Maxence Gollier, Sacha Benarroch-Lelong

MTH6412B, Polytechnique Montréal, Automne 2024

Projet voyageur de commerce - Phase 2

Notre code est disponible sur ce dépôt GitHub. Il est organisé selon une structure de projet Julia. L'environnement associé est nommé `STSP`.

La deuxième phase est dédiée à la conception de structures de données et d'algorithmes pour les problèmes de recouvrement minimal par arbres.

Réimport des structures précédentes

`AbstractNode`

Type abstrait dont d'autres types de noeuds dériveront.

`Node`

Type représentant les noeuds d'un graphe.

Exemple:

```
noeud = Node("James", [π, exp(1)])
noeud = Node("Kirk", "guitar")
noeud = Node("Lars", 2)
```

`AbstractEdge`

Type abstrait dont d'autres types d'arêtes dériveront.

`Edge`

Type représentant les arêtes d'un graphe.

Exemple:

```
arête = Edge("E411", 40000, "Ottignies-Louvain-la-Neuve", "Namur")
arête = Edge("E40", 35000, "Bruxelles", "Gand")
arête = Edge("Meuse", 45000, "Liège", "Namur")
```

Construit une arête à partir d'un poids et de deux identifiants de noeud sous forme de nombre.

Construit une arête à partir d'un poids et de deux identifiants de noeud.

`adjacency`

Construit un dictionnaire d'adjacence à partir d'une liste d'arêtes.

Exemple:

```
arête1 = Edge("E19", 50000, "Bruxelles, "Anvers")
arête2 = Edge("E40", 35000, "Bruxelles", "Gand")
arête3 = Edge("E17", 60000, "Anvers", "Gand")
arêtes = Vector{Edge{Int}}[arête1,arête2,arête3]
adjacency(arêtes) =
    ("Bruxelles" => [("Anvers", 50000), ("Gand", 35000)], "Gand" => [("Br
uxelles", 35000), ("Anvers", 60000)], "Anvers" => [("Bruxelles", 50000), ("Ga
nd", 60000)]).
```

`add_adjacency!`

```
add_adjacency!(adjacency::Dict{String, Vector{Tuple{String, U}}}, edge::Edge
{U})
```

Ajoute une arête à un dictionnaire d'adjacence.

`AbstractGraph`

Type abstrait dont d'autres types de graphes dériveront.

`Graph`

Type representant un graphe comme un ensemble de noeuds et d'arêtes.

Exemple :

```
node1 = Node("Joe", 3.14)
node2 = Node("Steve", exp(1))
node3 = Node("Jill", 4.12)
edge1 = Edge("Joe-Steve", 2, "Joe", "Steve")
edge1 = Edge("Joe-Jill", -5, "Joe", "Jill")
G = Graph("Ick", [node1, node2, node3], [edge1,edge2])
```

De façon interne, les noeuds et les arêtes sont stockés en tant que dictionnaire. Ceci permet de retrouver des noeuds/arêtes rapidement à partir de leurs identifiants. De plus, l'adjacence est stockée en tant que dictionnaire ce qui permet facilement d'accéder aux voisins d'un noeud. Attention, tous les noeuds doivent avoir des données de même type. Toutes les arêtes doivent également avoir des données du même type mais pas nécessairement le même type que celui des noeuds. De plus, les noms des noeuds et des arêtes doivent être uniques.

```
Graph(name::String, nodes::Vector{Node{T}}, edges::Vector{Edge{U}})
```

Construit un graphe à partir d'une liste de noeud et d'arêtes.

`read_header`

read_header(filename::String)

Analyse un fichier .tsp et renvoie un dictionnaire avec les données de l'entête.

`read_nodes`

read_nodes(header::Dict{String}{String}, filename::String)

Analyse un fichier .tsp et renvoie une liste d'objets de type `Node`. Si les coordonnées ne sont pas données, les noeuds sont instanciés avec leur identifiant et `NaN`. Le nombre de noeuds est dans header["DIMENSION"].

`n_nodes_to_read`

n*nodes*to_read(format::String, n::Int, dim::Int)

Fonction auxiliaire de read_edges, qui détermine le nombre de noeud à lire en fonction de la structure du graphe.

`read_edges`

read_edges(header::Dict{String}{String}, filename::String)

Analyse un fichier .tsp et renvoie une liste d'arêtes sous forme brute de tuples.

`read_stsp`

read_stsp(filename::String; quiet::Bool=true)s

Lit un fichier `.tsp` et instancie un objet `Graph` correspondant après avoir construit ses noeuds et ses arêtes.

