Rapport du projet d'Implémentation d'algo. de rech. opérationnelle

Phase 3 : 02/11/2020

Antonin Kenens, Vincent Perreault et Laura Kolcheva Dépôt github à l'adresse suivante : https://github.com/VincentPerreault0/mth6412b-starter-code.git

Question 1 : Implémenter les deux heuristiques d'accélération et répondre à la question sur le rang.

Nous avons réutilisé nos structures de données précédentes pour les noeuds et les liens. Nous avons modifié la structure de donnée des noeuds pour prendre en compte le parent de chaque noeud. Les composantes connexes sont donc des ensembles de noeuds qui ont des liens de parenté. La spécificité est que la racine d'une composant connexe n'a pas de parent. Nous avons fait ce choix pour éviter les références circulaires.

Nous avons aussi rajouté l'attribut rank à tous les noeuds pour pouvoir mettre en place l'heuristique de compression des chemins. L'attribut minweight est utilie pour la question 2.

Nous avons ensuite étendu les méthodes de notre implémentation de Node pour prendre en compte les nouveaux attributs de Node.

`Main.workspace53.minweight`

```
. begin
.     """Type abstrait dont d'autres types de noeuds dériveront."""
.     abstract type AbstractNode{T} end
.
.     """Type représentant les noeuds d'un graphe."""
.     mutable struct Node{T} <: AbstractNode{T}
.         name::String
.         data::T
.         rank::Int64
.         parent ::Union{Nothing,Node{T}}
.         minweight::Int64
.     end
.
.     """Type représentant les arêtes d'un graphe."""
.     struct Edge
.       weight::Float64
.       nodes::Tuple{AbstractNode,AbstractNode}
.     end
.
.     """initialise un noeud uniquement avec un nom et un data"""
.     function Node(name:: String, data)
.       return(Node{typeof(data)}(name, data, 0, nothing, 10000))
.     end
.
.     function Node(name:: String, data, rank :: Int64)
.      return(Node{typeof(data)}(name, data, rank, nothing, 10000))
.     end
.
.     function Node(name:: String, data, parent :: Node)
```

```
        return(Node{typeof(data)}(name, data, 0, parent, 10000))
    end

    # on présume que tous les noeuds dérivant d'AbstractNode
    # posséderont des champs `name`,`data` 'rank, 'parent' et 'minweight'.

    """Renvoie le rank du noeud."""
    rank(node::AbstractNode) = node.rank

    """Renvoie le parent du noeud."""
    parent(node::AbstractNode) = node.parent

    """ Renvoie minweight du noeud"""
    function minweight(node::AbstractNode)
        node.minweight
    end
end
```

Nous introduisons maintenant l'heuristique de la compression des chemins. Ceci est une recherche de la racine. Cette recherche s'effectue en même temps que l'actualisation de la racine des éléments le long du chemin vers la racine de la composante connexe.

Main.workspace53.find_root

```
begin
    """ Trouve la racine d une composante connexe a partir d'un noeud
    Actualise la racine de tous les noeuds sur le chemin """
    function find_root(node :: Node{T}, nodes=nothing) where T
      if nodes===nothing
        nodes=Node{typeof(node.data)}[]
      end
      if parent(node)===nothing #ce noeud est une racine
        for nodetmp in nodes
          nodetmp.parent=node
        end
        node.parent=nothing #this node was in the array
        return(node)
      else
        push!(nodes, parent(node))
        find_root(parent(node), nodes)
      end
    end
end
```

Nous introduisons maintenant l'heuristique de l'union via le rang. Elle prend en entree les racines de deux composantes connexes trouvées par 'find_root'. Si les racines sont égales, on ne fait rien. Si les racines sont différentes, on fait une union via le rang des racines.

Main.workspace58.union_roots

```
begin
    """fonction qui prend en entree les racines de deux composantes connexes et les unis si elles
sont différentes"""
    function union_roots(root1::Node{T}, root2::Node{T}) where T
      if root1==root2
        return
```

```
      else
        if rank(root1)<rank(root2)
          root1.parent=root2
        elseif rank(root2)<rank(root1)
            root2.parent=root1
        else
            root2.parent=root1
            root1.rank+=1
        end
      end
      return
    end
end
```

Nous rappelons ici les méthodes d'affichage pour les différents types.

