Projet `gemmified`

CSH : Initiation au C et au shell

Première année

L'objectif est de réimplémenter en C le petit jeu de puzzle disponible à l'adresse suivante : `http://www.popcap.com/gamepopup.php?theGame=diamondmine`. Il est à faire en binôme.

1 Informations générales

Évaluation de ce projet

- **10 points** seront attribués de manière semi-automatique par évaluation des fonctionnalités de votre programme. Après compilation, votre programme sera évalué au travers de plusieurs tests. Nous utiliserons pour cela le programme `testsuite`, que les 2A ont implémenté dans leur projet de système en début d'année.
  Vous perdrez des points si votre programme ne se comporte pas comme attendu. Pour le bon fonctionnement de ce processus (et l'attribution des points correspondants), la commande 'd' de votre programme doit se comporter exactement comme l'implémentation de référence (voir plus loin). La note *maximale* d'un projet **ne compilant pas en l'état** sur neptune est 10.
- **5 points** seront attribués en fonction de la qualité (subjective) du source de votre programme.
- **5 points** récompenseront la qualité du mini-rapport que vous joindrez dans votre archive. Vous y détaillerez les difficultés auxquelles vous avez été confronté, et comment vous les avez résolues. Vous indiquerez également le nombre d'heures passées par chaque membre du binôme sur les différentes étapes de ce projet (conception, codage, tests, rédaction du rapport).
- Des points de bonifications seront attribués pour chacune des extensions implémentées.

Comment rendre votre projet Vous devez rendre un mini-rapport de projet (7 pages maximum, format pdf), votre source ainsi qu'un makefile permettant de compiler votre projet (sous le nom `gemmes`). Si vous écrivez de nouvelles suites de test, rendez-les également avec votre projet (avec l'extension `.test`) : une bonification est prévue pour les projets dotés d'un ensemble de tests bien fourni.

Pour rendre votre projet, vous devez placer les fichiers nécessaires dans un répertoire sur neptune, vous placer dedans, et invoquer la commande `/home/EqPedag/quinson/bin/rendre_projet CSH` (vous pouvez invoquer cette commande autant de fois que vous le souhaitez)

Avant le lundi 16 avril à 23h59

(la commande ne fonctionnera plus après)

Rappel La tricherie sera **sévèrement punie**. Voir `http://www.loria.fr/~quinson/teaching.html`

Informations complémentaires Vous trouverez des informations complémentaires sur le projet, une implémentation de référence du jeu et quelques exemples de tests (ainsi que le binaire `testsuite` utilisé pour lancer les tests) à l'adresse suivante : `http://www.loria.fr/~quinson/teach-CSH.html`.

2 Présentation du projet `gemmified`

L'objectif est d'implémenter un petit jeu de puzzle. Le plateau est découpé en 8 x 8 cases, chacune contenant une gemme d'une couleur particulière. Le but du jeu est de faire disparaître un maximum de gemmes du plateau. Pour cela, on échange les positions de deux gemmes adjacentes. Si cela aligne trois gemmes (ou plus), elles disparaissent du plateau et celles placées plus haut glissent vers le bas. De nouvelles gemmes sont placées en haut des colonnes pour remplir les trous.

3 Réalisation

3.1 Préliminaires : séquences aléatoires

Afin de permettre l'évaluation automatique de votre projet, nous allons utiliser un mécanisme spécifique pour les tirages aléatoires. Le principe est d'avoir une longue chaîne de caractères, et chaque fois que l'on

«tire une lettre au hasard», on prend la suivante dans la chaîne. On peut arguer que ce n'est pas très «aléatoire», mais il n'y a pas de hasard complet sur ordinateur. Voici la structure associée :

```
typedef struct s_randseq {
  short len;  /* Longueur de la séquence */
  short pos;  /* Position courante dans la séquence */
  char *data; /* Contenu de la séquence */
} *randseq_t;
```

La fonction principale associée est `char randseq_next(randseq_t rs)` Elle retourne le contenu actuel de la séquence et avance le pointeur interne. Si `rs` est `{5,0,"ABCDE"}`, le premier appel renvera 'A', le second 'B', ... le sixieme 'A'. `rs` vaut alors `{5,2,"ABCDE"}`.

