Bioinformatics III

Fourth Assignment

Alexander Flohr (2549738)
Andrea Kupitz (2550260)

May 9, 2018

Exercise 4.1: Dijkstra's algorithm for finding shortest paths

(a) Figure 1 shows an example of a graph with one edge with negative edge weight, where Dijkstra's algorithm fails to find minimal paths.
If we take node 1 as the source node, the algorithm will chose node 4 as the second node and thus never find the shortest path $1 \rightarrow 2 \rightarrow 4$ because nodes, that have been visited, are never visited again. Thereby, the algorithm returns the path $1 \rightarrow 4$ with length 2 as shortest path from 1 to 4 whereby the path $1 \rightarrow 2 \rightarrow 4$ with length -5 would be shorter.

(b) The modified algorithm guarantees to find the shortest path, even if some edges have negative weights because by adding the absolute value of the smallest edge weight to all weights, transforms the graph in a graph with only positive edge weights. In this way the original assumption holds that the total weight of a path can never get smaller than the weight of each edge in it. Formally, $\sum_{i \in path} w_i >= w_k \forall k \in path$

(c) Breadth-first search can be applied in order to find shortest paths because it constructs a tree from the graph and visits all nodes. BFS is only guaranteed to find shortest paths if all edge weights are the same because the algorithm works with a queue and doesn't visit the nodes with minimal edge weights first like Dijkstra's algorithm. In this way BFS is only able to find path with minimal depth.

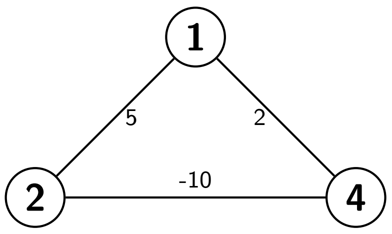

Figure 1: example of network with negative edge weight

Exercise 4.2: Force directed layout of networks

(a) Equation one and two show the force field for Coulomb energy, equation three and four for the harmonic energy.

$$\vec{F}_c(\vec{r}) = -\nabla E_c(\vec{r}) = -\frac{1}{4\pi\epsilon_0}\frac{q_1q_2}{\nabla\|\vec{r}\|} = -\frac{1}{4\pi\epsilon_0}\begin{pmatrix}\frac{\delta}{\delta x}\frac{q_1q_2}{\sqrt{x^2+y^2+z^2}}\\ \frac{\delta}{\delta y}\frac{q_1q_2}{\sqrt{x^2+y^2+z^2}}\\ \frac{\delta}{\delta z}\frac{q_1q_2}{\sqrt{x^2+y^2+z^2}}\end{pmatrix} = -\frac{1}{4\pi\epsilon_0}\begin{pmatrix}q_1q_2/\frac{x}{\sqrt{x^2+y^2+z^2}}\\ q_1q_2/\frac{y}{\sqrt{x^2+y^2+z^2}}\\ q_1q_2/\frac{z}{\sqrt{x^2+y^2+z^2}}\end{pmatrix} \tag{1}$$

$$= -\frac{1}{4\pi\epsilon_0}\begin{pmatrix}\frac{q_1q_2\sqrt{x^2+y^2+z^2}}{x}\\ \frac{q_1q_2\sqrt{x^2+y^2+z^2}}{y}\\ \frac{q_1q_2\sqrt{x^2+y^2+z^2}}{z}\end{pmatrix} = -\frac{1}{4\pi\epsilon_0}q_1q_2\sqrt{x^2+y^2+z^2}\begin{pmatrix}\frac{1}{x}\\ \frac{1}{y}\\ \frac{1}{z}\end{pmatrix} = -\frac{1}{4\pi\epsilon_0}\frac{q_1q_2\|\vec{r}\|}{\vec{r}} \tag{2}$$

$$\vec{F}_h(\vec{r}) = -\nabla E_h(\vec{r}) = -\frac{k}{2}\nabla\|\vec{r}\|^2 = -\frac{k}{2}\nabla(\sqrt{x^2+y^2+z^2})^2 = -\frac{k}{2}\nabla(x^2+y^2+z^2) \tag{3}$$

$$= -\frac{k}{2}\begin{pmatrix}2x+y^2+z^2\\ x^2+2y+z^2\\ x^2+y^2+2z\end{pmatrix} \tag{4}$$

(b) Equation five shows the force field for Coulomb energy, equation six for the harmonic energy.

$$\vec{F}_c(\vec{r}_{ij}) = -\nabla E_c(\vec{r}_{ij}) = -\frac{k_ik_j}{\nabla\|\vec{r}_{ij}\|} = -\frac{k_ik_j\|\vec{r}_{ij}\|}{\vec{r}_{ij}} \tag{5}$$

$$\vec{F}_h(\vec{r}_{ij}) = -\nabla E_h(\vec{r}_{ij}) = -\frac{1}{2}\nabla\|\vec{r}_{ij}\|^2 = -\frac{1}{2}\begin{pmatrix}2(x_i-x_j)+(y_i-y_j)^2\\ (x_i-x_j)^2+2(y_i-y_j)\end{pmatrix} \tag{6}$$

(c) If the degree of the nodes increases or decreases, the Coulomb energy decreases or increases, too because the Coulomb energy is proportional to the product of the node degrees.
Whereas the harmonic energy doesn't change if the degree of the nodes changes about the same amount because only the distance of them accounts to this energy.

(d)

(e) Listing 1 shows source code.