`Base.show`

```
begin
    import Base.show
    """Affiche un noeud."""
    function show(node::AbstractNode)
      println("Node ", name(node), ", data: ", data(node), " rank: ", rank(node), " parent: ", parent(node))
    end

    """Affiche un graphe"""
    function show(graph::AbstractGraph)
      println("Graph ", name(graph), " has ", nb_nodes(graph), " nodes and ", nb_edges(graph), " edges.")
      for node in nodes(graph)
        show(node)
      end
      for edge in edges(graph)
        show(edge)
      end
    end

    """Affiche une arête."""
    function show(edge::Edge)
      println("Edge weight : ", string(weight(edge)))
      for node in nodes(edge)
        print("  ")
        show(node)
      end
    end
end
```

Nous cachons ci-dessous la majorité des méthodes pour les graphes.

`Main.workspace53.create_graph_from_stsp_file`

Les méthodes add_ node et add_ edge ont été modifiées pour prendre en compte si un noeud/edge appartient déjà à un graphe avant de l'ajouter au graphe.

`Main.workspace53.add_edge!`

```
begin
```

```
    """Ajoute un noeud au graphe."""
    function add_node!(graph:: AbstractGraph{T}, node::AbstractNode{T}) where T
      if contains_node(graph,node)
        return(graph)
      else
        push!(graph.nodes, node)
        return(graph)
      end
    end

    """Ajoute une arête au graphe."""
    function add_edge!(graph::AbstractGraph{T}, edge::Edge) where T
      if contains_edge(graph, edge)
        return(graph)
      else
        push!(graph.edges, edge)
        # Si les noeuds du lien ne font pas partie du graphe, les rajouter
        for node in edge.nodes
          if !contains_node(graph, node)
            add_node!(graph, node)
          end
        end
        return(graph)
      end
    end
end
```

Nous pouvons finalement donner notre implémentation de l'algorithme de Kruskal avec les deux nouvelles heuristiques.

```
new_min_span_tree (generic function with 1 method)
```

```
begin
    function new_min_span_tree(graph :: AbstractGraph{T}, verbose:: Bool) where T
        #liste de liens dans le minimum spanning tree
        new_edges=Vector{Edge}()

        # Ordonnement des liens par poids croissants
        graph_edges = edges(graph)
        sort!(graph_edges, by=e -> e.weight)

        # Pour chaque lien,
        for edge in graph_edges
            if verbose
                println("Searching...")
                show(edge)
            end
            node1=nodes(edge)[1]
            node2=nodes(edge)[2]
            root1=find_root(node1)
            root2=find_root(node2)
            #connection de deux composantes
            if root1!=root2
                if verbose
                    println("merging components...")
                end
                #comparaison du rang et actualisation du pointeur vers la racine
                union_roots(root1,root2)

                #ajout de l arrete a l arbre
                push!(new_edges, edge)
            elseif verbose
                println("edge in only one component. On to next edge")
            end
        end
        #construction d arbre
        return( Graph("New minimal spanning tree of " * name(graph), nodes(graph), new_edges))
    end
```

```
end
```

Question 2 :implémenter l'algorithme de Prim

Nous avons implémenté le type PriorityQueue pour cet effet.

`Main.workspace64.add_item!`

```
begin
    """ type abstrait de file de priorite"""
    abstract type AbstractQueue{T} end

    """File de priorité (utilise pour les noeuds ou les edges)"""
    mutable struct PriorityQueue{T} <: AbstractQueue{T}
        items::Vector{T}
    end

    """ Cree une file de priorite vide"""
    function PriorityQueue{T}() where T
        PriorityQueue(T[])
    end

    """Indique si la file est vide."""
    function is_empty(q::AbstractQueue)
        length(q.items) == 0
    end

    """ Renvoie les elements de la file"""
    function items(q::AbstractQueue)
        q.items
    end

    """Donne le nombre d'éléments sur la file."""
    function nb_items(q::AbstractQueue)
        return(length(q.items))
    end

    """renvoie true si la file contien l objet"""
    function contains_item(q::PriorityQueue{T}, item::T) where T
        return(item in q.items)
    end

    """Ajoute `item` à la fin de la file."""
    function add_item!(q::AbstractQueue{T}, item::T) where T
        if contains_item(q,item)
            return(q)
        end
        push!(q.items, item)
        return(q)
    end


end
```

Nous avons fait le choid de ne pas implémenter les fonctions isless et == pour les objets des PriorityQueue. En effet, il s'agit des type Node et Edge que l'on veut comparer par l'attribut 'minweight' (pour Node) et l'attribut 'weight'(pour Edge). Si on implémente isless et == pour ces deux types avec ces atrtributs, cela change aussi le fonctionnement de toutes les fonctions contenant des expressions du type 'node in nodes' ou 'edge in edges'. Ce n'est pas un résultat voulu.