▷ **Question 1. Implémentez l'API suivante :**

```
randseq_t randseq_new(int len);
randseq_t randseq_new_from_str(char *seq);
char randseq_next(randseq_t rs);
```

randseq_new_from_str crée une séquence aléatoire en utilisant la chaine de caractères passée en paramètre.

randseq_new crée une séquence aléatoire de la longueur spécifiée, en la remplissant de caractères aléatoires (ie, obtenus avec la fonction rand(3)). Pensez à utiliser `srand(time())` pour initialiser la fonction `rand`.

▷ **Question 2. Testez votre API.**

Pour cela, écrivez un fichier randseq_test.c contenant une fonction main() qui crée une (ou plusieurs) séquence(s) aléatoire(s) et effectue plusieurs tests dessus. Il convient d'ordonner les tests de façon logique : il faut tester les fonctionnalités de base avant les fonctionnalités avancées, car si une fonctionnalité de base est cassée, il est probable qu'elle casse les tests avancés, et fausse ainsi leurs résultats. Vous présenterez rapidement ces tests dans votre rapport. Indiquez la partie du code qu'ils vérifient, et pourquoi vous avez décidé d'ajouter chacun d'entre eux.

3.2 Squelette du jeu : board_t et API associée

La prochaine étape est de créer un type représentant le plateau de jeu. Il est défini par ses dimensions et le contenu de chacune de ses cases. Il faut également ajouter un champ pour noter le score du joueur, et un autre pour la séquence aléatoire utilisée. La structure board_t doit donc comporter au moins 5 champs. En ce qui concerne le contenu, il est plus simple d'utiliser le type C «`char *`», et d'utiliser la macro suivante pour accéder à la case (`x`, `y`) du plateau `b` :

```
#define board_pos(b, x, y) ((b)->data[(y) + (x)*(b)->ysize])
enum dir_t { up=0,down=1,left=2,right=3 };
const int dx[4],dy[4];
```

▷ **Question 3.** Implémentez l'API suivante :

```
board_t board_new(int nlines,int nrows, randseq_t rs);
void board_print(board_t b);
```

▷ **Question 4.** Testez votre implémentation. La figure 1 présente un programme, et la sortie attendue. Ces deux fichiers sont disponibles sur la page du projet.

▷ **Question 5.** Écrivez une macro `board_neighbor(b, x,y, dir, dist)` comparable à `board_new`, mais retournant le contenu d'une case voisine de (x,y) dans la *dir*ection donnée à la *dist*ance donnée.

3.3 Analyse syntaxique des coups

Nous allons maintenant analyser les coups de l'utilisateur. Chacuns d'entre eux est composé d'une lettre indiquant la colonne suivie (sans espace) d'un chiffre indiquant la ligne, et d'une lettre indiquant la direction du changement. **u**(p) pour changer la case indiquée avec sa voisine au dessus, **d**(own) pour sa voisine en dessous, **l**(eft) pour sa voisine de gauche et **r**(ight) pour sa voisine de droite.

test_board.c

```c
#include "gemmes.h"
int main() {
   randseq_t rs = randseq_new_from_str(
    "FDGECECAADGECCDEAGDBGACFCCBCFBCF"
    "GEDBFDEAEEDBEDEGDFBCBECGDABGFAEB");
   board_t b = board_new(8,8,rs);
   board_print(b);
   return 0;
}
```

test_board.output

```
      a b c d e f g h
Board:   +---------------+  Score: 0
       1|F D G E C E C A |1
       2|A D G E C C D E |2
       3|A G D B G A C F |3
       4|C C B C F B C F |4
       5|G E D B F D E A |5
       6|E E D B E D E G |6
       7|D F B C B E C G |7
       8|D A B G F A E B |8
         +---------------+
          a b c d e f g h

```

FIG. 1 – Test pour **board_t**.

Un coup n'est valide que s'il permet de former une ligne de trois gemmes ou plus. Ainsi, sur la grille de la figure 1, *b4u* (qui inverse le `C` placé en B4 avec le `G` placé au dessus) n'est valide. En revanche, *d7r* crée une ligne verticale de `'B'` en `d`. C'est donc un coup valide, tout comme *b3r* ou *c4r*.

