Rapport de Projet de Résolution du jeu de Tentes

Nourry Celian
Numéro Étudiant : 22002895

Date de rendu : le 19/12/2024

Introduction

Description du jeu des tentes

Le jeu des tentes est un casse-tête logique qui se joue sur une grille rectangulaire ou carrée. Chaque case de la grille peut être occupée par un arbre, une tente ou rester vide. Le but du jeu est de placer les tentes sur la grille en respectant un ensemble de règles précises.

Règles du jeu

1. **Association des tentes et des arbres :**
   - Chaque tente doit être associée à un arbre adjacent (horizontalement ou verticalement, mais pas en diagonale).
   - Un arbre ne peut être associé qu'à une seule tente.
2. **Disposition des tentes :** Les tentes ne doivent pas être placées côte à côte, ni horizontalement, ni verticalement, ni en diagonale.
3. **Contraintes par ligne et par colonne :** Le nombre de tentes dans chaque ligne et chaque colonne est spécifié au début du jeu. Ces contraintes doivent être strictement respectées.
4. **Cases interdites :** Certaines cases peuvent être marquées comme inaccessibles (herbe), ce qui signifie qu'il est impossible d'y placer une tente.

Objectif du jeu

Le joueur doit placer toutes les tentes sur la grille de manière à satisfaire toutes les contraintes :

- Chaque tente doit être associée à un arbre.
- Les tentes ne doivent pas se toucher.
- Le nombre de tentes par ligne et par colonne doit correspondre aux indications données.

Objectif du projet

L'objectif principal de ce projet est de développer un programme capable de résoudre automatiquement les grilles du jeu des tentes. Pour ce faire, plusieurs objectifs spécifiques ont été définis :

1. **Résoudre automatiquement des grilles de jeu :** Le programme doit être capable de prendre une grille de jeu initiale, qui peut inclure des arbres, des cases vides et des espaces interdits (herbe, et d'en déduire une solution valide, c'est-à-dire une disposition correcte des tentes. L'objectif est d'automatiser le processus de résolution en utilisant des méthodes logiques et des techniques de recherche (notamment en exhaustive)

2. **Garantir la validité des solutions proposées :** Chaque solution générée par le programme doit respecter strictement les règles du jeu. Cela signifie que le programme doit vérifier plusieurs critères de validité :

   - Chaque tente doit être associée à un arbre et ne doit pas être placée côte à côte avec une autre tente.
   - Les contraintes sur le nombre de tentes par ligne et par colonne doivent être respectées.
   - Les espaces interdits (herbe) doivent être correctement évités.

   En outre, des tests doivent être effectués pour s'assurer que le programme ne produit que des solutions valides, et qu'aucune condition du jeu n'est enfreinte.

Conception

Structure de données du plateau

Le plateau se compose de 5 éléments :

- Un tableau 2D contenant des entiers pour représenter la grille du plateau
- Un tableau contenant le nombre de tentes que peut accueillir chaque ligne actuellement
- Un tableau contenant le nombre de tentes que peut accueillir chaque colonne actuellement
- Un tableau contenant le nombre de tentes que peut accueillir chaque ligne originalement
- Un tableau contenant le nombre de tentes que peut accueillir chaque colonne originalement

```c
typedef struct {
    int **grille;
    int *tentesParLigne;
    int *tentesParColonne;

    int *tentesParLigneOriginal;
    int *tentesParColonneOriginal;
} Plateau;
```

Les variables tentesParLigne et tentesParColonne sont utilisées par le programme durant la résolution pour traquer le nombre de tentes qu'il peut encore mettre sur une ligne ou une colonne. Tandis que pour tentesParLigneOriginal et tentesParColonneOriginal, elles sont utilisées pour l'affichage des valeurs originales.

Variables mises dans la grille du plateau Les variables mises dans la grille du plateau sont des entiers signés sous cette forme :

- "-1" pour de l'herbe
- "0" pour du vide
- "1" pour une tente
- "2" pour un arbre

La taille de chaque LIGNE et COLONNE du plateau est spécifiée au préalable dans "LIGNES" et "COLONNES". Ces valeurs doivent être modifiées si l'on veut utiliser d'autres dimensions pour notre plateau.

```c
#define HERBE -1
#define VIDE 0
#define TENTE 1
#define ARBRE 2

#define LIGNES 8
#define COLONNES 8
```

