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

Phase 2 : 05/10/2020

Antonin Kenens, Vincent Perreault et Laura Kolcheva Dépôt github à l'adresse suivante :
https://github.com/VincentPerreault0/mth6412b-starter-code/tree/phase2

Question 1 : Implémenter une structure de données pour les composantes connexes.

Nous avons réutilisé nos structures de données précédentes pour les noeuds et les liens.

`Main.workspace59.Edge`

```
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
    end

    """Type représentant les arêtes d'un graphe."""
    struct Edge
      weight::Float64
      nodes::Tuple{AbstractNode,AbstractNode}
    end
end
```

Nous avons ensuite étendu les méthodes de notre implémentation de AbstractGraph pour préparer le terrain aux nécessités des composantes connexes.

`Main.workspace62.add_edge!`

```
begin
    """Type abstrait dont d'autres types de noeuds dériveront."""
    abstract type AbstractGraph{T} end

    """Renvoie le nom du graphe."""
    name(graph::AbstractGraph) = graph.name

    """Renvoie la liste des noeuds du graphe."""
    nodes(graph::AbstractGraph) = graph.nodes

    """Renvoie la liste des arêtes du graphe."""
    edges(graph::AbstractGraph) = graph.edges
```

```
    """Renvoie le nombre de noeuds du graphe."""
    nb_nodes(graph::AbstractGraph) = length(graph.nodes)

    """Renvoie le nombre d'arêtes du graphe."""
    nb_edges(graph::AbstractGraph) = length(graph.edges)

    """Vérifie si le graphe contient un certain noeud."""
    contains_node(graph::AbstractGraph{T}, node::Node{T}) where T = node in graph.nodes

    """Ajoute un noeud au graphe."""
    function add_node!(graph::AbstractGraph{T}, node::Node{T}) where T
      push!(graph.nodes, node)
      graph
    end

    """Vérifie si le graphe contient un certain lien."""
    contains_edge(graph::AbstractGraph{T}, edge::Edge) where T = edge in graph.edges

    """Ajoute une arête au graphe."""
    function add_edge!(graph::AbstractGraph{T}, edge::Edge) where T
      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

      graph
    end
end
```

Nous rappelons ici les méthodes d'affichage pour tous ces types.

Base.show

```
begin
    import Base.show

    """Affiche un noeud."""
    function show(node::AbstractNode)
        println("Node ", name(node), ", data: ", data(node))
    end

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

Nous cachons ci-dessous les méthodes (très longues) pour lire les fichiers stsp.

`Main.workspace66.read_stsp`

Nous pouvons maintenant donner les méthodes de la structure concrète d'un graphe.

`Main.workspace69.create_graph_from_stsp_file`

```
begin
    """Structure concrète d'un graphe."""
    mutable struct Graph{T} <: AbstractGraph{T}
      name::String
      nodes::Vector{Node{T}}
      edges::Vector{Edge}
    end

    """Crée un graphe vide."""
    create_empty_graph(graphname::String, type::Type) = Graph{type}(graphname,[],[])

    """Crée un graphe symétrique depuis un ficher lisible par read_stsp."""
    function create_graph_from_stsp_file(filepath::String, verbose::Bool)
      # Utilisation de la fonction read_stsp
      graph_nodes, graph_edges = read_stsp(filepath, verbose)

      # Définition des constantes
      dim_nodes = length(graph_nodes)
      edges = Edge[]
      dim_edges = length(graph_edges)

      # Création des nodes
      if (dim_nodes != 0)
          nodes = Node{typeof(graph_nodes[1])}[]
          for node_ind in 1 : dim_nodes
              node = Node(string(node_ind), graph_nodes[node_ind])
              push!(nodes, node)
          end
      else
          # Si les nodes n'ont pas de data, on leur donne une data nulle : "nothing"
          nodes = Node{Nothing}[]
          for node_ind in 1 : dim_edges
              node = Node(string(node_ind), nothing)
              push!(nodes, node)
          end
      end

      # Création des edges à partir des nodes
      for i in 1 : dim_edges
          dim = length(graph_edges[i])
          for j in 1 : dim
              first_node = i
              second_node = graph_edges[i][j][1]
              edge_weight = graph_edges[i][j][2]
              edge = Edge(edge_weight, (nodes[first_node], nodes[second_node]))
              push!(edges, edge)
          end
      end

```

```
        # Création du nom du graphe à partir du nom du fichier
        split_filepath = split(filepath, "/")
        filename = split_filepath[length(split_filepath)]
        split_filename = split(filename, ".")
        graphname = String(split_filename[1])

        # Création du graphe
        return Graph(graphname, nodes, edges)
    end
end
```

Nous pouvons finalement donner notre implémentation de la structure de données concrète d'une composante connexe, ainsi que ses méthodes qui seront nécessaires pour l'algorithme.

Main.workspace73.merge_connected_components!