▷ **Question 6.** Écrivez une fonction en charge d'analyser le coup joué par l'utilisateur. Il ne s'agit que d'une analyse syntaxique (une lettre, un chiffre, une lettre). L'analyse sémantique nécessaire pour différencier les coups valides (comme *d7r*) des coups invalides (comme *b4u*) sera menée à la question 10, mais vous pouvez d'ors et déjà détecter *d1u* ou *a7l* comme invalides.

▷ **Question 7.** Mettez en place la boucle principale du programme qui demande le prochain coup à jouer, (ne fait rien d'autre pour l'instant), puis affiche de nouveau le plateau. Testez votre implémentation, et présentez succintement les tests effectués dans le rapport.

3.4 Longueur de segments

Pour déterminer les effets d'un coup sur le plateau, il faut calculer la longueur de segments. La fonction `board_searchline` retourne le nombre de lettres identiques à (x,y) dans la direction indiquée. Ainsi, pour (b,3,l), elle retourne 0, car il n'y a pas de G à gauche de (b,3). Pour (d,5,d), elle retourne 1, car il y a un B sous (d,5).

```c
int board_searchline(board_t b, int x, int y, enum dir_t dir);
```

▷ **Question 8.** Implémentez et testez cette fonction.

▷ **Question 9.** Modifiez `board_new()` pour qu'elle ne génère jamais de plateau avec trois lettres alignées dès le début. La séquence aléatoire ligne 1 doit générer le même plateau que celle ligne 2 (cf. figure 1).

```
FDGECECAADGECCDEAGDBGACFCCCCCCCCCCCCBCFBCFGEDBFDEAEEDBEDEGDFBCBECGDABGFAEB
FDGECECAADGECCDEAGDBGACFCCBCFBCFGEDBFDEAEEDBEDEGDFBCBECGDABGFAEB
```

3.5 Analyse sémantique des coups

▷ **Question 10.** Écrivez une fonction `board_move()` qui applique un coup (échange des deux gemmes concernées) et vérifie la validité sémantique du coup (cela crée une ligne de longueur 3 ou plus). Si le coup n'est pas valide, elle remet le plateau dans son état initial et retourne le code d'erreur 1. Intégrez cette fonction au reste du projet.

3.6 Cœur du jeu : mise en application des coups

Tout est maintenant en place pour écrire la fonction maîtresse du jeu, celle qui calcule l'effet d'un coup. Il faut faire disparaitre les gemmes alignées, mettre à jour le score en conséquence, faire descendre les gemmes placées au dessus d'espaces vacants, et reremplir le haut des colonnes avec des gemmes tirées de la séquence aléatoire.

La première tentation est de chercher à traiter les segments maximaux autour des deux gemmes ayant inversé leurs positions, puis autour de toutes gemmes déplacées lors de la disparition d'alignements. Mais

cette approche est extrêmement difficile à réaliser à cause de segments perpendiculaires qui peuvent apparaître. Ainsi, *g2u* sur le plateau de la figure 1 crèe deux segments : $[e2; g2]$ et $[g2; g4]$.

La bonne approche est de rechercher les segments sur tout le plateau sans optimisation particulière. Après tout, notre jeu n'est pas gourmand en temps processeur. Il faut faire attention cependant à ne compter chaque segment qu'une seule fois (pour ne pas attribuer les points plusieurs fois). Pour cela, on peut s'inspirer de la question 9 : Pour chaque case du plateau, on regarde si elle *commence* un segment vers le bas et/ou un segment vers la droite. La condition «*commence un segment vers le bas*» peut se réécrire en «*ne fait pas partie d'un segment vers le haut ET fait partie d'un segment vers le bas*».

Une fois que l'on a trouvé un segment, il faut un moyen de le «marquer» pour l'effacer ultérieurement. Il est cependant impossible de l'effacer tout de suite (de remplacer chacune des cases le composant par un espace), car sinon, on ne pourra plus trouver les segments perpendiculaires. Il s'agit du même problème que précédement : si après avoir joué *g2u*, on place des espaces dans $[e2; g2]$, on ne pourra plus détecter que $[g2; g4]$ forme également un ségment.

Il faut donc définir un nouvel espace de travail comportant autant de char qu'il n'y en a dans le plateau. Au début de la recherche, on remplit cet espace de 0, puis quand on trouve un segment, on met des 1 pour chaque case le composant.

▷ **Question 11.** Écrivez la fonction `board_update()`, qui cherche les segments à effacer du plateau en utilisant cet algorithme.

