Bioinformatics III

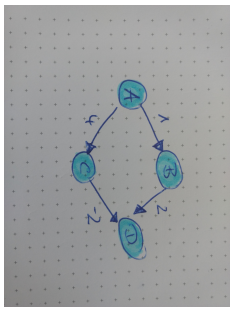
Fourth Assignment

Max Jakob (2549155)
Carolin Mayer (2552320)

May 14, 2018

Exercise 4.1: Dijkstra's algorithm for finding shortest paths

(a) Dijkstra finds the shortest path from A to D through B, since the path from A to C is already shorter than the complete upper path. However, the negative weight induce the way from A to C to D to be shorter. Hence, in this case the Dijkstra algorithm does not find the actual shortest path.
This example demonstrates that the Dijkstra algorithm in general does not yield the optimal result if negative weights are introduced. We likely reject paths with comparatively high weights, without taking into account that these might lead to the optimal solution through connection with negative edges and hence decreasing the total weight of the path.



(b) The discussed problem in a) can be solved by the proposed approach in b), since it prohibit the subsequent reduction of the path length by negative weights. By adding the absolute value of the minimal edge weight to all edges, the relative path-length is still the same. Hence, the proposed approach ensures finding the optimal solution for the shortest-path problem given a network with negative edges.

(c) The BFS can be used to find the shortest path in a network, if all edges weight the same and the network does not includes loops.
The aim of the BFS is to find a path from a given node A to another node B. It stops by the first occurrence of B. Hence, it finds the path from A to B with the minimal number of edges. If all edges have the same weight, this solution is consequentially equal to the shortest path. Having a network with different wights, the path with minimal number of edges is not necessary the shortest, as the sum of the weights of this path might be higher than a path with more edges but smaller weights. Hence, the BFS only works on networks with non or equal edge weights.

Exercise 4.2: Force directed layout of networks

(a) Given

$$\vec{F}(\vec{r}) = -\nabla E(\vec{x}) \tag{1}$$

$$E_c(\vec{r}) = \frac{1}{4\pi\epsilon_0}\frac{q_1 q_2}{||\vec{r}||} \tag{2}$$

$$E_h(\vec{r}) = \frac{k}{2}||\vec{r}||^2 \tag{3}$$

we can extend these to a 3D formula by replacing $||\vec{r}||$ with $\sqrt{x^2+y^2+z^2}$ in (2)

$$E_c(\vec{r}) = k_e\frac{q_1 q_2}{\sqrt{x^2+y^2+z^2}}$$

then

$$\frac{d}{dx}E_c(\vec{r}) = -\frac{1}{2}k_e\frac{q_1 q_2}{(x^2+y^2+z^2)^{3/2}} * 2x$$

resulting in

$$F_c(x,y,z) = \begin{pmatrix} \frac{k_e x q_1 q_2}{(x^2+y^2+z^2)^{3/2}} \\ \frac{k_e y q_1 q_2}{(x^2+y^2+z^2)^{3/2}} \\ \frac{k_e z q_1 q_2}{(x^2+y^2+z^2)^{3/2}} \end{pmatrix}$$

We can also rearrange and simplify (3) by

$$E_c(\vec{r}) = \frac{k}{2}(x^2+y^2+z^2)$$

$$\frac{d}{dx}E_c(\vec{r}) = kx$$

resulting in

$$F_h(x,y,z) = -k\begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

(b) For our application F_c and F_h will look like the following using $x_i - x_j = \Delta x$ and $y_i - y_j = \Delta y$:

$$F_h(\vec{r}_{i,j}) = -k\begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix}$$

$$F_c(\vec{r}_{i,j}) = \begin{pmatrix} \frac{\Delta x k_i k_j}{(\Delta x^2+\Delta y^2)^{3/2}} \\ \frac{\Delta y k_i k_j}{(\Delta x^2+\Delta y^2)^{3/2}} \end{pmatrix}$$

(c) An increase or decrease in the nodes degree ($k_{i,j}$) will not effect the harmonic energy E_h. However, the Coulomb energy will decrease/increase with an increase/decrease in the nodes degree. With this decrease/increase in energy, the force F_c will increase/decrease in the opposite direction. Considering the distance between two nodes, E_h will increase when the distance is shorter, which will lead to an decrease in the force F_h. E_c will increase with an increase in the distance which will lead to an decrease in F_c and vice versa.

