Projet Parallèlisation : Solveur de Tetravex

Tommy ANFOSSO

November 9, 2020

1 Manuel d'utilisation du programme

On passe en argument du programme le fichier .txt resprésentatif de la configuration Tetravex à résoudre. Puis le programme affiche la première solution trouvée, c'est à dire le contenu de la grille dans laquelle toutes les pièces ont été placées. Un message est affiché dans le cas où aucune solution n'a été trouvée.

2 Implémentation des algorithmes de backtracking

2.1 Implémentation en séquentiel

Je suis parti sur une implémentation itérative avec une stack qui permet de simuler la récursion. Je pensais que l'algorithme ne pouvait être implémenté en une fonction "tail-recursive", et que de ce fait le compilateur n'aurait pas pu optimiser le code.
J'utilise des shared_ptr à la place des pointeurs classiques afin de ne pas avoir de problèmes pour libérer l'espace mémoire alloué.

```
bool Solver::solve()
{
    auto& cur = this->currentBoard;

    //on ajoute la grille vide à la pile
    this->stack.push(cur);
    //tant qu'il y a des plateaux à traiter..
    while(!this->stack.empty())
    {
        //on récupère le prochain plateau à traiter
        cur = this->stack.top();
        this->stack.pop();

        //si le plateau est résolu, on retourne vrai
        if(cur->isSolved())
```

```
            return true;
        //sinon ..
        else
        {
            //on génère les candidats fils
            for(int i = 0 ; i < cur->tiles->size() ; ++i)
            {
                //uniquement si respect des régles tetravex
                if(cur->tileCanBePlaced(i))
                {
                    //on clone le plateau parent
                    auto c = std::make_shared<Board>(*cur);
                    //on y ajoute la nouvelle pièce
                    c->grid[i] = (*c->tiles)[c->tileToPlace];
                    //cette pièce n'a plus besoin d'être placée
                    c->tileToPlace++;
                    //on ajoute le plateau fils à la pile
                    this->stack.push(c);
                }
            }
        }
    }
    //si aucune solution n'a été trouvée, on retourne faux
    return false;
}
```

Listing 1: methode pour resoudre un plateau

2.2 Implémentation en parallèle

Pour la version multithread, j'ai adopté un design Thread Pool tout en réutilisant la méthode Solver::solve() de la version séquentielle.
Le nombre de threads en simultané est paramétrable lors de la construction du ThreadPool.
Je définis autant de Solvers que de thread, une pile remplie avec les premiers plateaux candidats (uniquement la première pièce de plaçée) et un mutex pour protéger l'accès à la pile.
La fonction callback utilisée par les threads du ThreadPool est la méthode ThreadPool::execute() qui permet de lancer l'exploration de plateaux candidats à partir d'un plateau pris dans la pile.

Cette méthode suit les étapes suivantes :

1. Il récupère le prochain candidat à explorer de la pile (protection avec mutex)
2. Il récupère le Solver dédié au thread

3. Il affecte le plateau de départ au Solver (racine de l'arbre à explorer)
4. Il nettoie la pile du Solver avant de commencer l'algorithme
5. Il lancer la résolution du plateau en appelant la méthode Solver::solve()
6. Si il trouve un résultat :
   - Il affiche la grille résultat
   - Il libère le Solver
   - Il lance un signal d'extinction à tous les autres threads

3 Interprétation des résultats

J'ai malheureusement eu quelques soucis lorsqu'il s'agissait de résoudre des instances de grille supérieures à 5x5, le programme prend beaucoup trop de temps. Je me suis donc contenté de reporter mes résultats pour les grilles allant jusqu'à 5x5. Pour ce qui est de l'interprétation, je me fierai davantage aux résultats escomptés contenus dans le sujet du projet.

Temps d'exécution en séquentiel :

- 2x2 : 0m0.014s
- 3x3 : 0m0.016s
- 4x4 : 0m0.215s
- 5x5 : + de 6 minutes

Pour une grille de Tetravex de petite taille (jusqu'à 6x6), l'implémentation séquentielle sera plus performante car la proportion de bons candidats générés est grande.
La construction de threads dans l'implémentation parallèle est gourmande et, dans ces conditions, elle va allonger le temps d'exécution plutôt que de le réduire. En revanche, à partir d'une dimension 7x7, l'implémentation parallèle se justifie et nous permet d'avoir de bien meilleures performances comparée à sa version séquentielle.

On remarque aussi que le temps d'exécution n'est pas linéaire en fonction de la taille du plateau.
Ceci est dû à l'algorithme de backtracking lui-même.
Plus le nombre de pièces augmentent, plus la proportion de mauvais candidats générés augmente, et plus l'algorithme perd du temps à explorer des branches en vain et à faire des manoeuvres de backtracking.

La parallélisation n'effectue pas moins de travail mais l'effectue plus rapidement, même parallélisées les performances du programme seront limitées par la complexité de l'algorithme utilisé.