Rapport de projet: Phase 3

Équipe: Elahe Amiri et Louis-Philippe Proulx

Lien Github du code: **Branche Phase 3 du projet**

Il y a une réplique du code dans "projet\phase3\main.jl"

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

```
begin
using Test

include("../node.jl")
include("../edge.jl")
include("../graph.jl")
include("../read_stsp.jl")

include("marked_node.jl")
include("marked_edge.jl")
include("marked_graph.jl")
include("Prime_Alg.jl")
include("kruskal_Alg.jl")
include("../display.jl")
end
```

Voici l'implémentation des deux heuristiques d'accélération:

1. L'union via le rang

```
function Union_sets(Root_node1::String, Root_node2::String, DS::DisjointSets)
    if DS.rank[Root_node1] > DS.rank[Root_node2]
        DS.parent[Root_node2] = Root_node1
    elseif DS.rank[Root_node1] < DS.rank[Root_node2]
        DS.parent[Root_node1] = Root_node2
    else
        DS.parent[Root_node2] = Root_node1
        DS.rank[Root_node1] =+1
    end
end
```

```
display("kruskal_Alg.jl",40,49)
```

Notre implémentation permet de prendre la valeur maximum de rang. Si les deux root nodes ont un rang égal, on ajoute 1 à un des deux noeuds. On met à jour le parent.

2. La compression des chemins

```
    for edge in Edge_sort
        root1 = find_set(edge.adjacentnodes[1].name, DS)
        root2 = find_set(edge.adjacentnodes[2].name, DS)
        if root1 != root2
            push!(MST_edges, edge)
            W = W + edge.weight
            Union_sets(root1, root2, DS)
        end
    end
```

```
display("kruskal_Alg.jl",65,73)
```

Chaque fois qu'il y a union de deux ensembles disjoints, cela se fait toujours au niveau de la racine. La racine d'un des ensemble devient le parent de l'autre racine. Cela permet la compression des chemins

en ajoutant uniquement de la profondeur si deux ensemble disjoints sont de même rang.

Question théorique: voir pièce jointe à la fin du rappport

Notre implémentation de l'algorithme de Prim

```
function Prime_Algo(graph::MarkedGraph{T}, edges_weight::Dict{Tuple{Int64,Int64},Float64}, sourc
e::MarkedNode{T}) where T
    W = 0.0
    PQ = PriorityQueue(MarkedNode{Array{Float64,1}}[])
    MST = MarkedNode{Array{Float64,1}}[]

    for U in graph.nodes
        if U.name == source.name
            set_min_weight!(U, 0.0)
            set_parent!(U, U)
        end
        enqueue!(PQ, U)
    end


    while !is_empty(PQ)
        V = dequeue!(PQ)
        set_visited!(V)
        push!(MST,V)
        W = W + V.min_weight
        for U in PQ.items
            if haskey(edges_weight,(parse(Int64, V.name) , parse(Int64, U.name)))
                if U.visited == false && U.min_weight > edges_weight[parse(Int64, V.name) , pars
e(Int64, U.name)]
                    U.min_weight = edges_weight[parse(Int64, V.name) , parse(Int64, U.name)]
                    set_parent!(U, V)
                end
            end
        end
    end
    #*header["NAME"]
    MST_Graph = MarkedGraph("MSTGraph_", MST, MarkedEdge{Array{Float64,1}}[])
    for node in MST[2:end]
        edge_name = "(*node.parent.name*","*node.name*")"
        new_edge = MarkedEdge(edge_name, node.min_weight, (node.parent , node))
        add_markededge!(MST_Graph, new_edge)
    end
    return W, MST_Graph
```

```
display("Prime_Alg.jl",41,76)
```

La fonction Prim*Algo dans le fichier *Prime*Alg.jl* calcule l'arbre de recouvrement minimal d'un graph et le poids total. Notre classe graphe ne contient pas un dictionnaire permettant facilement d'associer le poids d'une arrête à ses deux noeuds. Nous fournissons donc ce dictionnaire à la fonction Prim. Le noeud source sera entré dans la file de priorité avec un poids de 0 et tous les autres noeuds avec un poids Inf. Par la suite la boucle principale enlèvera le noeud avec un poids associé minimal jusqu'à ce que la file soit vide. Ce qui correspond à ajouter un arrête légère à notre sous-arbre de recouvrement minimum. On met à jour le poids des noeuds isolés en fonction du dernier noeud ajouté à l'arbre de recouvrement minimal, uniquement si cela réduit son poids. À la fin on crée un graph pour représenter l'arbre de recouvrement minimal.

