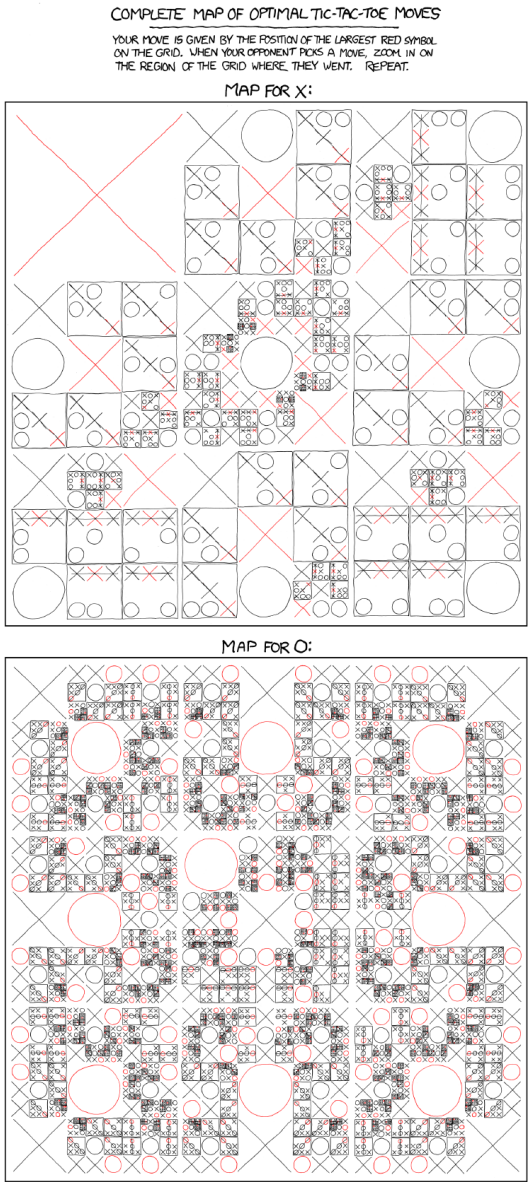
MPI* Info - TP1

Jeu de morpion



L. Valemberg

O. Caffier

1 Étude du jeu

1. Que peut-on dire du jeu `ttt(k,n)` quand $k > n$?
   Corrigé : Il est impossible de gagner à ce jeu car chaque ligne, colonne et diagonale peuvent comprendre au maximum n symboles.

2. Montrer qu'il existe une stratégie gagnante pour le premier joueur facile à mettre en place pour `ttt(2,n)` avec $n \geq 2$.
   Corrigé : Il suffit de placer son premier symbole dans un coin, on aura évidemment au moins 2 options au prochain tour, chacune amenant à une victoire certaine.

3. Montrer qu'il existe une stratégie gagnante pour le premier joueur facile à mettre en place pour `ttt(3,n)` avec $n \geq 4$.
   Corrigé :

Dans la suite du TP, on encode un jeu `ttt(k,n)` par le type suivant :

```c
struct TTT {
    int k;
    int n;
    int* grille;
};
typedef struct TTT ttt;
```

2 Quelques fonctions utiles

1. Écrire une fonction `ttt* init_jeu(int k, int n)` qui initialise un jeu de `ttt(k,n)` avec une grille vide (donc ne contenant que des zéros).
   Corrigé :

```c
ttt* init_jeu(int k, int n){
    ttt* res = malloc(sizeof(ttt));
    res->k = k;
    res->n = n;
    res->grille = malloc(n*n*sizeof(int));
    for (int i=0;i<n*n;i++){
        res->grille[i]=0;
    }
    return res;

}
```

2. Écrire une fonction `void liberer_jeu(ttt* jeu)` qui libère l'espace mémoire occupé par un jeu.
   Corrigé :

```c
void liberer_jeu(ttt* jeu){
    free(jeu->grille);
    free(jeu);
}
```

3. Écrire une fonction `int* repartition(ttt* jeu)` qui renvoie un tableau de taille 3 correspondant à la répartition des cases dans le jeu : `tab[i]` correspond au nombre d'occurrences de la valeur `i` dans la grille. La fonction renverra `NULL` si la grille contient des numéros autres que 0, 1 ou 2.
Corrigé :

```
int* repartition(ttt* jeu){
    int* res = malloc(3*sizeof(int));
    res[0]=0;
    res[1]=0;
    res[2]=0;
    int taille = jeu->n*jeu->n;
    for (int i =0; i<taille;i++){
        if (jeu->grille[i]!=0 && jeu->grille[i]!=1 && jeu->grille[i]!=2){
            return NULL;
        }
        res[jeu->grille[i]]++;

    }
    return res;

}
```