Nous avons tout de même implémenter des fonctions permettant de trouver le minimum d'une file, et deux fonctions popfirst! pertinentes dans ce contexte.
La fonction popfirst(q,type) renvoie: si type==Node, un noeud de minweight minimal, si type==Edge, un edge de weight minimal.
La fonction popfirst!(q,node) est spécifique. Dans une file on peut avoir plusieurs élements de type Edge ayant le même poids. On veut retirer le Edge correspondant à un noeud avec un minweight minimal. Il serait logique que ce Edge soit aussi de poids minimal. Mais si le Edge de poids minimal ne correspond pas au noeud de minweight minimal, c'est qu'il relie deux noeud qui sont déjà dans le minimum spanning tree. Cela veut dire que ce Edge ne sera jamais utilisé. On l'enlève donc de la file.

`Base.popfirst!`

```
begin
    import Base.popfirst!
    """renvoie le plus petit element de la file"""
    function minimum_item(q::AbstractQueue, type ::Type )
        min=items(q)[1]
        if type==Node
            for node in q.items[2:end]
                if minweight(node)<minweight(min)
                    min=node
                end
            end
        elseif type==Edge
            for edge in q.items[2:end]
                if weight(edge)<weight(min)
                    min=edge
                end
            end
        end
        return(min)
    end

    """Retire et renvoie un élément ayant la plus haute priorité.
    Ici la priorite est l ordre decroissant. """
    function popfirst!(q::PriorityQueue, type::Type)
        highest = minimum_item(q,type)
        idx = findall(x -> x == highest, q.items)[1]
        deleteat!(q.items, idx)
        return(highest)
    end

    """Retire et renvoie l'élément ayant la plus haute priorité
     et qui contient le noeud choisit et son parent.
     Si il y a plusieurs edge de meme poids, on choisit celui correspondant
     au noeud choisit. Si aucum edge de poids minimal ne correspond au noeud, on supprime les edge de
poids minimal et on reitere"""
    function popfirst!(q:: PriorityQueue{Edge}, node:: Node)
        min= minimum_item(q,Edge)
        idx = findall(x -> weight(x) == weight(min), q.items)
        tmp=-1
        for i in idx
            if node in nodes(items(q)[i]) && parent(node) in nodes(items(q)[i])
                tmp=i
                break
            end
        end
        if tmp==-1# si on est la c'est que les edge de poids min ne vont jamais etre utilise
            compteur=0
            for i in idx
                deleteat!(q.items, i-compteur)
                compteur=compteur+1
            end
```

```
            return(popfirst!(q, node))#on refait sur la file updatee
        else
            edge=items(q)[tmp]
            deleteat!(q.items, tmp)
            return(edge)
        end
    end
end
```

On peut maintenant montrer l'algorithme de Prim implémenté avec le type PriorityQueue

