INF4705

Laboratoire 3 : Analyse et conception d'algorithmes

Remis par :

Sébastien Crevier (1667643)

Andréanne Laurin (1738541)

Mercredi le 19 avril 2017

Introduction

L'objectif de ce travail est d'amener les étudiants à créer un algorithme unique permettant de trouver le meilleur tracé des chemins de randonnées pour le parc des Laurentides. Un classement des divers algorithmes soumis sera fait en fonction de la solution la plus optimale. En effet, il faut utiliser judicieusement les ressources disponibles de l'ordinateur, car ce classement détermine le nombre de points obtenu pour la qualité de l'algorithme. Afin que le concours soit juste, tous les ordinateurs sur lesquels les tests seront fait possèdent un processeur Intel Core i7-6700HQ à 2.60GHz et une mémoire vive de 16Go.

Présentation de l'algorithme

Nous avons décidé de créer quatre *threads* sur lesquels notre implémentation de l'algorithme de Kruskal est lancé en parallèle pour explorer différentes heuristiques avec des hyperparamètres différents. Lorsqu'une meilleure solution, celle-ci est projeté sur la sortie standard du programme. Les hyperparamètres utilisés sont générés aléatoirement afin d'explorer le plus de pistes de solutions possible.

<u>Étapes :</u>

1. Quatre *threads* sont exécutés avec des hyperparamètres variés.
2. En fonction de l'un des hyperparamètre, on ordonne la liste de tous les arcs en fonction de leur coût respectif. Une valeur différente de ce paramètre engendrera un ordonnancement aléatoire d'une certaine quantité des premiers arcs dans la liste déjà triée.
3. L'algorithme de Kruskal est appelé. Une variation de cet algorithme populaire a été utilisée afin de seulement lier les noeuds qui ne sont pas complets (degré maximal atteint) ou encore qui ne sont pas deux points d'observation époustouflants. On vérifie aussi si la nouvelle liaison n'engendre pas de cycle dans le graphe.
4. Une fois un premier graphe généré, on filtre les arcs inutiles.
5. Par la suite, on tente de compléter le graphes en ajoutant des arcs supplémentaires au

besoin.

6. On effectue l'étape 4 à nouveau.
7. On effectue ensuite les étapes 5 et 6 une seconde fois, avec une petite variation en fonction des hyperparamètres passés.
8. Le graphe ainsi trouvé est comparé avec la dernière meilleure solution, et s'il est meilleur, on met à jour cette solution commune et on affiche cette dernière sur la sortie standard.

Pseudo-code de notre implémentation de l'algorithme de Kruskal

```
Algorithme de Kruskal(bool randomizeEdgeSort, bool randomizeInsertion):
    vector arcs;
    arcs = this->arcs;

    //Heuristique 1 : reordonner la liste des arcs lors creation du graph
    IF(randomizeEdgeSort){
        shuffle(arcs)
    }

    For each arc in Graph{
        IF (arc.noeud1 != VIEWPOINT && arc.noeud2 != VIEWPOINT){
            arbreLongueurMinimum.push_back(arc);
            addEdge(arc.noeud1, arc.noeud2);

            IF (arbreLongueurMinimum contient déjà arc  ||  arc créer un cycle){
                arbreLongueurMinimum.pop_back(arc);
                removeEdge(arc.noeud1, arc.noeud2);
            }
        }
    }

    For each arc in arbreLongueurMinimum {
        IF(arc non necessaire){
            arbreLongueurMinimum.pop_back(arc);
            removeEdge(arc.noeud1, arc.noeud2);
        }

    For each noeud in arbreLongueurMinimum.arcNonUtilise {
        IF(noeud !relies || noeud !connecteEntree) ){
            //Heuristique 2 : inserer a un index aleatoire l'arc lors completion du graph
            IF(randomizeInsertion){
                arbreLongueurMinimum.insert(randomIndex, arcNonUtilise);
            }
            ELSE
            arbreLongueurMinimum.push_back(arcNonUtilise);
        }
    }

    nouvelleSolution = new Solution(arbreLongueurMinimum);
    solutionMutex.lock();

    IF(solution.cout > nouvelleSolution.cout){
        solution = nouvelleSolution
        solutionMutex.unlock();
        return true;
    }

    solutionMutex.unlock();
    return false;
```

Analyse de la complexité théorique

Où V est le nombre de points d'intérêts, n le nombre d'arcs et u le nombre d'arcs non utilisés.

