Karlsruhe Institute of Technology

Approximating One-Sided Crossing Minimization with Graph Networks

Bachelor Thesis of

Thomas Weidmann

At the Department of Informatics
Institute of Theoretical Informatics

Reviewers:	PD Dr. Torsten Ueckerdt
	Prof. Dr. Peter Sanders
Advisors:	Guido Brückner

Time Period: May 1st 2020 – August 31st 2020

Statement of Authorship

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, August 31, 2020

Abstract

To a certain graph there exist plenty of different drawings. In some of them one can see the relations between the nodes better than in other drawings. There are different characteristics of a good drawing of a graph. One of them is the number of crossings between the edges. The corresponding optimization problem is called the Crossing Minimization Problem (CMP) which is $\mathcal{NP}$-hard.

A quite restricted version of this problem has received considerable attention that called the One-Sided Crossing Minimization Problem (OCMP). In this problem the nodes are placed on two horizontal lines and the solution is an order of the nodes on the upper line such that the number of crossings is the minimum over all possible permutations. This optimization problem is also $\mathcal{NP}$-hard and there exist several well-studied efficient heuristics. We build two approximization algorithms using machine learning. One of them starts in a random permutation and switches neighbouring nodes to reduce the crossings and the other algorithm determines the leftest node in the order and by iterating we can build a complete order. We find that our approaches are competitive on graphs on sizes the network was trained on (up to 20 vertices) and even scale for sizes with up to 50 vertices.

Deutsche Zusammenfassung

Zu einem gegeben Graphen existieren viele verschiedene Zeichnungen. In manchen Zeichnungen kann man die Beziehungen zwischen den Knoten besser erkennen als in anderen. Ein Merkmal einer guten Zeichnung eines Graphen ist die Anzahl an Kreuzungen der Kanten. Das zugehörige Optimierungsproblem heißt Kreuzungsminimierungsproblem (KMP) welches $\mathcal{NP}$-schwer ist.

Das einseitige Kreuzungsminimierungsproblem (EKMP) ist eine stark eingeschränkte Version davon welche einige Aufmerksamkeit erreicht hat. Dabei werden die Knoten auf zwei horizontale Linien aufgeteilt sind und gesucht ist eine Permutation der Knoten auf der oberen Linie, sodass die Anzahl der Kreuzungen minimal ist, auch dieses Problem ist $\mathcal{NP}$-schwer. Es existieren einige lang erforschte Heuristiken und wir versuchen zwei Heuristiken mit Hilfe von maschinellem Lernen zu entwickeln. Ein Ansatz started in einer zufälligen Permutation und tauscht benachbarte Knoten um die Anzahl an Kreuzungen zu minimieren und in dem anderen Ansatz bestimmen wir den linkesten Knoten der Permutation und durch mehrfaches Anwenden dieses Vorgehens können wir eine komplette Permutation aufbauen. Wir finden, dass unsere Ansätze konkurrenzfähig zu existierenden Heuristiken sind auf Instanzgrößen auf denen das Netzwerk auch trainiert wurde (bis zu 20 Knoten) und dass diese sogar bis Instanzgrößen von 50 noch gut skalieren.

Contents

1 Introduction **1**
- 1.1 Graph Drawing . . . 1
- 1.2 Machine learning on graphs . . . 3
- 1.3 Contribution Outline . . . 6

2 Preliminaries **7**
- 2.1 Directed Graphs . . . 7
  - 2.1.1 Two Layered Network . . . 7
  - 2.1.2 Attributed directed graphs . . . 8
  - 2.1.3 Random Graphs . . . 8
    - 2.1.3.1 Erdős–Rényi model . . . 8
- 2.2 Neural Networks . . . 8
  - 2.2.1 Components . . . 9
    - 2.2.1.1 Perceptron . . . 9
    - 2.2.1.2 Multilayer Perceptron . . . 10
    - 2.2.1.3 Layer Normalization . . . 10
  - 2.2.2 Learning . . . 10
  - 2.2.3 Loss . . . 11
  - 2.2.4 Gradient Descent . . . 11
- 2.3 One-Sided Crossing Minimization . . . 12
  - 2.3.1 $\mathcal{NP}$-completeness of the decision problem . . . 13
  - 2.3.2 Reduction to Integer Linear Programm . . . 14
  - 2.3.3 Existing Heuristics . . . 15
    - 2.3.3.1 Barycenter . . . 15
    - 2.3.3.2 Sifting . . . 15
    - 2.3.3.3 Median . . . 16
    - 2.3.3.4 Greedy Switch . . . 16
    - 2.3.3.5 Greedy Insert . . . 16
    - 2.3.3.6 Split . . . 16

3 Graph Network **17**
- 3.1 Graph . . . 18
  - 3.1.1 Graph concatenation . . . 18
- 3.2 GN block Variants . . . 18
  - 3.2.1 Full GN Block . . . 20
  - 3.2.2 Graph Independent (GI) Block . . . 21
- 3.3 Models . . . 23
  - 3.3.1 n-Process Encode Decode . . . 23
  - 3.3.2 n-Layer Process Encode Decode . . . 24
  - 3.3.3 n-Recursive Layer Process Encode Decode . . . 24

4 Heuristics Using Machine Learning **25**
4.1 Learned Leftmost Node . . . 25
4.1.1 Score One-Sided Crossing Minimization . . . 26
4.1.1.1 Reduction to ILP . . . 26
4.1.2 Binary Score One-Sided Crossing Minimization . . . 26
4.1.3 $[0, 1]$-Iterpolated Score One-Sided Crossing Minimization . . . 27
4.1.4 Input Graph . . . 27
4.1.5 Algorithm . . . 28
4.2 Learned Switch . . . 28
4.2.1 Switch One-Sided Crossing Minimization . . . 29
4.2.1.1 Reduction to ILP . . . 29
4.2.2 Binary Switch One-Sided Crossing Minimization . . . 30
4.2.3 Input Graph . . . 30
4.2.4 Algorithm . . . 32

5 Training and Evaluation **33**
5.1 Trainingdata . . . 33
5.2 Training . . . 34
5.3 Comparison . . . 36
5.3.1 Existing Heuristics . . . 36
5.3.2 Interpolated Score . . . 38
5.3.3 Binary Score . . . 39
5.3.4 Binary Switch . . . 41
5.3.5 Model Comparison . . . 42

6 Conclusion **45**
6.1 Future Work . . . 45

Bibliography **47**

1. Introduction

1.1 Graph Drawing

Graph drawing is a scientific field in informatics that tries to find a geometrical representation of an abstract object called a graph. This is an important tool to make information for humans more accessible. One example is a network map. One can write every station with its reachable stations in a list but this would make it very hard to find a route from a given start to a given end so these maps are represented as graphs and drawn such that it's easy to perceive all these informations. Other examples are visualizing social networks with its users and relations between them, entity-relationship diagrams, data structures such as trees and moleculuar structurs in biology[FH01]. But there are also applications that have a more pracitical backround. One example is to design a hierarchical layout of VLSI circuits[KK90]. It consists of finding the position of an electronical component row by row where one row is fixed and the other one has to be permutated such that the number of crossings is the minimum.
The application of VLSI circuits has a strict metric of how good a solution is namely the number of crosses. In contrairy to that the metric of how good a graph drawing for human pereption is ambigously. There exist many metrics [Pur02] however one of the metrics is also minimizing the number of crosses.

Crossing Number

A drawing D of a graph G is a mapping from the nodes into the plane and mapping from the edges to a set of simple planar curves such that the curve has the adjacend nodes as endpoints and contains no other nodes. The number of crossings $\mathrm{cr}(D)$ of a given drawing D is the number of intersections between distinct edges. The crossing number $\mathrm{cr}(G)$ of a given graph G is the minimum number of crosses over all drawings. The descision problem to this is proven to be $\mathcal{NP}$-complete [MRG83] so there is probably no efficient algorithm to calculate this number. For complete bipartite graphs $K_{m,n}$ there exists a general drawing [Zar55] with

$$Z(m,n) = \lfloor \frac{m}{2} \rfloor \lfloor \frac{m-1}{2} \rfloor \lfloor \frac{n}{2} \rfloor \lfloor \frac{n-1}{2} \rfloor$$

crosses and therefore this is an upper bound for the number of crosses for this graph. There is also a general drawing for complete graphs K_n [Thr52] with

$$Z(n) = \lfloor \frac{n}{2} \rfloor \lfloor \frac{n-1}{2} \rfloor \lfloor \frac{n-2}{2} \rfloor \lfloor \frac{n-3}{2} \rfloor$$

crosses. There is also a lower bound proven independently by Ajtai, Chvátal, Newborn and Szemerédi [ACNS82] and by Leighton [Lei83] which is for a graph G with n nodes and m edges such that $e > 7m$ then the number of crosses

$$\mathrm{cr}(G) \geq \frac{e^3}{29n^2}$$

is bounded by this term.

Approximation Algorithm for Crossing Number

A graph that we can draw without crossings is called a planar graph. We can determine if a given graph is planar and then find a two dimensional embedding if so even in linear time[NC85]. However when there are more crosses this problem gets more difficult.

The Sugiyama algorithm [STT81] calculates to a given directed graph $G = (V, E)$ a two dimensional hierarichal drawing. It draws the nodes on distinct horizontal lines and edges between those lines and tries to minimize the number of crosses. It consists of four steps:

- step 1: Transform G into a equivalent graph with nodes on distinct horizontal lines called levels and edges just between neighboring levels
  - step 1.1: Change direction of edges to make the graph acyclic
  - step 1.2: Assign nodes to different levels
  - step 1.3: Add dummy nodes such that edges crossing multiple levels can be divided in multiple edges between neighboring levels
- step 2: Calculate for each level an order between the nodes
- step 3: Calculate concrete x-coordinates for nodes on each level
- step 4: Remove dummy nodes and edges and calculate a two dimensional assignment of every node

Some of these steps are hard optimization problems. For example in step 1.1 if we want to minimize the number of edges that have to be reversed we have to solve the $\mathcal{NP}$-hard problem Minimum Feedback Arc Set[Ced66] as well as in step 1.2 if we want to know the maximum number of levels and number of nodes per level we have to solve the $\mathcal{NP}$-hard Directed Layering Problem[PE91]. Step 1.3 and 4 are trivial.
For steps 2 and 3 a heuristic for reducing edge crossings is proposed. It starts with a random permutation on the lowest level. The algorithm then tries to find a permutation for the next higher level which is called the One-SidedCrossingMinimization. By iterating over the layers a permutation for every layer is found and then the direction changes and the algorithm tries to find a permutation for the second highest level and iterates down to the lowest level. This procedure is called the down-up procedure and is repeated at maximum a predefined number of rounds or until the permutation on the levels doesn't change.

The One-SidedCrossingMinimizationProblem is $\mathcal{NP}$-hard [Wor94] even for sparse graphs [Vrt01]. There exist many well studied heuristics for approximating this problem. In tests [MJ97] the barycenter heuristic lead to the best results regarding quality of the solution and computation time. In this thesis we want to build an algorithm using machine learning that approximates this optimization problem and compare them to existing heuristics.

1.2 Machine learning on graphs

There exist many tasks in which we want our nodes or edges in graphs to be mapped in a low-dimensional space. One example is mapping all nodes in a two dimensional space which indicate how to draw the graph. But there are some applications in which we want our whole graph to be mapped in a low dimensional space. Here an example is deciding whether a molecule is toxic or not. The basic idea of machine learning on graphs is that entities like nodes are real valued vectors attached to which are called embeddings. These are calculated with respect to the topological structure.

Node2Vec

This is a method of how to embed the nodes of an undirected graph in a p-dimensional space [AG16]. The function to be learned is $f : V \to \mathbb{R}^p$ which can be represented as a $|V| \times p$ matrix that has the p-dimensional embeddings of all nodes in the different columns. Define a strategy S like depth-first search or breadth-first search of sampling a random walk from a given node and with that define the network neighbourhood $N_S(u)$ as the set of nodes being discovered from node u by strategy S. Then the optimization problem is

$$\max_f \sum_{u \in V} \log(Pr(N_S(u)|f(u)))$$

where $Pr(N_S(u)|f(u))$ is the probability of discovering the set $N_S(u)$ when starting at u. By assuming the independence of discovering two different nodes the formula is equivalent to

$$\max_f \sum_{u \in V} \sum_{v \in N_S(u)} \log(Pr(v|f(u)))$$

and define

$$Pr(v|f(u)) = \frac{\exp(f(v) \cdot f(u))}{\sum_{v_i \in V} \exp(f(v_i) \cdot f(u))}$$

Now the optimization problem is defined and can be optimized by gradient descent with the embeddings of the nodes as variables. With the embeddings for the nodes the edges can be embedded as concatenation, sum or componentwise multiplication of the two adjacent nodes.