(d) To clarify the statement that F_c is the repulsive force and F_h is the attractive force, we only consider the distance between two nodes. If they are far apart from each other, F_c will be small and F_h will be big, as described in (b). However, if the distance is really small, F_c will be really large and F_h will be very small. As we try to minimize the overall force, one likes to minimize both forces. This means, F_c wants to pushes the nodes far away from each other and F_h wants to move them together as close as possible. The total minimum will be somewhere in the middle. But based on this characteristics, we can consider the harmonic force as an attractive force and the Coulomb force as an repulsive force.

(e) Below, our implementation for the `Layout` class in `layout.py` is shown.

Listing 1: Source code of the script `layout.py`

```
from random import gauss
import random as r
import decimal as d
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
        # total energy
        self.total_energy = 0

    def init_positions(self):
        """
        Initialise or reset the node positions, forces and charge.
        """
        r.seed(1)
        for node_name in self.network.nodes:
            node = self.network.get_node(node_name)
            node.pos_x = r.randint(1, self.size)
            node.pos_y = r.randint(1, self.size)

        self.calculate_forces()


    def calculate_forces(self):
        """
        Calculate the force on each node during the current iteration.
        """
        self.total_energy = d.Decimal(0)
        for n_name in self.network.nodes:
            n1 = self.network.get_node(n_name)
            n1.force_x = d.Decimal(0)
            n1.force_y = d.Decimal(0)

        for n1_name in self.network.nodes:
            n1 = self.network.get_node(n1_name)

            for n2_name in self.network.nodes:
                if n1_name != n2_name:
                    w = d.Decimal(0)
                    n2 = self.network.get_node(n2_name)
                    if n1.has_edge_to(n2):
                        w = d.Decimal(1)

                    delta_x = d.Decimal(n1.pos_x) - d.Decimal(n2.pos_x)
                    delta_y = d.Decimal(n1.pos_y) - d.Decimal(n2.pos_y)

                    F_h_x = d.Decimal(-1) * d.Decimal(delta_x)
                    F_h_y = d.Decimal(-1) * d.Decimal(delta_y)

                    z_x = d.Decimal(delta_x * d.Decimal(n1.degree()) * d.Decimal(n2.degree()))
                    z_y = d.Decimal(delta_y * d.Decimal(n1.degree()) * d.Decimal(n2.degree()))
                    n = ((delta_x ** d.Decimal(2)) + (delta_y ** d.Decimal(2))) ** d.Decimal((3
```

```
                    f_x =(z_x/n) + (w * F_h_x)
                    f_y = (z_y/n) + (w * F_h_y)

                    n1.force_x += f_x
                    n1.force_y += f_y

                    e_c = d.Decimal(d.Decimal(n1.degree()) * d.Decimal(n2.degree()))/(((delta_x
                    e_h = d.Decimal(0.5) * ((delta_x ** d.Decimal(2)) + (delta_y ** d.Decimal(2
                    self.total_energy += e_c + e_h

    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 += d.Decimal(gauss(0.0, self.interval * temperature))
            node.force_y += d.Decimal(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 n_name in self.network.nodes:
            n = self.network.get_node(n_name)
            n.pos_x += d.Decimal(self.alpha) * d.Decimal(n.force_x)
            n.pos_y += d.Decimal(self.alpha) * d.Decimal(n.force_y)

    def calculate_energy(self):
        """
        Calculate the total energy of the network in the current iteration.
        :return: total energy
        """
        return self.total_energy/2

    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 = 100

        for i in range(iterations):
            temperature -= 0.1
            # there is nothing to do here for you
```

