Implémentation d'algorithmes de recherche opérationnelle. Projet phase 3

Polytechnique Montréal

MTH6412B

Xavier Lebeuf et Geoffroy Leconte

Octobre 2021

1. Importation des fichiers

Main.workspace2.benchmark_table_KruskalPrim

```
begin
    using Plots
    using PlutoUI
    using Test
    using PrettyTables

    include("mainphase3.jl")
end
#hidden line
```

2. Modifications à la phase 1 et 2

Application des commentaires

1. L'attribut "data" du type Nodes a été changé pour "coordinates", un terme plus représentatif.
2. "Edges{I, J}" est maintenant "Edges{T, I}" pour correspondre à "Node{T}".
3. Les fonctions "add_edge!" et "add_node!" sont maintenant plus robustes en effectuant l'ajout seulement si l'élément à ajouter n'existe pas déjà.
4. La fonction "isless" a été implémentée pour comparer des arêtes par leur poids. Nous pouvons alors utiliser "sort!" sur un vecteur d'arêtes.
5. Dans la fonction Kruskal, il y avait deux vecteurs vides dans lesquels on ajoutait des éléments avec des boucles. Ces deux vecteur sont maintenant initialisés avec une longueur fixe.
6. Le test unitaire qui vérifiait la longueur de l'arbre de coût minimal (maintenant de longeur fixe) de Kruskal vérifie maintenant qu'aucun élément "nothing" se trouve dans l'arbre.
7. Dans tous les fichiers où c'est possible, la synthaxe "variable.attribut" a été remplacée par la synthaxe "attribut(variable)".
8. La fonction Kruskal affiche maintenant le coût total de l'arbre de recouvrement trouvé.

Autre modifications

9. Tout ce qui attrait à la structure "ConnexComp" a été déplacé dans un nouveau fichier "connexcomp.jl".
10. Nous avons ajouté un champ "name" à la structure "ConnexComp" et implémenté quelques fonctions qui facilitent son utilisation.
11. La fonction "test*kruskal" comprend un nouvel argument "kruskal*\func" qui permet de tester différentes implémentations de l'algorithme de Kruskal.

3. Implémentation de l'union via le rang et tests unitaires

Nous avons d'abord ajouté à la structure "Node" les attributs "parent" et "rank". Nous avons ensuite écrit une fonction unionRang!() qui prend deux variables de type "ConnexComp" et qui modifie la première de façon à ce qu'elle contienne l'union des deux. Pour accélérer l'exécution de recherches futures, la fonction effectue une compression des chemins avant de se terminer. La fonction ne renvoie rien car elle agit directement sur les sommets des composantes connexes. Dans le fichier "mainphase3.jl" nous avons créé plusieurs composantes connexes simples sur lesquelles faire des tests. Nous avons également créé une fonction "test_unionRang" qui vérifie que la racine après l'union correspond à l'une des deux racines des arbres unis, que le rang de la racine est 1+ le rang de la racine de l'un des deux arbres, et finalement que la racine ne possède pas de parent.

```
Union de compEx1 et compEx2 ✓
Union de compEx1 et compEx3 ✓
Union de compEx2 et compEx3 ✓
```

```
begin
    with_terminal() do
        test_unionRang(deepcopy(compEx1), deepcopy(compEx2))
        test_unionRang(deepcopy(compEx1), deepcopy(compEx3))
        test_unionRang(deepcopy(compEx2), deepcopy(compEx3))
    end
end
#hidden line
```

Preuve que rang(sommet) ≤ |S|-1

Soit un arbre de $|S|$ sommets. Le rang le plus élevé de ces sommets est celui de la racine r. L'arrangement de ces sommets pour lequel le rang de la racine est le plus grand est celui où tout sommet n'a qu'un enfant, sauf la feuille. Dans ce cas, rang(r) = $|S|$ - 1. Il en suit que tout sommet a un rang ≤ $|S|$ - 1.

Preuve que rang(sommet) ≤ $\lfloor \log_2(|S|) \rfloor$ pour la procédure d'union par le rang

Par récurrence sur le rang de la racine n (qui est la hauteur). On va montrer que $2^n \leq |S_n|$, où $|S_n|$ peut être n'importe quel arbre de rang n créé avec la procédure d'union par le rang.

Pour n = 0, l'ensemble des sommets de l'arbre est un singleton, et $2^0 = 1$ donc la propriété est vérifiée au rang 0.

Au rang n ≥ 0, on suppose la propriété vraie. Si le rang de la racine vaut n + 1 pour le graphe $|S_{n+1}|$, on sait que dans la procédure le graphe a été créé par l'union de deux graphes de rang au plus n. On a alors $|S_{n+1}| \geq 2\ |S_n| \geq 2^{n+1}$ par hypothèse de récurrence.

Donc la propriété est vraie pour tout n ≥ 0.