Création du plateau

La fonction de création de plateau ne prend rien en paramètre et retourne une structure de données de Plateau. Elle va dans un premier temps allouer de la mémoire pour chaque LIGNE de celui-ci, puis sur chacune de ces LIGNES, elle va allouer de la mémoire pour chaque COLONNE. Elle va faire de même pour les tableaux du nombre de tentes restantes à placer sur chaque ligne et chaque colonne. Enfin, elle va initialiser toutes les valeurs de la grille à 0 pour représenter du VIDE.

```c
Plateau *creation_plateau(void) {
    Plateau *p = malloc(sizeof(Plateau));
    if (p == NULL) return NULL;

    // Allocation pour la grille
    p -> grille = malloc(LIGNES * sizeof(int *));
    if (p -> grille == NULL) {
        free(p);
        return NULL;
    }
```

```
    for (int i = 0; i < LIGNES; i++) {
        p -> grille[i] = malloc(COLONNES * sizeof(int));
        if (p -> grille[i] == NULL) {
            for (int j = 0; j < i; j++) free(p -> grille[j]);
            free(p -> grille);
            free(p);
            return NULL;
        }
    }

    // Allocation des tentes requise sur chaque cases
    p -> tentesParLigne = malloc(LIGNES * sizeof(int));
    p -> tentesParColonne = malloc(COLONNES * sizeof(int));
    if (p -> tentesParLigne == NULL || p -> tentesParColonne ==
    NULL) {
        free(p -> grille);
        free(p);
        return NULL;
    }

    p -> tentesParLigneOriginal = malloc(LIGNES * sizeof(int));
    p -> tentesParColonneOriginal = malloc(COLONNES * sizeof(int));
    if (p -> tentesParLigneOriginal == NULL || p ->
    tentesParColonneOriginal == NULL) {
        free(p -> grille);
        free(p -> tentesParLigne);
        free(p -> tentesParColonne);
        free(p);
        return NULL;
    }

    // Initialisation du Plateau avec du VIDE sur chaque case
    for (int i = 0; i < LIGNES; i++) for (int j = 0; j < COLONNES;
    j++) p -> grille[i][j] = VIDE;

    return p;
}
```

Libération de mémoire du plateau

La fonction de libération d'un plateau prend un plateau en paramètre et ne renvoie rien. Elle libère chacun des éléments mis dans celui-ci.

```
void free_plateau(Plateau *p) {
    if (p == NULL) return;

    for (int i = 0; i < LIGNES; i++) free(p -> grille[i]);
    free(p -> grille);
    free(p -> tentesParLigne);
    free(p -> tentesParColonne);
    free(p -> tentesParLigneOriginal);
    free(p -> tentesParColonneOriginal);

    free(p);
}
```

Copie d'un plateau existant

La fonction de copie d'un plateau prend en paramètre un plateau et renvoie un plateau. Elle fait l'exacte copie du plateau mis en paramètre pour le renvoyer.

Pour faire cela, elle initialise un plateau avec la fonction de création de plateau, puis chaque élément du plateau mis en paramètre est copié dans le nouveau plateau renvoyé. Cette fonction est utile pour faire des sauvegardes d'états précédents de plateau durant l'algorithme de backtracking.

```
Plateau *copie_plateau(const Plateau *p) {
    if (p == NULL) return NULL;

    Plateau *copie = creation_plateau();
    if (copie == NULL) return NULL;

    for (int i = 0; i < LIGNES; i++) {
        for (int j = 0; j < COLONNES; j++) {
            copie -> grille[i][j] = p -> grille[i][j];
        }
    }

    for (int i = 0; i < LIGNES; i++) {
        copie -> tentesParLigne[i] = p -> tentesParLigne[i];
        copie -> tentesParLigneOriginal[i] = p ->
    tentesParLigneOriginal[i];
    }

    for (int j = 0; j < COLONNES; j++) {
        copie -> tentesParColonne[j] = p -> tentesParColonne[j];
        copie -> tentesParColonneOriginal[j] = p ->
    tentesParColonneOriginal[j];
    }

    return copie;
}
```

Restauration d'un plateau

La fonction de restauration d'un plateau prend en paramètre deux plateaux et renvoie un plateau. Elle copie les données du plateau source dans le plateau destination. Cela est utilisé pour restaurer l'état précédent d'un plateau si l'algorithme de backtracking mène à une mauvaise solution.

```
void restaurer_plateau(Plateau *dest, const Plateau *src) {
    for (int i = 0; i < LIGNES; i++) {
        for (int j = 0; j < COLONNES; j++) {
            dest -> grille[i][j] = src -> grille[i][j];
        }
    }

    for (int i = 0; i < LIGNES; i++) {
        dest -> tentesParLigne[i] = src->tentesParLigne[i];
        dest -> tentesParLigneOriginal[i] = src ->
    tentesParLigneOriginal[i];
    }

    for (int j = 0; j < COLONNES; j++) {
        dest -> tentesParColonne[j] = src -> tentesParColonne[j];
        dest -> tentesParColonneOriginal[j] = src ->
    tentesParColonneOriginal[j];
    }
}
```