Main.workspace76.prim

```
begin
    """ fonction qui prend un graphe et une source et renvoie
    un minimal spanning tree par l algorithme de Prim"""
    function prim(graph :: AbstractGraph, s :: AbstractNode)
        if (s in nodes(graph))==false
            return
        end
        #initialisation des listes
        new_edges=Edge[]
        s.minweight=0
        q=PriorityQueue{Node}()
        p=PriorityQueue{Edge}()

        #initialisation des dictionnaires contenant les listes d adjacences des noeuds
        #et les arretes incidentes pour chaque noeud
        dict_edges=Dict{Node, Vector{Edge}}()
        dict_nodes=Dict{Node, Vector{Node}}()
        for node in nodes(graph)
            dict_edges[node]=Edge[]
            dict_nodes[node]=Node{typeof(node)}[]
            add_item!(q,node)
        end

        # Calcul des listes d adjacence
        for edge in edges(graph)
            node1=nodes(edge)[1]
            node2=nodes(edge)[2]
            push!(dict_nodes[node1],node2)
            push!(dict_nodes[node2],node1)
            push!(dict_edges[node1],edge)
            push!(dict_edges[node2],edge)
        end

        #Boucle
        while nb_items(q)>0
            #on sort un noeud de minweight minimal
            u=popfirst!(q, Node)
            #on ajoute l arrete correspondante aux arretes du minimum spanning tree
            if (u==s)==false #si u ==s on est a la premiere iteration et p est vide
                tmp=popfirst!(p,u)
                push!(new_edges,tmp)
            end
            #on actualise les valeurs des noeuds voisins
            for v in dict_nodes[u]
                if contains_item(q,v)==true
                    for edge in dict_edges[v]
```

```
                        if u in nodes(edge) && v in nodes(edge) && weight(edge)<minweight(v)
                            v.parent=u
                            v.minweight=weight(edge)
                            add_item!(p,edge)
                        end
                    end
                end
            end
        end
        g=Graph("Minimum Spanning tree from Prim alg of "*name(graph), nodes(graph), new_edges)
        return(g)
    end
end
```

Le test sur l'exemple vu en cours sera fait dans la prochaine section.

Question 3 : tester votre implémentation sur l'exemple des notes de cours et diverses instances de TSP symétrique dans vos tests unitaires.

Pour tester les méthodes de Node, PriorityQueue, ainsi que la nouvelle implémentation de l'algorithme de Kruskal et l'implémentation de l'algorithme de Prim, nous avons implémenté la série de tests unitaires ci-dessous.

Nous avons implémenté la méthode 'total_weight' pour pouvoir calculer le le poids total des arrêtes d'un graphe

`Main.workspace90.total_weight`

```
begin
    """ donne la somme des poids des arretes d un graphe"""
    function total_weight(graph:: AbstractGraph)
      if nb_nodes(graph)==0
        return(0)
      else
        s=0
        for edge in graph.edges
          s+=weight(edge)
        end
        return(s)
      end
    end
end
```

Note : Nous n'arrivons pas à faire fonctionner la macro '@test' dans le carnet Pluto, mais tout fonctionne sans problème dans VS Code.

LoadError: UndefVarError: @test not defined

in expression starting at C:\Users\lora\Desktop\mth6412b-starter-code\rapport_phase3.jl#==#19deffd0-1b84-11eb-2b11-f98efe6d0313:15

