Ctrl + S

plot_graph

Affiche un graphe étant données un ensemble de noeuds et d'arêtes.

Exemple :

```
graph_nodes, graph_edges = read_stsp("bayg29.tsp")
plot_graph(graph_nodes, graph_edges)
savefig("bayg29.pdf")
```

Fonction de commodité qui lit un fichier stsp et trace le graphe.

```
#the repository can be found on the github link: https://github.com/houskkam/mth6412b-starter-code
begin
import Pkg
Pkg.add("Plots")
include("node.jl")
include("edge.jl")
include("graph.jl")
include("read_stsp.jl")
end
```

```
  Resolving package versions...
No Changes to `C:\Users\victo\.julia\environments\v1.9\Project.toml`
No Changes to `C:\Users\victo\.julia\environments\v1.9\Manifest.toml`
```

Selection deleted

show

```
show([io::IO = stdout], x)
```

Write a text representation of a value `x` to the output stream `io`. New types `T` should overload `show(io::IO, x::T)`. The representation used by `show` generally includes Julia-specific formatting and type information, and should be parseable Julia code when possible.

`repr` returns the output of `show` as a string.

For a more verbose human-readable text output for objects of type `T`, define `show(io::IO, ::MIME"text/plain", ::T)` in addition. Checking the `:compact` `IOContext` key (often checked as `get(io, :compact, false)::Bool`) of `io` in such methods is recommended, since some containers show their elements by calling this method with `:compact => true`.

See also `print`, which writes un-decorated representations.

Examples

```
julia> show("Hello World!")
"Hello World!"
julia> print("Hello World!")
Hello World!
```

```
show(io::IO, mime, x)
```

The `display` functions ultimately call `show` in order to write an object `x` as a given `mime` type to a given I/O stream `io` (usually a memory buffer), if possible. In order to provide a rich multimedia representation of a user-defined type `T`, it is only necessary to define a new `show` method for `T`, via: `show(io, ::MIME"mime", x::T) = ...`, where `mime` is a MIME-type string and the function body calls `write` (or similar) to write that representation of `x` to `io`. (Note that the `MIME""` notation only supports literal strings; to construct `MIME` types in a more flexible manner use `MIME{Symbol("")}`.)

Selection deleted

For example, if you define a `MyImage` type and know how to write it to a PNG file, you could define a function `show(io, ::MIME"image/png", x::MyImage) = ...` to allow your images to be displayed on any PNG-capable `AbstractDisplay` (such as IJulia). As usual, be sure to `import Base.show` in order to add new methods to the built-in Julia function `show`.

Technically, the `MIME"mime"` macro defines a singleton type for the given `mime` string, which allows us to exploit Julia's dispatch mechanisms in determining how to display objects of any given type.

The default MIME type is `MIME"text/plain"`. There is a fallback definition for `text/plain` output that calls `show` with 2 arguments, so it is not always necessary to add a method for that case. If a type benefits from custom human-readable output though, `show(::IO, ::MIME"text/plain", ::T)` should be defined. For example, the `Day` type uses `1 day` as the output for the `text/plain` MIME type, and `Day(1)` as the output of 2-argument `show`.

Examples

```
julia> struct Day
           n::Int
       end

julia> Base.show(io::IO, ::MIME"text/plain", d::Day) = print(io, d.n, " day")

julia> Day(1)
1 day
```

Container types generally implement 3-argument `show` by calling `show(io, MIME"text/plain"(), x)` for elements `x`, with `:compact => true` set in an `IOContext` passed as the first argument.

Affiche un noeud.

Affiche un edge.

Affiche un graphe

Affiche un edge.

```
### First exercice of Phase 2
# We decided to represent connected component as a oriented graph with a root. We created two new datatypes for that.
# The first one is EdgeOriented, which implements AbstractEdge. It is very similar to normal Edge but it has a start and an end.
# We also added a function that converts an Edge to EdgeOriented.
begin
    """Type abstrait dont d'autres types de edges orientés dériveront."""
    abstract type AbstractEdgeOriented{Z, T <: AbstractNode} <: AbstractEdge{Z, T}
end

    """Type représentant les edges orientés d'un graphe.

