Rapport phase 3

Ameliorations apportees a la phase 2

Composantes connexes

Une nouvelle structure

Premiérement nous avons changé le Type composante connexe, le type devient un dictionnaire ou chaque clef est un noeud, et chaque valeur son parent

```
mutable struct Component{T} <: AbstractComp{T}
    nodes::Dict{Node{T}, Node{T}}
end
```

La strategie est donc de représenter un arbre couvrant d'un graphe G comme un vecteur de composantes connexes: une par noeuds de G. On a donc besoin de plusieurs fonctions sur les composantes connexes, que nous détaillons dans la partie suivante.

Des fonctions pour modifier les composantes connexes et vecteurs de composantes connexes

Prend en argument un graphe et renvoi un vecteur de composantes connexes initiales (noeud n => noeud n)

```
function to_components(g::Graph{T}) where T
    tmp = Vector{Component{T}}()
    for n in nodes(g)
        d= Dict{Node{T}, Node{T}}()
        d[n] = n
        solo = Component{T}(d)
        push!(tmp, solo)
    end
    return tmp
end
```

Vide une composante connexe de ces noeuds

```
function empty!(comp::AbstractComp{T}) where T
    comp.nodes = Dict{Node{T}, Node{T}}()
    comp
end
```

Renvoi la composante connexe qui contient le noeud n

```
function get_component_with_node(tree::Vector{Component{T}}, n::Node{T}) where T
    for c in tree
        if haskey(nodes(c), n)
            return c
        end
    end
    return nothing
end
```

Joins la composante connexe comp2 a la composante connexe comp1 en les liant au niveau de l'arete e

```
function add_nodes_at!(comp1::AbstractComp{T}, comp2::AbstractComp{T}, e::Abstrac
tEdge{T}) where T
    new1, new2 = ends(e)
    if haskey(nodes(comp1),new1)
        nodes(comp1)[new1] = new2
    elseif haskey(nodes(comp1),new2)
        nodes(comp1)[new2] = new1
    end
    for (k,v) in nodes(comp2)
        nodes(comp1)[k] = v
    end
    comp1
end
```

Renvoi true si les deux composantes connexes sont les memes

```
function same_component(comp1::AbstractComp, comp2::AbstractComp)
    if length(nodes(comp1)) != length(nodes(comp2))
        return false
    else
        for n in keys(nodes(comp1))
            if !haskey(nodes(comp2), n)
                return false
            end
        end
    end
    return true
end
```

Algorithme de Kruskal

Prend en parametre un graphe et renvoi un arbre couvrant de poids minimum en utilisant l'algorithme de Kruskal

```
function kruskal(g::Graph{T}) where T
#Tri les aretes de g par poids croissant
edge_sorted = sort(edges(g), by=weight)
tree_comps = to_components(g)
#garde en memoire les aretes selectionnees pour l'arbre
edges_selected = Vector{Edge{T}}()
for e in edge_sorted
    (new1, new2) = ends(e)
    comp1 = get_component_with_node(tree_comps, new1)
    comp2 = get_component_with_node(tree_comps, new2)
    if !same_component(comp1, comp2)
        push!(edges_selected, e)
        add_nodes_at!(comp1, comp2, e)
        empty!(comp2)
    end
end
return Graph{T}("Kruskal de $(name(g))", nodes(g), edges_selected)
end
```

Implémentation de l'heuristique 1-union via le rang

-Prend en parametre un graphe.
-Renvoi un graphe qui en est un arbre couvrant a cout minimum en utilisant l'algorithme de Kruskal muni de l'heuristique du rang.
-les modifications sont faites dans la boucle for.

```
function kruskal_heur1(g::Graph{T}) where T
    #Tri les aretes de g par poids croissant
    edge_sorted = sort(edges(g), by=weight)
    tree_comps = to_components_rg(g)
    #garde en memoire les aretes selectionnees pour l'arbre
    edges_selected = Vector{Edge{T}}()
    for e in edge_sorted
        (new1, new2) = (get_node(g, name(ends(e)[1])), get_node(g, name(ends(e)[2
]))) 
        comp1 = get_component_with_node(tree_comps, new1)
        comp2 = get_component_with_node(tree_comps, new2)
        if !same_component(comp1, comp2)
            push!(edges_selected, e)
            if rang(comp1) > rang(comp2)
                #### ajoute new 2 a la composante de new 1
                add_nodes_at!(comp1, comp2, e)

                ### on enleve new 2 de sa composante
                empty!(comp2)
            else
                add_nodes_at!(comp2, comp1, e)
                empty!(comp1)
                if rang(comp1) == rang(comp2)
                    set_rang!(comp2, rang(comp2) +1)
                end
            end
        end
    end

    return Graph{T}("Heuristique 1 kruskal de $(name(g))", nodes(g), edges_select
ed)
end
```

Implémentation de l'heuristique 2-compression des chemins

-on a utilisé une fonction renvoi si les deux composantes ont la meme racine (et donc sont identiques) ou non.