1. **top-level scope** @ *:0*

```
. begin
.     using Test
.     include("../phase2/graph.jl")
.     include("new_min_span_tree.jl")
.     include("prim.jl")
.
.
.     #Tests pour nouvelles fonctions de Node
.     println("Testing Node methods...")
.     node1=Node("Joe", 1,5,nothing,6)
.     node2=Node("James",3)
.     node3=Node("Matt", 6, 2)
.     node4=Node("Rebeca",9, node3)
.
.     @test name(node1)=="Joe"
.     @test data(node1)==1
.     @test rank(node1)==5
.     @test parent(node1)===nothing
.     @test minweight(node1)==6
.
.     @test data(node2)==3
.     @test rank(node2)==0
.     @test parent(node2)===nothing
.     @test minweight(node2)==10000
.
.     @test data(node3)==6
.     @test rank(node3)==2
.     @test parent(node2)===nothing
.     @test minweight(node2)==10000
.
.     @test parent(node4)===node3
.     @test minweight(node4)==10000
.
.     node3.parent=node2
.     node2.parent=node1
.     @test find_root(node4, nothing)==node1
.     @test node2.parent==node1
.     @test node3.parent==node1
.
.     #Test pour nouvelles fonctions de Graph
.     g=create_empty_graph("BigG", Int)
.     add_node!(g,node1)
.     add_node!(g,node1)
.
.     @test nb_nodes(g)==1
.     @test contains_node(g, node1)==true
.     @test total_weight(g)==0
.
.     edge1=Edge(1000.0, (node1,node2))
.     add_edge!(g, edge1)
.
.     @test nb_nodes(g)==2
.     @test nb_edges(g)==1
.     @test total_weight(g)==1000.0
.
.     #Test pour New Min Span Tree
.     println("Testing New Minimum Spanning Tree Kruskal Algorithm with range and depth...")
.     # Exemple vu en cours
.     nodeA = Node("a", nothing)
.     nodeB = Node("b", nothing)
.     nodeC = Node("c", nothing)
.     nodeD = Node("d", nothing)
.     nodeE = Node("e", nothing)
.     nodeF = Node("f", nothing)
.     nodeG = Node("g", nothing)
.     nodeH = Node("h", nothing)
.     nodeI = Node("i", nothing)
.
```

```
    edge1 = Edge(4,(nodeA,nodeB))
    edge2 = Edge(8,(nodeB,nodeC))
    edge3 = Edge(7,(nodeC,nodeD))
    edge4 = Edge(9,(nodeD,nodeE))
    edge5 = Edge(14,(nodeD,nodeF))
    edge6 = Edge(4,(nodeC,nodeF))
    edge7 = Edge(2,(nodeC,nodeI))
    edge8 = Edge(11,(nodeB,nodeH))
    edge9 = Edge(8,(nodeA,nodeH))
    edge10 = Edge(7,(nodeH,nodeI))
    edge11 = Edge(1,(nodeG,nodeH))
    edge12 = Edge(6,(nodeG,nodeI))
    edge13 = Edge(2,(nodeF,nodeG))
    edge14 = Edge(10,(nodeE,nodeF))

    g3 = Graph("Class Example", [nodeA, nodeB, nodeC, nodeD, nodeE, nodeF, nodeG, nodeH, nodeI], [edge1, edge2, edge3, edge4, edge5, edge6, edge7, edge8, edge9, edge10, edge11, edge12, edge13, edge14])
    #show(g3)
    mst = new_min_span_tree(g3, false)
    #show(mst)

    @test nb_nodes(mst) == nb_nodes(g3)
    @test nb_edges(mst) == 8
    @test contains_edge(mst,edge1) == true
    @test contains_edge(mst,edge2) == true || contains_edge(mst,edge9) == true  # ces 2 liens ont le même poids dans le graphe et, selon l'ordre utilisé dans sa construction explicite, l'algorithme de Kruskal va finir par en utiliser un et un seul pour son arbre de recouvrement minimal
    @test contains_edge(mst,edge3) == true
    @test contains_edge(mst,edge4) == true
    @test contains_edge(mst,edge6) == true
    @test contains_edge(mst,edge7) == true
    @test contains_edge(mst,edge11) == true
    @test contains_edge(mst,edge13) == true

    #Tests pour PriorityQueue
    println("Testing PriorityQueue methods...")
    node1=Node("Joe", 1,5,nothing,6)
    node1.minweight=2
    node2=Node("James",3)
    node2.minweight=1
    node3=Node("Matt", 6, 2)
    node4=Node("Rebeca",9, node2)
    node4.minweight=1
    edge1=Edge(100, (node1,node2))

    q=PriorityQueue{Node}()

    @test is_empty(q) == true
    @test items(q)==[]
    @test nb_items(q)==0
    @test contains_item(q,node1) == false

    add_item!(q, node1)

    @test nb_items(q) == 1
    @test is_empty(q) == false
    @test items(q)==[node1]
    @test contains_item(q,node1) == true

    add_item!(q,node2)
    add_item!(q,node2)

    @test nb_items(q)==2
    @test minimum_item(q, Node)==node2

    tmp=popfirst!(q, Node)

