Rapport Projet LabyrinTrack

Dans le cadre de ce projet, je dois développer un algorithme permettant de résoudre un labyrinthe, en utilisant une approche de type backtracking. Le labyrinthe est représenté par une grille de caractères, dans laquelle différentes entités sont présentes. L'objectif du programme est de trouver un chemin allant du point de départ D au point d'arrivée A, en passant par les trois objets mentionnés (couronne, bouclier et épée respectivement pour C, B, E). Je dois écrire un programme en C++ qui permet de résoudre un labyrinthe via un algorithme de type backtracking. Mon programme doit prendre en entrée un fichier texte décrivant le labyrinthe et devra afficher le chemin de cette partie.

Implémentation Algorithme de Backtracking Séquentielle

- Ouvrir un terminal
- Se placer à la racine du projet `projet/LabyrinTrack`
- Vérifier la présence du fichier `labyrinthe.txt'
- Compiler avec la commande suivante pour compiler:

```
g++ -std=c++11 main.cpp Labyrinthe.cpp -o labyrintheV1
```

- Lancer le programme

```
./labyrintheV1
```

Un chrono donne le temps d'exécution avec l'algorithme de Backtracking Séquentielle en milliseconde.
La résolution du labyrinthe (chemin emprunté) est caractérisé par des `*`

Il y a 4 grilles affichées en couleur avec leurs résolutions.

Explication dur l'Algorithme de Backtracking (Séquentielle) utilisée

J'ai codé une fonction récursive: `bool Labyrinthe::backtracking(const Position &position, vector<Position> &cheminActuel` qui doit s'arréter quand on est sur la case A arrivée et, que les objets ont tous été ramassés.

Sinon, de façon récursive dans une pile `cheminActuel`:

je vérifie la présence: mur, monstre,, case déjà visitée (position), on arrête

Je vérifie la présence d'un objet à ramasser: on le ramasse.

On avance dans un ordre précis dans les 4 directions: droit gauche bas haut

Si cela ne marche pas, on revient en arrière en effaçant le chemin (positons dans la pile) et on efface l'objet ramassé.

Implémentation Algorithme de Backtracking Parallèle Variante 1

- Ouvrir un terminal
- Se placer à la racine du projet `projet/LabyrinTrackVariante1`
- Vérifier la présence du fichier `labyrinthe.txt'
- Compiler avec la commande suivante pour compiler:

```
g++ -std=c++11 -pthread main.cpp Labyrinthe.cpp -o labyrintheV2
```

- Lancer le programme

```
./labyrintheV2
```

Un chrono donne le temps d'exécution avec l'algorithme de Backtracking parallèle variante1 en milliseconde. La résolution du labyrinthe (chemin emprunté) est caractérisé par des `*` Il y a 4 grilles affichées en couleur avec leurs résolutions.

