Rapport développeur: Sokoban v2.0

aslan.malsagov[at]edu.esiee.fr
E3FR-2 MALSAGOV Aslan
2017

1. Présentation

Ce rapport met en avant les aspects de la réalisation des modification sur le projet Sokoban 1. Sokoban 2 est une nouvelle version comportant nombreux modification et amélioration au niveau de développement tout en gardant l'aspect visuel et fonctionnelle de la première version du jeu. Cela permet à l'utilisateur facilement de migrer vers la nouvelle version.

2. Décomposition du projet

Le projet est composé de cinq principaux parties:

makefile	Fichier utilisé pour exécuter la compilation du projet
main.c	Fichier principale appelant les fonctions externes
sokoban.c	Fichier comportant les définition des fonctions et des variables globales
sokoban.h	Fichier contenant la déclaration des prototypes des fonctions
grille1.sok	Fichier contenant la grille de premier niveau avec ces dimensions

La description des chaqu'un de ces fichiers simplifiera...

3. Description des composantes

makefile

La déclaration des sept variables au début du fichier rend son évolution simple et rapide. Ainsi, il n'est plus besoin d'écrire les longues commandes de compilation, il suffit de taper «make» dans le dossier du projet.

La désignation des variables:

CC	Compilateur utilisée
CFLAGS	Les options appliqué lors de la compilation
LDFLAGS	Les options d'édition de liens
EXEC	Nom de(s) l'exécutable(s) généré
BIN	Liste des objets binaires généré pour le lancement du programme
SRC	Liste des fichiers sources contenant les fonctions
HEAD	Les fichiers contenant les prototypes

A la suite de la déclaration des variables on définit les règles de compilation.
La forme générale de la définitions de la règle est la suivante:

```
cible: dépendances
       commandes
```

Le makefile du projet présenté six règles:

all	....
sokoban	....
clean	Supprime les objets binaires .o ainsi que les fichiers cachés de sauvegarde
clear	Supprimer l'exécutable et l'archive tar et dans le cas d'un affichage d'un message sur la sortie d'erreur redirige le vers le périphérique **null**.
zip	Permet de créer un un dossier temporaire prédéfinie d'un nom d'utilisateur dans lequel on copie dans les sources d'extension .c, .h et .sok puis créer un archive **tgz** et enfin supprimé le dossier temporaire

main.c

Le fichier **main.c** est l'un des fichiers principaux du projets. Après la phase d'appelle aux directives de préprocesseur défini par le symbole **#**, le programme exécute la fonction **main()** qui est responsable des appel des fonction externes.
Le début de la fonction on commence par la déclaration des variables ce qui leur alloue un espace de la mémoire nécessaire selon leur type.
Puis on rentre dans la partie d'affectation des ces variables par appel aux fonctions externes.

```
Grille jeu;
Grille gCache;

jeu=initGrille(argv[1]);
gCache = copieGrille(jeu);
```

Au début on initialise deux grilles: la grille jeu - dans laquelle l'utilisateur joue et la grille caché qui est composé de la grille de jeu sans le Sokoban et sans les caisses. Elle permettra par la suite la fonction d'annulation de la commande.

ajaxVitre est un alias avec la commande UNIX pour effacer l'écran du terminal.

On fait appele à la fonction **getSokoban()** pour trouver son position dans la grille. S'il y présente ses coordonnées sont affichée, sinon la fonction **error()** est appelé avec le code correspondant.
La fonction **verif()** vérifie les dimensions de la grille ainsi que la présence des caisses, des cibles et de leurs égalité.

```
nouveauJeu(gCache, jeu);
positionSokoban=getSokoban(jeu);
ajaxVitre;

if((positionSokoban.x==0) && (positionSokoban.y==0))
        error(11); /* NO_SOKOBAN */
else
        printf("La position du Sokoban est en (%d;%d)\n", positionSokoban.x, positionSokoban.y);

verif(jeu);
```