1. **void Graph::filterUnnecessaryEdges(vector<Edge*>& tree, vector<Point*>& points)**

La complexité de cette fonction est O(n). Une boucle while est utilisé sur la taille du vecteur donc O(n) de plus à l'intérieur de la boucle on utilise la fonction erase sur l'itérateur qui a aussi une complexité de O(n). Ce qui résulte en une complexité de O(n+n) donc O(n).

2. **void Graph::connectInvalidPoints(vector<Edge*>& tree, vector<Edge*>& unusedEdges, vector<Point*>& points, bool keepUnsuccessfulConnections, bool randomizeInsertion)**

Pour la boucle for, la complexité est de O(V). Cependant, celle-ci contient aussi une boucle while qui possède une complexité de O(u). Ce qui veut dire que la complexité de cette fonction serait de O(V*u).

3. **bool Graph::isCyclic(list<int>* adj)**

Cet algorithme de détection de cycle a été inspiré du site web geeksforgeeks (voir référence ci-dessous) et possède une complexité de O(V+n). En effet, il effectue une traversée du graphe à l'aide de la liste d'adjacence.

4. **bool Graph::kruskal(bool sortEdgesByCost, bool randomizeInsertion, float hyperparam)**

Pour toute les Initialisation de vecteurs, de liste et les opérations de *push_back* ou de *pop_back* nous avons une complexité de O(1). Pour les boucles for qui roule un nombre constant de fois, la complexité est aussi de O(1). L'utilisation de la fonction std::sort de la STL possède une complexité de O(n log n). L'utilisation de la fonction shuffle possède une complexité de O(n). La boucle while boucle un maximum de fois de la taille du vecteur d'arc donc O(n) et possède la

méthode *isCyclic* décrite plus haut qui a une complexité de O(V+n), ce qui lui donne une complexité de O(V+n). Dernièrement, l'algorithme de krukal utilise les fonctions *filterUnnecessaryEdges* et *connectInvalidPoints* qui ont chacune une complexité de = O(n) et O(V*u).

Au final, l'algorithme de kruskal qui a été implémenté a une complexité de O(n log n + V*u)

Conception de l'algorithme

Ce qui rend notre algorithme unique est la combinaison des différentes étapes. Malgré l'utilisation de l'algorithme de Kruskal et la détection de cycle qui ont été inspirés d'algorithmes déjà existants, ce sont les étapes intermédiaires qui font toute la différence. En effet, comme mentionnée plus haut, l'algorithme est lancé sur quatre processus en parallèle et ils possèdent un espace mémoire partagé qui permet d'accélérer la découverte de nouvelles solutions. De plus, au début de l'algorithme de Kruskal, l'ordonnancement des arcs peut être fait de manière aléatoire si le processus est lancé avec le paramètre sortEdgesByCost à vrai. Il y a aussi l'utilisation de la méthode *filterUnnecessaryEdges* qui permet d'éliminer les arcs en trop après la complétion d'un graphe. De plus, la méthode *connectInvalidPoints* permet de trouver une solution juste en insérant aléatoirement des arcs entre les noeuds encore disponibles (degré actuel plus petit que leur degré maximal). Ces passes sont appliqués à plusieurs reprises à chaque itération afin de potentiellement trouver une solution plus optimale.

Références

- Notre version de l'algorithme de Kruskal a été inspiré de: http://www.sciencedirect.com/science/article/pii/S0166218X0500301X

- L'algorithme de détection de cycles dans les graphes a été inspiré de: http://www.geeksforgeeks.org/detect-cycle-undirected-graph