Bioinformatics III

Fourth Assignment

Thibault Schowing (2571837)
Wiebke Schmitt (2543675)

May 9, 2018

Exercise 4.1: Dijkstra's algorithm for finding shortest paths

(a) *Draw a directed or undirected graph with at least one negative edge weight for which Dijkstra's algorithm does not find the shortest path from some node s to another node t. Use your example to explain why Dijkstra's algorithm only works on graphs with non–negative edge weights.*

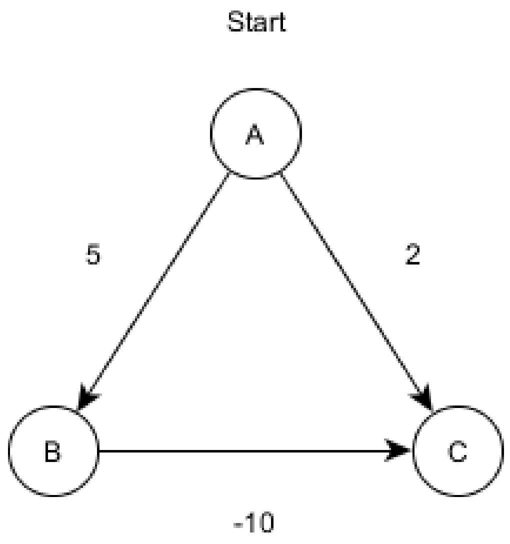

Figure 1: Example directed graph

A-B-C is the shortest path.

We have $V = A, B, C$, $E = (A, C, 2), (A, B, 5), (B, C, -10)$. So, A-C is found first but not A-B-C. Dijkstra assume that the minimality will never change when "closing" a node. The use of negative numbers change this rule and so is not compatible with Dijkstra.

(b) *Dijkstra's algorithm modifications*

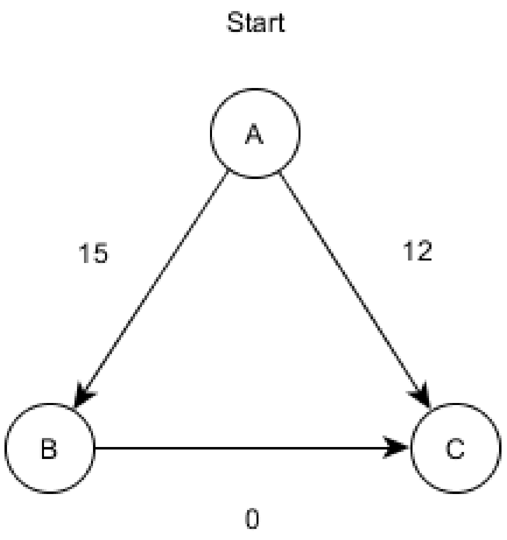

Figure 2: Graph with edge cost ≥ 0.

We can see that adding 10 to all weights doesn't work. A-C is still the path that Dijkstra find first and in this case, it is the shortest path which was not the case before.

(c) *Could BFS be used to find the shortest paths between nodes? If so, what would the edge weights have to look like for BFS to be guaranteed to find the shortest paths between nodes? Why (not)?*

With BFS, we can find a path with arbitrary weights but it is not guaranteed to be optimal.

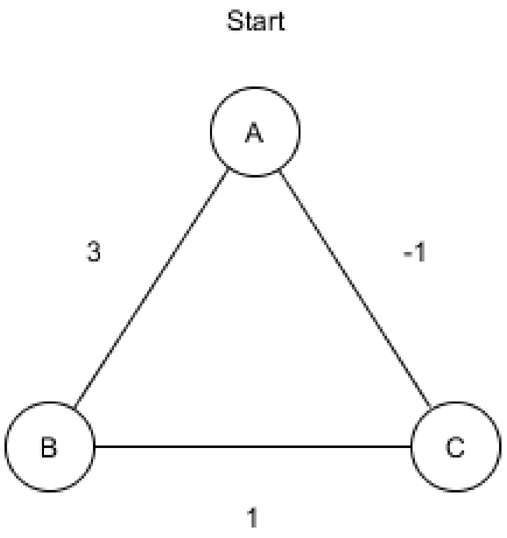

Figure 3: When we want to go from A to B, the shortest path is A - C - B, but with BFS we will first expand A and find the path A-B. When we expand the C node, B is already marked as "duplicate". BFS will ignore the link between B and C in this case.