    @test tmp==node2
    @test nb_items(q)==1
    @test contains_item(q,node2) == false
    @test contains_item(q,node1) == true

    #Test pour PriorityQueue avec Edge comme item
```

```
    edge1=Edge(100, (node1,node2))
    edge2=Edge(100, (node2,node3))
    edge3=Edge(1, (node3, node1))
    node3.parent=node2
    p=PriorityQueue{Edge}()

    @test is_empty(p) == true
    @test items(p)==[]
    @test nb_items(p)==0
    @test contains_item(p,edge1) == false

    add_item!(p,edge1)
    add_item!(p,edge1)

    @test is_empty(p) == false
    @test items(p)==[edge1]
    @test nb_items(p)==1
    @test contains_item(p,edge1) == true

    add_item!(p, edge2)

    @test popfirst!(p,Edge)==edge1

    add_item!(p,edge1)

    @test popfirst!(p, node3)==edge2

    add_item!(p,edge3)

    @test minimum_item(p,Edge)==edge3

    #Tests pour algorithme de Prim
    println("Testing Prim's algorithm...")
    # Exemple vu en cours
    nodeA = Node("a", nothing)
    nodeB = Node("b", nothing)
    nodeC = Node("c", nothing)
    nodeD = Node("d", nothing)
    nodeE = Node("e", nothing)
    nodeF = Node("f", nothing)
    nodeG = Node("g", nothing)
    nodeH = Node("h", nothing)
    nodeI = Node("i", nothing)

    edge1 = Edge(4,(nodeA,nodeB))
    edge2 = Edge(8,(nodeB,nodeC))
    edge3 = Edge(7,(nodeC,nodeD))
    edge4 = Edge(9,(nodeD,nodeE))
    edge5 = Edge(14,(nodeD,nodeF))
    edge6 = Edge(4,(nodeC,nodeF))
    edge7 = Edge(2,(nodeC,nodeI))
    edge8 = Edge(11,(nodeB,nodeH))
    edge9 = Edge(8,(nodeA,nodeH))
    edge10 = Edge(7,(nodeH,nodeI))
    edge11 = Edge(1,(nodeG,nodeH))
    edge12 = Edge(6,(nodeG,nodeI))
    edge13 = Edge(2,(nodeF,nodeG))
    edge14 = Edge(10,(nodeE,nodeF))

    g3 = Graph("Class Example", [nodeA, nodeB, nodeC, nodeD, nodeE, nodeF, nodeG, nodeH, nodeI],
[edge1, edge2, edge3, edge4, edge5, edge6, edge7, edge8, edge9, edge10, edge11, edge12, edge13,
edge14])
    #show(g3)
    mst_prim = prim(g3, nodeA)
    #show(mst_prim)

    @test name(mst_prim)=="Minimum Spanning tree from Prim alg of Class Example"
    @test nb_nodes(mst_prim) == nb_nodes(g3)
    @test nb_edges(mst_prim) == 8
    @test contains_edge(mst_prim,edge1) == true
    @test contains_edge(mst_prim,edge2) == true || contains_edge(mst_prim,edge9) == true  # ces 2
liens ont le même poids dans le graphe et, selon l'ordre utilisé dans sa construction explicite,
l'algorithme de Kruskal va finir par en utiliser un et un seul pour son arbre de recouvrement minimal
    @test contains_edge(mst_prim,edge3) == true
```

```
    @test contains_edge(mst_prim,edge4) == true
    @test contains_edge(mst_prim,edge6) == true
    @test contains_edge(mst_prim,edge7) == true
    @test contains_edge(mst_prim,edge11) == true
    @test contains_edge(mst_prim,edge13) == true

    println("Testing with stsp instances...")
    filenames = ["instances/stsp/bayg29.tsp",
                 "instances/stsp/bays29.tsp",
                 "instances/stsp/brazil58.tsp",
                 "instances/stsp/brg180.tsp",
                 "instances/stsp/dantzig42.tsp",
                 "instances/stsp/fri26.tsp",
                 "instances/stsp/gr17.tsp",
                 "instances/stsp/gr21.tsp",
                 "instances/stsp/gr24.tsp",
                 "instances/stsp/gr48.tsp",
                 "instances/stsp/gr120.tsp",
                 "instances/stsp/hk48.tsp",
                 "instances/stsp/pa561.tsp",
                 "instances/stsp/swiss42.tsp"]

    for i=1:length(filenames)
      graph = create_graph_from_stsp_file(filenames[i], false)
      println("Graph ", name(graph), " has ", nb_nodes(graph), " nodes and ", nb_edges(graph), "
edges.")
      mst1 = new_min_span_tree(graph, false)
      println(name(mst1), " has ", nb_nodes(mst1), " nodes and ", nb_edges(mst1), " edges and weight
",total_weight(mst1))
      mst2=prim(graph,nodes(graph)[10])
      println(name(mst2), " has ", nb_nodes(mst2), " nodes and ", nb_edges(mst2), " edges and weight
", total_weight(mst2))
      println()
    end
    println("All tests complete")
end
```

