Rapport de projet: Phase 2

Équipe: Elahe Amiri et Louis-Philippe Proulx

Lien Github du code: **Branche Phase 2 du projet**

Il y a une réplique du code dans "projet\phase2\main.jl"

```
display (generic function with 2 methods)
```

```
. begin
. include("../node.jl")
. include("../edge.jl")
. include("../graph.jl")
. include("../read_stsp.jl")
. include("../disjoint_set.jl")
. include("../display.jl")
. end
```

Voici le type DisjointSet que nous proposons. L'information de chaque noeud est emmagasiné dans un dictionnaire. L'information inclut le nom de l'ensemble connexe (String), l'ensemble des noeuds qui font parti de cet ensemble (Set(Node)) ainsi que l'ensemble des arrêtes qui font partis de cet ensemble (Set(Edge))

```
import Base.show
include("node.jl")
include("edge.jl")
include("graph.jl")

"""Type abstrait dont d'autres types d'ensemble dériveront."""
abstract type AbstractDisjointSet{T} end

"""Type representant les composantes connexes d'un graphe.

Exemple :
my_disjoint_set = DisjointSet{Vector{Float64}}()
create_disjointSet!(my_disjoint_set, Main_Graph)

Each nodes will point to the name of the set, to a set of nodes and to a set of edges
"""
mutable struct DisjointSet{T} <: AbstractDisjointSet{T}
  dict_sets::Dict{String, Tuple{String, Set{Node{T}}, Set{Edge{T}}}}
end

"""Creation of an empty disjoint sets"""
DisjointSet{T}() where T = DisjointSet(Dict{String,Tuple{String, Set{Node{T}}, Set{Edge{T}}}}())

"""Creation of disjoint sets from a graph"""
function create_disjointSet!(disjointset::DisjointSet{T}, graph::Graph{T}) where T
  #At first, each node has its own set
  for node in graph.nodes
```

```
    disjointset.dict_sets[node.name] = (node.name, Set([node]), Set(Edge{T}[]))
  end
end

"""Returns the edges"""
edges(disjointset::DisjointSet{T}) where T = collect(values(disjointset.dict_sets))[1][3]

"""Returns the nodes"""
nodes(disjointset::DisjointSet{T}) where T = collect(values(disjointset.dict_sets))[1][2]

"""Find the name of the set of the node"""
function find_root(disjointset::DisjointSet{T},node::Node{T}) where T
  disjointset.dict_sets[node.name][1]
end

"""Return true if the two nodes of the edge are form two different sets"""
function is_arc_disjoint(disjointset::DisjointSet{T},edge::Edge{T}) where T
  return(find_root(disjointset,edge.adjacentnodes[1]) != find_root(disjointset,edge.adjacentnode
s[2]))
end

"""Merge the two sets together based on the edge that join them """
function union_disjoint_set(disjointset::DisjointSet{T}, new_edge::Edge{T} ) where T
  node_name=new_edge.adjacentnodes[1].name
  node_name2=new_edge.adjacentnodes[2].name

  # The new sets includes all the nodes of the two sets
  New_Set = union(disjointset.dict_sets[node_name][2],
  disjointset.dict_sets[node_name2][2])
  #The new set includes all of the edges of the two sets
  New_Set_Edges = union(disjointset.dict_sets[node_name][3],
  disjointset.dict_sets[node_name2][3])
  #The new set also include the new edge
  New_Set_Edges = union(New_Set_Edges, Set([new_edge]))
  #The new set will be name base on the name of the first set
  New_Set_Name = disjointset.dict_sets[node_name][1]

  #update the information of all nodes that are part of the new set
  for element in New_Set
    disjointset.dict_sets[element.name]=  (New_Set_Name, New_Set, New_Set_Edges)
  end
end
```

```
. display("../disjoint_set.jl")
```

Nous allons maintenat implémenter l'algorithme the Kruskal sur l'exemple des notes de cours