This is an unsupervised mapping strategy since we don't need any labeling of the nodes. In the master thesis of Yani Kolev[Kol19] node2vec has been used for crossing minimization. Even though the outline was that node2vec didn't provide good enough representations for their purposes they think that node2vec is designed for larger graphs that don't need geometric informations.

Struct2Vec

This is a method of telling structural identity of some nodes in an undirected graph $G = (V, E)$. The paper [LFRR17] presents a general framework for determining to a given node u some nodes with equaling properties. It consists of four phases.
The first phase consists of predicting structural identity of two nodes. The formula for two nodes $u, v \in V$ is recursively defined as

$$f_k(u, v) = f_{k-1}(u, v) + g(s(R_k(u)), s(R_k(v)))$$
$$f_{-1}(u, v) = 0$$

where $R_k(u)$ is the set of nodes with distance of exactly k from u and with $U \subseteq V$ being a set of nodes $s(U)$ is defined as the ordered degree sequence of those nodes where g is a metric for measuring the distance between two of those sequences.

The second phase constructs a multilayer graph M with edge weights that represents the structural idendity of nodes. The graph M consits of $k^* + 1$ layers with k^* being the diameter of G. Each layer consists of a complete graph with the same set of nodes as the given graph G at beginning. Then for each layer $k \in \{0, 1, .., k^*\}$ the weight of an edge between nodes on the same layer is defined as

$$w_k(u, v) = e^{-f_k(u,v)}$$

and every node $v \in V$ in layer k is connected to its corresponding node in the neighbouring layers meaning v_{k-1} and v_{k+1}. The weight for these edges is

$$w(u_k, u_{k-1}) = 1$$
$$w(u_k, u_{k+1} = \log(\Gamma_k(u) + e)$$

where $\Gamma_k(u)$ is the number of edges in layer k adjacent to u that have more than average weight on this layer.

The third phase tells how to generate a random sequence. For a given node $v \in V$ the random sequence of fixed low lenght starts with the corresponding node in layer zero. Before deciding which node comes after node u_k the algorithm decides whether to stay in this layer with a constant probability $q > 0$ and in the other case changes the layer. In the nontrivial case where two neighbouring layers exist the unnormalized probabilities of changing to layer $k + 1$ is $w(u_k, u_{k+1})$ and to layer $k - 1$ is $w(u_k, u_{k-1})$. In layer k the unnormalized probability of going to node v after node u is $e^{-f_k u,v}$. This process is repeated such that a set of random sequences are determined.

The fourth step is learning an embedding of a node given those sequences. Skip-Gram[MCCD13] has proven to be effective for this.

Also struct2vec with its four presented phases is an unsupervised technique.

Learning Combinatorial Optimization Algorithms over Graphs

Combinatorial optimization problems are often $\mathcal{NP}$-hard which makes it not easy to handle them. One approach is to exactly solve the problem which gets very hard with rising instance size. An computationally easier approach is approximization. This is often realized by greedy algorithms. Given a partial solution the algorithm computes a bigger partial

solution until our solution fulfills the constraints of the problem for example if we want to approximate MinimumVertexCover we start with an empty set of nodes and try to add nodes until the nodes cover all edges.

This paper [HD17] provides a general framework of learning an approximation algorithm. We introduce a learned funtion $Q(h(S), v)$ which maps our partial solution S and a given node v to a single scalar that tells us the quality of our new solution $S' = S \cup \{v\}$ with the help of a helper function h. The helper function h maps our partial solution S to a combinatorial structure satisfying the constraints of the given problem. Given a optimal Q function we just have to add the node v which maximizes $Q(h(S), v)$ and repeat this until our solution fulfills the constraints of the problem.

Given a directed graph $G = (V, E)$ with a edge weight function $w : E \to \mathbb{R}^+$ the Q function works as follows. First an p-dimensional feature embedding μ_v for each node v is calculated recursively with μ_v^0 as the p-dimensional zero vector and

$$\mu_v^{t+1} = relu(\theta_1 x_v + \theta_2 \sum_{u \in N(v)} \mu_u^t + \theta_3 \sum_{u \in N(v)} relu(\theta_4 w(v, u)))$$

where relu is the rectified linear unit defined componentwise as $\text{relu}(x) = max\{0, x\}$, $N(v)$ is the neighbourhood of a node v, x_v is one if and only if $v \in S$ and $\theta_1 \in \mathbb{R}^p$, $\theta_2, \theta_3 \in \mathbb{R}^{p \times p}$, $\theta_4 \in \mathbb{R}$ are the model parameters. Then our embedding is $\mu_v = \mu_v^T$ which contains informations about its T-hop neighbourhood. Usually T is a small number like four. Then Q is approximated with

$$\hat{Q}(h(S), v, \Theta) = \theta_5^\top relu([\theta_6 \sum_{u \in N(v)} \mu_u, \theta_7 \mu_v])$$

where $[.,.]$ is the concatenation and $\theta_5 \in \mathbb{R}^{2p}, \theta_6, \theta_7 \in \mathbb{R}^{p \times p}$ are also model parameters which make $\Theta = \bigcup_{i=1}^{7} \theta_i$ the set of all model parameters. These are learned using reinforcement learning. In experiments they outperformed existing approximation algorithms by far with the average approximation ratio as metric.

Neurosat

Another approach was presented with neurosat [DS19]. It is a neural network based on graphs that learned the satisfiabilaty problem. A set of clauses is represented as a undirected graph by adding for every variable two nodes representing its two complementary literals. Each literal is connected with the clause it appears in and the complementary literals are also connetced with an different edge meaning they exchange different messages. Each node is an embedding accosciated which are at beginning L_{init} for each literal and C_{init} for each clause which are learned vectors.

Then the nodes send learned messages over the edges in one message passing iteration. The embeddings of the literals and clauses are stored in two learned LSTMs with the initial hidden states being just zeros. They used two multilayer perceptrons L_{msg} and C_{msg} which calculate an message vector from the embedding of a literal or a clause. L_{vote} is another multilayer perceptron which calculates the scalar vote of an embedding. We say $l \in c$ if and only if literal l is part of the clause c. In one message passing iteration one clause c gets the old hidden state and the sum over all literals that appear in c of the message vector $\sum_{l \in c} L_{\text{msg}}(l)$ as input. Then the literal l gets updated which also gets the old hidden state, the old state of its complement literal and the sum over all clauses that the literal appears

in of its message vector $\sum_{c \ni l} C_{\text{msg}}(c)$. They did 28 iterations of message passing iterations and trained the network to minimize the sigmoid cross-entropy between the true binary label and the mean of all literal votes mean($\{L_{\text{vote}}(l) | l \text{ is literal}\}$). The true binary label is 1 if and only if the set of clauses are satisfiable

After training they found out that if the network says a set of clauses is satisfiable usually one of the complementary literals has a higher vote than the other. This directly corresponded to the assignment of the variables and they could decode a satisfying assingment most of the time.

Even though the outline was that their approach didn't beat existing SAT solvers they say they think artificial intelligence can help in the future to improve the state of the art.

Graph Networks

Deepmind created a Graph Nets library. It works with Tensorflow and this framework creates a neural network that operates on graphs[BHB+18]. It works on attributed directed graphs which means that each edge, each node and the whole graph are one embedding attached to. A graph network updates these attributes only with respect to the underlying topology and its attributes. It can be used for supervised learning.

1.3 Contribution Outline

We developed two heuristics for approximating the one-sided crossing minimization using Deepmind's Graph Nets library with two different approaches.

Learned leftmost

In this approach we build our permutation from left to right node by node. We start with our given graph and then try to predict the node that has to be the leftest in a permutation. Since sometimes one can choose two nodes to be the leftest and still reach the optimum number of crosses we define functions that map a node to a number which tells us if we can put it in the left of a permutation.

Learned switch

In this approach we start in a given permutation. Then we try to predict for two neighbouring nodes if we have to switch them. We defined a global metric that tells us if we have to switch nodes because we don't want to act greedy and just reach a local minimum.

We were able to create heuristics that show good results in comparison to existing heuristics.

2. Preliminaries

In this chapter we introduce some preliminaries that are used in this thesis.

2.1 Directed Graphs

A *directed graph* is an abstract structure which represents to a given set of objects some relations. We call the objects nodes and the realtions between them edges that point from a sender node to a receiver node. A directed graph can be drawn by drawing points for the nodes and lines from the sender to the receiver node for the edges. An example could be a network map. Each stop is represented as a node and if a bus or train drives from one stop to another an edge is directed like this.

Formally we define a directed graph $G = (V, E)$ as tuple with V being the set of nodes and E being the set of edges. An edge $e \in E \subseteq V \times V$ is a ordered pair $e = (v_i, v_j)$ where the node is directed from v_i to v_j.

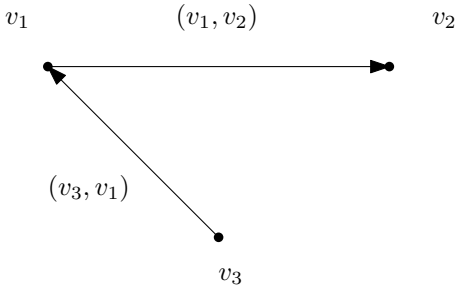

Figure 2.1: example of an directed graph

2.1.1 Two Layered Network

A *two layered network* $N = (V, U, E, x)$ with $V = \{v_1, v_2, .., v_n\}$ and $U = \{u_1, u_2, .., u_m\}$ describes the directed graph $G = (V \dot{\cup} U, E)$. The nodes of this graph can be divided into two distinct sets such that all the edges are between the two distinct sets and each edge $e \in V \times U$ is directed from V to U. Additionaly x is an *ordering* of U. Define an ordering x of a set S as

a bijective function $x : S \to \{1, 2, .., |S|\}$ where each element is assigned number called rank.

If we also have an ordering y of V we can draw the graph by placing the nodes of the two distinct sets on two horizontal distinct lines where the nodes are sorted by both orderings. If the elements in a set have a subscript like $B = \{b_1, b_2, .., b_m\}$ we say x is a trivial order when $x(b_i) = i$.

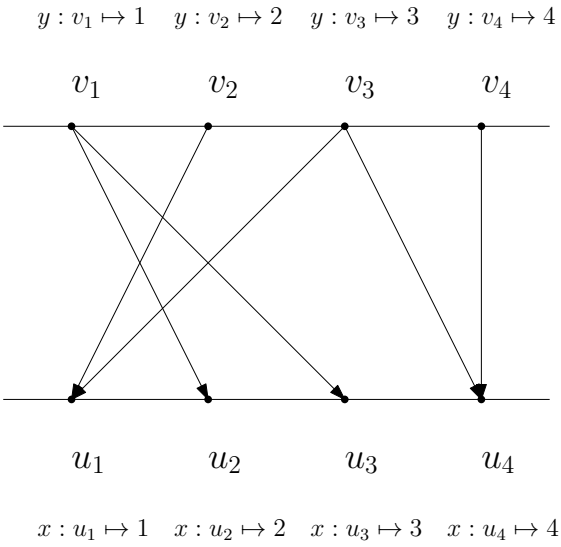

Figure 2.2: two layered network $N = (V, U, E, x)$ with trivial ordering x

2.1.2 Attributed directed graphs

An *attributed directed graph* $G = (V, E)$ is a graph where each node is a d_V dimensional, each edge a d_E dimensional and the whole graph a d_G dimensional vector called embedding associated. Define $e_V : V \to \mathbb{R}^{d_V}$ as the function that returns the embedding of a node, define $e_E : E \to \mathbb{R}^{d_E}$ as the function that returns the embedding of an edge and define $e_G : \{G\} \mapsto \mathbb{R}^{d_G}$ that returns the embedding of the whole graph also called the global embedding.

2.1.3 Random Graphs

In this subsection we will describe a random graph generator.

2.1.3.1 Erdős–Rényi model

Consider a Graph $G = (V, E)$ with n nodes and m edges. The $G(n, p)$ model decides independently for each edge with probability p whether it is included in the graph or not and therefore chooses a graph with n nodes and $m \sim \text{Bin}(\binom{n}{2}, p)$ edges.