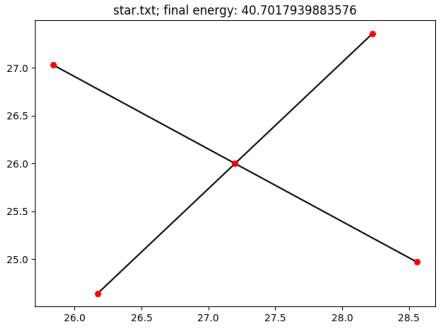
```
                self.calculate_forces()
                self.add_random_force(temperature)
                self.displace_nodes()
                energies.append(self.calculate_energy())

        return energies
```

(f) The code for task f) is also shown in Listing 1. As starting temperature, we choose 100 degree. In each step, we decrease this temperature by 0.1 degree. With simulated annealing, one can overcome local optima in the simulation step. This can be very useful in practice, as we can often improve out optimization.

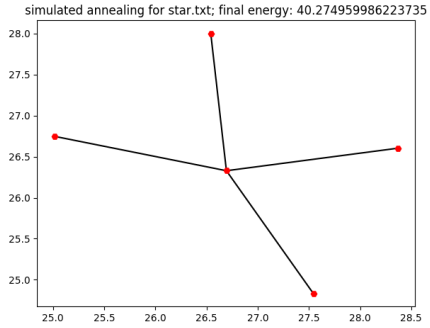
(g) The code for our script to create all the plots shown in Figure 1 is listed in Listing 2. Regarding the plots, where the force directed algorithm was compared to simulated annealing, one can see that both algorithms perform well in the four settings. The final layout energy of both algorithms are almost identical in most cases. Only at Figure 1g and Figure 1h a bigger difference can be observed. There, simulated annealing performs a little worse, but the structure of this layout can still be recognized (as a dog). The energy of each layout is written in each plot within the title.
In Figure 2 the system energy of the layout is shown during 1000 iterations. There, both algorithms are shown. Remarkable here is, that both algorithms decrease the energy drastically in their first iterations. It is also observable that the force directed algorithm decreases monotonically, whereas simulated annealing fluctuates while decreasing. The fluctuation reduces by the number of iterations, which is directly linked to the temperature decrease within each step. As the temperature falls, less random energy is added to the layout and thus the fluctuation stabilizes (after iteration 600 in this case).

Listing 2: Source code of the script `layout_main.py`

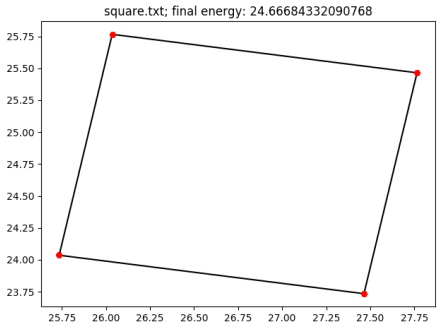
```
from layout import Layout
from tools import plot_layout, plot_energies

energy1 = []
energy2 = []
nets = ["star.txt", "square.txt", "star++.txt", "dog.txt"]
for name in nets:
    print("creating layout for " + name + " ...")
    temp_layout = Layout(name)
    Layout.init_positions(temp_layout)
    e1 = Layout.layout(temp_layout, 1000)
    # save energy
    energy1.append(e1)
    plot_layout(temp_layout, name + "; final energy: " + float(e1[999]).__str__())
    e2 = Layout.simulated_annealing_layout(temp_layout, 1000)
    # save energy
    energy2.append(e2)
    plot_layout(temp_layout,"simulated annealing for " + name + "; final energy: " + float(e2[9

plot_energies([energy1[0], energy2[0]], ["force directed", "simulated annealing"], "energy plot
```