Listing 1: Listing of source code

```
import math
import random
from random import gauss
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.
        """
        #random.seed()
        for node in self.network.nodes.values():
            node.pos_x = random.randrange(self.size)
            node.pos_y = random.randrange(self.size)

    def calculate_forces(self):
        """
        Calculate the force on each node during the current iteration.
        """
        pairwiseForce = {}
        for nodeid, node in self.network.nodes.items():
            pairwiseForce[nodeid] = {}
            for node2id, node2 in self.network.nodes.items():
                if node2id not in pairwiseForce:
                    coulomb = - node.degree() * node2.degree() * math.hypot(node.pos_x-node2.po
                    coulomb_x = float(coulomb) / (node.pos_x-node2.pos_x) if node.pos_x != node
                        else float(coulomb) / 0.1
                    coulomb_y = float(coulomb) / (node.pos_y - node2.pos_y) if node.pos_y != no
                        else float(coulomb) / 0.1
                    if node.has_edge_to(node2):
                        harmonic_x = - float(1/2) * (2*(node.pos_x-node2.pos_x) + pow(node.pos_
                        harmonic_y = - float(1 / 2) * (2 * (node.pos_y - node2.pos_y) + pow(noo
                        coulomb_x += harmonic_x
                        coulomb_y += harmonic_y
                    pairwiseForce[nodeid][node2id] = (coulomb_x, coulomb_y)
                elif node2id in pairwiseForce and node2 != node:
                    pairwiseForce[nodeid][node2id] = pairwiseForce[node2id][nodeid]
        for nodeid in self.network.nodes.keys():
            for node2id in node.neighbour_nodes:
                if node2id != nodeid:
                    node.force_x += pairwiseForce[nodeid][node2id][0]
                    node.force_y += pairwiseForce[nodeid][node2id][1]

    def add_random_force(self, temperature):
        """
        Add a random force within [- temperature * interval, temperature * interval] to each no
        (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 for
        """
        for node in self.network.nodes.values():
            node.pos_x += self.alpha * node.force_x
            node.pos_y += self.alpha * node.force_y
            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
        """
        totalE = 0
        for node in self.network.nodes.values():
            for node2 in self.network.nodes.values():
                if node2.identifier > node.identifier:
                    totalE += float(node.degree() * node2.degree()) / math.hypot(node.pos_x-noc
                    if node.has_edge_to(node2):
                        pow1 = pow(node.pos_x-node2.pos_x, 2)
                        pow2 = pow(node.pos_y-node2.pos_y, 2)
                        totalE += float(1/2) * (pow1 + pow2)
        return totalE

    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 = []

        for i in range(iterations):
            # TODO: DECREASE THE TEMPERATURE IN EACH ITERATION. YOU CAN BE CREATIVE.
            temperature = iterations-i
            # 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
```

(f) Listing 1 shows source code.

(g) Listing 2 shows source code.

Listing 2: Listing of source code

```
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
    energies_normal = layout.layout(1000)
    # plot the normal layout
    plot_layout(layout, '')
    # run the simulated annealing layout for 1000 iterations and store the total energies
    energiesSA = layout.simulated_annealing_layout(1000)
    # plot the simulated annealing layout
    plot_layout(layout, '')
```

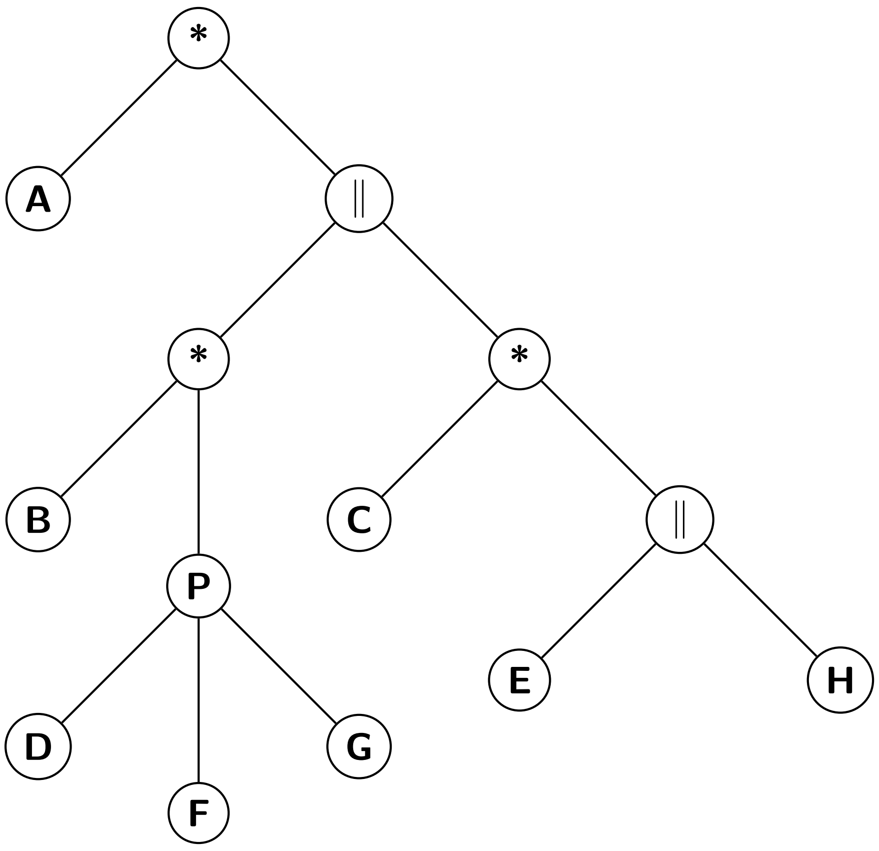

Figure 2: modular decomposition of set of protein complexes

```
# plot the total energies of the normal layout and the simulated annealing layout
plot_energies(energies_normal, '', '')
plot_energies(energiesSA, '', '')
```

Exercise 4.3: Graph Modular Decomposition

Figure shows the modular decomposition of the set of protein complexes.