Since we want to generate two layered networks the model has to be changed slightly. Consider a two layered network $N = (V, U, E, x)$ with n nodes in V and m nodes in U. The $G(n, m, p)$ model decides independently for each possible edge $e \in V \times U$ with probability p whether it is included in the graph or not and therefore $|E| \sim Bin(n \cdot m, p)$. The order x of U is random.

2.2 Neural Networks

A Neural Network is like a calculation rule of some kind of input with some variables. In most cases the Variables are initialized randomly. The goal is to change the variables to some values that the network output is what we want it to be by training the network.

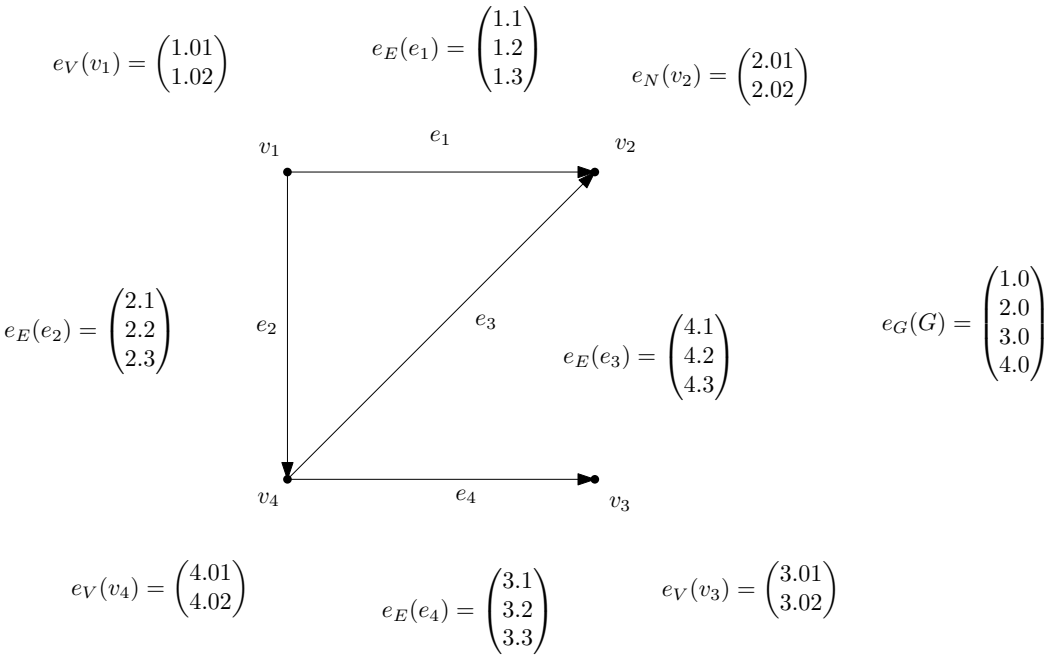

Figure 2.3: attributed directed graph $G = (V, E)$

2.2.1 Components

Now we will explain some components from which one can build a neural network. These small components are already a neural network, however composing different components can make a network stronger. Every component is a calculation rule of an input vector $x \in \mathbb{R}^n$ to an output vector $z \in \mathbb{R}^m$.

2.2.1.1 Perceptron

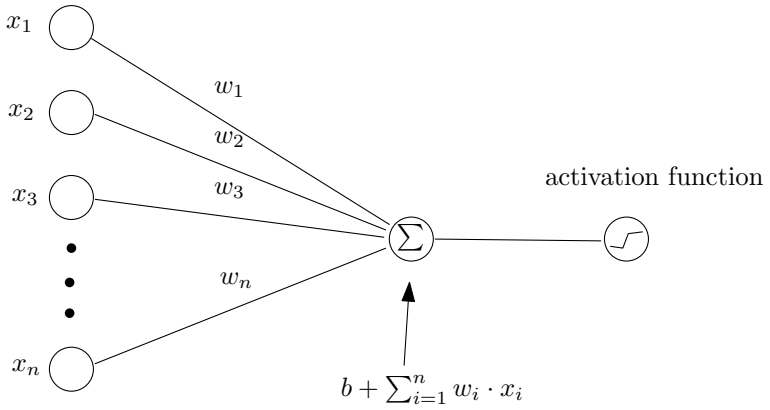

Figure 2.4: perceptron

One common known architechture is a *perceptron*. From our input x a single scalar $z \in \mathbb{R}^1$ is calculated from. In the first step the weighted sum of x is calculated and the bias is added. In the last step a so called activation function is applied to that. Some examples for activation functions are

- $\tanh : \mathbb{R} \to \mathbb{R}$
- sigmoid function : $\mathbb{R} \to \mathbb{R}, x \mapsto \frac{1}{1+e^{-x}}$
- rectified linear unit : $\mathbb{R} \to \mathbb{R}, x \mapsto max\{0, x\}$

In a perceptron the learned values are the weights $w_0, w_1, ..., w_n \in \mathbb{R}$ and the bias $b \in \mathbb{R}$.

2.2.1.2 Multilayer Perceptron

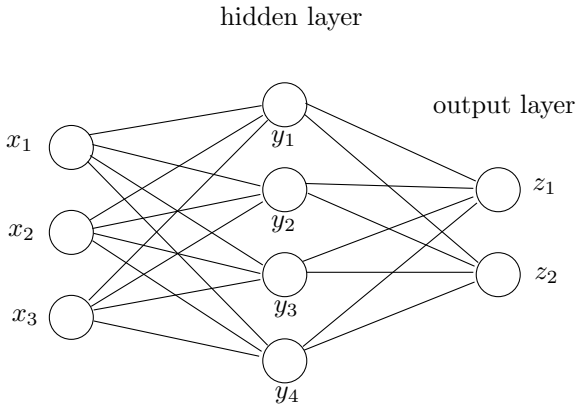

Figure 2.5: multilayer perceptron

One can also compose some perceptrons and then get a multilayer perceptron. The figure above shows a multilayer perceptron with one hidden layer however an arbitrary amount of hidden layers are possible. Note that in the figure for the perceptron there is an extra circle for the activation function which isn't in this figure. A multilayer perceptron also has activation functions but for simplicity they aren't pictured.

The value on a hidden layer or the output layer is the weighted biased sum of the values of the layer before and some activation function on top. Here our learned values are once more the weights and biases of the different layers. Our output vector $z \in \mathbb{R}^m$ can have any dimension, depending on the number of neurons on the output layer.

2.2.1.3 Layer Normalization

This component normilizes our input vector x which means that $z \in \mathbb{R}^n$ has the same dimension. Given the input vector x the output is

$$y = s \circ \frac{x - \mu}{\sigma + \epsilon} + o$$

with

$$\mu = \frac{1}{n} \sum_{i=1}^{n} x_i$$

$$\sigma = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (x_i - \mu)^2}$$

where $\circ$ is the componentwise multiplication. The learned variables are $o \in \mathbb{R}^n$ and $s \in \mathbb{R}^n$ and ϵ is a constant. Typically s is initialized with only ones as entries and o with only zeros as entries.

2.2.2 Learning

The basic idea is that we feed the network a chunk of data and the network tries to change the variables a little bit so it performs better on this perticular chunk of data and repeat this step.

2.2.3 Loss

To make our network better we need a function that tells us how good our network performs on data which means it shows us the difference between our actual network and the network we want it to be. One takes a function where lower values mean a better performance. Let's say we want our network to predict the degrees of each node. The prediction of our network is $z \in \mathbb{R}^n$ with the z_i being the node degree prediction for the node v_i. We can calculate the true label l for the graph which means l_i is the degree of v_i. With those vectors we have to define a metric called *loss* for how bad our prediction is. Given our prediction z and our true label l some common known loss functions are

- mean squared error $\frac{1}{n}\sum_{i=1}^{n}(z_i - l_i)^2$
- mean absolut error $\frac{1}{n}\sum_{i=1}^{n}|z_i - l_i|$

If we have a dataset containing some graphs we can calculate the average loss of all graphs. Now an easy approach to make our network better is to iterate over all learnable varibales and change them slightly higher or lower and check if our loss gets better or worse. If it gets worse then reverse our change of the variable. This algorithm will make our network better over time however a more mathematical approach and the actual state of the art is the gradient descent. The basic idea is the same but it is realized a little different.

2.2.4 Gradient Descent

Now consider our loss function as a function from $f : \mathbb{R}^n \to \mathbb{R}$ where n is the number of learnable variables. Now let's say our current variables are $v \in \mathbb{R}^n$. The negative gradient $-\nabla f(v)$ tells us the direction in which the function f at $f(v)$ decreases the fastest. So in other words if we move our actual variables v in the direction of $-\nabla f(v)$ our loss decreases the fastest in comparison to all other directions. So if we update our variables $v' = v - \gamma\nabla f(v)$ where γ is called the learning rate we move our variables in this direction and hope our loss gets better. If we repeat this algorithm our loss hopefully gets better.

One could calculate the average loss of all the data we have but this would end up in a not handable computational complexity. In practical applications we feed a predefined number inputs the network, this is called the *batch size*.

2.3 One-Sided Crossing Minimization

Consider a two layered network $N = (V, U, E, x)$ with n nodes in V and m nodes in U. The solution to the problem is an ordering y of the nodes V such that the number of crossings is the minimum of all possible orderings if we draw this graph. Note that the number of crossings just depends on the given graph and the permutation of V since the ordering x of U is given with the two layered network.

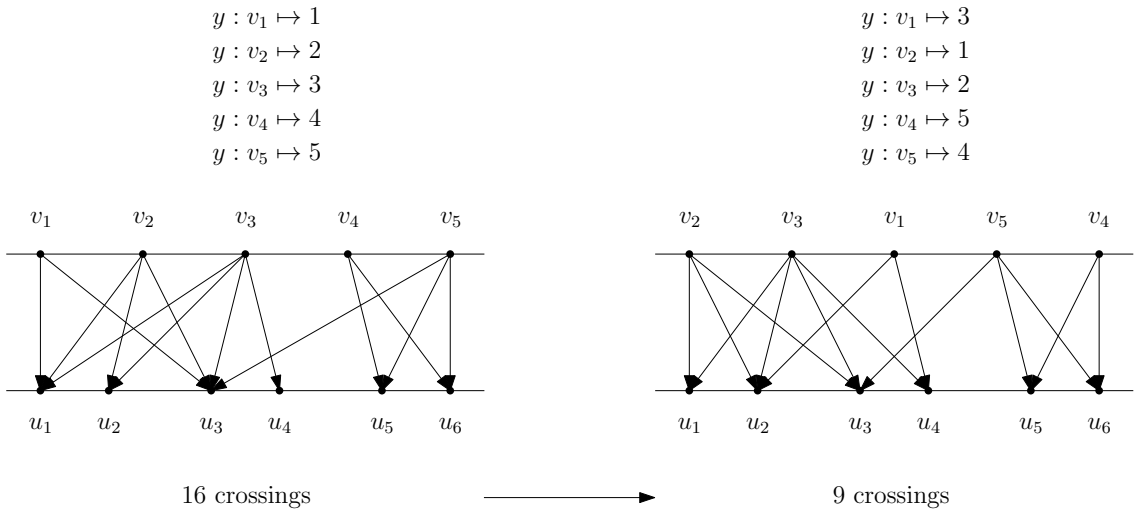

Figure 2.6: bad order on the left and optimal order on the right

Define $N(v) = \{u|(v,u) \in E \text{ or } (u,v) \in E\}$ as the Neighbourhood of v and define the crossing matrix $C = (c_{i,j})$ of a two layered network where

$$c_{i,j} = \begin{cases} \perp, & i = j \\ |\{(u_m, u_n) \in N(v_i) \times N(v_j) | x(u_m) > x(u_n)\}|, & i \neq j \end{cases}$$

is the number of crossings between the edges adjacent to v_i and v_j when v_i is on the left of v_j.

The order y of the nodes N can be also expressed as a order matrix $M = (m_{i,j}) \in \{0, 1, \perp\}^{n \times n}$. The order matrix is defined as

$$m_{i,j} = \begin{cases} \perp, & i = j \\ 1, & y(v_i) < y(v_j) \\ 0, & y(v_i) > y(v_j) \end{cases}$$

which means $m_{i,j}$ is 1 if and only if v_i is on the left of v_j. Given the order Matrix M one can calculate the ordering y of V with $y(v_n)$ being the sum of nonzero entries in the n-th column.

With the order Matrix M or its corresponding ordering y and the correlation matrix C the number of crosses is

$$\text{crosses}_C(y) = \text{crosses}_C(M) = \sum_{i=0}^{n} \sum_{\substack{j=0 \\ j \neq i}}^{n} m_{i,j} * c_{i,j}$$

Since a two layered network N defines its correlation matrix C the subscript can also be N.