```

Selection deleted

```
    Exemple:
            noeud1 = Node("James", "ahooj")
            noeud2 = Node("Kirk", "guitar")
            noeud3 = Node("Lars", "tdd")
            edge1 = EdgeOriented(noeud1, noeud2, 5)
            edge2 = EdgeOriented(noeud2, noeud3, 4)

    """
    mutable struct EdgeOriented{Z, T} <: AbstractEdgeOriented{Z, T}
    debut::T
    fin::T
    poids::Z
    end

    # on présume que tous les edges dérivant d'AbstractEdge
    # posséderont des champs `edge1`, `edge2` et `poids`.

    """Renvoie le nom du premiere noeud d'un edge."""
    debut(edge::AbstractEdgeOriented) = edge.debut

    """Renvoie le nom du deuxieme noeud d'un edge."""
    fin(edge::AbstractEdgeOriented) = edge.fin

    ==(e1::AbstractEdgeOriented, e2::AbstractEdgeOriented) = (debut(e1) == debut(e2)) && (fin(e1) == fin(e2)) && (poids(e1) == poids(e2))

    Base.convert(::Type{T}, e::Edge) where {T<:EdgeOriented} = EdgeOriented(node1(e), node2(e), poids(e))

    """Affiche un edge."""
    function show(edge::AbstractEdgeOriented)
    println("Parent node ", debut(edge), "child node: ", fin(edge) ,", weight: ", poids(edge))
    end
end
```

Selection deleted

show

```
show([io::IO = stdout], x)
```

Write a text representation of a value `x` to the output stream `io`. New types `T` should overload `show(io::IO, x::T)`. The representation used by `show` generally includes Julia-specific formatting and type information, and should be parseable Julia code when possible.

`repr` returns the output of `show` as a string.

For a more verbose human-readable text output for objects of type `T`, define `show(io::IO, ::MIME"text/plain", ::T)` in addition. Checking the `:compact` `IOContext` key (often checked as `get(io, :compact, false)::Bool`) of `io` in such methods is recommended, since some containers show their elements by calling this method with `:compact => true`.

See also `print`, which writes un-decorated representations.

Examples

```
julia> show("Hello World!")
"Hello World!"
julia> print("Hello World!")
Hello World!
```

```
show(io::IO, mime, x)
```

The `display` functions ultimately call `show` in order to write an object `x` as a given `mime` type to a given I/O stream `io` (usually a memory buffer), if possible. In order to provide a rich multimedia representation of a user-defined type `T`, it is only necessary to define a new `show` method for `T`, via: `show(io, ::MIME"mime", x::T) = ...`, where `mime` is a MIME-type string and the function body calls `write` (or similar) to write that representation of `x` to `io`. (Note that the `MIME""` notation only supports literal strings; to construct `MIME` types in a more flexible manner use `MIME{Symbol("")}`.)

Selection deleted

For example, if you define a `MyImage` type and know how to write it to a PNG file, you could define a function `show(io, ::MIME"image/png", x::MyImage) = ...` to allow your images to be displayed on any PNG-capable `AbstractDisplay` (such as IJulia). As usual, be sure to `import Base.show` in order to add new methods to the built-in Julia function `show`.

Technically, the `MIME"mime"` macro defines a singleton type for the given `mime` string, which allows us to exploit Julia's dispatch mechanisms in determining how to display objects of any given type.

The default MIME type is `MIME"text/plain"`. There is a fallback definition for `text/plain` output that calls `show` with 2 arguments, so it is not always necessary to add a method for that case. If a type benefits from custom human-readable output though, `show(::IO, ::MIME"text/plain", ::T)` should be defined. For example, the `Day` type uses `1 day` as the output for the `text/plain` MIME type, and `Day(1)` as the output of 2-argument `show`.

Examples

```
julia> struct Day
           n::Int
       end

julia> Base.show(io::IO, ::MIME"text/plain", d::Day) = print(io, d.n, " day")

julia> Day(1)
1 day
```

Container types generally implement 3-argument `show` by calling `show(io, MIME"text/plain"(), x)` for elements `x`, with `:compact => true` set in an `IOContext` passed as the first argument.

Affiche un noeud.

Affiche un edge.

Affiche un graphe

Affiche un edge.

Affiche un graphe.