Exercise 4.2: Force directed layout of networks

(a) *Implementation preparation*

For the harmonic force we have:

$$F_h(\vec{r}) = -\nabla E_h(\vec{r})$$

According to the definition:

$$= -\nabla \frac{k}{2} \|\vec{r}\|^2 = -\frac{k}{2} \nabla \|\vec{r}\|^2$$

$$= -\frac{k}{2}\nabla(\sqrt{x^2+y^2+z^2})^2$$

$$= -\frac{k}{2}\nabla(x^2+y^2+z^2)$$

$$= -\begin{pmatrix} kx \\ ky \\ kz \end{pmatrix}$$

For the Coulomb force we have:

$$F_c(\vec{r}) = -\nabla E_c(\vec{r})$$

$$= -\nabla(\frac{1}{4\pi\epsilon_0}\frac{q1q2}{\|\vec{r}\|})$$

$$= -\nabla(\frac{1}{4\pi\epsilon_0}\frac{q1q2}{\sqrt{x^2+y^2+z^2}})$$

$$= -\frac{q1q2}{4\pi\epsilon_0}\nabla(\frac{1}{\sqrt{x^2+y^2+z^2}})$$

We apply partial derivatives on $\frac{1}{\sqrt{x^2+y^2+z^2}}$ (chain rule):

$$= -\frac{q1q2}{4\pi\epsilon_0}\begin{pmatrix} -\frac{x}{(x^2+y^2+z^2)^{\frac{3}{2}}} \\ -\frac{y}{(x^2+y^2+z^2)^{\frac{3}{2}}} \\ -\frac{z}{(x^2+y^2+z^2)^{\frac{3}{2}}} \end{pmatrix} = \frac{q1q2}{4\pi\epsilon_0}\begin{pmatrix} \frac{x}{(x^2+y^2+z^2)^{\frac{3}{2}}} \\ \frac{y}{(x^2+y^2+z^2)^{\frac{3}{2}}} \\ \frac{z}{(x^2+y^2+z^2)^{\frac{3}{2}}} \end{pmatrix}$$

(b) ***Adapting the energy equations for networks***

For the Harmonic force we have:

$$F_h(\vec{r}_{ij}) = -\nabla E_h(\vec{r}_{i,j}) = -\begin{pmatrix} x_i - y_j \\ y_i - y_j \end{pmatrix}$$

For the Coulomb force:

$$F_c(\vec{r_{ij}}) = k_1 k_2 \begin{pmatrix} \frac{x_i - x_j}{((x_i-x_j)^2+(y_i-y_j)^2)^{\frac{3}{2}}} \\ \frac{y_i - y_j}{((x_i-x_j)^2+(y_i-y_j)^2)^{\frac{3}{2}}} \end{pmatrix}$$

(c) ***Understanding the Coulomb and harmonic energy:*** *How does the Coulomb energy and harmonic energy change if the degree of both nodes is increased or decreased? What happens if the distance between two nodes is increased or decreased?*

If the degree of both nodes increases, the harmonic force is not affected but Coulomb force will increase with a factor $-k_1k_2$ as seen in (b).

However, if the distance increase, both Coulomb and harmonic will be affected. The Harmonic energy is $\frac{1}{2}||\vec{r^2}||^2$ and the Coulomb $\frac{k_ik_j}{||\vec{r}_{ij}||}$ so if we increase the distance, Coulomb energy is going to decrease and Harmonic energy to increase.

(d) ***Understanding the forces: Why is the Coulomb force the repulsive force and the harmonic force the attractive force?***

In our case, all the node will have a positive charge (the degree) which means that they are all going to repel each other. However, the objective is to keep connected nodes close to each other and at the same time spread the rest of the graph away to give it a nice display. The Coulomb force is the basic force between every node according to their degrees and distance, disregarding their connection. The Harmonic force is applied here only when two nodes are connected to each other and because it has opposite sign, as seen in (b), it will temper the force between the two connected nodes and allow to keep them close ot each other.