```
begin
    """Type representant une composante connexe comme un graphe."""
    mutable struct ConnectedComponent{T} <: AbstractGraph{T}
        name::String
        nodes::Vector{Node{T}}
        edges::Vector{Edge}
    end

    """Crée une composante connexe à partir d'un noeud."""
    create_connected_component_from_node(node::Node{T}) where T = ConnectedComponent{T}
("Connected component containing node " * node.name,[node],[])

    """Calcule le nombre de noeuds d'un lien contenus dans la composante connexe."""
    function contains_edge_nodes(c_component::ConnectedComponent{T}, edge::Edge) where T
        nb_nodes_contained = 0
        for node in edge.nodes
            if contains_node(c_component, node)
                nb_nodes_contained += 1
            end
        end
        return nb_nodes_contained
    end

    """Fusionne deux composantes connexes reliées par un lien."""
    function merge_connected_components!(c_component1::ConnectedComponent{T},
c_component2::ConnectedComponent{T}, linking_edge::Edge) where T

        # Fusion des noeuds
        for node in c_component2.nodes
            add_node!(c_component1, node)
        end

        # Fusion des liens
        for edge in c_component2.edges
            add_edge!(c_component1, edge)
        end

        # Ajout du lien les reliant
        add_edge!(c_component1, linking_edge)

        return c_component1
    end
end
```

Question 2 : Implémenter l'algorithme de Kruskal pour un arbre de recouvrement minimal.

Avec les méthodes que nous avons défini dans la dernière question pour les composantes connexes, il nous est aisé d'implémenter l'algorithme de Kruskal.

Main.workspace77.find_minimum_spanning_tree

```
"""Algorithme de Kruskal pour calculer un arbre de recouvrement minimal d'un graphe
symétrique connexe."""
function find_minimum_spanning_tree(graph::AbstractGraph{T}, verbose::Bool) where T

  # Création une composante connexe pour chaque noeud du graphe
  connected_components = Vector{ConnectedComponent{T}}()
  for node in nodes(graph)
    push!(connected_components, create_connected_component_from_node(node))
  end

  # 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
      print("Searching ")
      show(edge)
    end

    # Trouver la ou les composantes connexes y touchant.
    linked_ccs = Vector{ConnectedComponent{T}}()
    for cc in connected_components
      nb_nodes_contained = contains_edge_nodes(cc, edge)

      # Si le lien touche à une seule composante, passer au lien suivant.
      if nb_nodes_contained == 2
        if verbose
          println("Found in " * cc.name * ".")
        end
        break
      elseif nb_nodes_contained == 1
        push!(linked_ccs, cc)
        if length(linked_ccs) == 2
          break
        end
      end
    end

    # Si le lien touche à 2 composantes connexes distinctes, les fusionner
    if length(linked_ccs) == 2
      if verbose
        println("Found between " * linked_ccs[1].name * " and " * linked_ccs[2].name * ". =>
Merging components.")
      end

      sort!(linked_ccs, by=cc -> nb_nodes(cc), rev = true)
      merge_connected_components!(linked_ccs[1], linked_ccs[2], edge)
      deleteat!(connected_components, findall(cc->cc==linked_ccs[2], connected_components))

      # Si nous n'obtenons plus qu'une seule composante connexe, éviter les boucles inutiles
```

```
      if length(connected_components) == 1
        break
      end
    end
  end

  # Si le graphe initial n'était pas connexe, choisir la plus grosse composante connexe finale
  if length(connected_components) > 1
    sort!(connected_components, by=cc -> nb_nodes(cc), rev = true)
  end

  return Graph("Minimal spanning tree of " * name(graph), nodes(connected_components[1]),
edges(connected_components[1]))
end
```

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

Question 3 : Tests unitaires.

Pour tester les méthodes de AbstractGraph, celles de ConnectedComponent ainsi que l'algorithme de Kruskal, nous avons implémenter la série de tests unitaires ci-dessous.

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\Vincent\Dropbox\2020- Maîtrise\Session 1\Impl d'Algo de Rech Oper\Projet\mth6412b-starter-code\rapport_phase2.jl#==#d6632060-03fa-11eb-22ed-6d4638537d55:16

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

