Bioinformatics III

Prof. Dr. Volkhard Helms
Nicolas Künzel
Winter Semester 2019/2020

Saarland University
Chair for Computational Biology

Exercise Sheet 4

Due: Nov 14, 2019 14:15

Submit your solutions on paper, hand-written or printed at the *beginning* of the lecture or in building E2.1, Room 3.01. Alternatively, you can send an email with a single PDF attachment to nicolas.kuenzel@bioinformatik.uni-saarland.de. Either way, please hand in your source code or programming problems can not be graded. The source code should be sent in a .zip file via email.

Dijkstra, force directed layouts and modular decomposition

We continue to look at networks. The assignment of this week deals with edge weights in Dijkstra's algorithm, energies and forces applied to layout networks, as well as modular decomposition.

Exercise 4.1: Dijkstra's algorithm for finding shortest paths (30 points)

Lecture 6 introduced Dijkstra's algorithm that finds the shortest paths between a given start node and all other nodes in a graph if the weight of each edge is non–negative.

(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.

(b) To run Dijkstra's algorithm on graphs with negative edge weights, someone proposes the following modification:

(1) If there are edges with negative weights in the graph, find the edge with the smallest (most negative) weight w_s.

(2) Increase the weight of all edges by $|w_s|$ so that all edges have weights ≥ 0.

(3) Run Dijkstra's algorithm on the modified graph.

Is this modified algorithm guaranteed to find the shortest paths (= smallest sum of weights in the original graph), even when some edges possess negative weights? Explain your answer.

(c) Dijkstra's algorithm is very similar to simple breadth–first search (BFS). BFS has a worst case performance of $\mathcal{O}(n+m)$, which is faster than Dijkstra's worst case runtime of $\mathcal{O}(n^2)$, with n nodes and m edges. 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)?

If you need a reminder how BFS works, you can find pseudocode and an animated example on Wikipedia.

Exercise 4.2: Force directed layout of networks (60 points)

In this exercise you implement a layout algorithm for networks in the **Layout**-class by using energy functions that mimic repulsive and attractive molecular forces. Subsequently, you read networks from files and visualise the final layouts and the energy trajectories.

General remarks:

- The small arrow over a variable indicates that it is a vector, e.g. $\vec{x}$.

- The gradient describes the direction and rate of change of a function f, the so called slope, and is denoted by the gradient (Nabla) operator ∇. In 1D, the gradient is the simple derivative $\nabla f(x) = \frac{\mathrm{d}}{\mathrm{d}x} f$. In n dimensions, the gradient is a vector of the partial derivatives

$$\nabla f(x_1, ..., x_n) = \begin{pmatrix} \frac{\partial}{\partial x_1} f \\ \frac{\partial}{\partial x_2} f \\ \dots \\ \frac{\partial}{\partial x_n} f \end{pmatrix}$$

  that can be understood as a multidimensional slope.

- The force F equals the negative gradient of the energy E. In other words, the force is a measure for how much the energy changes with an infinitesimal displacement. Given an n–dimensional vector $\vec{x} = (x_1, ..., x_n)$, the force is an n-dimensional vector

$$\vec{F}(\vec{x}) = -\nabla E(\vec{x}).$$

- The Coulomb energy between two point charges q_1 and q_2 with distance $\|\vec{r}\|$ is

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

  where $k_e = \frac{1}{4\pi\epsilon_0}$ is the Coulomb constant. The harmonic energy with spring constant k is

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

(a) **Implementation preparation:** Use the general definitions above to write down the force field $\vec{F_c}(\vec{r})$ for the Coulomb energy $E_c(\vec{r})$ and the force field $\vec{F_h}(\vec{r})$ for the harmonic energy $E_h(\vec{r})$ in 3D Cartesian coordinates.

Note that in 3D $\vec{r} = (x, y, z)$ with length $\|\vec{r}\| = \sqrt{x^2 + y^2 + z^2}$. Consequently, $\vec{F_c}(\vec{r})$ and $\vec{F_h}(\vec{r})$ are vectors in terms of x, y and z. Simplify the equations as much as possible and write down all steps needed to reach your solution.

(b) **Adapting the energy equations for networks:** Since the network layout is in 2D, two nodes i and j have coordinates (x_i, y_i) and (x_j, y_j), respectively. Thus, the distance between the two nodes is

$$\|\vec{r}_{ij}\| = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \quad \text{with} \quad \vec{r}_{ij} = \begin{pmatrix} x_i - x_j \\ y_i - y_j \end{pmatrix}.$$

Additionally, the charge q is replaced by the node degree k and the Coulomb and spring constants are dropped. That gives the adjusted Coulomb energy

$$E_c(\vec{r}_{ij}) = \frac{k_i k_j}{\|\vec{r}_{ij}\|}$$

and the adjusted harmonic energy

$$E_h(\vec{r}_{ij}) = \frac{1}{2} \|\vec{r}_{ij}\|^2.$$

Adjust your solutions for $\vec{F_c}(\vec{r}_{ij})$ and $\vec{F_h}(\vec{r}_{ij})$ from exercise part (a) accordingly and write them down. These are the energy and force equations you are going to need for your implementation.