Vérification de placement de tente

La fonction de vérification de placement de tente prend en paramètre un plateau et deux entiers signés. Ces deux entiers sont une représentation de coordonnées dans la grille du plateau. Elle retourne un entier signé. Cette fonction est indispensable pour vérifier si le placement d'une TENTE placée par un algorithme de résolution automatique est cohérent. Une TENTE peut être placée dans les coordonnées (i, j) de la grille du plateau quand la case :

- est VIDE
- a encore de la place pour accueillir une nouvelle TENTE
- n'a pas de case adjacente étant une TENTE (horizontale, verticale et diagonale)
- a une case adjacente étant un ARBRE (horizontale, verticale)

La fonction retourne 1 si les conditions énumérées ci-dessus sont respectées, sinon elle renvoie 0.

```
1  int peut_placer_tente(Plateau *p, int i, int j) {
2      // V rifier que la case donn e est VIDE
3      if (p -> grille[i][j] != VIDE) return 0;
4
5      // V rifier qu'il reste des tentes    placer sur la colonne
6      if (p -> tentesParColonne[i] <= 0 || p -> tentesParLigne[j] <=
       0) return 0;
7
8      // V rification de pr sence d'une tente adjacente    la case
       (i, j)
9      if (i > 0 && j > 0 && p -> grille[i - 1][j - 1] == TENTE)
       return 0;
10     if (i > 0 && p -> grille[i - 1][j] == TENTE) return 0;
11     if (i > 0 && j < COLONNES - 1 && p -> grille[i - 1][j + 1] ==
       TENTE) return 0;
12     if (j > 0 && p -> grille[i][j - 1] == TENTE) return 0;
13     if (j < COLONNES - 1 && p -> grille[i][j + 1] == TENTE) return
       0;
14     if (i < LIGNES - 1 && j > 0 && p->grille[i + 1][j - 1] == TENTE
       ) return 0;
15     if (i < LIGNES - 1 && p -> grille[i + 1][j] == TENTE) return 0;
16     if (i < LIGNES - 1 && j < COLONNES - 1 && p -> grille[i + 1][j
       + 1] == TENTE) return 0;
17
18     // V rification de la pr sence d'un ARBRE autour de la case (
       i, j). S'il n'y en a pas, on ne place pas la tente
19     if ((i == 0 || p -> grille[i - 1][j] != ARBRE) &&
20         (i == LIGNES - 1 || p -> grille[i + 1][j] != ARBRE) &&
21         (j == 0 || p -> grille[i][j - 1] != ARBRE) &&
22         (j == COLONNES - 1 || p -> grille[i][j + 1] != ARBRE))
23     return 0;
24
25     return 1;
26 }
```

Placement de tente La fonction de placement de tente prend également un plateau et deux entiers signés qui représentent les coordonnées dans une grille du plateau. Elle ne renvoie rien et ne fait pas de vérification poussée car elles sont déjà faites dans la fonction de vérification de placement de TENTE. Elle vérifie seulement si les deux entiers signés mis en paramètre ne sont pas en dehors des limites du plateau. Elle décrémente le nombre de TENTES requises sur la LIGNE et la COLONNE, puisqu'on vient d'en placer une. Elle met aussi de l'HERBE autour de la TENTE placée puisque aucune autre TENTE n'est supposée toucher la TENTE placée.

```
void placer_tente(Plateau *p, int i, int j) {
    // Place une TENTE dans la case (i, j)
    p -> grille[i][j] = TENTE;

    // On d cr mente le nombre de tente requises sur la ligne/
    colonne
    p -> tentesParColonne[i]--;
    p -> tentesParLigne[j]--;

    // Placement d'HERBE dans les cases adjacentes    la case (i, j
    )
    if (i > 0 && j > 0 && p -> grille[i - 1][j - 1] == VIDE) p ->
    grille[i - 1][j - 1] = HERBE;
    if (i > 0 && p -> grille[i - 1][j] == VIDE) p -> grille[i-1][j]
     = HERBE;
    if (i > 0 && j < COLONNES - 1 && p -> grille[i - 1][j + 1] ==
    VIDE) p -> grille[i - 1][j + 1] = HERBE;
    if (j > 0 && p -> grille[i][j - 1] == VIDE) p -> grille[i][j -
    1] = HERBE;
    if (j < COLONNES - 1 && p -> grille[i][j + 1] == VIDE) p ->
    grille[i][j + 1] = HERBE;
    if (i < LIGNES - 1 && j > 0 && p -> grille[i + 1][j - 1] ==
    VIDE) p -> grille[i + 1][j - 1] = HERBE;
    if (i < LIGNES - 1 && p -> grille[i + 1][j] == VIDE) p ->
    grille[i + 1][j] = HERBE;
    if (i < LIGNES - 1 && j < COLONNES - 1 && p -> grille[i + 1][j
    + 1] == VIDE) p -> grille[i + 1][j + 1] = HERBE;

    return;
}
```

