Rapport

Jean-emmanuel Chouinard (20246807), Timothe Payette (20239892)

12 juin 2023

1 Auto-évaluation

Notre programme fonctionne comme prévu. Il respecte les consignes et le format des sorties demandé. Par contre, la règle des longitudes et latitudes n'est pas respectée. Ceci est correct puisque le démonstrateur en tiendra compte lors de sa correction :) .

2 Analyse de la complexité temporelle (pire cas) théorique en notation grand O

```
// Start timing algorithm
long startTime = System.nanoTime();

// Find building with the most boxes and put truck in it
sortBuildingsByBox(buildings);
buildings.get(0).changeTruck();
truck.setTruckLatitude(buildings.get(0).getLatitude());
truck.setTruckLongitude(buildings.get(0).getLongitude());
truck.setCapacity(boxes);

// Do the distance travel optimization
Building location = buildings.get(0);
sortBuildingsByDistance(location, buildings);

// Stop timing algorithm
long endTime = System.nanoTime();
long duration = (endTime - startTime);
System.out.println(duration + "ns");
```

Notre programme peut être divisé en deux principales section. La première section tri la liste d'entrepôt (ici nommé building) en fonction du nombre de boites qu'il contient. De cette manière, il est assuré que le premier élément de la liste est celui avec le plus de boite. Voici la fonction $sortBuildingsByBox$.

```
public static void sortBuildingsByBox(ArrayList<Building> buildings) {
    int n = buildings.size();

    for (int i = 1; i < n; i++) {
        Building key = buildings.get(i);
        int j = i - 1;

        while (j >= 0 && buildings.get(j).getAvailableBoxes() < key.getAvailableBoxes()) {
            buildings.set(j + 1, buildings.get(j));
            j--;
        }
        buildings.set(j + 1, key);
    }
}
```

Dans cette fonction de tri, nous passons à travers la liste de buildings deux fois dans le pire des cas (soit que la liste est classée en ordre décroissante), ce qui rend notre algorithme de complexité $O(n^2)$ où n est le nombre de buildings dans la liste.

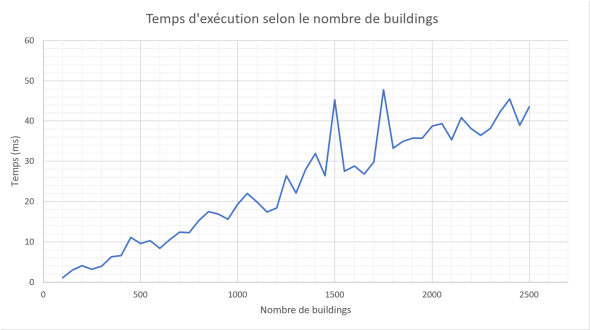
Observons la deuxième partie de l'algorithme où la liste est triée en fonction de la distance entre le premier élément de la liste et les autres.

```java
public static void sortBuildingsByDistance(Building location, ArrayList<Building> buildings) {
    int n = buildings.size();

    // Calculate distances from location
    for (int i = 1; i < n; i++) {
        double distance = calculateDistance(location.getLatitude(), location.getLongitude(),
                buildings.get(i).getLatitude(), buildings.get(i).getLongitude());
        buildings.get(i).setDistance(distance);
    }

    // Sort buildings from closest to longest distance
    for (int i = 1; i < n; i++) {
        Building key = buildings.get(i);
        int j = i - 1;

        while (j >= 0 && buildings.get(j).getDistance() > key.getDistance()) {
            buildings.set(j + 1, buildings.get(j));
            j--;
        }
        buildings.set(j + 1, key);
    }
}
```

Dans cette fonction de tri, nous passons deux fois à travers la liste dans le pire des cas. Comme celle de *sortBuildingsByBox*, la complexité est de $O(n^2)$. De plus, le calcul des distances entre les buildings et le camion a une complexité de $O(n)$. Finalement, cette fonction a une complexité de $O(n^2) + O(n) = O(n^2)$.

En bout de ligne, notre algorithme de tri a une complexité de $O(n^2) + O(n^2) = O(n^2)$.

3 Analyse de la complexité temporelle empirique



Les données utilisées (n) pour ce graphique vont de 100 à 2500 par bond de 50.