`plot_graph`

Affiche un graphe étant donnés 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")
```

Affiche un graphe étant donnés 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")
```

Affiche un graphe étant donnés 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")
```

Trace un graphe directement depuis un objet `Graph`.

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

Trace un graphe directement depuis un objet `Graph`.

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

Affiche un graphe étant donnés 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")
```

Trace un graphe directement depuis un objet `Graph`.

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

Affiche un graphe étant donnés 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")
```

Affiche un graphe étant donnés 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")
```

Affiche un graphe étant donnés 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")
```

Affiche un graphe étant donnés 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")
```

Affiche un graphe étant donnés 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")
```

Affiche un graphe étant donnés 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")
```

Affiche un graphe étant donnés 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")
```

Affiche un graphe étant donnés 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")
```

Affiche un graphe étant donnés 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")
```

Affiche un graphe étant donnés 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")
```

Affiche un graphe étant donnés 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")
```

Affiche un graphe étant donnés 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")
```

plot_graph

Affiche un graphe étant donnés 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")
```

Affiche un graphe étant donnés 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")
```

Affiche un graphe étant donnés 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")
```

Trace un graphe directement depuis un objet `Graph`.

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

Trace un graphe directement depuis un objet `Graph`.

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

Affiche un graphe étant donnés 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")
```

Trace un graphe directement depuis un objet `Graph`.

`plot_graph`

Affiche un graphe étant donnés 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")
```

Affiche un graphe étant donnés 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")
```

Affiche un graphe étant donnés 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")
```

Trace un graphe directement depuis un objet `Graph`.

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

Trace un graphe directement depuis un objet `Graph`.

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

Affiche un graphe étant donnés 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")
```

Trace un graphe directement depuis un objet `Graph`.

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

Composantes connexes : structure de données

Nous choisissons la structure d'arbre pour représenter les composantes connexes d'un graphe. Notons que cela est restrictif : une composante connexe n'est pas nécessairement acyclique. Cependant, en vue d'une implémentation de l'algorithme de Kruskal, cette structure semble la plus pertinente.

Nous définissons une forêt comme un ensemble d'arbres.

`Tree`

Type representant un arbre comme l'identifiant d'un parent et une taille d'arbre.

L'identifiant d'un parent dérive de l'identifiant du noeud d'un graphe. La taille d'un arbre est définie comme le nombre d'arbres qui ont cet arbre comme parent. Ce type est principalement utile pour le type `Forest`.

`Forest`

Type representant une forêt comme un ensemble d'identifiants de noeuds pointant vers des arbres.

Les identifiants dérivent des identifiants des noeuds d'un graphe. A partir de l'identifiant du noeud d'un graphe, cette structure permet de trouver l'arbre associé dans le dictionnaire `trees`. Si l'identifiant d'un noeud pointe vers un arbre dont le parent a le même identifiant, alors ce noeud est une "racine". Les "racines" de l'arbre sont utiles pour les fusionner et pour vérifier l'existence de cycles.

Voir la documentation de la fonction `merge` pour plus de détails.

Le nombre de "racines" contenue dans la forêt est également stockée dans l'attribut `num_roots`.

```
Forest(G)
```

Initialise une forêt de composantes connexes à partir d'un graphe. Un arbre de taille 1 est créé par noeud du graphe. Cette fonction est conçue pour servir à l'initialisation de l'algorithme de Kruskal.

Arguments

- `G (Graph)`: le graphe à partir duquel construire une forêt de composantes connexes

Type de retour

`Forest`

Utilitaires nécessaires à l'algorithme de Kruskal

`find`

```
find(forest, node_id)
```

Retourne la racine de l'arbre associé au noeud d'identifiant `node_id` dans la forêt `forest`. Itère de parent en parent jusqu'à trouver un identifiant dont le parent est lui-même.

Arguments

- `forest (Forest)`: forêt dans laquelle rechercher l'arbre auquel est rattaché le noeud
- `node_id (String)`: identifiant du noeud à rechercher dans la forêt

Type de retour

`String`

Exemple

```
julia> find("24", forest)
"7"
# Le noeud d'identifiant "24" est contenu dans l'arbre de la forêt dont la ra
cine est d'identifiant "7".
```

`merge!`

```
merge!(forest, root_id1, root_id2)
```

Fonction permettant de fusionner deux arbres à partir de deux identifiants qui ont la propriété d'être des "racines". La fusion s'opère en redéfinissant le parent d'une "racine" par l'autre "racine". Pour choisir quelle racine prend l'autre racine comme enfant, la taille des arbres associé est comparée ; l'arbre de plus grande taille englobe l'autre. Ce choix est justifié ici.