Selection deleted

```
# Then we used it to create the ComposanteConnexe type, which is a connected component. It is a subtype of AbstractGraph, so all methods that are implemented for AbstractGraph function for ComposanteConnexe too.
begin

    """Type abstrait dont d'autres types de graphes dériveront."""
    abstract type AbstractComposanteConnexe{T, Z} <: AbstractGraph{T, Z} end

    """Type representant une composante connexe comme un ensemble de noeuds, .

    Exemple :
```

```
        noeud1 = Node("James", "ahooj")
        noeud2 = Node("Kirk", "guitar")
        noeud3 = Node("Lars", "tdd")
        edge_oriented_1 = EdgeOriented(noeud1, noeud2, 5)
        edge_oriented_2 = EdgeOriented(noeud2, noeud3, 4)
        C = ComposanteConnexe(noeud1, [noeud1, noeud2, noeud3], [edge_oriented_1, edge_oriented_2])

    Attention, tous les noeuds doivent avoir des données de même type.
    """
    mutable struct ComposanteConnexe{T, Z} <: AbstractComposanteConnexe{T, Z}
    root::T
    nodes::Vector{T}
    edges::Vector{EdgeOriented{Z,T}}
    end

    """Ajoute un noeud et l'arret qui le relie au graphe."""
    function add_node_and_edge!(composante::ComposanteConnexe{Node{T}, Z}, node::Node{T}, edge::EdgeOriented{Z, Node{T}}) where {T, Z}
    push!(composante.nodes, node)
    push!(composante.edges, edge)
    composante
    end


    """Determines that connected components are equal if their contents equal."""
    ==(c1::ComposanteConnexe, c2::ComposanteConnexe) = (nodes(c1) == nodes(c2)) && (edges(c1) == edges(c2))


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

end
```

Selection deleted

kruskal

This function gets as an argument an instance of Graph and returns its spanning tree of type ComposanteConnexe while using Kruskal's algorithm

```
# Second exercice of Phase 2
# Firstly, here is my code for the Kruskal algorithm:
""" This function gets as an argument an instance of Graph and returns
    its spanning tree of type ComposanteConnexe while using Kruskal's algorithm
"""
begin
    function kruskal(graph::Graph{T, Z}) where {T, Z}
        sorted_edges = sort(graph.edges)
        connected_components = Vector{ComposanteConnexe{T, Z}}()
        num_added_edges = 0
        should_add = true
        for edge in sorted_edges
            should_add = true
            add_edge_to = Vector{ComposanteConnexe{T, Z}}()
            for component in connected_components
                # if both nodes already exist in the came cnnected component,
                # there is already a way between them, so we will not add this edge
                if (node1(edge) in nodes(component)) && (node2(edge) in
nodes(component))
                    should_add = false
                    break
                # if one of the nodes is already in one of the composed components,
we note the component
                elseif (node1(edge) in nodes(component)) || (node2(edge) in
nodes(component))
                    push!(add_edge_to, component)
                end
            end
            if should_add
                edge = convert(EdgeOriented{Z,T}, edge)
                #print("#\n")
                #print(edge, length(add_edge_to))
                # create new one and add it to the list of existing components
                if length(add_edge_to) == 0
                    new_component = ComposanteConnexe(debut(edge), [debut(edge),
fin(edge)], [edge])
                    push!(connected_components, new_component)
                # add the edge and the node that connects it to the connected
component
                elseif length(add_edge_to) == 1
                    if debut(edge) in nodes(add_edge_to[1])
                        add_node_and_edge!(add_edge_to[1], fin(edge), edge)
                    else
                        add_node_and_edge!(add_edge_to[1], debut(edge), edge)
                    end
                # I have to connect all components and the new edge from add_edge_to
into one,
```