(e) ***Implementing the force directed layout algorithm***

Listing 1: layout˙main.py

```python
from layout import Layout
from tools import plot_layout, plot_energies


file_paths = ['star.txt', 'square.txt', 'star++.txt', 'dog.txt']

for file_path in file_paths:
    # read the file into your layout class

    layout = Layout(file_path)

    # run the normal layout for 1000 iterations and store the total energies
    # plot the normal layout

    energies1 = layout.layout(1000)
    plot_layout(layout, file_path + " - Normal Layout")

    # run the simulated annealing layout for 1000 iterations and store the
        total energies
    # plot the simulated annealing layout

    energies2 = layout.simulated_annealing_layout(1000)
    plot_layout(layout, file_path + " - Simulated Annealing Layout")

    # plot the total energies of the normal layout and the simulated annealing
        layout

    plot_energies([energies1, energies2], ["Normal layout", "Simulated 
        Annealing Layout"], "Layout energies")
    pass
```

Listing 2: layout.py

```python
from random import gauss
import math
import random
import itertools
from generic_network import GenericNetwork


class Layout:
    def __init__(self, file_path):
        """
        :param file_path: path to a white-space-separated file that contains
            node interactions
        """
        # create a network from the given file
        self.network = GenericNetwork()
        self.network.read_from_tsv(file_path)
        # friction coefficient
        self.alpha = 0.03
        # random force interval
        self.interval = 0.3
        # initial square to distribute nodes
        self.size = 50


    def init_positions(self):
        """
        Initialise or reset the node positions, forces and charge.
        """

        netsize = len(self.network.nodes)
        # Set up the positions and charge
        for key, node in self.network.nodes.items():

            #Pick a coordinate between 0 and 50 (initial square)
            node.pos_x = random.randint(1,self.size + 1)
            node.pos_y = random.randint(1,self.size + 1)
            #print("Random posx: ", node.pos_x, " and posy ", node.pos_y)

            node.charge = node.degree()

        # Calculate the force
        self.calculate_forces()


    def calculate_forces(self):
        """
        Calculate the force on each node during the current iteration.
        """

        # For all pair of nodes...
        for pair in itertools.combinations(self.network.nodes.items(),2):

            node1 = pair[0][1]
            node2 = pair[1][1]

            coulombx = (node1.charge * node2.charge) * ((node1.pos_x - node2.
                pos_x)/((node1.pos_x - node2.pos_x)**2 + (node1.pos_y - node2.
                pos_y)**2)**(3/2))
            coulomby = (node1.charge * node2.charge) * ((node1.pos_y - node2.
                pos_y)/((node1.pos_x - node2.pos_x)**2 + (node1.pos_y - node2.
                pos_y)**2)**(3/2))
            harmonicx = 0.0
            harmonicy = 0.0

            # If the nodes are connected, we temper the force with the
                harmonic
            if node1.has_edge_to(node2):
                harmonicx = -(node1.pos_x - node2.pos_x)
                harmonicy = -(node1.pos_y - node2.pos_y)

            # Add the force to node1 .... opposite to node2
            fx = coulombx + harmonicx
            fy = coulomby + harmonicy

            node1.force_x += fx
            node1.force_y += fy

            node2.force_x -= fx
            node2.force_y -= fy


    def add_random_force(self, temperature):
        """
        Add a random force within [- temperature * interval, temperature *
            interval] to each node.
        (There is nothing to do here for you.)
        :param temperature: temperature in the current iteration
        """
        for node in self.network.nodes.values():
            node.force_x += gauss(0.0, self.interval * temperature)
            node.force_y += gauss(0.0, self.interval * temperature)

    def displace_nodes(self):
        """
        Change the position of each node according to the force applied to it
            and reset the force on each node.
        """
        for node in self.network.nodes.values():
            node.pos_x = node.pos_x + node.force_x * self.alpha
            node.pos_y = node.pos_y + node.force_y * self.alpha

            # Reset the forces to 0
            node.force_x = 0
            node.force_y = 0


    def calculate_energy(self):
        """
        Calculate the total energy of the network in the current iteration.
        :return: total energy
        """

        energy_total = 0

        for pair in itertools.combinations(self.network.nodes.values(), 2):
            node1 = pair[0]
            node2 = pair[1]

            # Coulomb energy
            Ec = (node1.degree() * node2.degree()) / (math.sqrt(((node1.pos_x
                - node2.pos_x)**2) + ((node1.pos_y - node2.pos_y)**2)))

            #Harmonic energy
            Eh = 0
            if node1.has_edge_to(node2):
                Eh = ((node1.pos_x - node2.pos_x)**2 + (node1.pos_y - node2.
                    pos_y)**2)/2
            energy_total += Ec + Eh


        return energy_total

    def layout(self, iterations):
        """
        Executes the force directed layout algorithm. (There is nothing to do
            here for you.)
        :param iterations: number of iterations to perform
        :return: list of total energies
        """
        # initialise or reset the positions and forces
        self.init_positions()
        energies = []

        for _ in range(iterations):
            self.calculate_forces()
            self.displace_nodes()
            energies.append(self.calculate_energy())

        return energies

    def simulated_annealing_layout(self, iterations):
        """
        Executes the force directed layout algorithm with simulated annealing.
        :param iterations: number of iterations to perform
        :return: list of total energies
        """
        self.init_positions()
        energies = []
        temperature = 100000
        for i in range(iterations):
            # TODO: DECREASE THE TEMPERATURE IN EACH ITERATION. YOU CAN BE
                CREATIVE.
            temperature = 0.2 * temperature
            # there is nothing to do here for you
            self.calculate_forces()
            self.add_random_force(temperature)
            self.displace_nodes()
            energies.append(self.calculate_energy())

        return energies
```

