INF4705: Analyse et conception d'algorithme

Laboratoire 3

Présenté à : Samuel Gagnon

Soumis par

Raphael Christian-Roy(1743344)

Louis-Charles Hamelin(1742949)

Section 01 (B1)

Mercredi 19 avril 2017

Introduction

Lors de ce dernier travail pratique, nous devions implémenter un algorithme de notre cru afin de résoudre un problème combinatoir. Le problème est fort simple, il s'agit de déterminer le chemin le plus court permettant de passer tous les points d'intérêts dans un parc. Par contre, il faut s'assurer que chaque point d'intérêt à un chemin vers une des entrées du parc. De plus, chaque entrée du parc doit être le point de départ d'un sentier. Les points étape doivent avoir au moins deux sentiers incidents. Finalement, les points de vue sont accessibles que par un seul sentier et chaque point d'intérêt à un nombre limité de sentiers incidents.

Cadre expérimental

Pour ce qui est de l'environnement de développement et d'analyse, nous avons utilisé un MacBook Pro 2016, ayant 16Gb de mémoire vive et un processeur 3.3 GHz Intel Core i7. La technologie utilisée pour concevoir les algorithmes est Java.

Jeux de données

Dans ce laboratoire, nous avions comme série de données 5 type de parc, dont chacune on 3 variantes enc e qui a trait le nombre de point d'intérêts. De plus, nous avions une liste des types de points d'intérêt, soit 1: époustouflant, 2: un point d'entrée, 3: point étape. Nous avions aussi une liste du nombre maximum de sentiers associés à chaque point d'intérêt. Finalement, nous avons une matrice permettant d'identifier le coût entre chaque paire de sommet.

Présentation de l’algorithme

Fonction `findMinPath`

Pseudo-code

```
tant que nbTour > 0 faire
    randomNodeIndex <- random.nextInt(nbreTourDeLoop)
    switch(typeList)
        Case1:
            Si nbEdgePermisRestant == 0 alors
                noeudRestantALier.remove()
            tant que nbEdgePermisRestant <= 0 faire
                Si testList.size() == nbTotalNode alors
                    noeudRestantALier.remove()
                Sinon
                    randomNumber = random.nextInt
                    Si !testList.contain(randomNumber) alors
                        testList.add(randomNumber)
            Si currentCostPath + costMatrix > bestCost alors
                Retrurn
            Sinon
                currentCostPAth += costMatrix
                nbEdgePermisRestant[randomNodeIndex] -= 1
                nbEdgePermisRestant[randomNumber] -= 1
        Case2:
            Si nbEdgePermisRestant == 0 alors
                noeudRestantALier.remove()
            tant que nbEdgePermisRestant <= 0 faire
                Si testList.size() == nbTotalNode alors
                    noeudRestantALier.remove()
                Sinon
                    randomNumber = random.nextInt
                    Si !testList.contain(randomNumber) alors
                        testList.add(randomNumber)
            Si currentCostPath + costMatrix > bestCost alors
                Retrurn
            Sinon
                currentCostPAth += costMatrix
                nbEdgePermisRestant[randomNodeIndex] -= 1
                nbEdgePermisRestant[randomNumber] -= 1
```

```
Case3:
    Si nbEdgePermisRestant == 0 alors
        noeudRestantALier.remove()
    tant que nbEdgePermisRestant <= 0 faire
        Si testList.size() == nbTotalNode alors
            noeudRestantALier.remove()
        Sinon
            randomNumber = random.nextInt
            Si !testList.contain(randomNumber) alors
                testList.add(randomNumber)
    Si currentCostPath + costMatrix > bestCost alors
        Retrurn
    Sinon
        currentCostPAth += costMatrix
        nbEdgePermisRestant[randomNodeIndex] -= 1
        nbEdgePermisRestant[randomNumber] -= 1

isEtapesEdgesMinAtteint = isEtapesEdgesMinAtteintFct
isEtapesReachEntree = isEtapesReachEntreeFct
isWowSpotReachEntree =  isWowSpotReachEntreeFct
isAllNodeInMinPath = isAllNodeInMinPathFct

Si currentCostPath < bestPath alors
    printPath(currentCouplesFound)
```

Fonction `isAllNodeInMinPathFct`

Pseudo-code

```
Pour i = 0 à nbCouple faire
    Si !allNodeVerif.contain(couple[0]) alors
        allNodeVerif.add(couple[0])
    Si !allNodeVerif.contain(couple[1]) alors
        allNodeVerif.add(couple[1])

Pour i = 0 à allNodeInParc faire
    Si !allNodeVerif.contain(couple[0]) alors
        isAllNodeInMinPath = false
        Break
    isAllNodeInMinPath = true
Return isAllNodeInMinPath
```

Complexité théorique

O(n+m)

Fonction `isEtapesEdgesMinAtteintFct`

Pseudo-code

```
Pour i = 0 à etapeIndexPosition faire
    Si nbMaxEdge - nbEdgePermisRestant < 2 alors
        isEtapesEdgesMinAtteint = false
        Break
    isEtapesEdgesMinAtteint = true

Return isEtapesEdgesMinAtteint
```

Complexité théorique

O(n)

Fonction `isEtapesReachEntreeFct`

Pseudo-code

```
Pour i = 0 à etapeIndexPosition faire
    Pour j = 0 à entreeIndexPosition faire
        Si graph.isReachable alors
            isEtapesReachEntree = true
            Break
    Si !isEtapesReachEntree alors
        Break

Return isEtapesReachEntree
```

Complexité théorique

O(m*n)

Fonction `isWowSpotReachEntreeFct`

Pseudo-code

```
Pour i = 0 à wowSpotIndexPosition faire
        Pour j = 0 à entreeIndexPosition faire
                Si graph.isReachable alors
                        isWowReachEntree = true
                        Break
        Si !isWowReachEntree alors
                Break

Return isWowReachEntree
```

Complexité théorique

O(m*n)

Fonction `isCoupleAlreadyThere`

Pseudo-code

```
Pour i = 0 à currentCoupleFound faire
                Si Arrays.equals(coupleATesterA)) || (Arrays.equals(coupleATesterB)))
                        Return true

Return false
```

Complexité théorique

O(n)

Originalité de notre algorithme

Notre algorithme tire son originalité du fait qu'il est de type aléatoire. En effet, ce dernier peut être surnommé l'algorithme de Vegas ou même Monte Carlos. La raison de ces surnoms est fort simple, notre algorithme effectue des essaies aléatoire afin de trouver le meilleur chemin selon certaines conditions et ce continuellement. Ainsi, si l'essaie qu'il est entrain de faire obtient un meilleur résultat que le meilleur précédent, on conserve ce dernier et on l'utilise comme barème pour la prochaine itération. Selon nous cet algorithme nous permet de fournir un résultat rapidement et ayant un coût relativement moyen. C'est un très bon compromis en ce qui attrait de difficulté d'implémentation et de résultats.