Selection deleted

```
                # delete the old unconnected components
                else
                    push!(connected_components, connect_into_one(add_edge_to, edge))
                    for component in add_edge_to
                        component_idx = findfirst(==(component), connected_components)
                        deleteat!(connected_components, component_idx)
                    end
                end
                num_added_edges = num_added_edges + 1
            end
            if num_added_edges >= length(graph.nodes)
                break
            end
        end
        if length(connected_components) > 1
            println("error, too many connected components left")
        end
        connected_components[1]
    end
```

connect_into_one

Takes a vector of connected components and merges them into one.

```
# It also use a function that I added to the ComposanteConnexe
begin
    """Takes a vector of connected components and merges them into one."""
    function connect_into_one(composantes::Vector{ComposanteConnexe{T, Z}}, edge::EdgeOriented{Z, T}) where {T, Z}
    new_component = composantes[1]
    for node in nodes(composantes[2])
        if !(node in nodes(new_component))
        add_node!(new_component, node)
        end
    end
    for edge in edges(composantes[2])
        add_edge!(new_component, edge)
    end
    if(length(composantes) > 2)
        print("have to connect more than 2 components")
    end
    add_edge!(new_component, edge)
    new_component
    end
end
```

Selection deleted

Multiple definitions for noeud1, G, edge1, noeud2, noeud3, noeud4 and edge2

Combine all definitions into a single reactive cell using a `begin ... end` block.

```
# Now I will be testing whether it gives the correct result for the graph that
# we saw on lab slides
begin
    using Test
    # The testing_components_equal function is implemented in ComposanteConnexe like
    this

    """Used to make sure two connected components contain the same nodes and edges."""
    function testing_components_equal(c1::ComposanteConnexe, c2::ComposanteConnexe)
        @test length(nodes(c1)) == length(nodes(c1))
        @test length(edges(c2)) == length(edges(c2))

        for each in nodes(c1)
           @test each in nodes(c2)
        end
        for each in edges(c1)
            @test each in edges(c2)
        end
    end

    # Initializing nodes from example from laboratories
    noeud1 = Node("a", "a")
    noeud2 = Node("b", "b")
    noeud3 = Node("c", "c")
    noeud4 = Node("d", "d")
    noeud5 = Node("e", "e")
    noeud6 = Node("f", "f")
    noeud7 = Node("g", "g")
    noeud8 = Node("h", "h")
    noeud9 = Node("i", "i")

    # Initializing edges from example from laboratories
    edge1 = Edge(noeud1, noeud2, 4.0)
    edge2 = Edge(noeud1, noeud8, 8.0)
    edge3 = Edge(noeud2, noeud8, 11.0)
    edge4 = Edge(noeud2, noeud3, 8.0)
    edge5 = Edge(noeud8, noeud9, 7.0)
    edge6 = Edge(noeud8, noeud7, 1.0)
    edge7 = Edge(noeud7, noeud9, 6.0)
    edge8 = Edge(noeud9, noeud3, 2.0)
    edge9 = Edge(noeud7, noeud6, 2.0)
    edge10 = Edge(noeud3, noeud4, 7.0)
    edge11 = Edge(noeud3, noeud6, 4.0)
    edge12 = Edge(noeud4, noeud6, 14.0)
    edge13 = Edge(noeud4, noeud5, 9.0)
    edge14 = Edge(noeud6, noeud5, 10.0)

    # Initializing graph from example from laboratories
    lab_nodes = [noeud1, noeud2, noeud3, noeud4, noeud5, noeud6, noeud7, noeud8,
noeud9]