Listing 3: node.py

```
class Node:
    def __init__(self, identifier):
        self.identifier = identifier
        # contains the identifiers of other nodes connected to this node
        self.neighbour_nodes = set()

        # fields for the layout algorithm
        self.pos_x = 0.0
        self.pos_y = 0.0
        self.force_x = 0.0
        self.force_y = 0.0
        self.charge = 0

    def __eq__(self, node):
        """
        :param node: Node-object
        :return: True if the other node has the same identifier, False
            otherwise
        """
        return self.identifier == node.identifier

    def __str__(self):
        """
        :return: string representation of the node identifier
        """
        return str(self.identifier)

    def has_edge_to(self, node):
        """
        :param node: Node-object
        :return: True if this node has an edge to the other node, False
            otherwise
        """
        return node.identifier in self.neighbour_nodes

    def add_edge(self, node):
        """
        Adds an edge to the other node by adding it to the neighbour-nodes.
        :param node: Node-object
        """
        self.neighbour_nodes.add(node.identifier)

    def remove_edge(self, node):
        """
        Removes the edge to the other node, if that edge exists, by removing
            the other node from the neighbour nodes.
        :param node: Node-object
        """
        self.neighbour_nodes.discard(node.identifier)

    def degree(self):
        """
        :return: the degree of this node (= number of neighbouring nodes)
        """
        return len(self.neighbour_nodes)
```

Listing 4: tools.py

```
import matplotlib.pyplot as plt
from itertools import combinations


def plot_layout(layout, title):
    """
    Plots the layout of a network.
    :param layout: Layout-object
    :param title: plot title
    """
    for node_1, node_2 in list(combinations(layout.network.nodes.values(), 2))
        :
        if layout.network.has_edge(node_1, node_2):
            # plot the edge between the two nodes, if it exists
            plt.plot([node_1.pos_x, node_2.pos_x], [node_1.pos_y, node_2.pos_y
                ], linestyle='-', color='black')

    # plot the nodes
    for node in layout.network.nodes.values():
        plt.plot(node.pos_x, node.pos_y, marker='H', color='red')

    # set the title, clean up the plot layout and show it
    plt.title(title)
    plt.tight_layout()
    plt.show()
    plt.clf()


def plot_energies(energy_lists, legend, title):
    """
    Plots list(s) of total energies.
    :param energy_lists: a list that contains a list of total energies
    :param legend: a list of curve labels
    :param title: plot title
    """
    # plot each list of total energies
    for energy_list in energy_lists:
        plt.plot(energy_list)

    # set the x-axis and y-axis labels
    plt.xlabel('iteration')
    plt.ylabel('Total Energy')

    # set the legend, title, clean up the plot layout and show it
    plt.legend(legend)
    plt.title(title)
    plt.tight_layout()
    plt.show()
    plt.clf()
```