2.3.1 $\mathcal{NP}$-completeness of the decision problem

We will now consider the decision problem of the One-Sided Crossing Minimization which is the following

$$\begin{aligned}\text{Instance: two layered network } N = (V, U, E, x), k \in \mathbb{N}_0\\ \text{Question: order } y \text{ of } V \text{ such that } \text{crosses}_N(y) \leq k\end{aligned}$$

and show that this problem is $\mathcal{NP}$-complete as a short review of the proof [Wor94]. This is done by reducing from the problem Minimum Feedback Arc Set which is the following

$$\begin{aligned}\text{Instance: directed graph } G = (V, E), k \in \mathbb{N}_0\\ \text{Question: } F \subseteq E, |F| \leq k \text{ such that } G' = (V, E \setminus F) \text{ is acyclic}\end{aligned}$$

If our graph has n self edges then the feedback arc set has to contain all these self edges because each of these edges result in one cycle. Therefore we have to decide if our graph with all self edges removed has a feedback arc set of size at most $k - n$ and if $k - n$ is less than zero we know the graph has no solution of the questioned size. This is why we can assume that graph G has no self edges.

Given now an instance of the feedback arc set $G = (V, E), k \in \mathbb{N}$ with $V = \{v_1, v_2, ..., v_n\}$ and $|E| = m$ we build our new two layered network $N' = (V', U', E', x'), k' \in \mathbb{N}_0$ as following:

$$\begin{aligned}V' =& V\\ U' =& \bigcup_{e \in E} C_e \text{ with } C_e = \{c_e^1, c_e^2, c_e^3, c_e^4, c_e^5, c_e^6\}\end{aligned}$$

So for each node in our graph G we put a node in V' and for each edge e in our graph we put a clump C_e consisting of six nodes in U'.
Now we have to add the edges. Each $v_i \in V'$ and each clump C_e are connected with two edges. We differentiate between three cases:
If e is an outgoing edge from v_i we connect v_i with c_e^1 and c_e^5. If e is an incoming edge to v_i we connect v_i with c_e^2 and c_e^6 and if e is not adjacent to v_i we connect it with the two middle nodes c_e^3 and c_e^4. All posibilities are in the figure below illustrated.

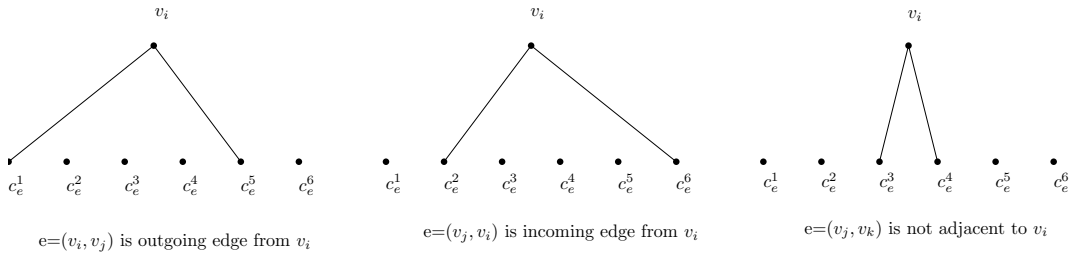

Figure 2.7: Three cases for edges

Let x' be an order of U' where each clump stays together. The order between different clumps is not defined nor nessecary. We prove now that our graph G has feedback arc set of size at most k if and only if our two layered network N' has a order y' of V' with at most

$$k' = 4\binom{n}{2}\binom{m}{2} + m\binom{n-2}{2} + 4m(n-2) + m + 2k$$

crosses.

Lemma 2.1. *Let y' an ordering for V' with $B = \{(v_i, v_j) \in E | y'(v_i) > y'(v_j)\}$. Then the number of crosses is*

$$m := 4\binom{n}{2}\binom{m}{2} + m\binom{n-2}{2} + 4m(n-2) + m + 2|B|$$

For $e_1, e_2 \in E$ and $e_1 \neq e_2$ we have two clumps. Each of the clump has two edges to each node in V_1. Now consider the crosses between edges from different clumps. So for each pair of nodes v_1, v_2 and each pair of edges e_1, e_2 we have four crosses which makes in total $4\binom{n}{2}\binom{m}{2}$ crosses between different clumps.
Now consider the crosses between edges from the same clump $C_{(v_i,v_j)}$. Since there are $n-2$ nodes that aren't incident to (v_i, v_j) in G, in our constructed network N' these nodes are connected to $c^3_{(v_i,v_j)}$ and $c^4_{(v_i,v_j)}$. So for each pair of nodes in these $n-2$ nodes we get one crossing giving $\binom{n-2}{2}$ crosses. Now summing over all clumps this gives $m\binom{n-2}{2}$ crosses. The node v_i is connected to $c^1_{(v_i,v_j)}$ and $c^5_{(v_i,v_j)}$ and the node v_j is connected to $c^2_{(v_i,v_j)}$ and $c^6_{(v_i,v_j)}$ which both cross exactly two edges incident to the $n-2$ other nodes giving $4m(n-2)$ crosses. Now the only uncounted crosses are between the edges $(v_i, c^1_{(v_i,v_j)})$, $(v_i, c^5_{(v_i,v_j)})$ and $(v_j, c^2_{(v_i,v_j)})$, $(v_j, c^6_{(v_i,v_j)})$. If $y'(v_i) < y'(v_j)$ then there will be one cross and otherwise then there will be three crosses. Summing over all clumps this gives $m + 2|B|$ crosses which leads to the lemma. □

Let $G = (V, E)$ now have a feedback arc set F of size at most k. Since $G = (V, E \setminus F)$ has no cycles we can find an ordering y by topological sort such that for every edge $(v_i, v_j) \in E \setminus F$ the nodes in the new graph are ordered $y'(v_i) < y'(v_j)$. The just shown lemma implies that the two layered network N' has at most k' crosses.

Let N' now have an order y' of V' such that there are at most k' crosses. Define $B = \{(v_i, v_j) \in E | y'(v_i) > y(v_j)\}$. The lemma implies that $|B| <= k$. Now we know that for each path $(v_a, v_b), (v_b, v_c), ..., (v_y, v_z) \subset E \setminus B$ the ordering is $y(v_a) < y(v_z)$ so there can't be any cycle and therefore B is a feedback arc set of size at most k.

2.3.2 Reduction to Integer Linear Programm

Define the entries of the order matrix as binary variables $m_{i,j}$ for $i, j = 1..n_1$ and $i \neq j$, but there are many assignments of the order matrix which don't represent a permutation. The first constraint is that v_i is on the left of v_j if and only if v_j is on the right of v_i. This equivalence can be expressed in constraints as

$$m_{i,j} = 1 - m_{j,i} \qquad i \neq j$$

The second one is keeping the transitivity. When node v_i is on the left of v_j and v_j is on the left of v_k then v_i is on the left of v_k. This implication can be expressed in constraints as

$$m_{i,j} + m_{j,k} - m_{i,k} \leq 1 \qquad i \neq j, j \neq k, i \neq k$$

Then the number of crosses is

$$\sum_{i=0}^{n} \sum_{\substack{j=0 \\ j \neq i}}^{n} m_{i,j} \cdot c_{i,j}$$

which makes our ILP to

$$\text{minimize} \sum_{i=0}^{n} \sum_{\substack{j=0 \\ j \neq i}}^{n} m_{i,j} \cdot c_{i,j} \text{ subject to}$$

1.	$m_{i,j} + m_{j,k} - m_{i,k} \leq 1$	$i \neq j, j \neq k, i \neq k$
2.	$0 \leq m_{i,j} \leq 1$	$i \neq j$
3.	$m_{i,j} = 1 - m_{j,i}$	$i \neq j$

2.3.3 Existing Heuristics

Given a two layered network $N = (V, U, E, x)$ there exist many approaches to approximate the ordering y of V. Some algorithms will need the crossing matrix C.

2.3.3.1 Barycenter

The barycenter heuristic can be used to approximate optimization problems. However the minimization of crossings in a hierarchical drawing of a graph is the most famous application of the algorithm. It calculates the position of each node in V as the average position of all its adjacent nodes in U. In practice this simple algorithm performs very well and is often used in graph drawing.

On the theoretical side this algorithm does not have an upper bound for its error. It has been shown that the relative number of crossings to the optimum is up to $\theta(|V|)$ times higher.

The computational complexity is the result of computing for each node the average position of its adjacent nodes which takes $\theta(|U|)$ and then sorting the nodes takes $\theta(|V| \log(V)))$ which leads to $\theta(|U| + |V| \log(V)))$ for the complete algorithm[MS05].

2.3.3.2 Sifting

The sifting algorithm was originally used for reducing the number of vertices in reduced ordered binary decision diagrams but can also be used in the one-sided crossing minimization. The algorithm starts with a given permutation and then sifts one node after another until every node has been sifted once. Sifting a node means fixing every other node and place this node on every possible position and compute every time the number of crosses. Then place the node at the local minimum and continue with the next node.

There exist different variants of the algorithm. Since many use cases already have a permutation for both of the vertices and the question is to find a better permutation one can start with this given permutation or start with a random permutation. There exist also three different methods to choose the next node. The first variant chooses the nodes in the given permutation for example from the left to the right. The second one chooses the nodes randomly. A more accurate variant sorts the nodes by the degree of them.

By swapping one node u with its right neighbour v the number of crossings is $cross_{new} = cross_{old} - c_{u,v} + c_{v,u}$ which takes constant time when given the crossing matrix $(c_{i,j})$. The implementation of sifting is swapping the node until it is on the leftmost or rightmost position and then swap it on the other side. This takes $\theta(|V|)$ which makes sifting all nodes in $\theta(|V|^2)$[MSM98].

2.3.3.3 Median

The median heuristic is very similar to the barycenter heuristic. Instead of computing the average position of all adjacent nodes it computes the median of all nodes. If a node has no adjacent nodes, then it is placed on the left and if two nodes are assigned to the same position then the one with odd degree is placed on the left of the one with even degree.

It has been shown that the median heuristic computes drawings with at least three times worse crossings than the optimal solution. There exist graphs where the median heuristic is exactly three times worse than the optimum so there can't exist a lower bound. However for some graphs there exist better bounds. Consider the following theorem.

Theorem 2.2. *Suppose* $\epsilon > 0$ *and* $0 < c < 1$. *Then there exist an* N_0 *such that the graph has at least* N_0 *nodes on both horizontal lines then the error is at most*

$$\frac{3 - c^2}{1 + c^2} + \epsilon$$

With c being close to one this error is close to one which means that for dense graphs the median heuristic is close to the optimum.
Computing the median position of its adjacent nodes takes $\theta(|U|)$ and sorting takes $\theta(|V|)$ because all the values are in $0..|U|$ by using bucket sort wich leads to a complete complexity of $\theta(|U| + |V|)$[Wor94].

2.3.3.4 Greedy Switch

The greedy switch heuristic also starts in a given permutation. Then it iterates over the nodes in V from left to right. If swapping the actual node with its neighbour will reduce the number of crosses then swap. This is done with every pair of nodes in one iteration. If there had been at least one swap in an iteration the just repeat this algorithm.

2.3.3.5 Greedy Insert

This heuristic works like insertion sort. We start with an empy ordered set for the nodes in V. Then we take one random node and place it in the position that minimizes the number of crosses in this ordered set. This will be repeated until we have an order for the whole set V.

2.3.3.6 Split

This heuristic works recursively like quick sort. First a pivot element $v_i \in V$ is chosen. Then we choose another node $v_j \in V$ and place it in the left partition if and only if $c_{i,j} > c_{j,i}$ which means that the number of crossings between the edges adjacent to v_i and v_j is lower when v_j is on the left of v_i. Otherwise the node is placed in the right. Then we pick a pivot element in those partitions and try to recursively build an order of the elements[Ism12].

3. Graph Network

Neural networks often get sequence of data like a voice sample or a vector like the data of some sensors as input. In these cases the ordering matters because the same sentence with the words reordered can make its content completely different or in the other case if we change the output of two sensors our network won't make good predictions anymore.

But there exist problems where we want our input to be independent of the order and instead define relations on our own between data points. One example of a representation of such a problem are graphs. Many problems can be represented as graphs for example molecular structures. Each atom can be represented as node and the edges between the nodes can indicatate some chemical relations between the nodes.