Vérification de la validité d'un plateau

La fonction de vérification de la validité d'un plateau prend en paramètre et renvoie un entier signé. Elle vérifie si l'aménagement du plateau est la solution du puzzle. Pour le faire simplement, elle vérifie si toutes les lignes et les colonnes ont exactement le bon nombre de TENTES placées sur celles-ci.

```
int verif_solution(Plateau *p) {
    for (int i = 0; i < LIGNES; i++) if (p -> tentesParLigne[i] !=
    0) return 0;
    for (int j = 0; j < COLONNES; j++) if (p -> tentesParColonne[j]
     != 0) return 0;
    return 1;
}
```

Résolution logique

La fonction de résolution logique prend en paramètre un plateau et renvoie un entier signé. Elle fonctionne sur une boucle qui continue tant qu'une modification du plateau est faite par cette même fonction. Les modifications du plateau sont faites quand :

- le nombre de TENTES requises sur une ligne ou une colonne est égal à 0 (on met de l'HERBE tout le long de celle-ci)
- le nombre de case VIDE sur le long d'une ligne ou une colonne est égal au nombre de TENTES restantes à placer sur celle-ci (on place donc une TENTE sur toutes les cases VIDES le long de la LIGNE/COLONNE)
- un seul espace VIDE est disponible autour d'un ARBRE (on place donc une TENTE sur celui-ci)

La fonction retourne 1 si la solution est valide, sinon 0. Elle sert à aider la fonction de backtracking pour rapidement supprimer des arbres de parcours en utilisant les contraintes du jeu.

```
int resolution_logique(Plateau *p){
    int modifications = 1;
    while (modifications){
        modifications = 0;
        for (int i = 0; i < LIGNES; i++) {
            int countVide = 0;

            // Si aucune tente n'est requise sur une ligne, on met de l'HERBE tout le long de celle-ci
            if (p -> tentesParLigne[i] == 0) for (int j = 0; j < COLONNES; j++) if (p -> grille[j][i] == VIDE) p -> grille[j][i] = HERBE;

            // Compter le nombre de VIDE sur la ligne
            for (int j = 0; j < COLONNES; j++) {
                if (p -> grille[j][i] == VIDE) {
                    countVide++;
                }
            }

            // Si le nombre de VIDE est egal au nombre de tente placer sur la LIGNE
            if (countVide == p -> tentesParLigne[i]) {
                for (int j = 0; j < COLONNES; j++) {
                    if (peut_placer_tente(p, j, i)) {
                        placer_tente(p, j, i); // On place les tentes sur toutes les cases vides
                        modifications = 1;
                    }
                }
            }
        }

        // La meme chose pour les colonnes
        for (int i = 0; i < COLONNES; i++) {
            int countVide = 0;
```

```
32
33          if (p -> tentesParColonne[i] == 0) for (int j = 0; j <
    LIGNES; j++) if (p -> grille[i][j] == VIDE) p -> grille[i][j] =
     HERBE;
34
35          for (int j = 0; j < LIGNES; j++) {
36              if (p -> grille[i][j] == VIDE) {
37                  countVide++;
38              }
39          }
40
41          if (countVide == p -> tentesParColonne[i]) {
42              for (int j = 0; j < COLONNES; j++) {
43                  if (peut_placer_tente(p, i, j)) {
44                      placer_tente(p, i, j);
45                      modifications = 1;
46                  }
47              }
48          }
49      }
50
51      // S'il y a seulement un espace vide autour d'un ARBRE, on
    y place une TENTE
52      for (int i = 0; i < LIGNES; i++){
53          for (int j = 0; j < COLONNES; j++){
54              if (p -> grille[i][j] == ARBRE){
55                  int espaces_vides = 0;
56                  int direction_i = -1, direction_j = -1;
57
58                  if (i > 0 && p -> grille[i - 1][j] == VIDE) {
    espaces_vides++; direction_i = i - 1; direction_j = j;}
59                  if (i < LIGNES - 1 && p -> grille[i + 1][j] ==
    VIDE) { espaces_vides++; direction_i = i + 1; direction_j = j;}
60                  if (j > 0 && p -> grille[i][j - 1] == VIDE) {
    espaces_vides++; direction_i = i; direction_j = j - 1;}
61                  if (j < COLONNES - 1 && p -> grille[i][j + 1]
    == VIDE) { espaces_vides++; direction_i = i; direction_j = j +
    1;}
62
63                  if (espaces_vides == 1) {
64                      if (peut_placer_tente(p, direction_i,
    direction_j)) {
65                          placer_tente(p, direction_i,
    direction_j);
66                          modifications = 1;
67                      }
68                  }
69              }
70          }
71      }
72
73  }
74
75  if (verif_solution(p)) return 1;
76  return 0;
77 }
```