```
begin
    using Test

    # Tests pour les méthodes de Graph
    println("Testing Graph methods...")
    println()

    node1 = Node("Joe", 3.14)
    node2 = Node("Steve", exp(1))
    node3 = Node("Jill", 4.12)
    edge1 = Edge(500, (node1, node2))
    edge2 = Edge(1000, (node2, node3))

    g1 = create_empty_graph("g1",Float64)

    @test name(g1) == "g1"
    @test nb_nodes(g1) == 0
    @test nb_edges(g1) == 0
    @test contains_node(g1,node1) == false
```

```
    @test contains_edge(g1,edge1) == false

    add_node!(g1,node1)

    @test nb_nodes(g1) == 1
    @test contains_node(g1,node1) == true

    add_edge!(g1,edge1)

    @test nb_edges(g1) == 1
    @test contains_edge(g1,edge1) == true
    @test nb_nodes(g1) == 2
    @test contains_node(g1,node2) == true

    add_node!(g1,node3)
    add_edge!(g1,edge2)

    #show(g1)
    #println()

    @test nb_nodes(g1) == 3
    @test contains_node(g1,node3) == true
    @test nb_edges(g1) == 2
    @test contains_edge(g1,edge2) == true

    g2 = Graph("g2", [node1, node2, node3], [edge1, edge2])

    #show(g2)
    #println()

    @test name(g1) != name(g2)
    @test nodes(g1) == nodes(g2)
    @test edges(g1) == edges(g2)


    # Tests pour les méthodes de Connected Component
    println("Testing ConnectedComponent methods...")
    println()

    cc1 = create_connected_component_from_node(node1)

    @test name(cc1) == "Connected component containing node Joe"
    @test nb_nodes(cc1) == 1
    @test contains_node(cc1,node1) == true
    @test nb_edges(cc1) == 0

    @test contains_edge_nodes(cc1, edge2) == 0
    @test contains_edge_nodes(cc1, edge1) == 1

    cc2 = create_connected_component_from_node(node2)

    @test contains_edge_nodes(cc2, edge1) == 1
    @test contains_edge_nodes(cc2, edge2) == 1

    merge_connected_components!(cc1,cc2,edge1)

    @test nb_nodes(cc1) == 2
    @test contains_node(cc1,node1) == true
    @test contains_node(cc1,node2) == true
    @test nb_edges(cc1) == 1
    @test contains_edge(cc1,edge1) == true

    @test contains_edge_nodes(cc1, edge1) == 2
    @test contains_edge_nodes(cc1, edge2) == 1

    cc3 = create_connected_component_from_node(node3)
    merge_connected_components!(cc1,cc3,edge2)

    #show(cc1)
    #println()

    @test name(g1) != name(cc1)
```

```
    @test nodes(g1) == nodes(cc1)
    @test edges(g1) == edges(cc1)


    # Tests pour l'algorithme de Kruskal pour un arbre de recouvrement minimal
    println("Testing Minimum Spanning Tree Kruskal Algorithm...")
    println()

    # 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)
    #println()

    mst = find_minimum_spanning_tree(g3, true)
    println()

    #show(mst)
    #println()

    @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

    println("All tests complete!")
end
```

Avec cete série de tests, nous obtenons la sortie suivante.

```
Testing Graph methods...

Testing ConnectedComponent methods...

Testing Minimum Spanning Tree Kruskal Algorithm...

Searching Edge weight : 1.0
  Node g, data: nothing
  Node h, data: nothing
Found between Connected component containing node g and Connected component containing node
 h. => Merging components.
Searching Edge weight : 2.0
  Node c, data: nothing
  Node i, data: nothing
Found between Connected component containing node c and Connected component containing node
 i. => Merging components.
Searching Edge weight : 2.0
  Node f, data: nothing
  Node g, data: nothing
Found between Connected component containing node f and Connected component containing node
 g. => Merging components.
Searching Edge weight : 4.0
  Node a, data: nothing
  Node b, data: nothing
Found between Connected component containing node a and Connected component containing node
 b. => Merging components.
Searching Edge weight : 4.0
  Node c, data: nothing
  Node f, data: nothing
Found between Connected component containing node c and Connected component containing node
 g. => Merging components.
Searching Edge weight : 6.0
  Node g, data: nothing
  Node i, data: nothing
```

