Problem Set 3 – Graph drawing by stress minimization

Graph drawing, particularly the visualization of large graphs, is a huge subject featuring, for example, its own annually conference[1] and an extensive handbook[2].

[1] http://www.graphdrawing.org/

[2] https://cs.brown.edu/people/rtamassi/gdhandbook/

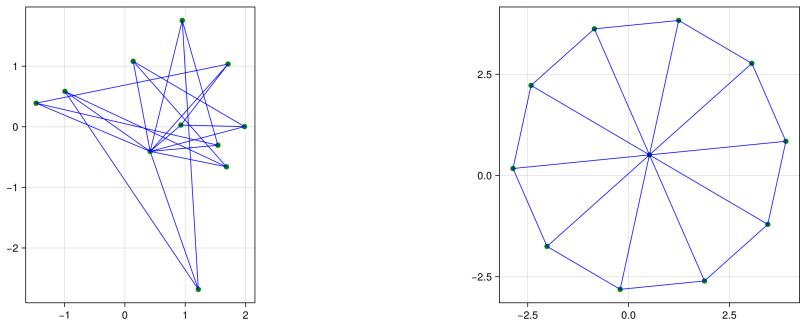


Figure 1: Two drawings of the same graph W_{11}.

For some reason, humans prefer the right picture of a wheel graph with 11 nodes over the left one. So that's what a good graph layout algorithm should deliver.

One family of graph layout algorithms are the *stress minimization* algorithms, also known as *spring embedders* or *force directed layout*: Build the graph. Use electrically charged balls which repel each other for the nodes. Use springs or rubber bands as edges. They will pull connected nodes towards each other. Allow the system to settle into a configuration that minimizes the overall stress or energy.

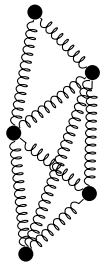


Figure 2: A spring graph.

1 *Graphs and graph embeddings*

Definition 1.1. For our purposes, a Graph $G = (V, E)$ is

(i) a finite set V of *vertices* or *nodes* and

(ii) a set $E \subseteq \{\{x, y\} \mid x, y \in V\}$ of *edges*.

Let's unroll this definition:

- Edges are sets of two nodes, i.e., unordered pairs. They have no direction.
- No edge can connect a node to itself.
- Two nodes can not be connected by more than one edge.

Thus, our graphs are *finite, undirected, and simple.* We use *graph embeddings* to visualize this purely combinatorial structure.

Definition 1.2. A graph embedding is an injective map $p : V \to \mathbb{R}^2$ from vertices to points in a plane or higher-dimensional space. The edges are mapped into the corresponding straight-line segments.

2 *Graph drawing by stress minimization*

An embedding p maps every vertex $v \in V$ to a point $p(v)$ in the plane. Let the term $\|p(u) - p(v)\|$ denote the Euclidean distance between two points in the plane.

2.1 FR model

In this model (Fruchterman and Reingold 1991) the stress (or energy or potential) of an embedding $p(G)$ is defined as[3]:

$$S(p(G)) = \frac{1}{6} \sum_{\text{edges}\{u,v\}\in E} \|p(u) - p(v)\|^3 \quad - \sum_{\text{pairs } u,v\in V} \log\|p(u) - p(v)\|. \tag{1}$$

[3] The second sum is over all unordered pairs, i.e., $n(n-1)/2$ terms for n nodes.

The first term increases stress for an edge between two vertices if they are far apart, while the second term increases stress for an arbitrary pair of vertices if they are close together. The goal is to minimize this stress function for an aesthetically pleasing layout.

There are many possible modifications. One can change the exponent in the first term to make it more or less sensitive to long-range interactions etc.

One alternative is the following:

2.2 ERLL model

The *Edge repulsion LinLog* model (Noak 2004) uses another exponent in the first term and modifies the second term such that 'Big Boss nodes' with many edges have a larger repulsion between them.

$$S(p(G)) = \frac{1}{2} \sum_{\text{edges}\{u,v\}\in E} \|p(u)-p(v)\| \quad - \sum_{\text{pairs } u,v\in V} \deg(u)\deg(v)\log\|p(u)-p(v)\|. \tag{2}$$

Here, $\deg(v)$ denotes the *edge degree* of a node v, the number of edges incident on it.

So, given a graph G we have to find the embedding p which minimizes $S(p)$.

3 Minimum search by gradient descent

Numerically searching for the minimum of a function $f : \mathbb{R}^n \to \mathbb{R}$ where $n \gg 1$ is one of the central problems of optimization and machine learning. In a sense, training of an artificial neural network (ANN) is essentially such a minimum search problem.

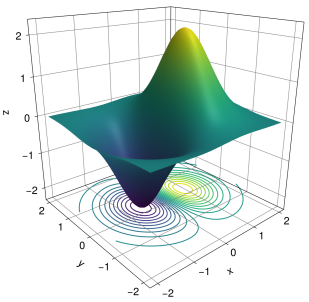


Figure 3: A function $z = f(x, y)$ as a landscape.