There exist approaches of neural networks that work on graphs. The basic idea is that nodes which are connected by edges can send messages over the edges. After such a so called message passing step each node can behave differently depending on the messages it receives. Deepmind now build a machine learning framework that works on graphs which will be presented in this chapter [BHB+18]. One message passing step is done by a graph network (GN). It gets a graph as input and the output also a graph.

3.1 Graph

The graph network works on directed attributed graphs. A graph network just updates the embeddings attached to the graph with respect to the structure to it. After updating these embeddings the size of the embeddings can change.

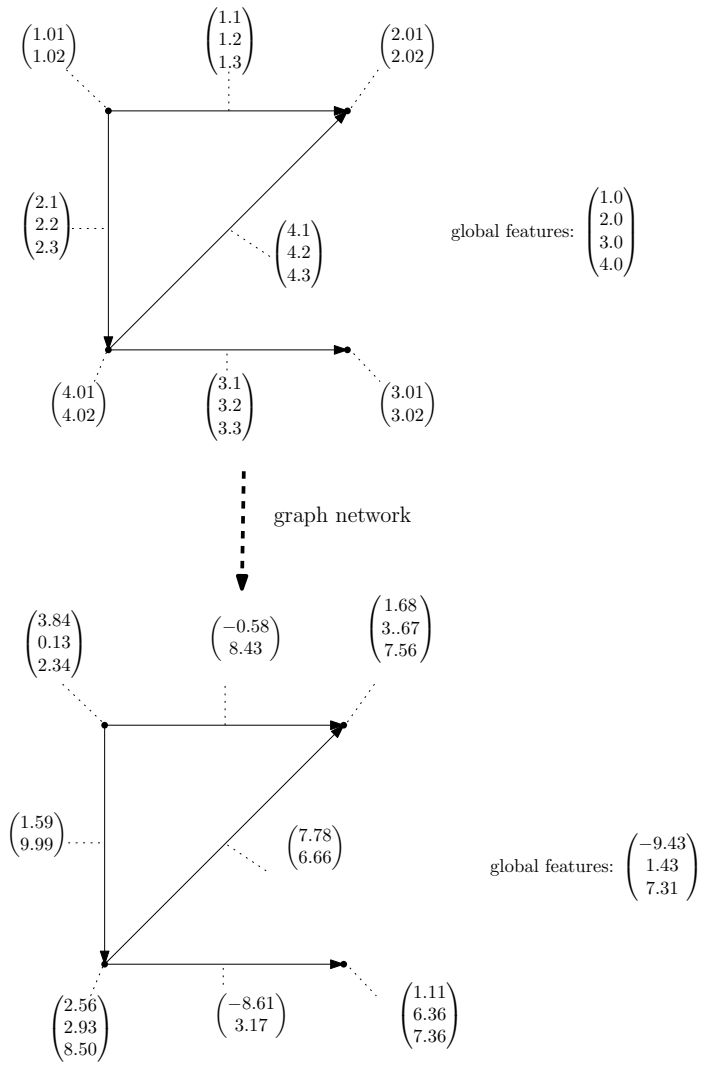

Figure 3.1: graph network operating on a given attributed directed graph

3.1.1 Graph concatenation

One basic operation on graphs is the graph concatenation. If we have two same graphs with possibly different attributes we can concatenate these. This means we consider the same graph but the attributes are the concatenation of both. This means in case of a specific node that we have two different embeddings. This node in the concatenated graph has a vector as attribute which is both vectors concatenated on another. One can see an example in figure 3.2. In this example two graphs are concatenete however an finate number of graphs can be concatenated. We use the symbol $\bigoplus$ to indicate the concatenation of graphs.

3.2 GN block Variants

The framework provides six diffent variants of a GN block, however we only use two of them and build with them complex networks.

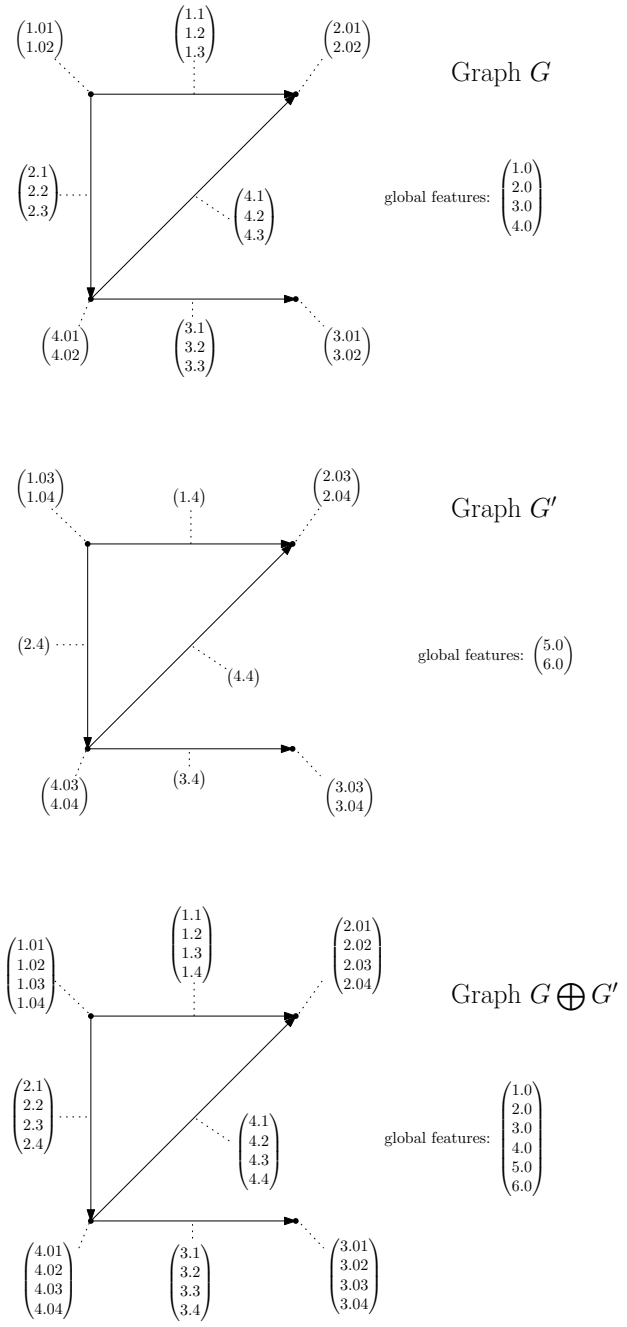

Figure 3.2: graph concatenation of graph G and G'

The variants use update functions ϕ. Update functions are the learned functions that should determine how the embeddings have to be updated. We use multilayer perceptrons with a layernorm layer.

The following variants consist of three phases:

step 1: edge update by ϕ^e

step 2: node update by ϕ^n

step 3: global update by ϕ^u

We will use the term old embedding for the embedding before it gets updated and new embedding for the updated one.

3.2.1 Full GN Block

In the following text we will describe each of the three phases for a full GN block in detail.

Step 1: edge update

In order to update all the edge embeddings for each edge embedding e_k the edge update function ϕ^e gets as input

- old edge embedding e_k
- old node embedding of receiver node v_{r_k}
- old node embedding of sender node v_{s_k}
- old global embedding u

and returns the new edge embedding e'_k.

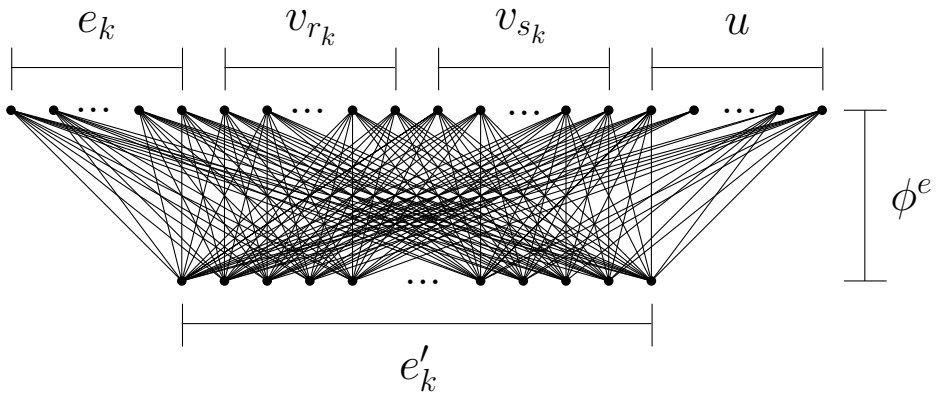

Figure 3.3: edge update function as single layer perceptron

Step 2: node update

In order to update all the node embeddings for each node embedding v_i we first need to determine the set of the old embeddings of edges E'_i that point to this node. Then these are summed up to $\overline{e}'_i$ and with that the node update function ϕ^n gets as input

- summed new embeddings of edges that point to this node $\overline{e}'_i$
- old node embedding v_i
- old global embedding u

and returns the new node embedding v'_i.

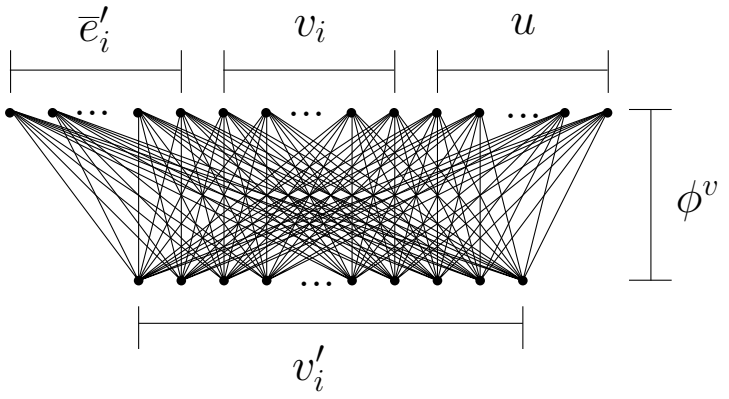

Figure 3.4: node update function as single layer perceptron

global update

In order to update the global embedding u the new embeddings of all nodes are summed up to $\overline{v}'$ and the new embeddings of all edges are summed up to $\overline{e}'$. Then the global update function ϕ gets as input

- summed new embeddings of all edges $\overline{e}'$
- summed new embeddings of all nodes $\overline{v}'$
- old global embedding u

and returns the new global embedding u'.

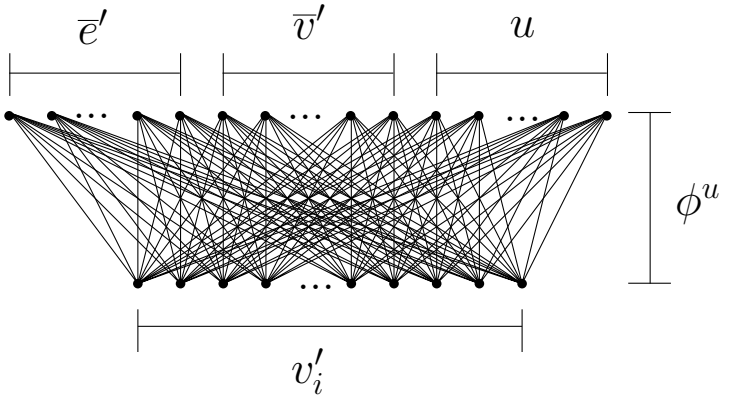

Figure 3.5: global update function as single layer perceptron

3.2.2 Graph Independent (GI) Block

In the following text we will describe each of the three phases for a graph independent block in detail.

edge update

In order to update all the edge embeddings for each edge embedding e_k the edge update function ϕ^e gets as input

- old edge embedding e_k

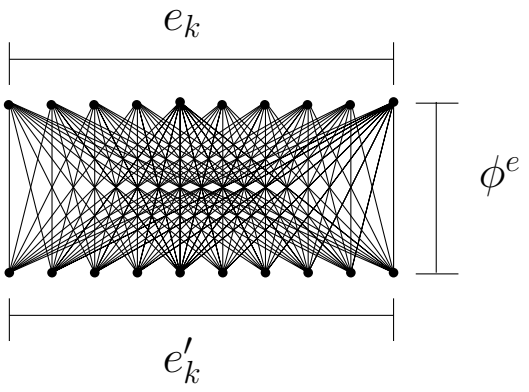

Figure 3.6: edge update function as single layer perceptron

and returns the new edge embedding e'_k.

node update

In order to update all the node embeddings for each node embedding v_i the node update function ϕ^v gets as input

- old node embedding v_i

and returns the new node embedding v'_i.