Nous créons un graph avec les même caractéristiques que celui des notes de cours

```
Graph{Array{Float64,1}}(
name = "Graph_Ex_Notes_de_Cours"
```

```
nodes =   Node[
            1:   Node{Array{Float64,1}}("a",  Float64[])
            2:   Node{Array{Float64,1}}("b",  Float64[])
            3:   Node{Array{Float64,1}}("c",  Float64[])
            4:   Node{Array{Float64,1}}("d",  Float64[])
            5:   Node{Array{Float64,1}}("e",  Float64[])
            6:   Node{Array{Float64,1}}("f",  Float64[])
            7:   Node{Array{Float64,1}}("g",  Float64[])
            8:   Node{Array{Float64,1}}("h",  Float64[])
            9:   Node{Array{Float64,1}}("i",  Float64[])
          ]
edges =
  Edge[
   1:
    Edge{Array{Float64,1}}("(a,b)",  4.0,  (Node{Array{Float64,1}}("a",  Float64[]),  Node
   2:
    Edge{Array{Float64,1}}("(a,h)",  8.0,  (Node{Array{Float64,1}}("a",  Float64[]),  Node
   3:
    Edge{Array{Float64,1}}("(b,c)",  8.0,  (Node{Array{Float64,1}}("b",  Float64[]),  Node
   4:
    Edge{Array{Float64,1}}("(b,h)",  11.0,  (Node{Array{Float64,1}}("b",  Float64[]),  No
   5:
    Edge{Array{Float64,1}}("(c,d)",  7.0,  (Node{Array{Float64,1}}("c",  Float64[]),  Node
   6:
    Edge{Array{Float64,1}}("(c,f)",  4.0,  (Node{Array{Float64,1}}("c",  Float64[]),  Node
   7:
    Edge{Array{Float64,1}}("(c,i)",  2.0,  (Node{Array{Float64,1}}("c",  Float64[]),  Node
   8:
    Edge{Array{Float64,1}}("(d,e)",  9.0,  (Node{Array{Float64,1}}("d",  Float64[]),  Node
   9:
    Edge{Array{Float64,1}}("(d,f)",  14.0,  (Node{Array{Float64,1}}("d",  Float64[]),  No
   10:
    Edge{Array{Float64,1}}("(e,f)",  10.0,  (Node{Array{Float64,1}}("e",  Float64[]),  No
   11:
    Edge{Array{Float64,1}}("(f,g)",  2.0,  (Node{Array{Float64,1}}("f",  Float64[]),  Node
   12:
    Edge{Array{Float64,1}}("(g,h)",  1.0,  (Node{Array{Float64,1}}("g",  Float64[]),  Node
   13:
    Edge{Array{Float64,1}}("(g,i)",  6.0,  (Node{Array{Float64,1}}("g",  Float64[]),  Node
   14:
    Edge{Array{Float64,1}}("(h,i)",  7.0,  (Node{Array{Float64,1}}("h",  Float64[]),  Node
  ]
)
```

```
. begin
. Ex_Graph = Graph("Graph_Ex_Notes_de_Cours", Node{Array{Float64,1}}[], Edge{Array{Float64,1}}[])
. my_node_dict = Dict{String,Any}()
. for node_name in ["a","b","c","d","e","f","g","h","i"]
.     my_node_dict[node_name]=Node{Array{Float64,1}}(node_name,[])
.     add_node!(Ex_Graph,my_node_dict[node_name])
. end
. add_edge!(Ex_Graph,Edge("(a,b)",4.0,(my_node_dict["a"],my_node_dict["b"])))
. add_edge!(Ex_Graph,Edge("(a,h)",8.0,(my_node_dict["a"],my_node_dict["h"])))
. add_edge!(Ex_Graph,Edge("(b,c)",8.0,(my_node_dict["b"],my_node_dict["c"])))
. add_edge!(Ex_Graph,Edge("(b,h)",11.0,(my_node_dict["b"],my_node_dict["h"])))
. add_edge!(Ex_Graph,Edge("(c,d)",7.0,(my_node_dict["c"],my_node_dict["d"])))
. add_edge!(Ex_Graph,Edge("(c,f)",4.0,(my_node_dict["c"],my_node_dict["f"])))
. add_edge!(Ex_Graph,Edge("(c,i)",2.0,(my_node_dict["c"],my_node_dict["i"])))
. add_edge!(Ex_Graph,Edge("(d,e)",9.0,(my_node_dict["d"],my_node_dict["e"])))
. add_edge!(Ex_Graph,Edge("(d,f)",14.0,(my_node_dict["d"],my_node_dict["f"])))
. add_edge!(Ex_Graph,Edge("(e,f)",10.0,(my_node_dict["e"],my_node_dict["f"])))
. add_edge!(Ex_Graph,Edge("(f,g)",2.0,(my_node_dict["f"],my_node_dict["g"])))
. add_edge!(Ex_Graph,Edge("(g,h)",1.0,(my_node_dict["g"],my_node_dict["h"])))
  add_edge!(Ex_Graph,Edge("(g,i)",6.0,(my_node_dict["g"],my_node_dict["i"])))
```

```
. add_edge!(Ex_Graph,Edge("(h,i)",7.0,(my_node_dict["h"],my_node_dict["i"])))
. end
```