3.1 Simple gradient descent

The idea of gradient descent (GD) is simple: we start at a random point $\mathbf{x}_0 \in \mathbb{R}^n$ and find the minimum iteratively by moving in the direction $\mathbf{g}$ of steepest descent (negative gradient) at each step with some fixed step size $\alpha \in \mathbb{R}_{>0}$.

$$\text{simple GD:} \quad \mathbf{x}_{k+1} = \mathbf{x}_k + \alpha \mathbf{g}(\mathbf{x}_k) \tag{3}$$

$$\text{where} \quad \mathbf{g}(\mathbf{x}) = -\operatorname{grad} f(\mathbf{x}) = -\nabla f(\mathbf{x}) = -\left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n}\right)\Big|_{\mathbf{x}} \tag{4}$$

As usual for an iterative algorithm we will stop if either (i) the steps no longer create a reasonable improvement, $|f(\mathbf{x}_{k+1}) - f(\mathbf{x}_k)| < \epsilon$ or (ii) a certain maximal number of steps has been reached.

The simple GD algorithm is far from optimal. It performs poorly in narrow valleys and other regions where the gradient ∇f is small.

3.2 Nesterov gradient descent

So-called momentum GD algorithms improve upon simple GD by incorporating a momentum term, i.e., by taking account not only the local gradient but also the gradients of previous steps. Think of a heavy ball rolling downhill, where the ball's velocity from previous steps helps it to avoid smaller local minima and passing regions of low gradient.

The Nesterov GD algorithm is a variant[4] where additionally the gradient is taken at a projected future position which improves convergence for a large class of functions. We start with a zero momentum vector $\mathbf{v}_0$ and update the position and momentum as follows:

$$\mathbf{v}_{k+1} = \beta \mathbf{v}_k - \alpha \nabla f(\mathbf{x}_k + \beta \mathbf{v}_k) \tag{5}$$

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{v}_{k+1} \tag{6}$$

with a momentum decay constant $\beta < 1$ and a learning rate α.

Of course, nothing can fully protect us from getting caught in a local minimum. There are some tricks to increase the chance of escaping local minima, such as random restarts or the technique of stochastic gradient descent (SGD). But no numerical algorithm guarantees finding the global minimum for a general function.

[4] For other variants, see e.g. S. Ruder, 'An overview of gradient descent optimization algorithms'.

4 Implementation in Julia

4.1 Packages

The following packages are recommended:

(i) `Graphs.jl` as a source for a graph data type and many examples,
(ii) `Distances.jl` for the fast calculation of all pairwise distances,
(iii) `Zygote.jl` for automatic differentiation and
(iv) A plot package, e.g., `Plots.jl` or `CairoMakie.jl`.

4.1.1 Graphs.jl

```
julia> using Graphs

julia> g = wheel_graph(3)
{3, 3} undirected simple Int64 graph

julia> nv(g), ne(g)  # number of nodes and edges, resp.
(3, 3)

julia> adjacency_matrix(g)
3×3 SparseArrays.SparseMatrixCSC{Int64, Int64} with 6 stored entries:
 ⋅  1  1
 1  ⋅  1
 1  1  ⋅

julia> for e ∈ edges(g) # iterator object for edges
           i, j = e.src, e.dst   # edge is a struct with members src, dst
           println(i,"  ", j)
       end
1  2
1  3
2  3
```

Some graph examples from `Graphs.jl` you should try:

- `g1 = wheel_graph(11)`
- `g2 = smallgraph(:icosahedron)`
- `g3 = smallgraph(:karate)`
- `g4 = dorogovtsev_mendes(25)`
- `g5 = circular_ladder_graph(12)`
- `g6 = watts_strogatz(25, 4, .2)`
- `g7 = static_scale_free(14, 40, 5)`

4.1.2 Distances.jl

In every step of our minimum stress search we have to calculate all $n(n-1)/2$ distances $\|p(v_i) - p(v_j)\|$ between the n nodes. There are some tricks to do this efficiently. Our suggestion is:

- Keep the coordinates of the n nodes in a $2 \times n$-matrix p which represents the embedding and is updated in every step of the algorithm.
- Use the `Distances.jl` package to calculate all distances efficiently:

```
using Distances
...
D = pairwise(Euclidean(), p, dims=2) # returns a symmetric n×n matrix
            # D[i,j] is the distance between node i and node j
```

4.1.3 Zygote.jl

`Zygote.jl` is a Julia package for reverse mode automatic differentiation. Given a Julia function it hooks into the compiler and automatically produces the source code for the gradient.

Let's say we have a function `stress(graph, p, further_params,...)` where `p` is an array and we seek the gradient with respect to this argument `p` at a given value `p0`.

```
using Zygote
...
s, (grad,) =  Zygote.withgradient(p -> stress(graph, p, params), p0)
# s is the stress value at p0,
# grad is an array of the same size as p0, the gradient of stress at p0
```

4.2 Coding tasks

- Write a function `plot_graph(g::SimpleGraph, p::Matrix{Float64})` that takes a graph and its node positions as input and plots the graph, as in Fig. 1.
- Initialize the positions array with random coordinates: `p = rand(2,n)`
- During gradient descent, collect the stress values at each step and plot them to visualize the convergence. Use this to compare different methods.
- Call the `plot_graph()` function every 30 steps to watch progress.
- Start with a simple graph like `wheel_graph(7)` to test and debug your code.
- It may be useful to regularize the log, i.e., use $\log(\|...\| + 10^{-6})$ in the stress function.
- Try step sizes $\alpha = 0.01 \ldots 0.05$ and a momentum decay of $\beta = 0.9$.
- Stop if the relative change of stress is less than, e.g., 10^{-7}.
- Bonus: explore other minimizers and/or other stress functions.
- Bonus: apply these algorithms to 3-dimensional graph embeddings.