Résolution par backtracking

La fonction de backtracking est la plus important pour résoudre les puzzles. Elle prend en paramètre un plateau et un booléen. Elle renvoie un entier signé. Premièrement, la fonction va vérifier si le plateau est une solution valide. Elle retourne 1 si c'est le cas. Ensuite, elle fait une sauvegarde du plateau actuel au cas ou la solution recherchée par le backtracking serait non valide. Puis, elle vérifie pour l'intégralité des cases du plateau si on peut y placer une TENTE. Si oui, elle l'a place puis fait une résolution logique du nouveau plateau pour écarter les contraintes logiques du jeu. Puis on rappelle la fonction de backtrack récursivement pour pourvoir continuer à placer des TENTES. Si à la fin du parcours d'un arbre de possibilité on tombe sur une solution non juste, on retourne 0 et on restaure le plateau précédent avec l'ancienne sauvegarde. Enfin, si booléen est à true, la fonction permet de voir tous les états différents du backtracking avec des pauses de 1 seconde après chaque modification de celle-ci.

```c
int backtrack(Plateau *p, bool slowMode) {
    // Si la solution est trouvee, on arrete
    if (verif_solution(p)) return 1;
    if (slowMode){
        clear_terminal();
        afficher_plateau(p);
        sleep(1);
    }

    // Parcourir toutes les cases du plateau
    for (int i = 0; i < LIGNES; i++) {
        for (int j = 0; j < COLONNES; j++) {
            // Essayer de placer une tente
            if (peut_placer_tente(p, i, j)) {
                Plateau *backup = copie_plateau(p);
                placer_tente(p, i, j);
                if (slowMode){
                    clear_terminal();
                    afficher_plateau(p);
                    sleep(1);
                }
                resolution_logique(p);
                if (slowMode){
                    clear_terminal();
                    afficher_plateau(p);
                    sleep(1);
                }

                if (backtrack(p, slowMode)) return 1;

                // Annuler le placement si a ne mene pas une solution
                restaurer_plateau(p, backup);
                free_plateau(backup);

                if (slowMode){
                    clear_terminal();
                    afficher_plateau(p);
                    sleep(1);
                }
            }
```

```
41          }
42      }
43
44      return 0;
45  }
```

Fonction de lecture d'un fichier de jeu de Tentes

La fonction de lecture d'un fichier de Tentes prend une chaîne de caractère constante et un plateau. Elle renvoie 1 si aucune erreur ne s'est produite, -1 sinon. Elle lit un fichier pour en extraire les informations nécessaires d'un jeu de Tentes. Format d'un fichier valide :

- doit commencer par "debutPlan"
- si "lignes" est lu : lit les tentes nécéssaires sur chaque ligne du jeu. Le séparateur pour chaque nouvelle ligne est ";"
- si "colonnes" est lu : lit les tentes nécéssaires sur chaque colonne du jeu. Le séparateur pour chaque nouvelle colonne est ";"
- si "arbres" est lu : lit les coordonées (x, y) de chaque arbre. Le séparateur entre chaque coordonéee x et y est ";"
- doit terminer par "finPlan"

Fonction jouer

La fonction jouer sert aux utilisateurs à jouer au jeu de Tentes sans aucune résolution automatique.

Fonction main

La fonction principale (main) initialise un plateau, puis lit un fichier pour remplir celui-ci. Elle donne la possibilité de jouer, résoudre par logique et résoudre par backtrack. Elle libère le plateau ensuite.

Tests

Les résultats des tests effectués montrent que la résolution en backtrack et relativement rapide sur des grilles de 8x8. Cependant, certaines grilles sont plus rapidement résolues avec une simple utilisation de l'algorithme de résolution logique. Les grilles en 20x20 restent insolvable avec l'algorithme de backtracking actuel, car prenant trop de temps.

Améliorations

Une des améliorations possibles serait l'utilisation d'heurestique qui priorise le remplissage de Tente dans les cases où le nombre de tente requise sur une ligne et une colonne est petit.