Tant que la variable **encore** est vrai, la boucle **do while()** affiche la grille et le prompt, récupère la saisi de l'utilisateur dans le tableau d'instructions de **char** ayant la de taille MAXCH+1 et dans le switch selon la commande on fait appele a la fonction **joue()** en passant dans les paramètres, la grille jeu, les instructions(caractère par caractère) pour déplacer le Sokoban.
Dans le cas ou l'utilisateur rentre le caractère **q** qui correspond à l'instruction de quitter, on met la valeur de la variable **encore** à **false**, du coup cela permet de sortir de la boucle.
La condition **if()** vérifie que si la fonction **gagne()** renvoie la valeur true, dans ce cas on affiche le message correspondant à la félicitation avec le nombre de coups effectué pour gagner et on affecte la valeur **false** à la variable **encore** pour quitter la boucle **do while()**.

```
do{
        int i;
        afficheGrille(jeu);
        printf("> ");
        if(fgets(instruction, MAXCH, stdin)==NULL) continue;
        for(i=0; instruction[i]!='\n'; i++){
                ajaxVitre;
                positionSokoban=getSokoban(jeu);
                switch(instruction[i]){
                        case HAUT:    joue(jeu, instruction[i], positionSokoban, gCache); break;
                        case BAS:     joue(jeu, instruction[i], positionSokoban, gCache); break;
                        case GAUCHE:  joue(jeu, instruction[i], positionSokoban, gCache); break;
                        case DROITE:  joue(jeu, instruction[i], positionSokoban, gCache); break;
                        case AIDE:    aide(); break;
                        case QUITTER: encore=false; break;
                        default: printf("Instruction inconnu\n"); break;
                }
                coups++;
        }
        if(gagne(jeu, getSokoban(jeu), gCache)==true){
                printf("Félicitations! Tu as gagné en %d coups\n", coups);
                encore=false; /* On met fin au jeu */
        }
} while(encore==true);
```

sokoban.c

Le fichier Sokoban**.c** contient l'implémentation des fonctions du jeu.

Tout en début de la fonction **getSokoban()** on déclare la variable **position** de la structure **Position** et affectant les valeurs **0;0** aux variables contenant dans la structure. La double boucle **for()** cherche le caractère **'S'** dans la grille de jeu récupéré dans le paramètre en parcourant chaque ligne. Elle retourne la position du Sokoban dans la grille passée en paramètre, ou **{0,0}** affecté à la variable **position** s'il n'est pas trouvé.

```
Position getSokoban(Grille g){
        int i, j;
        Position position = {0, 0};

        for(i=0; i<g.h; i++){
                for(j=0; j<g.L; j++){
                        if(g.grille[i][j]=='S'){
                                position.x = i;
                                position.y = j;
                        }
                }
        }
        return position;

}
```

La fonction **nouveauJeu()** prend en paramètre deux grilles. Elle recopie la grille initiale, **init.grille,** passée en premier paramètre dans une nouvelle grille de jeu, **g.grille,** passée en deuxième paramètre. Puis elle remplace tout les caisses et le Sokoban trouvé dans la grille initiale par le caractère **VIDE**.

```
void nouveauJeu(Grille init, Grille g){
        int i, j;
        for(i=0; i<g.h; i++){
                for(j=0; j<g.L+1; j++){
                        g.grille[i][j] = init.grille[i][j];
                        if((init.grille[i][j]==CAISSE)||(init.grille[i][j]==SOKOBAN))
                                init.grille[i][j]=VIDE;
                }
        }
}
```

La fonction **verif()** effectue quelques vérification sur la grille passé en paramètre. Notamment elle vérifie l'initialisation correcte de la grille, sinon elle appelle la fonction **error()** avec le code correspondant. Elle vérifie également qu'il a autant des caisses que de cibles en parcourant la grille et comptent leur nombre, puis suit la vérification dans la condition **if()**.

```
Boolean verif(Grille g){
        int i, j, nbCaisses=0, nbCibles=0;

        for(i=0; i<g.h; i++){
                j=0;

                while(g.grille[i][j]!='\0')
                        j++;
                if(g.L!=j)
                        error(12); /* G_INCORECT */
        }


        for(i=0; i<g.h; i++){
                for(j=0; j<g.L; j++){
                        if(g.grille[i][j]==CAISSE)
                                nbCaisses++;
                        if(g.grille[i][j]==CIBLE)
                                nbCibles++;
                }
        }

        if(nbCaisses==0)
                error(13); /* UNE_CAISSE */

        if(nbCaisses!=nbCibles)
                error(14); /* DIF_CAICIB */
        return true;
}
```

La fonction **compte()** permet de compter par la variable **nb_occ** dans la grille en la parcourant, le nombre de fois qu'apparaît un caractère **c** passé en paramètre dans les valeurs de **grille[i][j]**.

```
int compte(Grille g, char c){
        int i, j, nb_occ=0;

        for(i=0; i<g.h; i++)
                for(j=0; j<g.L; j++)
                        if(g.grille[i][j]==c)
                                nb_occ++;
        return nb_occ;
}
```