On prend le log2 de la propriété au rang n: $n \leq \log_2(|S_n|)$.

Mais comme n est entier, on peut écrire: $n \leq \lfloor \log_2(|S_n|) \rfloor$.

4. Implémentation de la compression des chemins

Le fonction "compressionChemins!()" prend en argument une composante connexe et un sommet à partir duquel effectuer la compression. Comme cette fonction sera utilisée par la fonction unionRang!() qui trouve déjà la racine de la composante connexe, nous la passons également en argument. La fonction "compressionChemins!()" n'a alors pas besoin de trouver la racine et est plus rapide. La fonction ne renvoie rien car elle agit directement sur les sommets de la composante connexe. Dans le fichier "mainphase3.jl" nous avons créé une composante connexe sur lesquelle faire des tests. Nous avons également créé une fonction "test_unionRang()" qui prend en argument une composante connexe, le sommet sur lequel lancer la recherche, et un vecteur solution. La solution permet de vérifier que chaque sommet comporte exactement le bon parent après la compression.

```
compEx4 ✓
```

```
begin
    with_terminal() do
        test_compression(compEx4, solution_parents_compEx4, n6)
    end
end
#hidden line
```

5. Amélioration de l'implémentation de l'algorithme de Kurskal avec l'union par le rang

Nous avons créé une nouvelle fonction se nommant "kruskal_acc()". Il s'agit d'un copier coller de la fonction de la phase 2, à l'exception de l'union des composantes connexes qui est effectuée par la fonciton "unionRang()", qui elle à son tour utilise la fonction "compressionChemins()". Un simple test qui consiste à ajouter un compteur dans la fonction "unionRang()" a permis de montrer que le nombre total d'itérations est diminué sur tous les graphes des fichiers ".tsp" lorsque la compression des chemins est effectuée.

6. Tests sur la fonction "kruskal_acc()"

La fonction est d'abord testée sur le graphe en exemple dans les notes de cours. Les tests sont les mêmes que ceux de la phase 2. On trouve une solution optimale du problème. La fonction est ensuite testée sur toutes les instances ".tsp". On observe entre autres que bayg29.tsp a un cout de 1319, ce qui est optimal au vu des commentaires sur notre dernier rapport.

```
Edge g↔h, data: (g, h), weight: 1
Edge f↔g, data: (f, g), weight: 2
Edge c↔i, data: (c, i), weight: 2
Edge a↔b, data: (a, b), weight: 4
Edge c↔f, data: (c, f), weight: 4
Edge c↔d, data: (c, d), weight: 7
Edge b↔c, data: (b, c), weight: 8
Edge d↔e, data: (d, e), weight: 9
G exemple du cours ✓
```

```
begin
    with_terminal() do
        arbrecoutmin = kruskal(Gexcours)
        for e in arbrecoutmin
            show(e)
        end
        @test nothing ∉ arbrecoutmin
        @test sommeweights(arbrecoutmin) == 37
        println("G exemple du cours ✓")
    end
end
#hidden line
```

```
bayg29.tsp ✓    avec cout total: 1319
bays29.tsp ✓    avec cout total: 1557
brazil58.tsp ✓  avec cout total: 17514
brg180.tsp ✓    avec cout total: 1920
dantzig42.tsp ✓ avec cout total: 591
fri26.tsp ✓ avec cout total: 741
gr120.tsp ✓ avec cout total: 5805
gr17.tsp ✓  avec cout total: 1421
gr21.tsp ✓  avec cout total: 2161
gr24.tsp ✓  avec cout total: 1011
gr48.tsp ✓  avec cout total: 4082
hk48.tsp ✓  avec cout total: 9905
pa561.tsp ✓ avec cout total: 2396
swiss42.tsp ✓   avec cout total: 1079
```

```
begin
    with_terminal() do
        test_kruskal(raw"./instances/stsp", kruskal_acc)
    end
end
#hidden line
```

7. Implémentation de l'algorithme de Prim et tests unitaires

Nous avons d'abord ajouté l'attribut "min_weight" à la structure "Node". Ensuite, nous avons créé un nouveau fichier "nodequeue.jl" qui contient les types "AbstractQueue" et "NodeQueue" ainsi que plusieurs fonctions qui agissent sur ces types. Entre autres, la fonction "popfirst!()" retire et renvoie l'élément dont la valeur "min_weight" est la plus petite dans une "NodeQueue".

Avec ces outils, l'algorithme a été implémenté tel qu'il est suggéré de le faire dans l'énoncé de la phase 3. Alors l'algorithme prend en entrée un graph et un sommet de départ et n'a pas besoin de renvoyer quoi que ce soit. Chaque sommet sauf celui de départ aura un parent dans son attribut "parent" et un cout dans sont attribut "min_weight". Ces coûts et parents correspondent à l'arbre de recouvrement de coût minimal.