```

```
    lab_edges = [edge1, edge2, edge3, edge4, edge5, edge6, edge7, edge8, edge9, edge10, edge11, edge12, edge13, edge14]
    G = Graph("Lab", lab_nodes, lab_edges)

    # Initializing expected kruskal connected component
    kruskal_expected_edges = [edge1, edge2, edge6, edge8, edge9, edge10, edge11, edge13]
    kruskal_expected_edges = convert(Array{EdgeOriented{Float64, Node{String}}}, kruskal_expected_edges)
    expected_connected_component_kruskal = ComposanteConnexe(noeud1, lab_nodes, kruskal_expected_edges)

    # Testing kruskal connected component
    kruskal_component = kruskal(G)
    print(kruskal_component)
    print("\n")
    print(expected_connected_component_kruskal)

    testing_components_equal(kruskal_component, expected_connected_component_kruskal)
```

```
Main.var"workspace#2".ComposanteConnexe{Main.var"workspace#2".Node{String}, Float64}(Main.var"workspace#2".Node{String}("a", "a"), Main.var"workspace#2".Node{String}[Main.var"workspace#2".Node{String}("a", "a"), Main.var"workspace#2".Node{String}("b", "b"), Main.var"workspace#2".Node{String}("h", "h"), Main.var"workspace#2".Node{String}("g", "g"), Main.var"workspace#2".Node{String}("f", "f"), Main.var"workspace#2".Node{String}("i", "i"), Main.var"workspace#2".Node{String}("c", "c"), Main.var"workspace#2".Node{String}("d", "d"), Main.var"workspace#2".Node{String}("e", "e")], Main.var"workspace#2".EdgeOriented{Float64, Main.var"workspace#2".Node{String}}[Main.var"workspace#2".EdgeOriented{Float64, Main.var"workspace#2".Node{String}}(Main.var"workspace#2".Node{String}("a", "a"), Main.var"workspace#2".Node{String}("b", "b"), 4.0), Main.var"workspace#2".EdgeOriented{Float64, Main.var"workspace#2".Node{String}}(Main.var"workspace#2".Node{String}("h", "h"), Main.var"workspace#2".Node{String}("g", "g"), 1.0), Main.var"workspace#2".EdgeOriented{Float64, Main.var"workspace#2".Node{String}}(Main.var"workspace#2".Node{String}("g", "g"), Main.var"workspace#2".Node{String}("f", "f"), 2.0), Main.var"workspace#2".EdgeOriented{Float64, Main.var"workspace#2".Node{String}}(Main.var"workspace#2".Node{String}("i", "i"), Main.var"workspace#2".Node{String}("c", "c"), 2.0), Main.var"workspace#2".EdgeOriented{Float64, Main.var"workspace#2".Node{String}}(Main.var"workspace#2".Node{String}("c", "c"), Main.var"workspace#2".Node{String}("f", "f"), 4.0), Main.var"workspace#2".EdgeOriented{Float64, Main.var"workspace#2".Node{String}}(Main.var"workspace#2".Node{String}("c", "c"), Main.var"workspace#2".Node{String}("d", "d"), 7.0), Main.var"workspace#2".EdgeOriented{Float64, Main.var"workspace#2".Node{String}}(Main.var"workspace#2".Node{String}("a", "a"), Main.var"workspace#2".Node{String}("h", "h"), 8.0), Main.var"workspace#2".EdgeOriented{Float64, Main.var"workspace#2".Node{String}}(Main.var"workspace#2".Node{String}("d", "d"), Main.var"workspace#2".Node{String}("e", "e"), 9.0)])
Main.var"workspace#2".ComposanteConnexe{Main.var"workspace#2".Node{String}, Float64}(Main.var"workspace#2".Node{String}("a", "a"), Main.var"workspace#2".Node{String}[Main.var"workspace#2".Node{String}("a", "a"), Main.var"works
```