Testons note implémentation sur l'exemple des notes de cours

```
7.0
```

```
begin
Ex_Graph1 = MarkedGraph("Graph_Ex_Notes_de_Cours", MarkedNode{Array{Float64,1}}[],
MarkedEdge{Vector{Float64}}[])
my_node_dict = Dict{String,Any}()
for node_name in ["1","2","3","4","5","6","7","8","9"]
  my_node_dict[node_name]=MarkedNode(Float64[],name=node_name)
  add_markednode!(Ex_Graph1,my_node_dict[node_name])
end
add_markededge!(Ex_Graph1,MarkedEdge("(1,2)",4.0,
(my_node_dict["1"],my_node_dict["2"])))
add_markededge!(Ex_Graph1,MarkedEdge("(1,8)",8.0,
(my_node_dict["1"],my_node_dict["8"])))
add_markededge!(Ex_Graph1,MarkedEdge("(2,3)",8.0,
(my_node_dict["2"],my_node_dict["3"])))
add_markededge!(Ex_Graph1,MarkedEdge("(2,8)",11.0,
(my_node_dict["2"],my_node_dict["8"])))
add_markededge!(Ex_Graph1,MarkedEdge("(3,4)",7.0,
(my_node_dict["3"],my_node_dict["4"])))
```

```
add_markededge!(Ex_Graph1,MarkedEdge("(3,6)",4.0,
(my_node_dict["3"],my_node_dict["6"])))
add_markededge!(Ex_Graph1,MarkedEdge("(3,9)",2.0,
(my_node_dict["3"],my_node_dict["9"])))
add_markededge!(Ex_Graph1,MarkedEdge("(4,5)",9.0,
(my_node_dict["4"],my_node_dict["5"])))
add_markededge!(Ex_Graph1,MarkedEdge("(4,6)",14.0,
(my_node_dict["4"],my_node_dict["6"])))
add_markededge!(Ex_Graph1,MarkedEdge("(5,6)",10.0,
(my_node_dict["5"],my_node_dict["6"])))
add_markededge!(Ex_Graph1,MarkedEdge("(6,7)",2.0,
(my_node_dict["6"],my_node_dict["7"])))
add_markededge!(Ex_Graph1,MarkedEdge("(7,8)",1.0,
(my_node_dict["7"],my_node_dict["8"])))
add_markededge!(Ex_Graph1,MarkedEdge("(7,9)",6.0,
(my_node_dict["7"],my_node_dict["9"])))
add_markededge!(Ex_Graph1,MarkedEdge("(8,9)",7.0,
(my_node_dict["8"],my_node_dict["9"])))
my_edge_weight = Dict{Tuple{Int64,Int64},Float64}()
my_edge_weight[(1,2)]=4.0
my_edge_weight[(1,8)]=8.0
my_edge_weight[(2,3)]=8.0
my_edge_weight[(2,8)]=11.0
my_edge_weight[(3,4)]=7.0
my_edge_weight[(3,6)]=4.0
my_edge_weight[(3,9)]=2.0
my_edge_weight[(4,5)]=9.0
my_edge_weight[(4,6)]=14.0
my_edge_weight[(5,6)]=10.0
my_edge_weight[(6,7)]=2.0
my_edge_weight[(7,8)]=1.0
my_edge_weight[(7,9)]=6.0
my_edge_weight[(8,9)]=7.0
end
```

```
(37.0, MarkedGraph{Array{Float64,1}}("MSTGraph_", MarkedNode{Array{Float64,1}}[MarkedNo
```

```
W1, Prime_MST = Prime_Algo(Ex_Graph1, my_edge_weight, Ex_Graph1.nodes[1])
```

The weight of MST using Prim Algorithm:

```
md"The weight of MST using Prim Algorithm: "
```

```
37.0
```

```
W1
```

Le fichier *MST_Alg_test.jl* contient le résultat des tests sur les instances stsp. Il vérifie les structures implémentés tels que les files et les ensembles disjoinsts. Par la suite, il compare le poids de l'arbre de recouvrment minimal trouvé par l'algorithme de Kruskal à celui de Prim. Les réponses devraient être égal s'il n'y a pas d'erreurs d'implémentation. Après avoir rouler le fichier, il n'y avait pas d'erreurs. Voici un example pour *bayg29.tsp*

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

header = read_header(filepath)
graph_nodes, graph_edges, edges_weight = read_stsp(filepath)
Main_Graph = MarkedGraph("Graph_"*header["NAME"], MarkedNode{Array{Float64,1}}[], MarkedEdge{Arr
ay{Float64,1}}[])
create_MarkedGraph!(Main_Graph, graph_nodes, graph_edges, edges_weight)

W1, Prime_MST = Prime_Algo(Main_Graph, edges_weight, Main_Graph.nodes[1])
W2, Kruskal_MST = Kruskal(Main_Graph)

@test W1 == W2
```

```
display("MST_Alg_test.jl",65,80)
```