Arguments

- `forest (Forest)`: forêt dans laquelle se trouvent les 2 arbres à fusionner
- `root_id1 (String)`: identifiant de la racine du premier arbre participant à la fusion
- `root_id2 (String)`: identifiant de la racine du second arbre participant à la fusion

Type de retour

Aucun : fonction *in-place*.

Implémentation de l'algorithme de Kruskal

`kruskal`

```
kruskal(G)
```

Implémentation de l'algorithme de Kruskal pour identifier un arbre de recouvrement minimal d'un graphe. Renvoie un tuple contenant le coût et une liste des arêtes formant l'arbre de poids minimal. Si le graphe n'est pas connexe, une erreur est renvoyée.

Arguments

- `G (Graph)`: le graphe dans lequel il faut identifier un arbre de recouvrement minimal

Type de retour

`Float64, Vector{Edge}`

Exemples

```
julia> kruskal(graph)
```

Test de l'algorithme sur l'exemple des notes de cours

```
nodes =
 [Node("a", Int64[-2, 0]), Node("b", [-1, 1]), Node("c", [0, 1]), Node("d", [1, 1]), Node("e",
```

```
nodes = [
    Node("a", [-2, 0]),
    Node("b", [-1, 1]),
    Node("c", [0, 1]),
    Node("d", [1, 1]),
    Node("e", [2, 0]),
    Node("f", [1, -1]),
    Node("g", [0, -1]),
    Node("h", [-1, -1]),
    Node("i", [-1, 0])
]
```

```
edges =
```
```
[Edge("a-b", 4, "a", "b"), Edge("a-h", 8, "a", "h"), Edge("b-c", 8, "b", "c"), Edge("b-h", 1
```

```
edges = [
    Edge("a", "b", 4),
    Edge("a", "h", 8),
    Edge("b", "c", 8),
    Edge("b", "h", 11),
    Edge("c", "d", 7),
    Edge("c", "i", 2),
    Edge("c", "f", 4),
    Edge("d", "e", 9),
    Edge("d", "f", 14),
    Edge("e", "f", 10),
    Edge("f", "g", 2),
    Edge("g", "i", 6),
    Edge("g", "h", 1),
    Edge("h", "i", 7)
]
```

```
example_graph =
```
```
Graph("example", Dict("i" ⇒ Node("i", [ more]), "f" ⇒ Node("f", [ more]), "g" ⇒ Node(
```

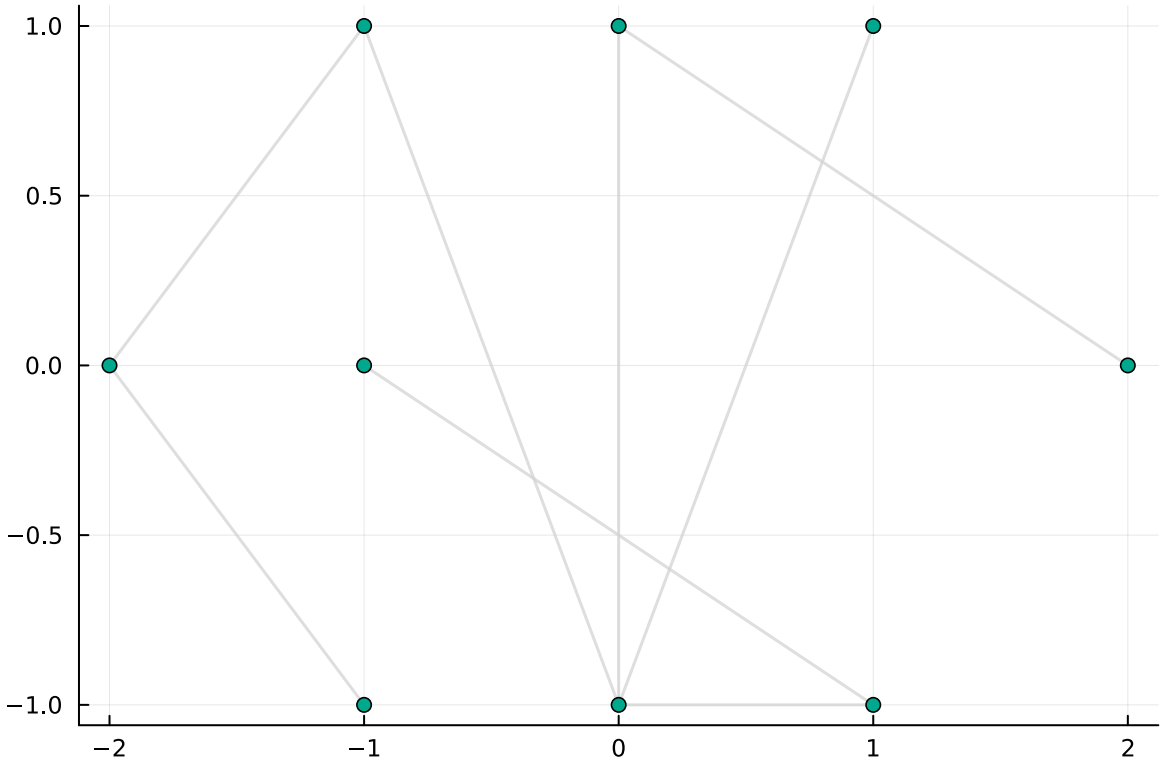
```
example_graph = Graph("example", nodes, edges)
```

```
example_kruskal =
```
```
(37, [Edge("g-h", 1, "g", "h"), Edge("c-i", 2, "c", "i"), Edge("f-g", 2, "f", "g"), Edge("c-
```

```
example_kruskal = kruskal(example_graph)
```

```
graph_kruskal =
```
```
Graph("Example after Kruskal", Dict("i" ⇒ Node("i", [ more]), "f" ⇒ Node("f", [ more]
```

```
graph_kruskal = Graph("Example after Kruskal", nodes, example_kruskal[2])
```




```
1 plot_graph(graph_kruskal, "alph")
```