heuristique_compression (generic function with 1 method)

```
# Third exercice of Phase 2
#because composante connexe can lead to long chains and thus is time inefficient, another strategy can be used to form a chain.
begin
    #Include the other files we use: already done above

    #This new strategy starts from disjoints sets. Because this is not yet implemented, a new type is created.
    #This is DisjointSet which gives a parent and rank for a certain node.
    mutable struct DisjointSet{T}
        parent::T
        rank::Int
    end


    #There are only graphs given so the first step consists of creatin a disjoint-set for each node in the graph.
    function create_disjoint_sets(graph::Graph{T, Z}) where {T, Z}
        #an empty vector is made where everything will be stored.
        #when iterating through all the nodes in the graph, the disjoint set gets filled up.
        disjoint_sets= [DisjointSet(node, 0) for node in graph.nodes]
        #A vector of disjoint sets is returned.
        return disjoint_sets
    end

    #For this method the roots have to be found to compare the ranks of each node
    #This method does that without path compression
    function find_roots(disjoint_sets::Vector{DisjointSet{T}}, node::T) where T
        # iteratively follows the parent pointers until it reaches a node that is its own parent,
        # which indicates it is the root of the set.
        while disjoint_sets[node].parent != node
            node = disjoint_sets[node].parent
        end
        #the node that is the root of the given node is returned
        return node
    end

    #This method uses path compression
    function find_roots_compression(disjoint_sets::Vector{DisjointSet{T}}, node::T) where T
        # iteratively follows the parent pointers until it reaches a node that is its own parent,
        # which indicates it is the root of the set.
        if disjoint_sets[node].parent != node
            #but here the parent of each node is set as the root, which differentiates from the previous method.
            disjoint_sets[node].parent = find_roots(disjoint_sets, disjoint_sets[node].parent)
        end
        #the root (parent) is given
        return disjoint_sets[node].parent
```

```
    end

    #Union-by-Rank method
    function heuristique_union(graph::Graph{T, Z}) where {T, Z}
        # get all edges
        all_edges = edges(graph)
        # Make a disjoint-set for each node in the graph
        disjoints = create_disjoint_sets(graph)
        #an empty vector is made to
        tree_edges = Vector{Edge{T, Z}}()

        #all the edges in the graph are looked at
        for edge in all_edges
            #two nodes will get compared eached time to know whos rank is higher
            node1 = edge.node1
            node2 = edge.node2
            #to know the rank we will have to find their roots so the function find_roots is called upon. This will be done without path compression
            root_1 = find_roots(disjoints, node1)
            root_2 = find_roots(disjoints, node2)

            #if they both have the same parent they can't be connected because you would have a cycle
            if root_1!=root_2
                #the node with the highest rank becomes the parent of the other node
                if disjoints[root_1].rank < disjoints[root_2].rank
                    disjoints[root_1].parent = root_2
                    #rank goes up when they become the parent of someone new
                    disjoints[root_2].rank += 1
                #the node with the highest rank becomes the parent of the other node
                elseif disjoints[root_1].rank > disjoints[root_2].rank
                    disjoints[root_2].parent = root_1
                    #rank goes up when they become the parent of someone new
                    disjoints[root_1].rank += 1
                else
                    #if they have the same rank we can choose the parent
                    disjoints[root_2].parent = root_1
                    #rank goes up when they become the parent of someone new
                    disjoints[root_1].rank += 1
                end
                #the new edge that is made will be pushed into the tree
                push!(tree_edges, edge)
            end
        end
        #this gives back all the connections
        for edge in tree_edges
            println("Edge from node ", edge.node1, " to node ", edge.node2)
        end

    end

    #Montrer que le rang d'un noeud sera toujours inférieur à |S|-1.
    "Let us presume a random node x and we want to show that the rang will always be smaller than |S|-1, with S the number of nodes.
```

```
    1.When initialising every node gets a rank of 0 (base case)
    2.When comparing two sets we look at the rank of the roots of the nodes. When the nodes are different, the node with the highest rank becomes the parent.
    Node 'x' can only be a root when it is in a set with a lower rank node.
    3.At each union  the rank of the root with lowest rank will be minimum highered with one. So if node 'x' is in a set with rank 0 and gets connected with another set, the new root of the set will have
    a rank of at least 1.
    4.This will be repeated for all the nodes. All nodes except one will be connected, so we work with |S|-1 unions. This means the rank will never be higher than |S|-1.
    "
    #Montrer ensuite que ce rang sera en fait toujours inférieur à floor[log_{2}(|S|)] ;
    "
    1.Every node gets a rank 0 at creation
    2.With every union of two sets with different ranks, the new parents increases with minimum 1 in rank.
    3.To prove that the rank of each node is smaller than floor[log_{2}(|S|)] induction will be used.
    4.Base case: for one node with |S|=1 , the rank is zero and floor[log_{2}(|1|)]=0. -> this is true
    5.Induction hypothesis: the rank of a node is always smaller than floor[log_{2}(|n|)] for every set dimension |S| going to n.
    rank ( of set n) < floor[log_{2}(|n|)]
    6.Induction: prove this for n+1
    We split this set into a set of n and a set of 1. When connecting the two sets the new root will have a rank of at least 1 (because the ranks are different).
    This will lead to a rank, of a set of (n+1), lower than floor[log_{2}(|n|)]+1
    Because: rank ( of set n+1) < floor[log_{2}(|n|)]+1
    rank ( of set n) < floor[log_{2}(|n|)]
    rank ( of set n)+1 < floor[log_{2}(|n|)] +1
    This shows that the rank of every node will always be lower than floor[log_{2}(|S|)] vor every given set size |S|.
    "

    # Using path compression
    function heuristique_compression(graph::Graph{T, Z}) where {T, Z}
        # get all edges
        all_edges = edges(graph)
        # Make a disjoint-set for each node in the graph
        disjoints = create_disjoint_sets(graph)
        #empty vector is made to collect all the edes
        tree_edges = Vector{Edge{T, Z}}()

        #all the edges in the graph are looked at
        for edge in all_edges
            #two nodes will get compared eached time to know whos rank is higher
            node1 = edge.node1
            node2 = edge.node2
            #to know the rank we will have to find their roots so the function find_roots is called upon. This will be done with path compression
            root_1 = find_roots_compression(disjoints, node1)
            root_2 = find_roots_compression(disjoints, node2)
```

```
137
138             #if they both have the same parent they can't be connected because you
    would have a cycle
139             if root_1 != root_2
140                 #the node with the highest rank becomes the parent of the other node
141                 #rank goes up when they become the parent of someone new
142                 if disjoints[root_1].rank < disjoints[root_2].rank
143                     disjoints[root_1].parent = root_2
144                     disjoints[root_2].rank += 1
145                 elseif disjoints[root_1].rank > disjoints[root_2].rank
146                     disjoints[root_2].parent = root_1
147                     disjoints[root_1].rank += 1
148                 else
149                     #if they have the same rank we can choose the parent
150                     disjoints[root_2].parent = root_1
151                     disjoints[root_1].rank += 1
152                 end
153                 #the new edge that is made will be pushed into the tree
154                 push!(tree_edges, edge)
155             end
156         end
157
158         #this gives back all the connections
159         for edge in tree_edges
160             println("Edge from node ", edge.node1, " to node ", edge.node2)
161         end
162     end
163
```