J'avais essayé de garder ma fonction récursive `Labyrinthe::backtracking(const Position &position, vector<Position> &cheminActuel)` et de créer des thread soit par direction ou par grille mais à chaque fois, il y avait un dépassement de mémoire et donc des erreurs impossibles pour moi à résoudre simplement. Je me suis dis que c'était la récursivité de ma fonction qui posait un problème. J'ai donc décidé de reécrire cette fonction et de la rendre non récursive avant toute création de thread. J'ai utilisé une pile pour stocker les états. J'ai trouvé les informations sur ce concept sur internet. Le codage a été plutôt long et pas aussi facile que je le pensais.

```
// Structure pour mémoriser l'état en cours de mon backtracking
struct EtatBacktracking {
    Position position;               // La position en cours
    vector<Position> cheminActuel;   // Le chemin parcouru jusqu'à la position mémorisée
    int direction;                   // La prochaine direction à essayer (0 = droite, 1 = gauche, 2 = bas, 3 = haut)
    bool objetRamasse;               // Objet ramassée à cette position
    char symboleObjet;               // le synbole de l'objet ramassé

    // Initialisation de l'état avec une position (depart) et un chemin (vide)
    EtatBacktracking(const Position& positionEncours, const vector<Position>& chemin, int dir = 0) {
        position = positionEncours;             // On fixe la position en cours
        cheminActuel = chemin;      // chemin déjà parcouru
        direction = dir;            // direction 0 (droite) par défaut
        objetRamasse = false;       //  aucun objet
        symboleObjet = ' ';         // au départ (vide)
    }
};
```

```cpp
// Ma version non récursive de backtracking en utilisant une pile
bool Labyrinthe::backtracking(const Position &positionDepart, vector<Position> &cheminTrouve) {

    // Pile pour gérer les étapes du backtracking
    stack<EtatBacktracking> pile;

    // Ensemble (set) pour stocker les positions déjà visitées, plus rapide d'accès que <vector>
    set<Position> positionsVisitees;

    // On pousse l'état de départ dans la pile + chemin vide
    pile.push(EtatBacktracking(positionDepart, vector<Position>()));
    positionsVisitees.insert(positionDepart);

    // Tant que la pile n'est pas vide, il reste des positions à vérifier
    while (!pile.empty()) {

        // On regarde l'état au sommet de la pile sans le retirer
        EtatBacktracking &etat = pile.top();

        // Si on a essayé les 4 directions (0 à 3), on doit revenir en arrière tout annuler les actions et les etats
        if (etat.direction > 3) {
            // Si un objet a été ramassé à cette étape, annuler le ramassage
            if (etat.objetRamasse) {
                for (auto &objet : objets) {
                    if (objet.symbole == etat.symboleObjet && objet.grille == etat.position.grille) {
                        objet.ramasse = false;
                        break;
                    }
                }
            }
            // On enlève la position du chemin et on la retire des positions visitées
            if (!etat.cheminActuel.empty()) {
                positionsVisitees.erase(etat.position);
                etat.cheminActuel.pop_back();
            }
            // On enlève cet état de la pile (on revient en arrière)
            pile.pop();
            continue; // On passe à l'itération suivante
        }

        // si position direction droite = 0
        if (etat.direction == 0) {
            // Vérifie si on est arrivé à la position d'arrivée
```

```
            if (etat.position == arrivee && (tousObjetsRamasses() ||
grilleActuelle == 3)) {

                // Copie la positon dans le chemin final
                chemin = etat.cheminActuel;
                return true;
            }

            // On vérifie si la position en cours est valide
            if (!estValide(etat.position) || estObstacle(etat.position) ||
                (grilleActuelle != 3 && estMonstre(etat.position))) {
                // Si position pas valide, on pop sur la pile
                pile.pop();
                continue;
            }

            // on peut ramasser un objet à cette position ?
            if (peutRamasserObjet(etat.position)) {
                char symbole = getCase(etat.position); // symbole de l'objet
                for (auto &objet : objets) {
                    if (objet.symbole == symbole && objet.grille ==
etat.position.grille && !objet.ramasse) {
                        objet.ramasse = true;          // ramasse
                        etat.objetRamasse = true;    // marquage
                        etat.symboleObjet = symbole; // mémorise le symbole
                        break;
                    }
                }
            }

            // on mémorise la position actuelle
            etat.cheminActuel.push_back(etat.position);
        }

        // on décide la prochaine position par ordre
        Position prochainePosition = etat.position;
        switch (etat.direction) {
            case 0: prochainePosition.x++; break; // Droite
            case 1: prochainePosition.x--; break; // Gauche
            case 2: prochainePosition.y++; break; // Bas
            case 3: prochainePosition.y--; break; // Haut
        }

        // On passe à la direction suivante pour la prochaine fois
        etat.direction++;

        // Si la prochaine position pas été visitée
        if (positionsVisitees.find(prochainePosition) == positionsVisitees.end())
{
```

```
            // On la marque comme visitée
            positionsVisitees.insert(prochainePosition);

            // On ajoute un nouvel état dans la pile avec cette nouvelle position et le chemin actuel
            pile.push(EtatBacktracking(prochainePosition, etat.cheminActuel));
        }
    }

    // Si on sort de la boucle
    return false;
}
```