▷ **Question 12.** Faites en sorte que cette fonction modifie le score du plateau de façon adéquate. Pour chaque segment trouvé, il faut mettre à jour les points du joueur (le champ `score` de la structure `board_t`). Un segment de longueur l rapporte $(l-1) \times 5$ points. De plus, si un coup crèe plusieurs segments, le gain associé au $i^{\text{ième}}$ segment est multiplié par 2^i. Ainsi, le coup *g2u* rapporte 30 points. $2^0 \times 2 \times 5 = 10$ pour $[e2; g2]$ et $2^1 \times 2 \times 5 = 20$ pour $[g2; g4]$. On parcours le plateau depuis le coin haut gauche d'abord par ligne puis par colonne. Testez votre travail.

▷ **Question 13.** Modifiez `board_update()` pour qu'après avoir trouvé tous les segments à effacer, elle remplace par des espaces dans le plateau les parties à effacer. Cette fonction parcours donc le plateau deux fois. La première fois, elle cherche les segments et les indique dans l'espace de travail (le plateau n'est pas modifié). Lors du second parcours, le plateau est modifié pour appliquer ce qui est indiqué dans l'espace de travail. Testez votre travail.

▷ **Question 14.** Modifiez cette fonction pour qu'elle «*fasse tomber les gemmes placées au dessus d'espaces vacants*». Pour cela, vous parcourrez chaque colonne du plateau. Pour chacune, parcourrez chaque case en partant du bas. S'il s'agit d'un espace, il faut trouver une case au dessus pour remplir le trou. Faites une trosième boucle imbriquée pour trouver cette gemme. S'il n'y a aucune gemme au dessus (toutes les cases au dessus sont vides), utilisez la séquence aléatoire pour tirer la nouvelle gemme à placer.

▷ **Question 15.** Modifiez à nouveau cette fonction pour qu'elle réapplique le même traitement tant que des segments ont été supprimés car les gemmes nouvellement placées peuvent former de nouveaux segments. Testez votre travail.

Voila, vous avez implémenté le cœur du jeu, qui est maintenant jouable.

3.7 Options et commandes

▷ **Question 16.** Ajoutez les nouvelles commandes suivantes (en plus des coups valides) :

- **d :** Affiche l'état courant du plateau. Cela doit être sous la forme d'une chaîne de caractères composées de 'A'...'D' sur une seule ligne, puis le score sur une autre ligne. La syntaxe doit être exactement la même que celle de l'implémentation de référence puisque c'est ce qui sera utilisé pour les tests automatiques.
- **? :** Affiche une petite aide.
- **q :** Termine le jeu.

▷ **Question 17.** Votre programme doit accepter les arguments suivants :

- **-h :** Si cette option est passée, votre programme affiche les options acceptées et se termine immédiatement.
- **-s :** L'argument suivant est la chaîne à utiliser pour initialiser la séquence aléatoire.
- **-q :** Supprime tous les affichages sauf celui de la commande 'd'.
- **-b :** augmente la taille de l'affichage (cf. l'implémentation de référence)
- **-x**, **-y** permet de modifier la taille du plateau au lancement du programme.

- **-c** spécifie le nombre de couleurs à utiliser.

▷ **Question 18.** Ajoutez la commande 'h', qui donne un indice (*hint*) au joueur. Le programme indique une position à partir de laquelle un coup est possible.

▷ **Question 19.** Ajoutez un système détectant automatiquement si aucun coup n'est possible, et termine la partie quand c'est le cas.

4 Aller plus loin : extensions possibles

1. **Affichage ascii art** Réaliser un affichage en ascii art. Dans le mode -b, au lieu de remplir les cases de la lettre, vous utiliserez des petits dessins composés de lettres. Testez l'implémentation de référence avec l'option -f pour un exemple.
2. **Affichage graphique sur la console** Réaliser un mode où l'affichage se fait avec avec la bibliothèque `ncurse`. Vous trouverez un tutoriel pour l'usage de cette bibliothèque à l'adresse suivante : `http://www.apmaths.uwo.ca/~xli/ncurses.html`.
3. **Affichage graphique** Réaliser un affichage graphique à l'aide d'une bibliothèque telle que SDL.
4. **Autre** Vous êtes bien entendu libres d'implémenter toute autre extension ou option à ce jeu.