Nous créons maintenant une composante connexe par noeud

```
. begin
. my_disjoint_set_toy = DisjointSet{Vector{Float64}}()
. create_disjointSet!(my_disjoint_set_toy, Ex_Graph)
. end
```

Nous trions les arrêtes selon leur poids

```
A =  Edge[Edge{Array{Float64,1}}("(g,h)",  1.0,  (Node{Array{Float64,1}}("g",  Float64[])
```

```
. A = sort(Ex_Graph.edges, by = x -> x.weight)
```

Nous appliquons l'algorithme de Kruskal

```
. for a in A
.   if is_arc_disjoint(my_disjoint_set_toy,a)
.     union_disjoint_set(my_disjoint_set_toy, a )
.   end
. end
```

Nous affichons le poids de l'arbre de recouvrement minimal

```
37.0
```

```
. sum(x->x.weight,edges(my_disjoint_set_toy))
```

Dans cet 2e partie nous appliquerons l'algorithme à une instance TSP symétrique: "bayg29.tsp"

Création du graph à paritir du fichier de données

```
. begin
. Main_Graph = Graph{Vector{Float64}}()
```

```
. graph_nodes, graph_edges, edges_weight = read_stsp("../../instances/stsp/bayg29.tsp")
. create_graph!(Main_Graph, graph_nodes,graph_edges, edges_weight)
. end
```

Création des composantes connexes pour chaque noeud

```
. begin
. my_disjoint_set = DisjointSet{Vector{Float64}}()
. create_disjointSet!(my_disjoint_set, Main_Graph)
. end
```

Nous trions les arrêtes selon le poids

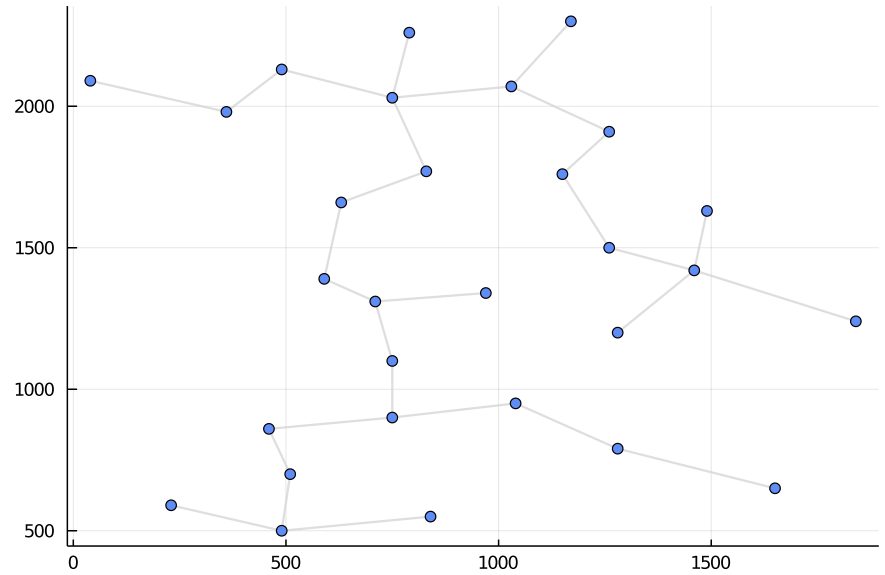
```
A1 =  Edge[Edge{Array{Float64,1}}("(10,20)",  25.0,  (Node{Array{Float64,1}}("10",  Floate
```

```
. A1 = sort(Main_Graph.edges, by = x -> x.weight)
```

Nous appliquons encore une fois l'algorithme de Kruskal

```
. for a in A1
.   if is_arc_disjoint(my_disjoint_set,a)
.     union_disjoint_set(my_disjoint_set, a )
.   end
. end
```