Tests sur des instances de STSP

```
dantzig42 =
```

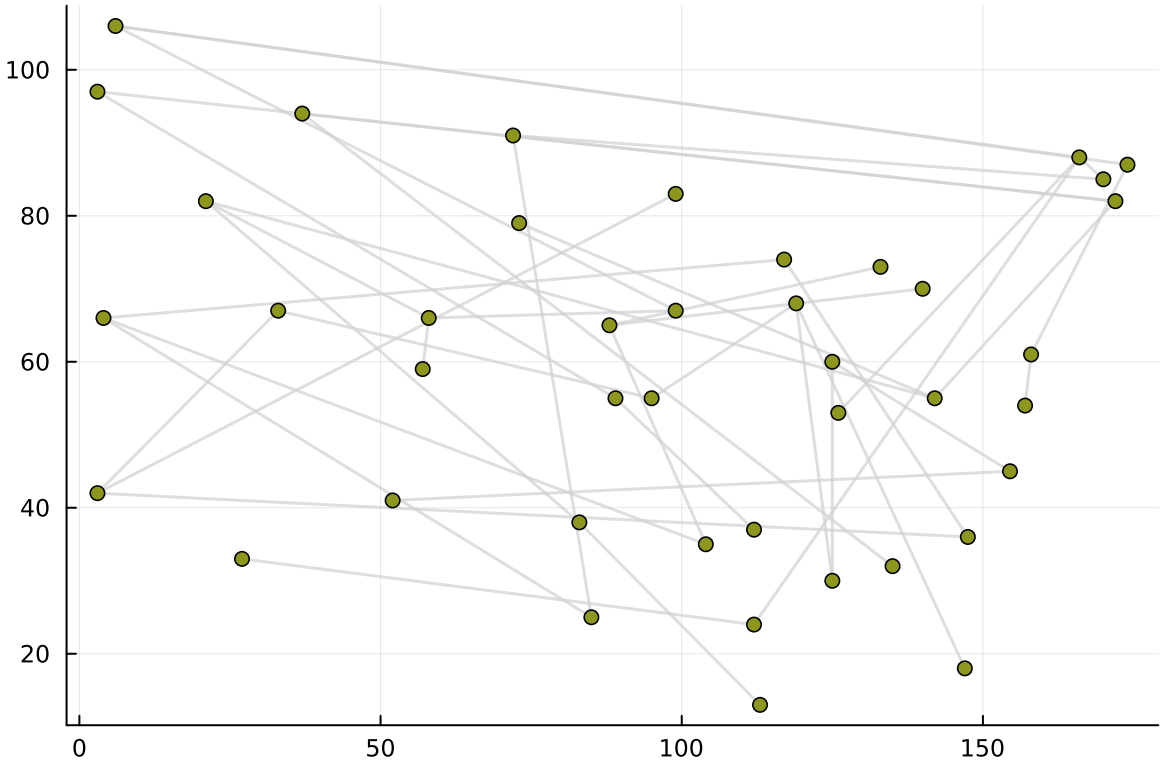
```
Graph("dantzig42", Dict("24" ⇒ Node("24", [ more]), "4" ⇒ Node("4", [ more]), "1" ⇒ N
```

```
1 dantzig42 = read_stsp("/Users/sacha/dev/optim/mth6412b-starter-code/instances/stsp/
  dantzig42.tsp")
```

```
d42_kruskal =
```

```
(591.0, [Edge("1-41", 3.0, "1", "41"), Edge("26-27", 3.0, "26", "27"), Edge("1-42", 5.0, "1
```

```
1 d42_kruskal = kruskal(dantzig42)
```