Listing 5: generic˙network.py

```
from node import Node


class GenericNetwork:
    def __init__(self):
        # key: node identifier, value: Node-object
        self.nodes = {}

    def read_from_tsv(self, file_path):
        """
        Reads white-space-separated files that contain two or more columns.
            The first two columns contain the
        identifiers of two nodes that have an undirected edge. The two nodes
            are added to the network.
        :param file_path: path to the file
        """
        # clear the prior content of the network
        self.nodes = {}

        # open the file for reading
        with open(file_path, 'r') as file:
            # iterate over the lines in the file
            for line in file:
                #
                columns = line.split()

                # skip lines that do not have two node identifiers
                if len(columns) < 2:
                    continue

                # create the two nodes and remove potential whitespace such as
                    new-line from their identifiers
                node_1 = Node(columns[0].strip())
                node_2 = Node(columns[1].strip())
                # add the nodes and the edge between them to the network
                self.add_node(node_1)
                self.add_node(node_2)
                self.add_edge(node_1, node_2)

    def add_node(self, node):
        """
        Adds the specified node to the network.
        :param node: Node-object
        """
        if node.identifier not in self.nodes.keys():
            self.nodes[node.identifier] = node

    def add_edge(self, node_1, node_2):
        """
        Adds an (undirected) edge between the two specified nodes.
        :param node_1: Node-object
        :param node_2: Node-object
        :raises: KeyError if either node is not in the network
        """
        # raise an error if the nodes are not in the network
        if node_1.identifier not in self.nodes.keys():
            raise KeyError('There␣is␣no␣node␣in␣the␣network␣with␣identifier:',
                node_1)
        if node_2.identifier not in self.nodes.keys():
            raise KeyError('There␣is␣no␣node␣in␣the␣network␣with␣identifier:',
                node_2)

        # add the (undirected) edge
        self.nodes[node_1.identifier].add_edge(node_2)
        self.nodes[node_2.identifier].add_edge(node_1)

    def get_node(self, identifier):
        """
        :param identifier: node identifier
        :return: Node-object corresponding to the given node identifier, if
            the node is in the network
        :raises: KeyError if there is no node with that identifier in the
            network
        """
        if identifier not in self.nodes.keys():
            raise KeyError('There␣is␣no␣node␣in␣the␣network␣with␣identifier:',
                identifier)
        return self.nodes[identifier]

    def has_edge(self, node_1, node_2):
        """
        :param node_1: Node-object
        :param node_2: Node-object
        :return: True if the two nodes have an (undirected) edge, False
            otherwise
        :raises: KeyError if either node is not in the network
        """
        # raise an error if the nodes are not in the network
        if node_1.identifier not in self.nodes.keys():
            raise KeyError('There is no node in the network with identifier:',
                 node_1)
        if node_2.identifier not in self.nodes.keys():
            raise KeyError('There is no node in the network with identifier:',
                 node_2)

        return node_1.has_edge_to(node_2) and node_2.has_edge_to(node_1)

    def size(self):
        """
        :return: number of nodes in the network
        """
        return len(self.nodes.keys())

    def max_degree(self):
        """
        :return: highest node degree in the network, 0 if there are no nodes
            in the network
        """
        return max([node.degree() for node in self.nodes.values()], default=0)
```

(f) ***Simulated annealing:*** *Explain why simulated annealing is a worthwhile optimisation principle in practice.*
Simulated annealing will approximate the global minimum (unlike Gradient descent, which will find a precise local minima). These methods avoid getting stuck in a local minima by making big random jumps in the beginning and reduce their size as we stabilize the layout. If a node is ”stuck” in a force-field in which it does not belong, a random force might help it to get out. If a node is already in a good position, this random force might false a little bit the scheme but won't break the overall structure. As the temperature is decreasing, the random force won't have the ability to displace the node far away of their position. The more iterations, the less amount of temperature.