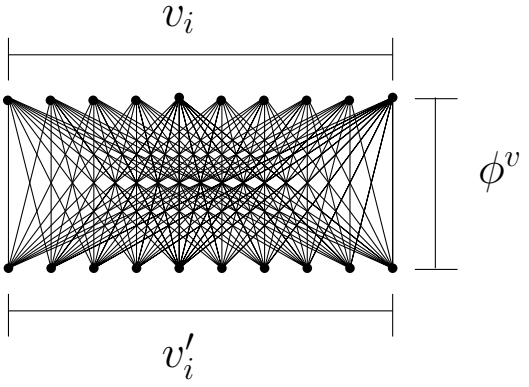

Figure 3.7: node update function as single layer perceptron

global update

In order to update the global embeding u the global update function ϕ^u gets as input

- old global embedding u

and returns the new global embedding u'.

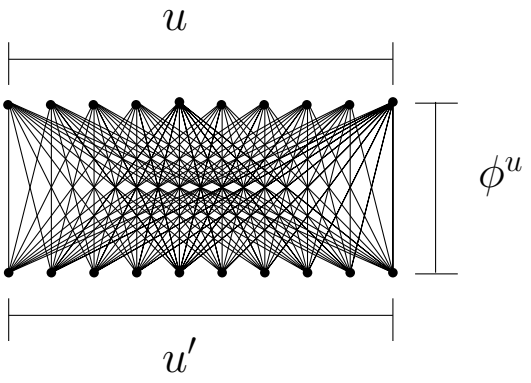

Figure 3.8: global update function as single layer perceptron

3.3 Models

In this section we wil describe some possibilites of composing graph networks together to make them more powerful.

3.3.1 n-Process Encode Decode

This model consists of two GI blocks $GI_{\text{enc}}, GI_{\text{dec}}$ and one core GN block GN_{core}. The blocks are used in three phase with the first phase being the encoder phase, followed by the core phase and the last phase is the decoder phase.

In the encode phase the input graph is given to GI_{enc}. This network updates all embeddings and the output is given to the next phase. The reason for this phase is that in the beginning we give some informations in each embedding and this GI block should transform them in embeddings that the following phases can work with better.

The core phase is a composition of the GN block GN_{core}. Every GN_{core} is given the result of the block before concatenated with the output of the encoder phase. This is repeated n times and the output of the last GN block is given to the next phase.

In the decode phase the graph is given to GI_{dec} which returns the result of this model. This GI block should update the embeddings such that they are the size that is needed. For example if we want a this model to predict the degree of a node we want every node to have a one dimensional embedding which is the node's prediction for the degree.

The number of how often the core GN block is used can be a constant like n or it can be a function depending on the input graph G_{in}. For example we can use as many core iterations as number of nodes the input graph has.

This model is visualized in the figure below.

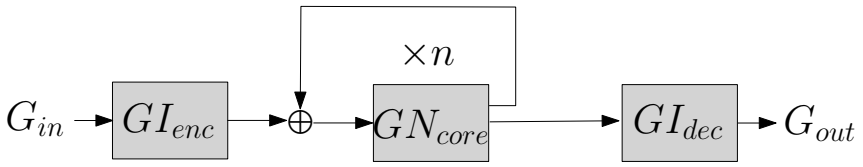

Figure 3.9: Encode-process-decode

3.3.2 n-Layer Process Encode Decode

This model consists of two GI blocks $GI_{\text{enc}}, GI_{\text{dec}}$ and n GN blocks $GN^1_{\text{core}}, GN^2_{\text{core}}, ..., GN^n_{\text{core}}$ which are used in the same phases as the n-process encode decode. The only difference is that this time the core networks are different with different learnable parameters and the graph is passed to each GN block just once. Therefore the number of iterations is fixed in this model in comparison to the n-process encode decode.

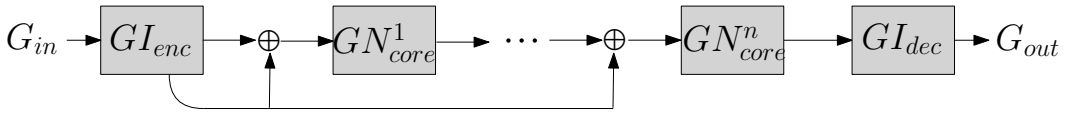

Figure 3.10: Encode-process-decode

3.3.3 n-Recursive Layer Process Encode Decode

This model consists of n GN blocks $GN^1_{\text{core}}, GN^2_{\text{core}}, ..., GN^n_{\text{core}}$ and two GI blocks GI_{enc} and GI_{dec}. The input graph is given to the encoder GI block GI_{enc}. This is given to the first graph network GN^1_{core}. The second graph network gets the concatenation of the encoded input graph and the output of the graph network before. The i-th graph network GN^i_{core} gets the concatenation of the encoded input graph and the output of the $n-1$ graph networks before. Then the decoder network GI_{dec} gets the concatenation the output of the last GN block GN^n_{core} and returns the output graph.

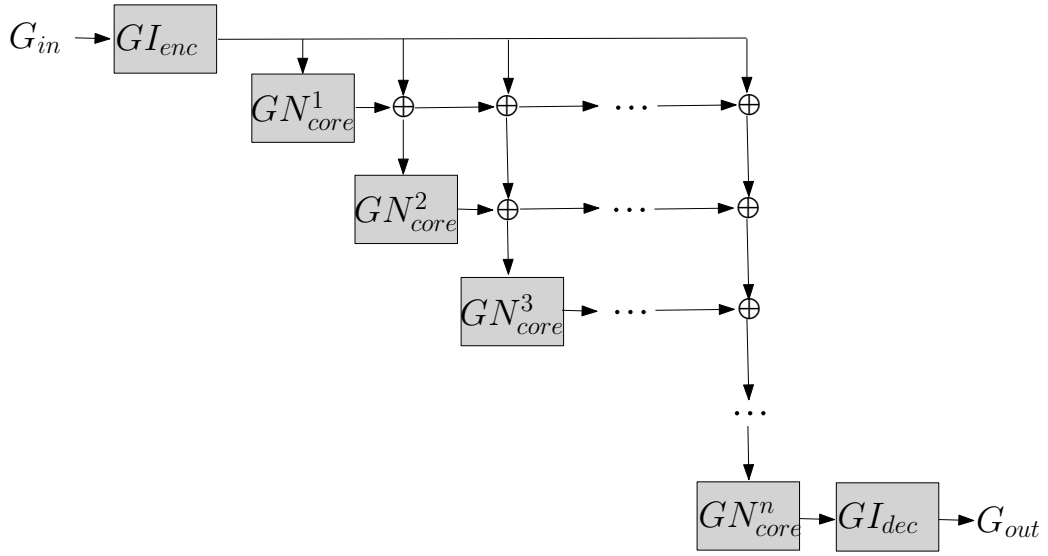

Figure 3.11: n- recursive process encode decode

4. Heuristics Using Machine Learning

4.1 Learned Leftmost Node

In this approach we want to build part by part our permutation and consequently reducing our graph to smaller instances. We want to start with the given graph and then determine the leftest node then delete this node from the graph and then determine the leftest node in this one. By iterating over the number of nodes in V_1 we then have our perfect permutation. This algorithm is visualized in the figure below.

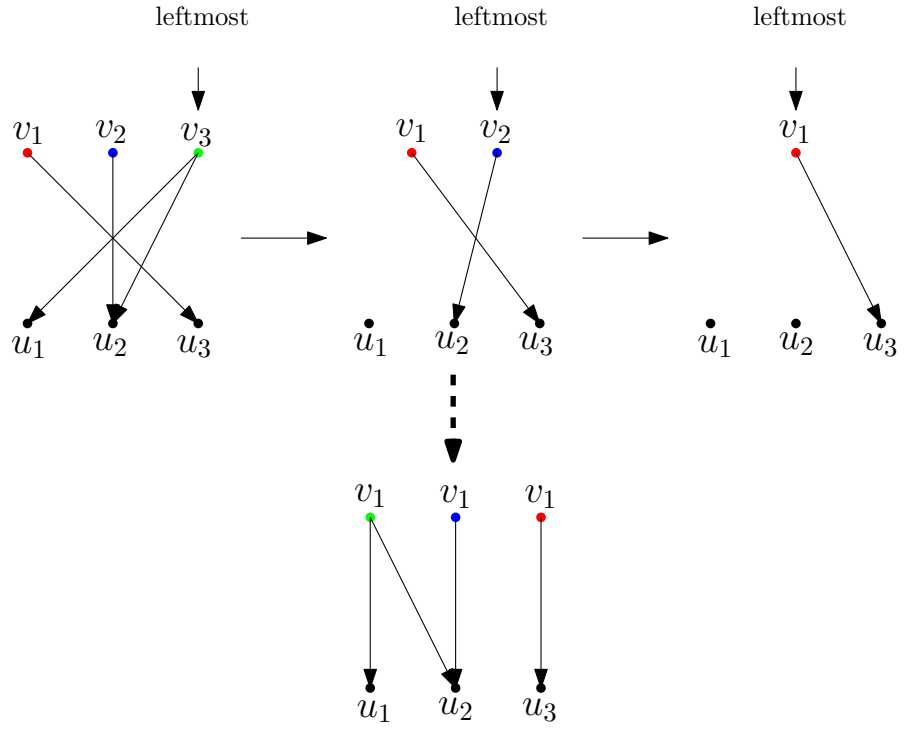

Figure 4.1: Leftmost algorithm visualization

Now we have to describe when a node should be the leftest node. First of all this is nondeterministic since there often exist many permutations with the optimal number of crossings and therefore the leftmost node is not unique. We have to define a metric from which one can tell which node can be the leftest node.

4.1.1 Score One-Sided Crossing Minimization

Consider a two layered network $N = (V, U, E, x)$ with $V = \{v_1, v_2, ..., v_n\}$, $U = \{u_1, u_2, ..., u_m\}$ and a trivial ordering x.The solution to the problem is a vector

$$s = \begin{pmatrix} s_1 \\ s_2 \\ \vdots \\ s_n \end{pmatrix} \in \mathbb{N}_0^n$$

with

$$s_i = \text{score}(v_i)$$

which is defined as the minimum number of crosses if v_i is placed on the leftest position on the upper horizontal line. We call this the score of this node.

4.1.1.1 Reduction to ILP

For every node in V we need to calculate the minimum number of crosses under all permutations with this node being on the leftmost position. This is the one-sided crossing minimization with the additional costraint of one node being on the leftmost position. Since our binary variables tell for every pair of nodes the relative position, we need to tell that one node is on the left of every other node. So for the l-th entry s_l we need to solve the ILP

$$\text{minimize} \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} m_{i,j} \cdot c_{i,j} \text{ subject to}$$

1.	$m_{i,j} + m_{j,k} - m_{i,k} \leq 1$	$i \neq j, j \neq k, i \neq k \quad i,j,k \in \{1,..,n\}$
2.	$0 \leq m_{i,j} \leq 1$	$i \neq j \quad i,j \in \{1,..,n\}$
3.	$m_{i,j} = 1 - m_{j,i}$	$i \neq j \quad i,j \in \{1,..,n\}$
4.	$m_{n,i} = 1$	$i \neq l \quad i \in \{1,..,n\}$
5.	$m_{i,n} = 0$	$i \neq l \quad i \in \{1,..,n\}$

4.1.2 Binary Score One-Sided Crossing Minimization

Consider a two layered network $N = (V, U, E, x)$ with $V = \{v_1, v_2, ..., v_n\}$, $U = \{u_1, u_2, ..., u_m\}$ and a trivial ordering x. The solution to the problem is a vector

$$s = \begin{pmatrix} s_1 \\ s_2 \\ \vdots \\ s_n \end{pmatrix} \in \{0,1\}^n$$

with s_i is one if and only if the score of v_i is the lowest among all scores. Note that there can be more than one entry one.

4.1.3 $[0,1]$-Iterpolated Score One-Sided Crossing Minimization

Consider a two layered network $N = (V, U, E, x)$ with $V = \{v_1, v_2, ..., v_n\}$, $U = \{u_1, u_2, ..., u_m\}$ and a trivial ordering x.The solution to the problem is a vector

$$s = \begin{pmatrix} s_1 \\ s_2 \\ \vdots \\ s_n \end{pmatrix} \in [0,1]^n$$

where this vector is the affine transformated score one-sided crossing minimization vector onto the interval $[0, 1]$. We here map the lowest score to one and the highest score to zero. If all scores are the same we map all entries to one.