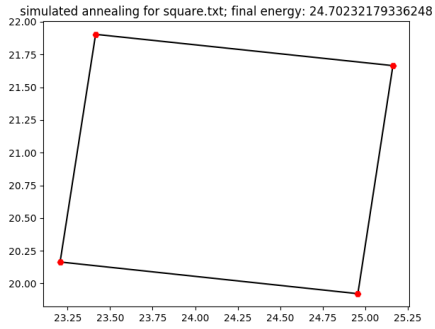
(a) force directed for star.txt



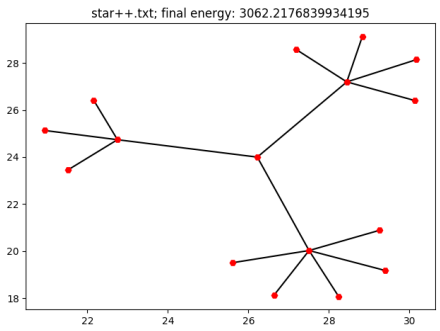
(b) simulated annealing for star.txt



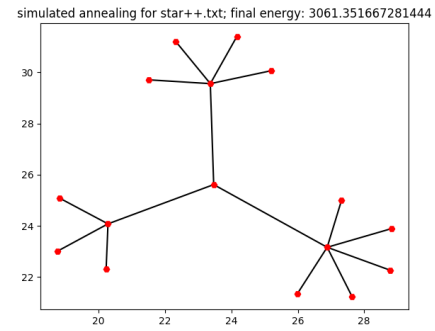
(c) force directed for square.txt



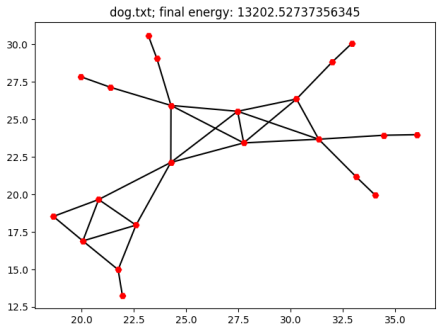
(d) simulated annealing for square.txt



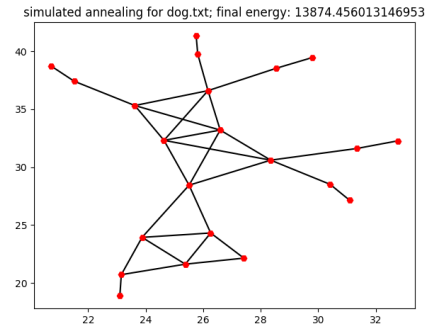
(e) force directed for star++.txt



(f) simulated annealing for star++.txt



(g) force directed for dog.txt



(h) simulated annealing for dog.txt

Figure 1: Layout plots for all given graphs.

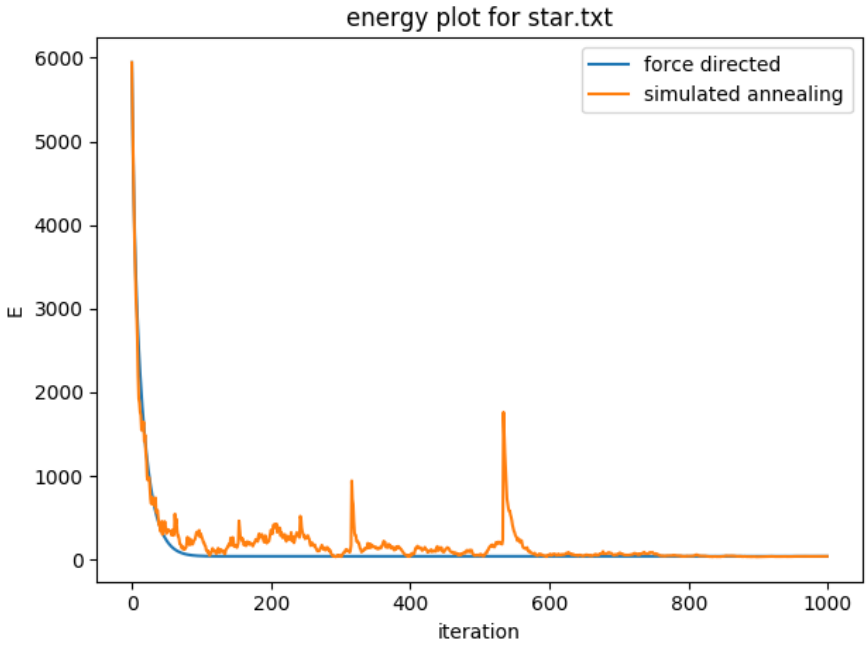


Figure 2: The system energy plotted over 1000 iterations of the two different algorithms.

Exercise 4.3: Graph Modular Decomposition