4. En déduire une fonction `int joueur_courant(ttt* jeu)` qui renvoie le numéro du joueur dont c'est le tour (en supposant que c'est toujours le joueur 1 qui commence). La fonction renverra 0 si toutes les cases de la grille sont remplies.
Corrigé :

```
int joueur_courant(ttt* jeu){
    int* rep = repartition(jeu);
    if (rep[0]==0){
        free(rep);
        return 0;
    }
    if (rep[1]>rep[2]){
        free(rep);
        return 2;
    }
    free(rep);
    return 1;


}
```

5. Écrire une fonction `void jouer_coup(ttt* jeu, int lgn, int cln)` qui joue un coup pour le joueur courant dans la grille à une ligne et une colonne données. La fonction devra afficher un message d'erreur et ne rien faire si la case correspondante ne contient pas 0. On fera la conversion entre un couple d'indices de ligne et de colonne et l'indice de la case dans le tableau unidimensionnel.
Corrigé :

```
void jouer_coup(ttt* jeu, int lgn, int cln){
    int joueur = joueur_courant(jeu);
    int nb_case = lgn*jeu->n + cln;
    if (jeu->grille[nb_case]!=0){
        printf("attention coup interdit\n");
        return;
    }
    jeu->grille[nb_case]=joueur;
}
```

3 Résolution du jeu

1. À quelle valeur `di`, en fonction de n, correspondent les directions Est, Sud-Est, Sud et Sud-Ouest?
   Corrigé : On a
   - Est : `di` = 1
   - Sud-Est : `di` = $n+1$
   - Sud : `di` = n
   - Sud-Ouest : `di` = $n-1$
2. Pourquoi n'est-il pas nécessaire d'envisager les 4 autres directions pour vérifier si un joueur est gagnant?
   Corrigé : Étant donné que l'on explore les cases "dans l'ordre croissant", les autres directions évoquées sont celles où les cases ont déjà été traitées par le programme, inutile d'y retourner.
3. Écrire une fonction `bool alignement(ttt* jeu, int i, int di, int joueur)` qui prend en argument un jeu, un indice de départ, une direction et un joueur et indique s'il existe un alignement gagnant pour le joueur depuis la case de départ, dans la direction donnée. On prendra garde, en gardant en mémoire la ligne et la colonne de la case courante, à ne pas sortir de la grille.
   Corrigé :

```
bool alignement(ttt* jeu, int i, int di, int joueur){
    int lgn = i/jeu->n;
    int cln = i % jeu->n;
    for (int j = 0;j<jeu->k;j++){
        if (cln<0||cln>=jeu->n || lgn<0 || lgn >= jeu->n){
            return false;
        }
        if (jeu->grille[cln+lgn*jeu->n]!=joueur){
            return false;
        }
        cln += ((di+1) % jeu->n)-1;
        lgn += (di+1)/jeu->n ;
    }
    return true;
}
```

4. En déduire une fonction `bool gagnant(ttt* jeu, int joueur)` qui indique si un joueur est gagnant ou non.
   Corrigé :

```
bool gagnant(ttt* jeu, int joueur){
    for (int i=0;i<jeu->n*jeu->n;i++){
        if (alignement(jeu,i,1,joueur)||alignement(jeu,i,jeu->n+1,joueur)||alignement(jeu,i,jeu->n-1,
            joueur)||alignement(jeu,i,jeu->n,joueur)){
            return true;
        }
    }
    return false;
}
```

5. Pour faire le calcul des attracteurs, il est nécessaire de garder des résultats pré-calculés en mé- moire. Pour ce faire, on encode une grille d'un jeu `ttt(k, n)` par un entier à n2 chiffres en base 3. Écrire alors une fonction `int encodage(ttt * jeu)` qui calcule un tel entier.
   Corrigé :

```
int encodage(ttt* jeu){
    int res =0;
    int puiss = 1;
    for (int i=0;i<jeu->n*jeu->n;i++){
        res+=jeu->grille[i]*puiss;
        puiss*=3;
    }
    return res;

}
```

On se donne une structure de dictionnaire `dict` implémentée par une table de hachage. On dispose des primitives suivantes :

— `void dict_free(dict* d)` libère la mémoire occupée par un dictionnaire
— `dict* create(void)` crée un dictionnaire vide
— `bool member(dict* d, int c)` teste l'appartenance d'une clé à un dictionnaire
— `int get(dict* d, int c)` renvoie la valeur associée à une clé dans un dictionnaire
— `void add(dict* d, int c, int v)` ajoute une association (clé, valeur) à un dictionnaire