La fonction **aide()** est appelé lorsque l'utilisateur demande de l'aide par la commande **a**. Elle est composé de trois simples fonction **printf()** qui affichent les symboles et les commandes.

```
void aide(){
        printf("Aide:\n");
        printf("  (%c)MUR, (%c)CAISSE, (%c)CIBLE, (%c)SOKOBAN, (%c)VIDE\n", MUR, CAISSE, CIBLE,
                                                                        Sokoban, VIDE);
        printf("  (%c)ide, (%c)quitter, (%c)aut, (%c)as, (%c)auche, (%c)roite\n", AIDE, QUITTER,
                                                                HAUT, BAS, GAUCHE, DROITE);
}
```

pas() - fait(retourne) un pas en incrémentant ou décrémentant les coordonnées de la position de départ affecté à la variable **position_actuelle** de type **Postion**, dans la direction indiquée par la commande **cmd** récupéré dans le paramètre de la fonction.

```
Position pas(Position pos_depart, Commande cmd){
        Position position_actuelle;
        position_actuelle=pos_depart;

        switch (cmd){
                case HAUT:      position_actuelle.x--; break;
                case BAS:       position_actuelle.x++; break;
                case GAUCHE:    position_actuelle.y--; break;
                case DROITE:    position_actuelle.y++; break;
                default:break;
        }
        return position_actuelle;
}
```

possible() - vérifie que les coordonnées de la position “future” passé en paramètre ne correspondant pas au **MUR** et **CAISSE** dans quel cas il renvoie la valeur **true**, sinon **false**.

```
Boolean possible(Grille g, Position pos){
        if((g.grille[pos.x][pos.y]==MUR)||(g.grille[pos.x][pos.y]==CAISSE))
                return false;
        return true;
}
```

deplace() - prend en paramètre 4 valeurs: la grille de jeu, les positions **a** et **b** correspondant au position actuelle et future, et la grille caché. Dans la première ligne on affecte le contenue des coordonnées de la position actuelle ou future position. Puis, le contenue de la position actuelle est remplacé par le contenue de la coordonnée correspondant dans la grille caché.
(NB: La grille cache ne contient que les symboles **VIDE** et **MUR**. Cf.: **nouveauJeu()**)

```
void deplace(Grille jeu, Position pos_a,Position pos_b, Grille g){
        jeu.grille[pos_b.x][pos_b.y]=jeu.grille[pos_a.x][pos_a.y];
        jeu.grille[pos_a.x][pos_a.y]=g.grille[pos_a.x][pos_a.y];
}
```

Ces trois fonctions sont sollicité par la fonction **joue()** qui calcule la future position ainsi que sa validité. Elle prend en paramètre les grilles de jeu et caché, ainsi que l'instruction et la position de départ (position actuelle).
On déclare la variable **futur_pos** et on fait un pas par appelle de la fonction **pas()** qui n'a aucun contrôle sur la validité de pas vers cette direction. Pour cela on fait appele a la fonction **possible()** qui nous le vérifie et on vérifie la valeur de retour. S'il est correct, on déplace les valeurs des coordonnées correspondantes. Sinon, si c'est à cause d'un mur affiche le message approprié. Ou si c'est une caisse, déplace la caisse d'un cran et met le Sokoban à sa place.

```
Position joue(Grille jeu, Commande dir, Position depart, Grille gCache){
        Position futur_pos;
        futur_pos=pas(depart, dir);

        if(possible(jeu, futur_pos)==true)
                deplace(jeu, depart, futur_pos, gCache);
        else{
                if(jeu.grille[futur_pos.x][futur_pos.y]==MUR)
                        printf("Impossible à cause d'un Mur devant\n");
                if(jeu.grille[futur_pos.x][futur_pos.y]==CAISSE){
                        if(possible(jeu, pas(futur_pos, dir))==true){
                                deplace(jeu, futur_pos, pas(futur_pos, dir), gCache);
                                deplace(jeu, depart, futur_pos, gCache);
                        }
                }
        }
        return futur_pos;
}
```

La fonction **gagne()** détecte que le joueur a gagné en parcourant en boucle la grille de jeu par le fait qu'il n'a plus de cible dans la grille et que Sokoban n'en occupe pas une dans la grille.

```
Boolean gagne(Grille jeu, Position soko, Grille init){
        int i, j, nbCibles=0;
        for(i=0; i<jeu.h; i++){
                for(j=0; j<jeu.L; j++){
                        if(jeu.grille[i][j]==CIBLE)
                                nbCibles++;
                }
        }
        if((nbCibles==0)&&(init.grille[soko.x][soko.y]!=CIBLE))
                return true;
        return false;
}
```