```
Found in Connected component containing node g.
Searching Edge weight : 7.0
  Node c, data: nothing
  Node d, data: nothing
Found between Connected component containing node d and Connected component containing node
 g. => Merging components.
Searching Edge weight : 7.0
  Node h, data: nothing
  Node i, data: nothing
Found in Connected component containing node g.
Searching Edge weight : 8.0
  Node b, data: nothing
  Node c, data: nothing
Found between Connected component containing node a and Connected component containing node
 g. => Merging components.
Searching Edge weight : 8.0
  Node a, data: nothing
  Node h, data: nothing
Found in Connected component containing node g.
Searching Edge weight : 9.0
  Node d, data: nothing
  Node e, data: nothing
Found between Connected component containing node e and Connected component containing node
 g. => Merging components.

All tests complete!
```

Si nous examinons la sortie de l'algorithme, nous reconnaissons l'algorithme de Kruskal tel qu'utilisé dans le cours. La seule exception est le lien [b,c] qui est utilisé au lieu du lien [a,h] puisqu'il est vu en premier par l'algorithme. Ces deux liens ayant le même poids, nous retrouvons bien un arbre de recouvrement minimal équivalent.

Question 4 : Tests sur instances de TSP symétriques.

Nous avons testé notre implémentation sur toutes les instances de TSP symétriques fournies avec le code suivant.

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

```
begin
    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"]

    filepath = "C:/Users/Vincent/Dropbox/2020- Maîtrise/Session 1/Impl d'Algo de Rech
 Oper/Projet/mth6412b-starter-code/"

```

```
    for filename in filenames
      graph = create_graph_from_stsp_file(filepath * filename, false)
      println("Graph ", name(graph), " has ", nb_nodes(graph), " nodes and ",
nb_edges(graph), " edges.")

      mst = find_minimum_spanning_tree(graph, false)
      println("Graph ", name(mst), " has ", nb_nodes(mst), " nodes and ", nb_edges(mst), "
edges.")

      println()
    end
end
```

Avec ce code, nous obtenons la sortie suivante.

```
Graph bayg29 has 29 nodes and 406 edges.
Graph Minimal spanning tree of bayg29 has 29 nodes and 28 edges.

Graph bays29 has 29 nodes and 841 edges.
Graph Minimal spanning tree of bays29 has 29 nodes and 28 edges.

Graph brazil58 has 58 nodes and 1653 edges.
Graph Minimal spanning tree of brazil58 has 58 nodes and 57 edges.

Graph brg180 has 180 nodes and 16110 edges.
Graph Minimal spanning tree of brg180 has 180 nodes and 179 edges.

Graph dantzig42 has 42 nodes and 903 edges.
Graph Minimal spanning tree of dantzig42 has 42 nodes and 41 edges.

Graph fri26 has 26 nodes and 351 edges.
Graph Minimal spanning tree of fri26 has 26 nodes and 25 edges.

Graph gr17 has 17 nodes and 153 edges.
Graph Minimal spanning tree of gr17 has 17 nodes and 16 edges.

Graph gr21 has 21 nodes and 231 edges.
Graph Minimal spanning tree of gr21 has 21 nodes and 20 edges.

Graph gr24 has 24 nodes and 300 edges.
Graph Minimal spanning tree of gr24 has 24 nodes and 23 edges.

Graph gr48 has 48 nodes and 1176 edges.
Graph Minimal spanning tree of gr48 has 48 nodes and 47 edges.

Graph gr120 has 120 nodes and 7260 edges.
Graph Minimal spanning tree of gr120 has 120 nodes and 119 edges.

Graph hk48 has 48 nodes and 1176 edges.
Graph Minimal spanning tree of hk48 has 48 nodes and 47 edges.
```

```
Graph pa561 has 561 nodes and 157641 edges.
Graph Minimal spanning tree of pa561 has 561 nodes and 560 edges.

Graph swiss42 has 42 nodes and 1764 edges.
Graph Minimal spanning tree of swiss42 has 42 nodes and 41 edges.
```

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, l'algorithme de Kruskal assure que ces arbres de recouvrement sont bel et bien minimaux.