][ac][gg]
AB[aa][ba][da][fa][ab][ag]
```