4.1.4 Input Graph

Given a two layered network $N = (V, U, E, x)$ with $V = \{v_1, v_2, ..., v_n\}$ and $U = \{u_1, u_2, ..., u_m\}$ with trivial ordering x we have to transform this in a directed, attributed graph. One important question is how to tell the network the properties of this problem in the structure of the graph and its embeddings. Since the two layered network N represents a directed graph we can obtain this structure and add for every edge its reversed edge. We also have to somehow tell the network the order x of the nodes in U. Our idea was to encode this by adding edges between neighbouring in U in condext of the ordering x. Therefore the graph given to the graph network is $G_{in} = (V \dot{\cup} U, E')$ with

$$\begin{aligned} E' &= E'_1 \cup E'_2 \cup E'_3 \cup E'_4 \\ E'_1 &= E \\ E'_2 &= \{(u_i, v_j) \in U \times V | (v_j, u_i) \in E\} \\ E'_3 &= \{(u_i, u_{i+1}) \in U \times U | i \in \{1, 2, .., m-1\}\} \\ E'_4 &= \{(u_{i+1}, u_i) \in U \times U | i \in \{1, 2, .., m-1\}\} \end{aligned}$$

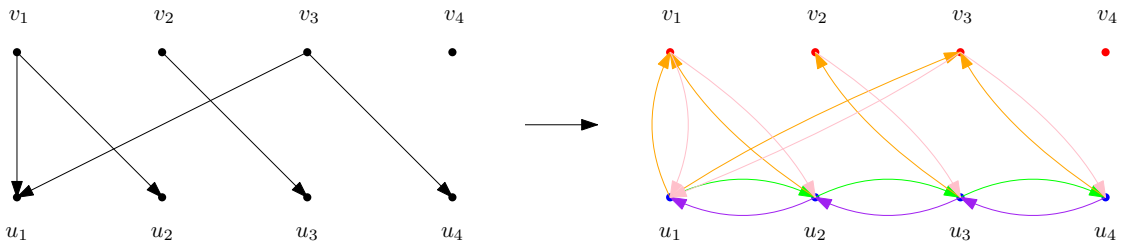

Figure 4.2: Construction of the input graph

Now we have to add embeddings as node, edge and global attributes. The goal is to add some informations that the model could help to update the embeddings. Even though every information from the two layered network N is saved in the structure of G_{in} we can help the network by adding the information. We added every embedding the numbers $|V|, |U|$ and additionally we wanted to add the rank of the nodes in U.

The global feature vector is

$$\begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_{e_G} \end{pmatrix}$$

where u_{e_G-1} is $|V|$, u_{e_G} is $|U|$ and u_1 until u_{e_G-2} is a learned vector.

The node feature vector is

$$\begin{pmatrix} n_1 \\ n_2 \\ \vdots \\ n_{e_V} \end{pmatrix}$$

where n_{e_V-1} is $|V|$ and n_{e_V} is $|U|$. For nodes in V we have n_1 until n_{e_V-2} as a learned vector and the node u_i in U has n_{e_V-2} as i which is the rank of the node and n_1 until n_{e_V-3} is a learned vector.

The edge feature vector is

$$\begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e_{e_E} \end{pmatrix}$$

where e_{e_E-1} is $|V|$ and e_{e_E} is $|U|$. Every E'_i has its own learned vector for the entries from e_1 until e_{e_E-2}.

We want the models to update the embeddings such that the embeddings of the nodes in V predict its binary score or interpolated score.

4.1.5 Algorithm

Our algorithm works like in the beginning of this section described and since we trained our network to predict the binary scores or the interpolated scores we interpreted the node with the hightes vote from our network as the leftmost node.

4.2 Learned Switch

In this approach we want to start in a random order y for V and then we want to iterate and make our solution better and better by switching the positions of some nodes. The only question is how to formally describe when we have to switch nodes. Let v_i and v_j be two nodes in V who appear to be next to each other with v_i being on the left of v_j. When we calculate the optimal number of crossings one time with the restriction v_i is on the left of v_j and the other time v_j is on the left of v_i and we find out that when v_j is on the left of v_i our best possible number of crosses is better then we have to switch these nodes. If these two numbers are the same then that means the there exist permutations with $y(v_i) < y(v_j)$ and $y(v_i) > y(v_j)$ with a perfect solution so we don't switch these nodes.

We can make a graph network predict for two given nodes if they have to be switched or not but this would be very inefficient. First of all our algorithm would take $\theta(|V_1|^2)$ predictions of our neural network. Additionally every node and edge has at the end important informations in its feature vectors and we would like to use more of its information. So we can train our network to predict this decision for every pair of nodes.

With this one undefined state comes up. What should we do when the network predicts for two or even more pairs directly next to each other that they have to be switched? Consider

this question with two pairs and let call pair one with nodes u and v and pair two with nodes v and w. We know we have to swap both of these. We know v should be on the left of u and w should be on the left of v. Then there exists just one way how to order these, they have to be reversed. This is visualized in the figure below.

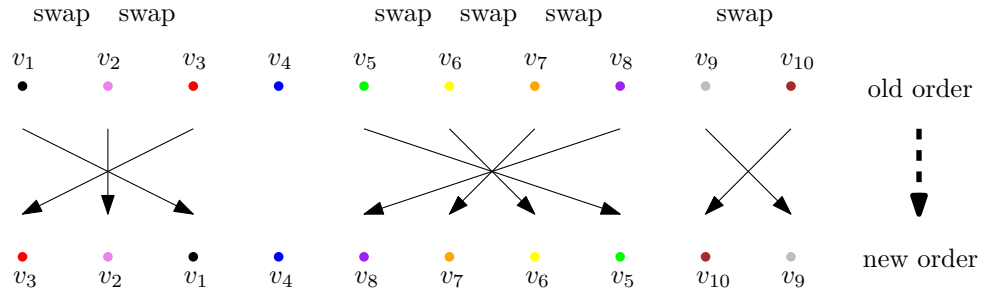

Figure 4.3: Switch algorithm visualization

Now we have to formally describe what we want our graph network to do.

4.2.1 Switch One-Sided Crossing Minimization

Consider a two layered network $N = (V, U, E, x)$ with trivial ordering y of V where $V = \{v_1, v_2, ..., v_n\}$, $U = \{u_1, u_2, ..., u_m\}$ and x is also trivial ordering for U. The solution to the problem is a matrix

$$S = \begin{pmatrix} s_{1,1} & s_{1,2} \\ s_{2,1} & s_{2,2} \\ \vdots & \vdots \\ s_{n-1,1} & s_{n-1,2} \end{pmatrix} \in \mathbb{N}_0^{n-1\times 2}$$

with

$$s_{i,1} = \min_{\substack{y \text{ order of } V \\ y(v_i)<y(v_{i+1})}} crosses_G(y)$$
$$s_{i,2} = \min_{\substack{y \text{ order of } V \\ y(v_i)>y(v_{i+1})}} crosses_G(y)$$

Each row in the solution Matrix S corresponds to a pair of nodes. In row i the left entry is the optimal number of crosses over all orders where v_i is somewhere on the left of v_{i+1} and the right entry is the optimal number of crosses over all orders where v_i is somewhere on the right of v_{i+1}.

4.2.1.1 Reduction to ILP

For every entry in the matrix we need to calculate the minimum number of crosses with an additional constraint that is for $s_{l,1}$ that v_l is somewhere on the left of v_{l+1} and for $s_{l,2}$ that v_l is somewhere on the right of v_{l+1}. So for $s_{l,1}$ we add the constraint $m_{l,l+1} = 1$ and

$m_{l+1,l} = 0$ since we want to force the l-th node before the $l+1$th node we similary set $m_{l,l+1} = 0$ and $m_{l+1,l} = 1$ for $s_{l,2}$. So for $s_{l,1}$ is the solution to the ILP

$$\text{minimize} \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} m_{i,j} \cdot c_{i,j} \text{ subject to}$$

1.	$m_{i,j} + m_{j,k} - m_{i,k} \leq 1$	$i \neq j, j \neq k, i \neq k \quad i,j,k \in \{1,..,n\}$
2.	$0 \leq m_{i,j} \leq 1$	$i \neq j \quad i,j \in \{1,..,n\}$
3.	$m_{i,j} = 1 - m_{j,i}$	$i \neq j \quad i,j \in \{1,..,n\}$
4.	$m_{l,l+1} = 1$	
5.	$m_{l+1,l} = 0$	

and for $s_l, 2$

$$\text{minimize} \sum_{i=1}^{n_1} \sum_{\substack{j=1 \\ j \neq i}}^{n_1} m_{i,j} \cdot c_{i,j} \text{ subject to}$$

1.	$m_{i,j} + m_{j,k} - m_{i,k} \leq 1$	$i \neq j, j \neq k, i \neq k \quad i,j,k \in \{1,..,n\}$
2.	$0 \leq m_{i,j} \leq 1$	$i \neq j \quad i,j \in \{1,..,n\}$
3.	$m_{i,j} = 1 - m_{j,i}$	$i \neq j \quad i,j \in \{1,..,n\}$
4.	$m_{l,l+1} = 0$	
5.	$m_{l+1,l} = 1$	

4.2.2 Binary Switch One-Sided Crossing Minimization

Consider a two layered network $N = (V, U, E, x)$ with trivial ordering y of V where $V = \{v_1, v_2, ..., v_n\}$, $U = \{u_1, u_2, ..., u_m\}$ and x is also trivial ordering for U.The solution to the problem is a vector

$$s = \begin{pmatrix} s_1 \\ s_2 \\ \vdots \\ s_{n-1} \end{pmatrix} \in \{0,1\}^{n-1}$$

with

$$s_i = 1 \leftrightarrow \min_{\substack{y \text{ order of } V \\ y(v_i) > y(v_{i+1})}} crosses_G(y) < \min_{\substack{y \text{ order of } V \\ y(v_i) < y(v_{i+1})}} crosses_G(y)$$

If we solve the switch one-sided crossing minimization matrix S we can easy and fast calculate s because s_i is 1 if and only if $s_{i,2} < s_{i,1}$.

4.2.3 Input Graph

Consider a two layered network $N = (V, U, E, x)$ with trivial ordering y of V where $V = \{v_1, v_2, ..., v_n\}$, $U = \{u_1, u_2, ..., u_m\}$ and x is also trivial ordering for U. We construct the input graph like for the learned leftmost approach but we have to add some more informations since in this approach we have a current ordering y of V. The network is told

this ordering the same way as the its told the ordering x of U. Therefore the graph given to the graph network is $G_{in} = (V \dot{\cup} U, E')$ with

$$
\begin{aligned}
E' &= E'_1 \cup E'_2 \cup E'_3 \cup E'_4 \cup E'_5 \cup E'_6 \\
E'_1 &= E \\
E'_2 &= \{(u_i, v_j) \in U \times V | (v_j, u_i) \in E\} \\
E'_3 &= \{(u_i, u_{i+1}) \in U \times U | i \in \{1, 2, .., m-1\}\} \\
E'_4 &= \{(u_{i+1}, u_i) \in U \times U | i \in \{1, 2, .., m-1\}\} \\
E'_5 &= \{(v_i, v_{i+1}) \in V \times V | i \in \{1, 2, .., n-1\}\} \\
E'_6 &= \{(v_{i+1}, v_i) \in V \times V | i \in \{1, 2, .., n-1\}\}
\end{aligned}
$$

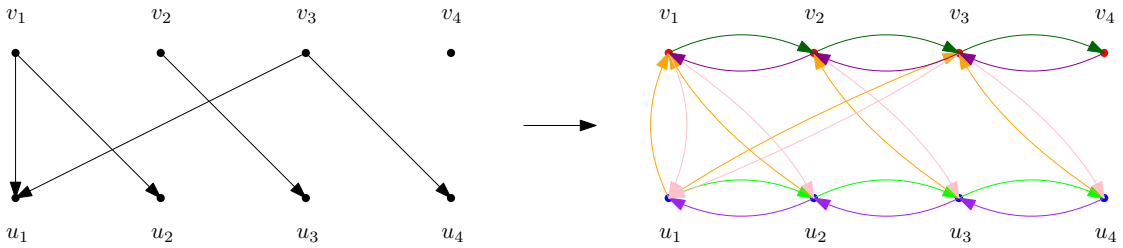

Figure 4.4: Construction of the input graph

Now we have to add embeddings as node, edge and global attributes. We added like in the learned leftmost approach every embedding the numbers $|V|, |U|$ and the rank of the nodes in U and additionally the rank of the nodes in V.