6. Écrire une fonction `int attracteur(ttt* jeu, dict* d)` qui prend en argument un jeu et un dictionnaire et renvoie le numéro de l'attracteur auquel appartient la grille du jeu. La fonction devra mémoriser le résultat dans le dictionnaire s'il n'y est pas déjà avant de renvoyer la valeur. On considèrera qu'une position nulle est dans l'attracteur 0.
Corrigé :

```
int attracteur(ttt* jeu, dict* d){
    int nb = encodage(jeu);
    if (member(d, nb)){
        return get(d,nb);
    }
    int joueur = joueur_courant(jeu);
    if (gagnant(jeu,joueur)){
        add(d,nb,joueur);
        return joueur;

    }
    else if (gagnant(jeu,3-joueur)){
        add(d,nb,3-joueur);
        return 3-joueur;

    }
    else if (joueur==0) {
        add(d,nb,0);
        return 0;
    }
    int case_autre_joueur=0;
    int case_libre=0;
    for (int i = 0 ; i< jeu->n;i++){
        for (int j = 0;j<jeu->n;j++){
            if (jeu->grille[i*jeu->n+j]==0){
                jouer_coup(jeu,i,j);
                case_libre++;
                int att = attracteur(jeu,d);
                jeu->grille[i*jeu->n+j]=0;
                if (att==joueur){
                    add(d,nb,joueur);
                    return joueur;
                }
                if (att==3-joueur){
                    case_autre_joueur++;
                }

            }
        }
    }
        if (case_autre_joueur==case_libre){
            add(d,nb,3-joueur);
            return (3-joueur);
        }
        else {add(d,nb,0); return 0;}


}
```

7. En déduire une fonction `int strategie_optimale(ttt* jeu, dict* d)` qui détermine le coup optimal à jouer étant donné un jeu et un dictionnaire contenant des numéros d'attracteurs.
Corrigé :

```c
int strategie_optimale(ttt* jeu, dict* d){
    int joueur = joueur_courant(jeu);
    int att = attracteur(jeu,d);
    for (int i = 0; i<jeu->n * jeu->n; i++){
        if(jeu->grille[i]==0){
            jeu->grille[i]=joueur;
            int attbis=attracteur(jeu,d);
            if (att==attbis){
                return i;
            }
            jeu->grille[i]=0;
        }
    }
    return 0;
}
```

4 Jouer une partie

1. Écrire une fonction `void afficher(ttt* jeu)` qui affiche le jeu en console.
Corrigé :

```c
void afficher_ligne_separante(int n){
    printf("\n");
    for (int i=0;i<n;i++){
        printf("+-");
    }
     printf("+-\n");

}
void afficher(ttt* jeu){
    int n= jeu->n;
    printf(" ");
    for (int i=0;i<n;i++){
        printf(" %d",i);

    }
    afficher_ligne_separante(n);
    for (int i=0;i<n;i++){

        printf("%d|",i);
        for (int j=0;j<n;j++){
            if (jeu->grille[i*n+j]==1)
                printf("X|");
            else if (jeu->grille[i*n+j]==0)
                printf(" |" );

            else
                printf(" O|" );
        }
        afficher_ligne_separante(n);
    }

}
```

2. Écrire une fonction `void jouer_partie(int k, int n)` qui crée une partie de ttt(k, n) et permet de jouer contre l'ordinateur. La fonction devra :
   - demander si le joueur humain souhaite commencer ou non
   - demander à chaque coup du joueur humain la ligne et la colonne où il souhaite jouer
   - afficher la grille après chaque coup joué (par l'humain ou l'ordinateur)
   - arrêter la partie dès qu'un joueur a gagné ou quand la grille est pleine, et afficher le joueur gagnant.

   Corrigé :

```c
void jouer_partie(int k, int n){
    printf("souhaitez vous commencer? (o/n) \n");
    char x;
    scanf("%c",&x);
    ttt* jeu = init_jeu(k,n);
    int joueur_machine;
    if (x=='o'){
        joueur_machine = 2;

    }
    else if (x=='n'){
        joueur_machine=1;
    }
    else printf("erreur de saisie\n");
    dict* d = create();
    int joueur=1;
    int tour=0;
    while(joueur!=0 && !gagnant(jeu,1) && !gagnant(jeu,2)){
        printf("Tour %d\n", tour);
        tour++;
        afficher(jeu);
        if (joueur==joueur_machine){
            int i= strategie_optimale(jeu,d);
            jeu->grille[i]=joueur_machine;
        }
        else{
            int lg,c;
            printf("que voulez-vous jouer?\n");
            printf("donner la ligne :\n");
            scanf("%d",&lg);
            printf("donner la colonne\n");
            scanf("%d",&c);
            if (c<0 || c>=jeu->n || lg<0 || lg >=jeu->n){
                printf("cette case n'existe pas\n");
            }
            else{
                jouer_coup(jeu,lg,c);
            }
        }
        joueur=joueur_courant(jeu);

    }
    afficher(jeu);
    if (gagnant(jeu,joueur_machine)){
        printf("vous avez perdu\n");
    }
    else if (gagnant(jeu, 3-joueur_machine)){
        printf("vous avez gagne\n");
    }
    else printf("match nul\n");
    liberer_jeu(jeu);
    dict_free(d);

}
```