(g) ***Applying the layout algorithms***

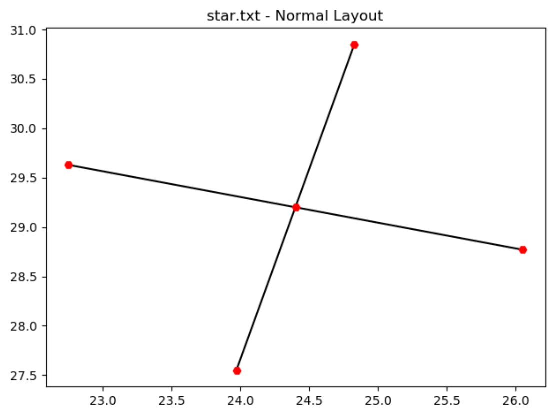

Figure 4:

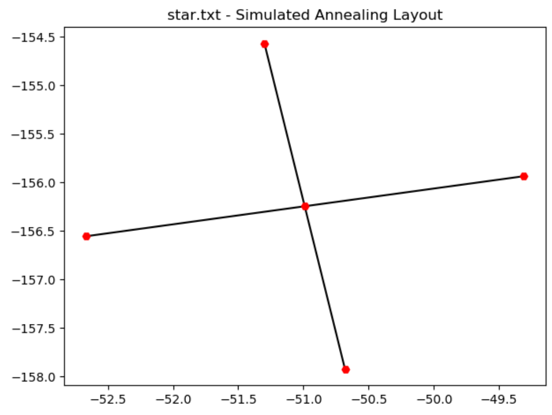

Figure 5:

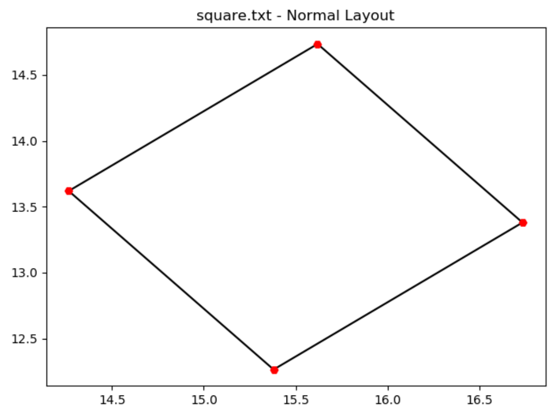

Figure 6:

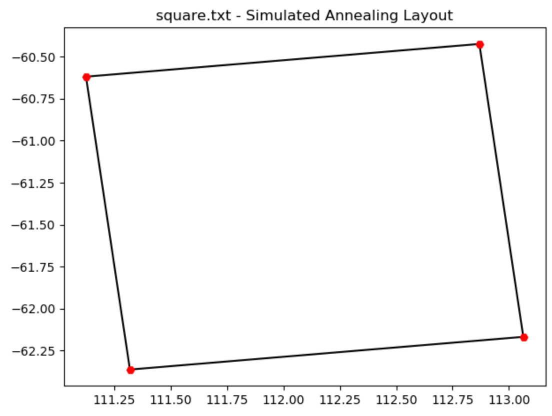

Figure 7:

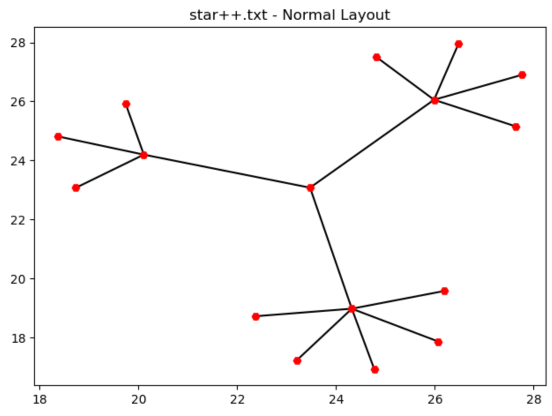

Figure 8:

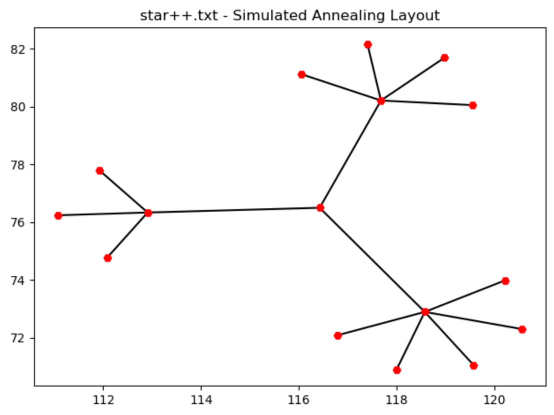

Figure 9:

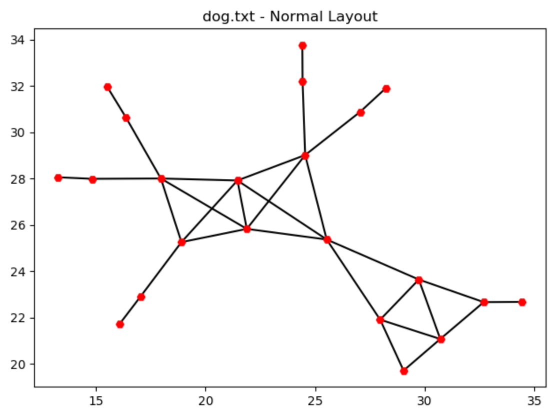

Figure 10:

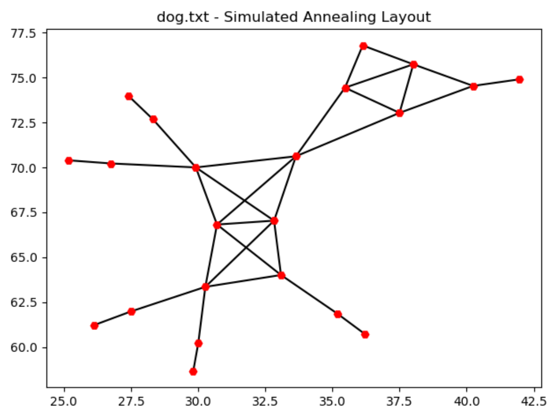

Figure 11:

Exercise 4.3: Graph Modular Decomposition

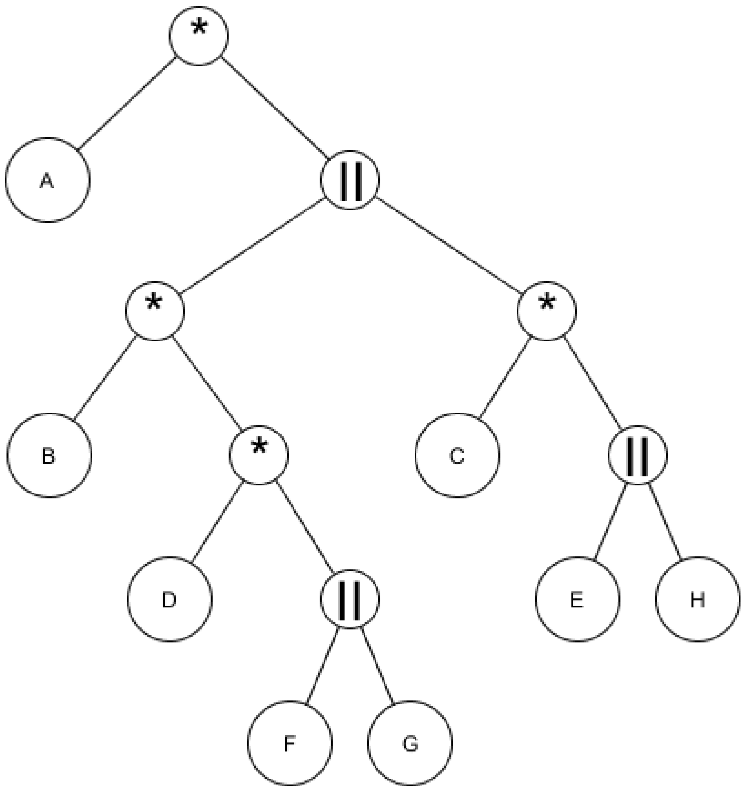

Figure 12: Modular decomposition of the network