Suguru (C) (DM3, Informatique, MP2I Carnot 21)

Tristan Gierczak–Galle, MP2I

Ce document constitue le compte-rendu complet du deuxième exercice du Devoir Maison d'informatique à rendre le vendredi 22 mai, portant sur l'implémentation et l'élaboration d'un algorithme de résolution du jeu logique Suguru. L'explication de l'implémentation du jeu suit la trame proposée dans le devoir.

(Voir les commentaires in-code pour plus de détails)

(Un récapitulatif des différentes fonctions/procédures, leur signature et leur rôle est présent dans la section Fonctions/Procédures)

I] Implémentation du jeu

Corps central

Le jeu du Suguru se jouant sur une grille `n*m`, avec `n` et `m` des entiers, il est naturel de représenter la grille de jeu sous la forme d'une matrice d'entiers. On définit alors le type `Grid`, de la manière suivante:

```
struct Grid {
  int n;
  int m;
  Cell **cells;
  int *zone_sizes;
  int zones_n;
};
```

On explique le rôle et l'utilité de chacun des champs de `Grid`:

- `n`, `m`: les dimensions de la grille
- `cells`: la grille en elle-même, faites de cellules de type `Cell` (que l'on explicite ensuite)
- `zones_sizes`: la taille de chaque zone (tableau indexé par des identifiants de zone)
- `zones_n`: nombre total de zones dans la grille instanciée

Le type `Cell`, qui construit entièrement la grille, est défini comme suit:

```
struct Cell {
  int val;
  int zone_id;
};
```

et stocke `val` *la valeur* et `zone_id` *l'identifiant de zone* (découpage dans la grille de Suguru) de la cellule.

Instanciation d'une grille de jeu

Ici, je suis allé un peu plus loin en proposant un *parseur de configurations*.

Dans un fichier dont le format exact est fourni ci-après, l'utilisateur peut utiliser l'exécutable pour créer une **grille initiale**, dont les formes et valeurs contenues sont choisies.

Le fichier doit **impérativement** être formaté de la manière suivante (un exemple valide est fourni dans `config.txt`):

```
n m
a b ...
.
.
.
x y ...
d e ...
.
.
.
t u ...
```

- La première ligne contient les **dimensions** `n` **et** `m` de la grille voulue **séparées par** `' '`.
- Les `n` prochaines lignes sont les **différentes lignes de valeurs** à insérer dans la grille. Chacune de ces lignes contient alors `m` valeurs, encore une fois **séparées par** `' '`.
- Les `n` dernières lignes cartographient la grille selon les **identifiants de zones**, à partir de `0`.

Par exemple, pour une grille `3 * 3`, les `n` lignes suivantes:

```
0 0 1
0 1 1
2 2 2
```

donne le découpage suivant (sans représenter les valeurs):

```
+---+---+---+
|       |   |
+   +---+   +
|   |       |
+---+---+---+
|           |
+---+---+---+
```

Ce fichier est parsé en un type `Config`, dont la définition est la suivante:

```
struct Config {
  int n;
  int m;
  int zones_n;
  int **zone_ids;
  int **vals;
};
```

suivant une structure équivalente à celle de `Grid`, et qui joue le rôle d'interface de construction entre le fichier à parser et l'instance de `Grid` utilisée. On construit une instance de `Grid` à partir d'une instance de `Config` par la procédure `init_grid_cfg`.

La procédure de parsing se déroule **ligne par ligne**, selon le modèle présenté ci-dessus, et se situe dans la procédure `parse_to_cfg`, qui regroupe **l'extraction des dimensions** ainsi que **le parsing** de **la matrice de valeurs** et de **la matrice de zones**.

II] Affichage

L'algorithme d'affichage de la grille se trouve entièrement dans `miniprojet.c/draw_grid`. Le programme affiche systématiquement l'état initial de la grille, avant toute résolution.

III] Vérification de solution

La vérification de solution est implémentée dans `_valid`, qui vérifie la validité de l'ajout d'une Cellule portant la valeur `val` à la ligne `row` et à la colonne `col` dans la grille. On définit alors `valid`, un wrapper autour de `_valid` qui l'appplique à chacune des cellules.

IV] Résolution

On se propose ici d'implémenter un algorithme de résolution par ***retour sur trace***. Le principe est le suivant:

- On parcourt l'entièreté de la grille, et on ne s'attarde que sur les cellules à remplir à l'état actuel de la grille. Pour chaque valeur dans `1..n`, `n` étant la taille de la zone de la cellule en exploration, on génère **une solution partielle**
  - Si cette solution est **valide**, on passe à la *case suivante (vide)*
  - Sinon, on *change de valeur*
- Si on arrive **à la fin de la grille** (au-delà de la `n`-ième ligne), on a trouvé une **solution totale**, et on *retourne `true`*
- Sinon, si aucune de ces valeurs ne pour cet étage du backtracking ne construit une solution partielle valide, *on retourne `false`*

La grille est **modifiée en place** par cette fonction. Si une solution existe, elle est affichée après l'affichage de l'état initial.

V] Appendix

- Pour plus de rigueur, un test de fuites de mémoire a été réalisé à l'aide de Valgrind-3.25.0, dont les logs sont contenus dans `valgrind-out.txt`.
- Pour refaire le test, compiler avec les flags `-Wall` et `-ggdb3` comme suit:

```
gcc miniprojet.c parser.c -Wall -ggdb3
```

puis:

```
valgrind --log-file=[fichier_logs] ./a.out config.txt
```

- Finalement, pour lancer le programme avec une configuration écrite correctement dans `config.txt`

```
./a.out config.txt
```

Fonctions/Procédures

- `parser.c`
  - `Config *parse_to_cfg(char *path)` : fonction principale du parsing de configurations
  - `void parse_vals(FILE *fp, int **vals, int n, int m)` : procédure auxiliaire utilisée par `parse_to_cfg` ; parse la matrice de valeurs textuelle dans `vals`
  - `void parse_zone_ids(FILE *fp, int **ids, int n, int m)` : procédure auxiliaire utilisée par `parse_to_cfg` ; parse la matrice d'identifiants de zones textuelle dans `ids`
  - `void cfg_free(Config *cfg)` : procédure de libération de mémoire allouée pour une instance de `Config`
  - `int split_into_array(char *line, char *sep, char *tokens[], int max_tokens)` : hache `line` en plusieurs tokens/sous-chaînes de caractères (au maximum de `max_tokens`) séparés par `sep` et stockés dans `tokens`
  - `int max(int *arr[], size_t n, size_t m)` : fonction auxiliaire pour le calcul de maximum de matrices d'entiers
- `miniprojet.c`
  - `void init_grid_cfg(Grid *grid, Config *cfg)` : initialise `grid` avec `cfg`, créé lors du parsing en amont
  - `Grid *new_grid(Config *cfg)` : thin wrapper autour de `init_grid_cfg`
  - `void grid_free(Grid *g)` : procédure de libération de mémoire allouée pour une instance de `Grid`
  - `int count_digits(int n)` : fonction utilitaire pour déterminer le nombre de chiffres en base `10` de `n`
  - `void draw_grid(Grid *g)` : afficher la grille `g`
  - `bool _valid(Grid *g, int row, int col, int val)` : détermine si le placement de le cellule portant `val` à la ligne `row` et à la colonne `col` est valide selon les règles du Suguru
  - `bool valid(Grid *g)` : `_valid` : appliquée à toutes les cellules de la grille
  - `bool solve(Grid *g, int i, int j)` : génère une solution au Suguru si elle existe, et retourne `false` sinon