UndefVarError: `G` not defined

```
# Fourth exercice of Phase 2
begin

    using DataStructures

    #I introduce a new structure so that in the priority queue the nodes can be added multiple times, also the edge is added,
    #this way we can get it easier when putting in the minimum spanning tree.
    mutable struct NodeKey{T,Z}
        node::Node{T}
        edge::EdgeOriented{Z,Node{T}}
        it::Int
    end

    #start of the function that uses prim algorithm to find the minimum spanning tree
    """ This function gets as an argument an instance of Graph
  and a startpoint of type Node.
  It returns the graph's spanning tree of type ComposanteConnexe
  while using Prim's algorithm beginning at the node startpoint
"""
    function prim_alg(graph::Graph{Node{T}, Z}, startpoint::Node{T}) where {T, Z}
        nodes_gr = nodes(graph)
        minimum_spanning_tree = ComposanteConnexe{Node{T}, Z}(startpoint, [startpoint], [])

        min_weights= Dict{Node{T}, Float64}()
        parents = Dict{Node{T}, Node{T}}()

        # un attribut min_weight -> Initialement, min_weight = -1 ; Initialement, le parent de chaque noeud est nothing
        for node in nodes_gr
            # everything will be added to the dictionary
            min_weights[node] = Inf
        end

        #débute en un noeud source s(startpoint) choisi par l'utilisateur et l'attribut min_weight de s est 0 ;
        #startpoint.min_weight = 0
        min_weights[startpoint] = 0

        # Initialisation de l'ensemble inTree pour suivre les nœuds inclus dans l'arbre de couverture minimale
        inTree =  Set{Node{T}}()

        #Priorityqueue contains all the nodes in increasing min weight that can be added to the minimum spanning tree.
        pq = PriorityQueue{NodeKey{T,Z},Int}()
        it=0
        edge_nodekey=EdgeOriented(startpoint, startpoint, 0.0)
        enqueue!(pq,NodeKey(startpoint,edge_nodekey,it),min_weights[startpoint])

        while !isempty(pq)