Nous avons créé une nouvelle méthode "plot_subgraph()" qui permet d'afficher un sous-ensemble d'arrêtes. Nous affichons donc l'arbre de recouvrement minimal que nous venons de trouver




```
. plot_subgraph(Main_Graph, edges(my_disjoint_set))
```

Le poids de l'arbre de recouvrement minimal:

```
1319.0
```

```
. sum(x->x.weight,edges(my_disjoint_set))
```

Les arrêtes de l'arbre de recouvrement minimal.

```
Set{Edge{Array{Float64,1}}} with 28 elements:
  Edge{Array{Float64,1}}("(24,27)", 38.0, (Node{Array{Float64,1}}("24", [1260.0, 1500.0…
  Edge{Array{Float64,1}}("(10,20)", 25.0, (Node{Array{Float64,1}}("10", [710.0, 1310.0]…
  Edge{Array{Float64,1}}("(4,10)", 39.0, (Node{Array{Float64,1}}("4", [750.0, 1100.0]),…
  Edge{Array{Float64,1}}("(15,19)", 49.0, (Node{Array{Float64,1}}("15", [750.0, 900.0])…
  Edge{Array{Float64,1}}("(5,9)", 42.0, (Node{Array{Float64,1}}("5", [750.0, 2030.0]), …
  Edge{Array{Float64,1}}("(14,18)", 32.0, (Node{Array{Float64,1}}("14", [510.0, 700.0])…
  Edge{Array{Float64,1}}("(6,12)", 46.0, (Node{Array{Float64,1}}("6", [1030.0, 2070.0])…
  Edge{Array{Float64,1}}("(1,24)", 52.0, (Node{Array{Float64,1}}("1", [1150.0, 1760.0])…
  Edge{Array{Float64,1}}("(6,28)", 52.0, (Node{Array{Float64,1}}("6", [1030.0, 2070.0])…
  Edge{Array{Float64,1}}("(17,22)", 47.0, (Node{Array{Float64,1}}("17", [230.0, 590.0])…
  Edge{Array{Float64,1}}("(8,27)", 39.0, (Node{Array{Float64,1}}("8", [1490.0, 1630.0])…
  Edge{Array{Float64,1}}("(4,15)", 34.0, (Node{Array{Float64,1}}("4", [750.0, 1100.0]),…
  Edge{Array{Float64,1}}("(23,27)", 74.0, (Node{Array{Float64,1}}("23", [1840.0, 1240.0…
  ⋮
```

```
. edges(my_disjoint_set)
```

Finalement, nous avons créé deux fichiers de test unitaire qui permettent de valider la bon fonctionnment du type DisjointSet et Graph

```
Test Passed
```

```
. begin
. include("graph_test.jl")
. include("disjoint_set_test.jl")
. end
```

Voici le code de "disjoint_set_test.jl"

```
using Test
include("../disjoint_set.jl")
include("../graph.jl")
include("../read_stsp.jl")

filename_stsp = "bayg29.tsp"
root = normpath(joinpath(@__FILE__,"..","..","..",".."))
filepath_to_stsp = "instances\\stsp"
filepath = joinpath(root, filepath_to_stsp)
filepath = joinpath(filepath, filename_stsp)

my_graph = Graph{Vector{Float64}}() #The type of data containted in a node is a vector, (x,y) co
ordinate
graph_nodes, graph_edges, edges_weight = read_stsp(filepath)
create_graph!(my_graph, graph_nodes,graph_edges, edges_weight)

my_disjoint_set = DisjointSet{Vector{Float64}}()
create_disjointSet!(my_disjoint_set, my_graph)

# Verify that the number of sets is equal to the number of nodes at first
@test length(my_disjoint_set.dict_sets) == length(my_graph.nodes)

#Every edge should be disjoint_set
for a in my_graph.edges
  @test is_arc_disjoint(my_disjoint_set,a)
end

#Apply Kruskal Algo
A = sort(my_graph.edges, by = x -> x.weight)

for a in A
  if is_arc_disjoint(my_disjoint_set,a)
    union_disjoint_set(my_disjoint_set, a )
  end
end

#The number of edges should be the number of nodes minus one
@test length(edges(my_disjoint_set)) == length(my_graph.nodes)-1
```

```
. display("disjoint_set_test.jl")
```