Dans la cellule suivante, on teste l'algorithme sur le graphe en exemple dans le cours avec la fonction "test_prim()". Les sommets contenant l'information sur l'arbre trouvé sont affichés et cette solution est la même que celle des notes de cours. Deux tests unitaires sont ensuite effectués. Premièrement, on vérifie que tous les parents sont les bons en comparant un vecteur contenant la solution aux deux solutions possibles du graphe. Deuxièmement, on teste l'optimalité en vérifiant que la somme des coûts est de 37, comme dans les notes de cours.

```
G exemple du cours ✓
Node a, coordonnées: [0, 0] rang: 0 min_weight: 0.0 parent: nothing
Node b, coordonnées: [0, 0] rang: 0 min_weight: 4.0 parent: a
Node c, coordonnées: [0, 0] rang: 0 min_weight: 8.0 parent: b
Node d, coordonnées: [0, 0] rang: 0 min_weight: 7.0 parent: c
Node e, coordonnées: [0, 0] rang: 0 min_weight: 9.0 parent: d
Node f, coordonnées: [0, 0] rang: 0 min_weight: 4.0 parent: c
Node g, coordonnées: [0, 0] rang: 0 min_weight: 2.0 parent: f
Node h, coordonnées: [0, 0] rang: 0 min_weight: 1.0 parent: g
Node i, coordonnées: [0, 0] rang: 0 min_weight: 2.0 parent: c
```

```
begin
    with_terminal() do
        test_prim(Gexcours, a)
        for node in nodes(Gexcours)
            show(node)
        end
    end
end
#hidden line
```

8. Test de "prim!()" sur toutes les instances

La fonction "test_prim_all()" performe l'algorithme sur tous les fichiers ".tsp". On observe entre autres que le coût total de bayg29.tsp est de 1319, ce qui est optimal au vu des commentaires sur notre dernier rapport.

```
bayg29.tsp✓ cout min: 1319.0
bays29.tsp✓ cout min: 1557.0
brazil58.tsp✓   cout min: 17514.0
brg180.tsp✓ cout min: 1920.0
dantzig42.tsp✓  cout min: 591.0
fri26.tsp✓  cout min: 741.0
gr120.tsp✓  cout min: 5805.0
gr17.tsp✓   cout min: 1421.0
gr21.tsp✓   cout min: 2161.0
gr24.tsp✓   cout min: 1011.0
gr48.tsp✓   cout min: 4082.0
hk48.tsp✓   cout min: 9905.0
pa561.tsp✓  cout min: 2396.0
swiss42.tsp✓    cout min: 1079.0
```

```
begin
    with_terminal() do
        test_prim_all(raw"./instances/stsp")
    end
end
#hidden line
```

9. Comparaison des temps d'exécution

On affiche dans un tableau les résultats des temps de calculs des différentes fonctions implémentées jusqu'à maintenant.

On constate que l'algorithme de Prim est le plus lent. Kruskal accéléré est un peu plus rapide que Kruskal.

Ce qui ralentit notre algorithme de Prim est probablement la fonction "isinfile" qui parcourt beaucoup de noeuds à chaque itération. Une solution possible serait de créer un nouveau type de noeud qui contient un champ permettant de savoir si le noeud en question est dans une "NodeQueue" ou non. La comparaison des algorithmes et l'obtention de performances optimales n'étant pas demandée dans cette phase du projet, nous avons fait le choix de ne pas implémenter cette modification et d'attendre les instructions futures pour avoir une vue d'ensemble de la manière dont seront utilisés nos algorithmes. Nous pourrons alors implémenter les éventuelles modifications nécessaires à ce moment-là.

Comparaison des performances entre les algorithmes implémentés

	prim!	kruskal	kruskal_acc
bayg29	7.47e-04	5.58e-05	5.41e-05
bays29	6.65e-04	5.48e-05	5.29e-05
brazil58	4.35e-03	2.77e-04	2.76e-04
brg180	1.15e-01	3.21e-03	3.40e-03
dantzig42	1.73e-03	1.22e-04	1.20e-04
fri26	5.39e-04	4.15e-05	3.96e-05
gr120	3.51e-02	1.69e-03	1.53e-03
gr17	3.21e-04	2.05e-05	1.81e-05
gr21	3.66e-04	2.88e-05	2.63e-05
gr24	4.75e-04	3.51e-05	3.38e-05
gr48	3.00e-03	1.71e-04	1.66e-04
hk48	2.47e-03	1.74e-04	1.63e-04
pa561	4.63e+00	9.18e-02	8.44e-02
swiss42	1.72e-03	1.33e-04	1.27e-04

```
begin
    with_terminal() do
        benchmark_table_KruskalPrim(raw"./instances/stsp")
    end
end
#hidden line
```

9. Branche phase3 sur GitHub

Lien: https://github.com/XavierLebeuf/mth6412b-starter-code/tree/phase3