Avec cete série de tests, nous obtenons la sortie suivante

Testing Node methods...
Testing New Minimum Spanning Tree Kruskal Algorithm with range and depth...
Testing PriorityQueue methods...
Testing Prim's algorithm...
Testing with stsp instances...

Graph bayg29 has 29 nodes and 406 edges. New minimal spanning tree of bayg29 has 29 nodes and 28 edges and weight 1319.0 Minimum Spanning tree from Prim alg of bayg29 has 29 nodes and 28 edges and weight 1319.0

Graph bays29 has 29 nodes and 841 edges. New minimal spanning tree of bays29 has 29 nodes and 28 edges and weight 1557.0 Minimum Spanning tree from Prim alg of bays29 has 29 nodes and 28 edges and weight 1557.0

Graph brazil58 has 58 nodes and 1653 edges. New minimal spanning tree of brazil58 has 58 nodes and 57 edges and weight 17514.0 Minimum Spanning tree from Prim alg of brazil58 has 58 nodes and 57 edges and weight 17514.0

Graph brg180 has 180 nodes and 16110 edges. New minimal spanning tree of brg180 has 180 nodes and 179 edges and weight 1920.0 Minimum Spanning tree from Prim alg of brg180 has 180 nodes and 179 edges and weight 1920.0

Graph dantzig42 has 42 nodes and 903 edges. New minimal spanning tree of dantzig42 has 42 nodes and 41 edges and weight 591.0 Minimum Spanning tree from Prim alg of dantzig42 has 42 nodes and 41 edges and weight 591.0

Graph fri26 has 26 nodes and 351 edges. New minimal spanning tree of fri26 has 26 nodes and 25 edges and weight 741.0 Minimum Spanning tree from Prim alg of fri26 has 26 nodes and 25 edges and weight 741.0

Graph gr17 has 17 nodes and 153 edges. New minimal spanning tree of gr17 has 17 nodes and 16 edges and weight 1421.0 Minimum Spanning tree from Prim alg of gr17 has 17 nodes and 16 edges and weight 1421.0

Graph gr21 has 21 nodes and 231 edges. New minimal spanning tree of gr21 has 21 nodes and 20 edges and weight 2161.0 Minimum Spanning tree from Prim alg of gr21 has 21 nodes and 20 edges and weight 2161.0

Graph gr24 has 24 nodes and 300 edges. New minimal spanning tree of gr24 has 24 nodes and 23 edges and weight 1011.0 Minimum Spanning tree from Prim alg of gr24 has 24 nodes and 23 edges and weight 1011.0

Graph gr48 has 48 nodes and 1176 edges. New minimal spanning tree of gr48 has 48 nodes and 47 edges and weight 4082.0 Minimum Spanning tree from Prim alg of gr48 has 48 nodes and 47 edges and weight 4082.0

Graph gr120 has 120 nodes and 7260 edges. New minimal spanning tree of gr120 has 120 nodes and 119 edges and weight 5805.0 Minimum Spanning tree from Prim alg of gr120 has 120 nodes and 119 edges and weight 5805.0

Graph hk48 has 48 nodes and 1176 edges. New minimal spanning tree of hk48 has 48 nodes and 47 edges and weight 9905.0 Minimum Spanning tree from Prim alg of hk48 has 48 nodes and 47 edges and weight 9905.0

Graph pa561 has 561 nodes and 157641 edges. New minimal spanning tree of pa561 has 561 nodes and 560 edges and weight 2396.0 Minimum Spanning tree from Prim alg of pa561 has 561 nodes and 560 edges and weight 2396.0

Graph swiss42 has 42 nodes and 1764 edges. New minimal spanning tree of swiss42 has 42 nodes and 41 edges and weight 1079.0 Minimum Spanning tree from Prim alg of swiss42 has 42 nodes and 41 edges and weight 1079.0

All tests complete

Note : Le filepath a dû être ajouté sur le carnet Pluto pour faire fonctionner le code.

Comme on peut le lire, nous obtenons la sortie attendue, c'est-à-dire que nous obenons pour chaque graphe d'entrée un graphe connexe en sortie avec le même nombre de noeud *N* et un nombre de lien égal à *N*-1. Les graphe de sortie étant connexes par construction, nous obtenons ainsi bien des arbres de recouvrement. De plus, les algorithmes de Kruskal et de Prim assurent que ces arbres de recouvrement sont bel et bien minimaux.