```

```
        #we add to it every time so that the same nodes can be added to the queue
different times but they won't be exactly the same
            it+=1
            node_key = dequeue!(pq)
            #we need to get the edge and the node
            w, h = node_key.node, node_key.edge
            #w = dequeue!(pq).node
            #println(w)

            # Si le nœud est déjà inclus dans l'arbre, passez au suivant
            if  w in inTree
                continue
            end

            #we push both in node and in the minimum spanning tree
            push!(inTree,w)
            add_node_and_edge!(minimum_spanning_tree, w, h)


            # Iterate through all adjacent nodes of w
            for edge in get_oriented_edges(graph, w)
                #println(min_weights)
                u= ifelse(debut(edge) == w, fin(edge), debut(edge))  # Trouver
l'autre nœud connecté par l'arête
                weight= poids(edge)
                #println(u,weight)


                # If v is not in min. spanning tree and the weight of (u, v) is
smaller than the current key of v
                if u ∉ inTree && weight < min_weights[u]
                    min_weights[u] = weight
                    parents[u] = w

                    enqueue!(pq, NodeKey(u,edge,it), min_weights[u])

                    #add edge to minimum spanning tree
                    #add_node_and_edge!(minimum_spanning_tree, u, edge)
                end
            end
        end

        #we delete the initial zero value we gave for initializing with the startpoint
        component_idx = findfirst(==(startpoint), nodes(minimum_spanning_tree))
        deleteat!(nodes(minimum_spanning_tree), component_idx)

        #we delete the initial zero value we gave for initializing with the edges
        component_idx = findfirst(==(edge_nodekey), edges(minimum_spanning_tree))
        deleteat!(edges(minimum_spanning_tree), component_idx)
        return minimum_spanning_tree
    end


#This will be tested with the following example
```

```
#everything that is necessary will be included

# Initializing nodes from example from laboratories: already done above

# Initializing edges from example from laboratories: already done above

# Initializing graph from example from laboratories: already done above

#Initializing expected prim connected components

#calling up the written function to get a result
result= prim_alg(G,noeud1)

#testing wether they are the same
testing_components_equal(result, kruskal_component)
print(result)
end

#This gives an error for an undifined G, when looking in the codes of visual studio
```

Multiple definitions for noeud1, G, edge1, noeud2, noeud3, noeud4 and edge2

Combine all definitions into a single reactive cell using a `begin ... end` block.

```
#Fifth exercise of Phase2
# Testing node.jl
begin
    noeud1 = Node("James", "ahooj")
    @test name(noeud1) == "James"
    @test data(noeud1) == "ahooj"

    noeud2 = Node("Kirk", "guitar")
    noeud3 = Node("Lars", 2)
    noeud4 = Node("Lars", "char")


    # Testing edge.jl
    edge1 = Edge(noeud1, noeud2, 5)
    @test node1(edge1) == noeud1
    @test node2(edge1) == noeud2
    @test poids(edge1) == 5

    @test_throws MethodError Edge(noeud1, noeud3, 5)

    oriente_edge = EdgeOriented(noeud1, noeud2, 3)
    @test debut(oriente_edge) == noeud1
    @test fin(oriente_edge) == noeud2
    @test poids(oriente_edge) == 3

    edge2 = Edge(noeud1, noeud2, 5)

    # Testing graph.jl
    G = Graph("Ick", [noeud1, noeud2, noeud4], [edge1, edge2])

    @test name(G) == "Ick"
    @test nodes(G) == [noeud1, noeud2, noeud4]
    @test edges(G) == [edge1, edge2]

    edge_oriented_1 = EdgeOriented(noeud1, noeud2, 5)
    edge_oriented_2 = EdgeOriented(noeud2, noeud4, 4)

    C = ComposanteConnexe(noeud1, [noeud1, noeud2, noeud4], [edge_oriented_1,
    edge_oriented_2])

    @test nodes(C) == [noeud1, noeud2, noeud4]
    @test edges(C) == [edge_oriented_1, edge_oriented_2]

    @test isless(edge_oriented_1, edge_oriented_2) == false
    @test <(edge_oriented_1, edge_oriented_2) == false
    @test >(edge_oriented_1, edge_oriented_2) == true

    v = [edge_oriented_1, edge_oriented_2, edge_oriented_2]
    sort!(v)

    @test sort([edge_oriented_1, edge_oriented_2]) == [edge_oriented_2,
    edge_oriented_1]
```

```
51      @test convert(EdgeOriented, edge1) == EdgeOriented{Int64, Node{String}}
        (Node{String}("James", "ahooj"), Node{String}("Kirk", "guitar"), 5)
52  end
```