(c) **Understanding the Coulomb and harmonic energy:** The force directed layout algorithm attempts to find a good layout by finding the lowest energy conformation of the nodes in the network. Look at the adjusted Coulomb energy $\vec{E}_c(\vec{r}_{ij})$ and harmonic energy $\vec{E}_h(\vec{r}_{ij})$ from part (b) and answer the following questions:

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?

(d) **Understanding the forces:** To find the lowest energy conformation, the Coulomb and harmonic energy are used to compute the pairwise force between two nodes. The interaction between two nodes i and j is defined as

$$W_{ij} = \begin{cases} 1, & \text{if edge } i \to j \text{ exists} \\ 0, & \text{otherwise} \end{cases} .$$

The pairwise force between two nodes i and j is then

$$\vec{F}_{ij} = \vec{F}_c(\vec{r}_{ij}) + W_{ij} \cdot \vec{F}_h(\vec{r}_{ij}).$$

The total force on node i is the sum of pairwise forces between i and all other nodes j:

$$\vec{F}_i = \sum_j \vec{F}_{ij} \qquad i \neq j.$$

The total force $\vec{F}_i$ determines the direction in which node i should be moved, as well as how far it should be moved in that direction. The larger the total force, the further it is moved from its current position in the network.

Look at the equation of the pairwise force $\vec{F}_{ij}$ between two nodes, as well as the equations for $\vec{F}_c(\vec{r}_{ij})$ and $\vec{F}_h(\vec{r}_{ij})$ that you obtained in part (b). Why is the Coulomb force the repulsive force and the harmonic force the attractive force?

(e) **Implementing the force directed layout algorithm:** You are going to implement the force directed layout algorithm in the **layout**– class in *Layout.py.* The basic outline of the layout algorithm is as follows:

(1) **init_positions()**: Assign random x and y positions to all nodes in the network. Set the initial force in the x and y coordinate for each node to the correct value. Set the charge of each node. The provided **Node**–class has already been extended with the required fields.

(2) **layout(iterations)**: In each iteration

i. **calculate_forces()**: Calculate the pairwise forces $\vec{F}_{ij}$ between all pairs of nodes and then compute the total force $\vec{F}_i$ for all nodes. Note that the forces between two nodes are symmetric, meaning $F_{ij} = -F_{ji}$. Remember not to count pairs of nodes twice!

ii. **displace_nodes()**: Use the forces computed in the previous step to update the position of each node

$$\begin{pmatrix} x_i \\ y_i \end{pmatrix} := \begin{pmatrix} x_i \\ y_i \end{pmatrix} + \alpha \cdot \vec{F}_i$$

A reasonable value is $\alpha = 0.03$. Do not forget to reset all the forces after this step!

iii. **compute_energy()**: Calculate the total energy, which is the sum of all individual interaction energies

$$E_{\text{tot}} = \sum_{i<j} E_c(\vec{r}_{ij}) + W_{ij} \cdot E_h(\vec{r}_{ij})$$

The energy of each iteration is stored and returned.

(f) **Simulated annealing:** The **Layout**–class contains an alternative layout function that adds a random thermal contribution as an additional force on each node in each iteration. This thermal contribution should decrease in each step, which you have to implement in the function **simulated_annealing_layout(iterations)**. Choose a sensible starting "temperature".

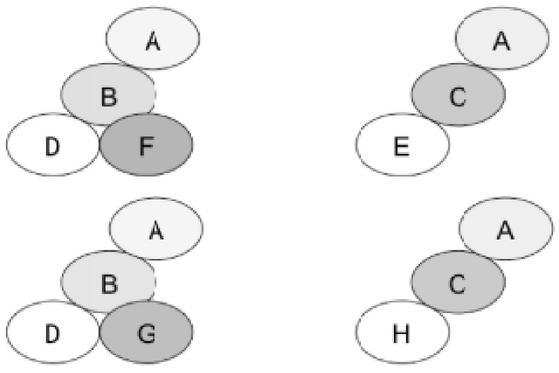
Explain why simulated annealing is a worthwhile optimisation principle in practice.

(g) **Applying the layout algorithms:** The supplement contains the test files "`star_net.txt`", "`square_net.txt`", "`star++_net.txt`" and "`dog_net.txt`". Write a simple Python script that uses the **Layout**–class to read the test files and do 1000 iterations with both implementations of the algorithm. Report the final energies and plot the layouts. For one of the networks also plot the energies per step for the basic and the simulated annealing method and compare the results.

You can use the plotting methods provided in *tools.py*.

Exercise 4.3: Graph Modular Decomposition (10 points)

A cell contains the 4 variants of a protein complex shown below:



Draw the tree for the modular decomposition of this set of protein complexes and indicate the "prime", "parallel" and "serial" symbols.