-* ici pour quoi vous avez ajouter cette fonction :::

-Prend en parametre un graphe et renvoi un graphe qui en est un arbre couvrant a cout minimum en utilisant l'algorithme de Kruskal muni de l'heuristique 2 (compression des chemins). –les modifications sont faites dans la boucle for.

```
function same_root(comp1::Component_root{T}, comp2::Component_root{T}) where T
    return name(root(comp1)) == name(root(comp2))
end



function kruskal_heur2(g::Graph{T}) where T
    #Tri les aretes de g par poids croissant
    edge_sorted = sort(edges(g), by=weight)
    tree_comps = to_components_root(g)
    #garde en memoire les aretes selectionnees pour l'arbre
    edges_selected = Vector{Edge{T}}()
    for e in edge_sorted
        (new1, new2) = (get_node(g, name(ends(e)[1])), get_node(g, name(ends(e)[2
]))
        comp1 = get_component_with_node(tree_comps, new1)
        comp2 = get_component_with_node(tree_comps, new2)
        if !same_root(comp1, comp2)
            push!(edges_selected, e)
            if rang(comp1) > rang(comp2)
                #### ajoute new 2 a la composante de new 1
                add_nodes!(comp1, comp2)

                ### on enleve new 2 de sa composante
                empty!(comp2)
            else
                add_nodes!(comp2, comp1)
                empty!(comp1)
                if rang(comp1) == rang(comp2)
                    set_rang!(comp2, rang(comp2) +1)
                end
            end
        end
    end

    return Graph{T}("Heuristique 2 kruskal de $(name(g))", nodes(g), edges_select
ed)
end
```

Algorithme de Prim

```
md""" ## Algorithme de Prim """
```

Pour l'implémentation de cet algorithme des Fonctions utilitaires sont utilisées

Déffinisson une fonction *get*all*edges*with*node qui
-Prend en argument un graphe et un noeud.
-Retourne toutes les aretes du graphe incidente au noeud.

```
function get_all_edges_with_node(g::AbstractGraph, node::AbstractNode)
    edges = Vector{AbstractEdge}()
    for n in nodes(g)
        e = get_edge(g, node, n)
        if !isnothing(e)
            push!(edges,e)
        end
    end
   return edges
end
```

Déffinisson une fonction *node*to*add qui

-Prend en parametre un vecteur des noeuds deja ajoutes a l'arbre de recouvrement et une arete

-Retourne l'extremité de l'arete qui n'appartient pas encore a l'arbre nothing sinon

```
function node_to_add(nodes_added::Vector{Node{T}}, new_edge::Edge{T}) where T
    (n1, n2) = ends(new_edge)
    i1 = findfirst(x -> name(x) == name(n1), nodes_added)
    i2 = findfirst(x -> name(x) == name(n2), nodes_added)
    if (sum(isnothing.([i1, i2])) == 1)
        if isnothing(i1)
            return n1
        else
            return n2
        end
    end
    return nothing
end
```

Algorithme Prim

Déffinisson une fonction *prim qui

Prend en parametre un grapheet renvoi un graphe qui est un de ses arbres de recouvrement minimum.

-Une brève documentation est présentée dans la fonction.

```
function prim(g::Graph{T}) where T

    edges_selected = Vector{Edge{T}}()

    #Toutes les aretes sont dans une structure mutable ordonnee. Le poids de l'ar
ete sert d'indice de priorité. Plus l'arete est legere, plus elle est prioritaire
    edges_sorted = MutableBinaryHeap{Edge{T}}(Base.By(weight))

    #on choisi au hasard une racine
    current_node = nodes(g)[rand(1:nb_nodes(g))]
    #On garde en memoire les noeuds couverts par l'arbre
    nodes_added =[current_node]

    #boolean qui indique quand il faut ajouter de nouvelles aretes aux aretes can
didates
    node_updated = true

    #tant que tous les noeuds n<ont pas ete atteinds
    while length(nodes_added) < nb_nodes(g)

        if node_updated
            #On cherche toutes les aretes incidentes au noeud qu<on vient d'ajout
er
            for e in get_all_edges_with_node(g, current_node)
                push!(edges_sorted, e)
             end
        end
        node_updated = false
        #On recupere l'arete la moins chere ATTEIGNABLE
        new_edge = pop!(edges_sorted)

        #On identifi quel noeud est ajouté avec l'ajout de cet arete
        new_node = node_to_add(nodes_added, new_edge)
        if !(isnothing(new_node))
            #On ajoute l'arete a l'arbre
            push!(edges_selected, new_edge)
            #On ajoute le nouveau noeud a notre liste
            push!(nodes_added, new_node)
            current_node = new_node
            node_updated = true
        end
    end

    return Graph("Prim arbre couvrant min de $(name(g))", nodes(g), edges_selecte
d)
end
```

Tests unitaires

Des tests unitaires on été implémentés, en prenant en compte un l'exemple ainsi que des cas limites.

Main

L'execution de la commande

```
julia main.jl $(instance)
```

produit en output le benchmark de chacunes de trois implementations de l'algorithme de Kruskal et de l'algorithme de Prim.

Pour toutes les instances symetriques, les resultats sont comparables:

- Les heuristiques de Kruskal permettent de diminuer le temps d'execution. C'est surtout la memoire allouee et le nombre d'allocation qui diminu drastiquement (du simple - kruskal_heur2 - au triple - kruskal -)
- L'algorithme de Prim implemente ainsi semble particulierement moins efficace que l'algorithme de Kruskal; meme sans heuristiques.