A partir de cette base, j'ai cherché à implémenter une première variante d'algo de backtracking parallèle en utilisant la librairie thread de C++11.

mon programme cherche un chemin dans un labyrinthe à plsueurs grilles en utilisant un backtracking,, c'est à dire qu'il explore tous les chemins possibles jusqu'à trouver une solution. J'ai amélioré avec cette variante de backtracking en crant plusieurs threads en parallèle, un epile pour stocker les positions à vérifier (en mettant ma fonction en non récursive) et une protection des données partagées avec l'utilisation de fonctionalités des librairies `mutex` et `atomic` Mon backtracking fonctionne en utilisant un pile (stack) poru garder en mémoire les positions à vérifier (evite une récursive) Chauqe élément de la pile est élement de type `EtatBacktracking` qui contient la position en cours, la direction à réaliser et le chemin parcouru, la liste des objets ramassés. On essaye ne boucle 4 directions dans un ordre précis. On vérifie que l'on est arrivé à la case finale `A` et que tous les objets ont été ramassés.: dans ce cas, la solution a été trouvée. Si la case est invalide (mur, monstre, hors de la grille), on revient en arrière en supprimant de la pile. Pour le ramassage des objets, j'ai placé un mutex pour éviter que deux threads ramassent le même objet en même temps.

Pour mon parallèlisme, je lance plsuieurs threads (selon la disponibilité avec la fonction `thread::hardware_concurrency()`). Chaque thread explore le labyrinthe de façonb indépendante avec un même position de départ. Quand un thread trouve la colution (`solutionTrouveeGlobale = true` qui est un vairiable atomique afin que les thread puisse partager l'information et que tous les threads soient au courant), on arrête les autres threads. Le mutex sert à protéger les données partagées comme la solution globale (cheminSolution) et l'état des objets trouvés et ramassés.

Implémentation Algorithme de Backtracking Parallèle Variante 2

- Ouvrir un terminal
- Se placer à la racine du projet `projet/LabyrinTrackVariante2`
- Vérifier la présence du fichier `labyrinthe.txt'
- Compiler avec la commande suivante pour compiler:

```
g++ -std=c++11 -pthread main.cpp Labyrinthe.cpp -o labyrintheV3
```

- Lancer le programme

```
./labyrinthev3
```

J'ai essayé de partir avec le programme de la variante 1 mais cette fois-ci au lieu que tous les threads commencent sans direction spéciale et font la même chose, cette fois-ci chaque thread a sa direction imposée: par exemple le thread 1 commence directement à droite, le thread 2 directement à gauche. On crée 4 threads avec un boucle et on donne à chacun une direction différente 0=droite, 1=gauche, 2=bas, 3=haut). Chaque thread démarre dans une direction différente mais ensuite il continue comme un backtracking normal, il explore les autres directions. Cela permet de répartir le travail et de trouver une solution plus rapidement. Dès qu'un thread trouve la solution, il arrête les autres avec la variable `solutionTrouveeGlobale`

Un exemple simple:

Thread 1 à l'ordre d'aller à droite: il va à droite mais trouve un mur

Thread 2 à l'ordre d'aller à gauche: il va à gauche et il trouve la bonne case

Thread 3 à l'ordre d'aller à bas: il va en bas et il trouve un monstre

Thread 4 à l'ordre d'aller en haut: il va en haut et trouve un mur

le Thread 2 a trouve la solution le premier et il arrête les autres.

Cela est plus efficace mais aussi peut avoir des varaiations de chemin trouvé selon si il y a plusieurs bonnes case trouvées et la vitesse du premier thread qui va envoyer le signale d'avoir trouve la solution.