Cette fonction gère les erreur en affichant le message correspondant au code passé en paramètre.

```
void error(CodeErreur code){
   switch(code){
      case ERR_OPEN :   fprintf(stderr,"Impossible d'ouvrir le fichier!\n"); break;
      case NO_SOKOBAN : fprintf(stderr,"Pas de Sokoban!\n"); break;
      case G_INCORECT : fprintf(stderr,"Grille initiale incorrecte !\n"); break;
      case UNE_CAISSE : fprintf(stderr,"Au moins une Caisse doit etre presente\n"); break;
      case DIF_CAICIB : fprintf(stderr,"Nombre de Caisses et des Cibles n'est pas egal\n"); break;
      case ALLOC_TAB : fprintf(stderr,"Allocation de la mémoire pour la boîte de 12 ligne a
                                                                        échoué!\n"); break;
      case ALLOC_TABn2 : fprintf(stderr,"Allocation de la mémoire pour 25 cases d'une des ligne a
                                                                         échoué\n"); break;
      default: fprintf(stderr,"Code erreur inconnu!\n");
    }
    exit(code);
}
```

La fonction **afficherGrille()** parcours la grille et affiche caractère par caractère son contenue.

```
void afficheGrille(Grille g){
        int i, j;
        for(i=0; i<g.h; i++){
                for(j=0; j<g.L; j++)
                        printf("%c", g.grille[i][j]);
                printf("\n");
        }
}
```

L'initialisation de la grille se fait grâce à la fonction **initGrille** qui prend en paramètre le nom du fichier. Dedans, après avoir déclaré les variables, on ouvre le fichier en mode "lecture seule" et tout de suite on vérifie son contenue. S'il ne pointe nul part, ce qu'une erreur d'ouverture se produise, dans ce cas l'appelle à fonction **error()** avec le code **11** indique le message approprié.
Si l'ouverture a abouti, la fonction **fgets()** lit le flux de **fichier** de la taille de **TAILLE_MAX** et stocke le dans le **buffer**. Ce **buffer** est utilisé car la fonction **sscanf()** ne travaille pas avec les fichiers directement. Avec cette fonction on lit dans le **buffer** deux entiers séparé par un espace et on les met dans les variables **hauteur** et **largeur** en passant par leur adresses.
La variable **g** de type **Grille** contiendra la grille crée à partir des dimensions récupéré dans le fichier. Puis on remplit la grille **g** de taille **g.L+1** (NB: +1 pour le caractère '\0') à partir de flux du **fichier**.

```
Grille initGrille(char* nomFichier){
        FILE* fichier = NULL;
        char buffer[TAILLE_MAX] = ""; /* Buffer-chaine vide de taille TAILLE_MAX */
        Grille g;
        int i, hauteur, largeur;
        fichier = fopen(nomFichier, "r");
        if(fichier == NULL)
                error(10); /* ERR_OPEN */
        fgets(buffer, TAILLE_MAX, fichier); /* printf("%s", buffer); */
        sscanf(buffer, "%d %d", &hauteur, &largeur);
        g = creerGrille(hauteur, largeur);
        for(i=0; i<g.h; i++){
                fgets(g.grille[i], g.L+1, fichier);
                fgetc(fichier);
        }
        fclose(fichier);
        return g;
}
```

Cette fonction crée une grille suivant les dimensions passée en paramètre.
On crée une grille **g** et affectant les bonnes dimensions à ces champs en parcourant le tableau et en allouant l'espace nécessaire de taille de **Ligne** (NB: char* Ligne)fois la hauteur de la grille. Si elle échoue on appelle la fonction **error(15)**. Puis on alloue on parcourant chaque case de la hauteur et allouant l'espace de taille **char*g.L+1**. Dans le cas d'une anomalie, on de libérer l'espace mémoire allouée. Enfin on renvoie cette grille.

```
Grille creerGrille(int hauteur, int largeur){
        Grille g;
        int i, j;
        g.h = hauteur;
        g.L = largeur;


        g.grille = malloc(sizeof(Ligne) * g.h);
        if(g.grille == NULL)
                error(15); /* ALLOC_TAB */

        for(i=0; i<g.h; i++){
                g.grille[i] = malloc(sizeof(char) * (g.L+1));
                if(g.grille[i] == NULL){
                        for(j=0; j<g.L+1; j++) /* j < g.L+1     j < i*/
                                free(g.grille[j]);
                        free(g.grille);
                        error(16); /* ALLOC_TABn2 */
                }
        }

        return g;
}
```