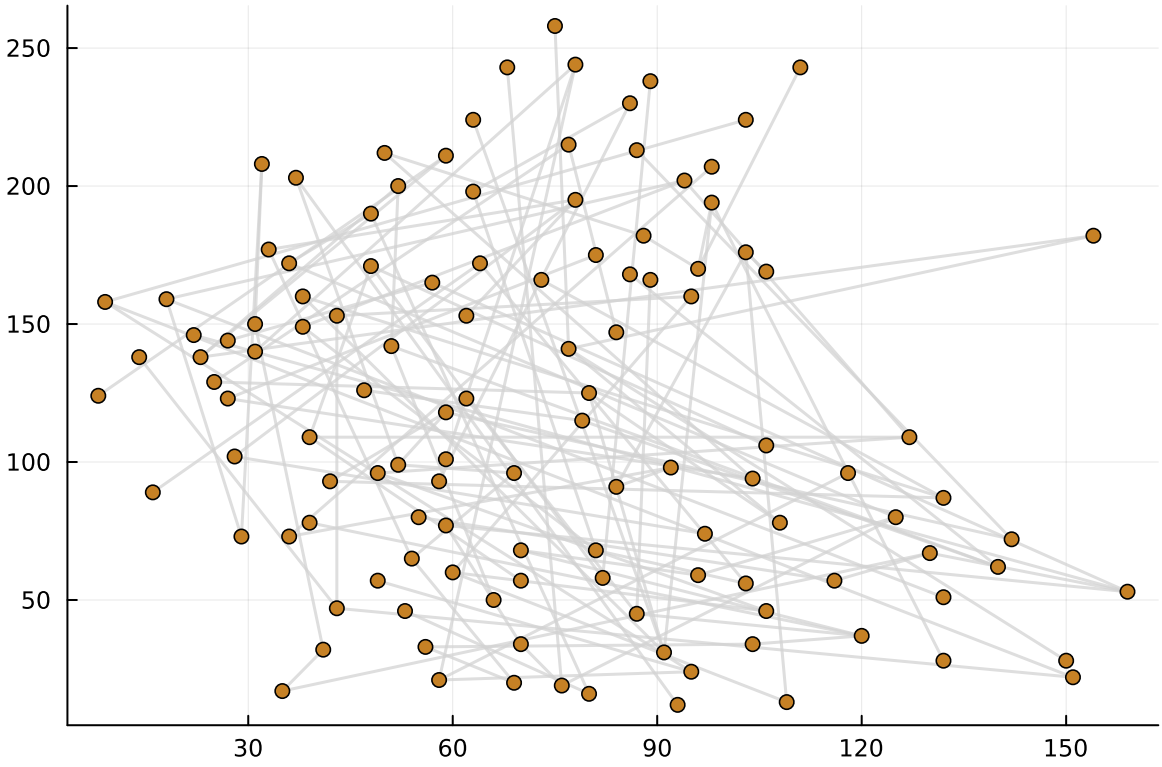

```
1 plot_graph(Graph("d42_kruskal", collect(values(dantzig42.nodes)), d42_kruskal[2]))
```

```
gr120 =
  Graph("gr120", Dict("32" ⇒ Node("32", [ more]), "29" ⇒ Node("29", [ more]), "1" ⇒ Nod
```

```
1 gr120 = read_stsp("/Users/sacha/dev/optim/mth6412b-starter-code/instances/stsp/
  gr120.tsp")
```

```
g120_kruskal =
  (5805.0, [Edge("70-116", 12.0, "70", "116"), Edge("42-98", 19.0, "42", "98"), Edge("47-71",
```

```
1 g120_kruskal = kruskal(gr120)
```




```
1 plot_graph(Graph("g120_kruskal", collect(values(gr120.nodes)), g120_kruskal[2]))
```

L'affichage de ces graphes permet au moins de constater que l'on obtient des structures connexes.