La fonction **creerGrille()** permet de copier dans une nouvelle grille la grille passée en paramètre et retourne la grille copié.

```
Grille copieGrille(Grille init){
        Grille g;
        int i;
        g = creerGrille(init.h, init.L);
        for(i=0; i<g.h; i++){
                strcpy(g.grille[i], init.grille[i]);
        }

        return g;
}
```

sokoban.h

Le fichier Sokoban**.h** est un fichier qui contient la déclaration des prototypes des fonctions décrites ci-dessous qui sont nécessaires au compilateur pour exécuter le processus de compilation. Dedans, on déclare aussi les nouveau types, les structures et les variables globales.

Tout ce qui se trouve entre **#ifndef** et **#endif** permet la protection contre les inclusions multiples de la partie du code se trouvant entre ce qui peut causé si ce n'est pas problèmes, alors l'alourdissement du programme. Quand le compilateur lit le fichier il vérifie l'existence de la variable **__SOKO__**, si elle n'existe pas, on la définie une seule fois et la prochaine parcours, on n'aura pas inclure le contenue.

```
#ifndef __SOKO__
#define __SOKO__

[...]

#endif
```

Alliance **ajaxVitre** avec commande UNIX permet d'effacer l'écran du terminal. Ici dans le cas sous Unix on appelle la fonction du système **clear**

```
#ifndef __UNIX__
       #define ajaxVitre system("clear");
#endif
```

Les bibliothèques externes permettent l'utilisation d'un large nombre des fonction définie dans C.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

Les variables globales sont définie car ils sont utilisé par un nombre des fonctions. Ça évite d'encombrer beaucoup de places pour les mêmes variables. Dans le cas d'évolution du programme, on n'aura que de modifier la valeur associé la variable.

```
#define MAXCH 40
#define TAILLE_MAX 10
```

La déclaration des nouveaux énumération qui ont un fonctionnement semblable au structures. Elle contient une liste des variables possibles avec leurs valeurs approprié. En définissant une valeur au premier variables les autres increment cette valeur de 1. Par exemple, **false** a pour valeur **0**, la variables suivante prend une valeur +1, donc 1. Pour **CodeErreur**, **ERR_OPEN=10**, **NO_SOKOBAN=11**, **G_INCORECT=12** ...

```
typedef enum {false=0, true}Boolean;
typedef enum {MUR='#', CAISSE='O', CIBLE='o', Sokoban='S', VIDE='.'}Symbole;
typedef enum {HAUT='h', BAS='b', GAUCHE='g', DROITE='d', AIDE='a', QUITTER='q'}Commande;
typedef enum {ERR_OPEN=10, NO_SOKOBAN, G_INCORECT,UNE_CAISSE, DIF_CAICIB, ALLOC_TAB,
                                                                    ALLOC_TABn2}CodeErreur;
```

Le type **Grille** est une structure qui contient deux entiers (hauteur et largeur) et un pointeur de Ligne. Cette structure permet indépendance entre les différentes grilles.

```
typedef struct _grille{
        int h;  /* Hauteur */
        int L;  /* Largeur */
        Ligne* grille;
} Grille;
```

Structure **Position** permet de stocker la ligne et la colonne où se trouve le personnage Sokoban.

```c
typedef struct{
    int x, y;
} Position;
```

Ensuite, on retrouve la définition des prototypes avec les types de retour précis ainsi que le paramètres attendue par ces fonctions.

```c
int compte(Grille g, char c);
void aide();
void freeGrille(Grille g);
void error(CodeErreur code);
void afficheGrille(Grille g);
void nouveauJeu(Grille init, Grille g);
void deplace(Grille jeu, Position pos_a, Position pos_b, Grille init);
Position getSokoban(Grille g);
Position pas(Position pos_depart, Commande instruction);
Position joue(Grille jeu, Commande dir, Position depart, Grille init);
Boolean verif(Grille g);
Boolean possible(Grille g, Position pos);
Boolean gagne(Grille jeu, Position soko, Grille init);
Grille initGrille(char* nomFichier);
Grille creerGrille(int hauteur, int largeur);
Grille copieGrille(Grille g);
```