The global feature vector is

$$
\begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_{e_G} \end{pmatrix}
$$

where u_{e_G-1} is $|V|$, u_{e_G} is $|U|$ and u_1 until u_{e_G-2} is a learned vector.

The node feature vector is

$$
\begin{pmatrix} n_1 \\ n_2 \\ \vdots \\ n_{e_V} \end{pmatrix}
$$

where n_{e_V-1} is $|V|$ and n_{e_V} is $|U|$. For nodes in V we have n_1 until n_{e_V-3} as a learned vector and N_{e_V-2} as the rank of y and the node u_i in U has n_{e_V-2} as i which is the rank of the node and n_1 until n_{e_V-3} is a learned vector.

The edge feature vector is

$$
\begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e_{e_E} \end{pmatrix}
$$

where e_{e_E-1} is $|V|$ and e_{e_E} is $|U|$. Every E'_i has its own learned vector for the entries from e_1 until e_{e_E-2}.

We want the models to update the embeddings such that the embeddings of the edges in E'_5 predict the binary switch value.

4.2.4 Algorithm

We described the algorithm at the beginning ob this chapter. We started in a random permutation and then let our neural network predict which nodes to swap. This is repeated up to $|V|$ times or the network sais no swaps have to be done and save the permutation that has the lowest number of crosses. Since we trained the networks to predict the binary switch we interpreted values as a switch command if and only if they were higher than 0.5.

However this algorithm did work very bad on graphs with a low density so we added one optimization.

Let $N = (V, U, E, x)$ be a two layered network with trivial ordering y of V and a trivial ordering x for U. The problem is that if node v_k has no adjacent edges then the position of this node $y(v_k)$ is irrelevant. When we now look at our switch one-sided crossing minimization matrix S then

$$s_{k,1} = \min_{\substack{y \text{ is order of } V \\ y(v_k)<y(v_{k+1})}} \text{crosses}_G(y) = \min_{\substack{y \text{ is order of } V \\ y(v_k)>y(v_{k+1})}} \text{crosses}_G(y) = s_{k,2}$$

which tells our algorithm to not swap this node with its right neigbour and also

$$s_{k-1,1} = \min_{\substack{y \text{ is order of } V \\ y(v_{k-1})<y(v_k)}} \text{crosses}_G(y) = \min_{\substack{y \text{ is order of } V \\ y(v_{k-1})>y(v_k)}} \text{crosses}_G(y) = s_{k-1,2}$$

which tells our algorithm to not swap this node with its left neighbour and so a node with no adjacent edges will never change its position.

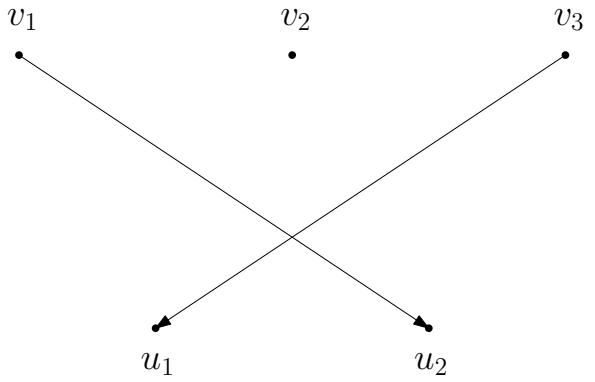

Figure 4.5: Bad example for initial switch algorithm

Therefore we optimized our algorithm by deleting all nodes with no adjacent edges in the beginning and then continue how already described.

5. Training and Evaluation

In this chapter we will explain how we trained the models and compare the different with different properties among each other and also compare these with existing heuristics. The models are trained on Google Colab and uploaded to Github.

5.1 Trainingdata

We generated two layered networks with the Erdős–Rényi model for two layered networks $G(n, m, p)$ with all three independent discrete uniformly distributed random variables

$$\begin{aligned} n &\sim U(5, 20) \\ m &\sim U(5, 20) \\ p &\sim U(5, 95) \end{aligned}$$

Learned Leftmost

Since we train with supervised learning we need to label our generated two layered networks. Therefore we have to calculated the scores for each two layered network using our reduction to ILP. With the nodes being $V = \{v_1, v_2, ..., v_n\}$ and the corresponding crossing matrix C we need to solve the optimization problem

$$\text{minimize} \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} m_{i,j} \cdot c_{i,j} \text{ subject to}$$

1.	$m_{i,j} + m_{j,k} - m_{i,k} \leq 1$	$i \neq j, j \neq k, i \neq k \quad i, j, k \in \{1, .., n\}$
2.	$0 \leq m_{i,j} \leq 1$	$i \neq j \quad i, j \in \{1, .., n\}$
3.	$m_{i,j} = 1 - m_{j,i}$	$i \neq j \quad i, j \in \{1, .., n\}$
4.	$m_{n,i} = 1$	$i \neq l \quad i \in \{1, .., n\}$
5.	$m_{i,n} = 0$	$i \neq l \quad i \in \{1, .., n\}$

for the l-th node. Since we need to solve for two layered network $|V|$ ILPs and these have many things in common we can optimize building these ILPs. Every ILP has the constraints 1., 2., and 3. because they make sure that the solution is really a permutation and every ILP has the same objective which is the number of crosses. So this is our base

and for every optimization problem of this perticular two layered network we just add the constraints 4. and 5. and delete them after optimizing our ILP model. We saved the scores of the nodes and if we wanted to know the binary scores or the interpolated scores we calculated them at runtime.

Learned Switch

We need to label our two layered networks for learned switch as well. With the nodes being $V = \{v_1, v_2, ..., v_n\}$ and the corresponding crossing matrix C we need to solve the optimization problem for $s_{l,1}$

$$\text{minimize} \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} m_{i,j} \cdot c_{i,j} \text{ subject to}$$

1.	$m_{i,j} + m_{j,k} - m_{i,k} \leq 1$	$i \neq j, j \neq k, i \neq k \quad i,j,k \in \{1,..,n\}$
2.	$0 \leq m_{i,j} \leq 1$	$i \neq j \quad i,j \in \{1,..,n\}$
3.	$m_{i,j} = 1 - m_{j,i}$	$i \neq j \quad i,j \in \{1,..,n\}$
4.	$m_{l,l+1} = 1$	
5.	$m_{l+1,l} = 0$	

and for $s_l, 2$

$$\text{minimize} \sum_{i=1}^{n_1} \sum_{\substack{j=1 \\ j \neq i}}^{n_1} m_{i,j} \cdot c_{i,j} \text{ subject to}$$

1.	$m_{i,j} + m_{j,k} - m_{i,k} \leq 1$	$i \neq j, j \neq k, i \neq k \quad i,j,k \in \{1,..,n\}$
2.	$0 \leq m_{i,j} \leq 1$	$i \neq j \quad i,j \in \{1,..,n\}$
3.	$m_{i,j} = 1 - m_{j,i}$	$i \neq j \quad i,j \in \{1,..,n\}$
4.	$m_{l,l+1} = 0$	
5.	$m_{l+1,l} = 1$	

which also have a lot of constraints in common. The constraints 1., 2. and 3. are our base and are used in every of the $2 \cdot (n-1)$ ILPs. So for every optimization problem we add the constraints 4. and 5. and remove them afterwards.

We use Gurobi to solve the ILPs and use Protobuf to save our trainingdata.

5.2 Training

In this section we want to present the different trained models. We trained those to minimize the mean squared error between the prediction and the target function. We used the Adam optimizer with a learning rate of 0.0001 and batched graph with about 3750 nodes.

Models

All models have two layer perceptrons with a layernormalization afterwards as edge, node and global update functions for the encoder and core networks. The decoder network's update functions have the same structure but have additionally one layer with one neuron

afterwards that has no activation function. All activation functions are the rectified linear unit. All models have one parameter k that tells how big the embeddings of the input graph for node, edge and global attributes are and k is also the size for all embeddings of the output of the encoder and core networks. Since the decoder network has one neuron on the last layer the embedding sizes from the output of the decoder network is one.

We have the n-layer process encode decode called 9-LPED with $n = 9$ and $k = 80$.

We have the n-process encode decode called 9-PED with the core network being used nine times and $k = 80$.

We have the n-process encode decode called U-PED with the core network being used $|U|$ times and $k = 80$.

We have the n-recursive layer process encode decode called 9-RLPED with $n = 9$ and $k = 20$.

Target Function

We trained each of those networks to predict the binary scores, the interpolated scores and the binary switch.

Loss and Accuracy

We created one data file containing 10.000 graphs from the same distribution as our Trainingdata for the learned leftmost approach and for the learned switch approach. We measured in the first step the average loss on this data file and then we measured the percentage how many problems the network solved correctly called accuracy.

In context of learned leftmost we say that our network predicted one problem correctly if the node with the highest vote has actually the lowest score for both the binary scores and the interpolated scores.

In context of learned switch we say that our network predicted one problem correctly if the rounded vector our model predicts has the same entries as the binary switch vector.

Model	Target function	Loss	Accuracy
9-LPED	interpolated score	0.00398	0.928
9-PED	interpolated score	0.00688	0.905
$\|U\|$-PED	interpolated score	0.00960	0.851
9-RLPED	interpolated score	0.00604	0.901
9-LPED	binary score	0.01227	0.971
9-PED	binary score	0.02422	0.964
$\|U\|$-PED	binary score	0.02787	0.959
9-RLPED	binary score	0.01670	0.967
9-LPED	binary switch	0.01940	0.763
9-PED	binary switch	0.02216	0.740
$\|U\|$-PED	binary switch	0.02408	0.725
9-RLPED	binary switch	0.02646	0.692

Table 5.1: All models with its loss and accuracy

5.3 Comparison

In this subsection we will compare all models and all existing heuristics. We used the implementation of the Open Graph Drawing Framework for the existing heuristics.

We tested all heuristics, learned heuristics and existing heuristics, by calculating an order of a given two layered newtork $N = (V, U, E, x)$ on four datasets which all consist of 2.000 two layered networks but from different distributions.

We wanted to see the average performance in comparison to the optimum depending on the density of the two layered network. Therefore we created two datasets one with a lower size. So dataset 1 is distributed from the $G(10, 10, p)$ model with $p \sim U(5, 95)$ and dataset 2 is distributed from the $G(20, 20, p)$ model with $p \sim U(5, 95)$.

Another interesting aspect is how the algorithm scales on problems whith a size it was not trained on. So dataset 3 is distributed from the $G(n, n, 0.25)$ model with $n \sim U(4, 49)$ and dataset 4 is distributed form the $G(n, n, 0.75)$ model with $n \sim U(4, 49)$.

We created plots where the average relative error in dependency of density or number of nodes. We said a graph has density of for example 0.2 when the number of edges in a graph divided of the number of possible edges is higher than 0.15 and lower than 0.25 is.

5.3.1 Existing Heuristics

In this subsection the existing heuristics are tested against each other on the four datasets.

In the figure below one can see the performance on the first two datasets. In the first two plots all heuristics are portrayed and on the two plots afterwards just the barycenter and median heuristic since they have by far the best performance. We can see that all algorithms have a better performance with higher density.

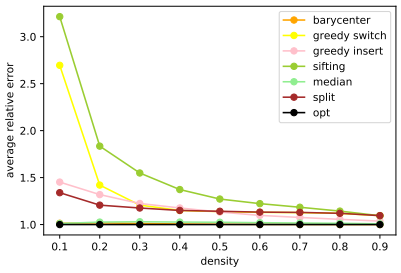

Figure 5.1: performance on the distribution from $G(10, 10, p)$ with $p \sim U(5, 95)$

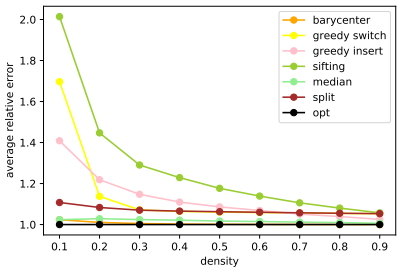

Figure 5.2: performance on the distribution from $G(20, 20, p)$ with $p \sim U(5, 95)$

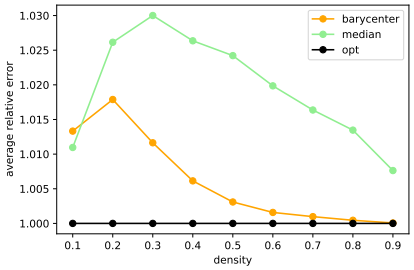

Figure 5.3: performance on the distribution from $G(10, 10, p)$ with $p \sim U(5, 95)$

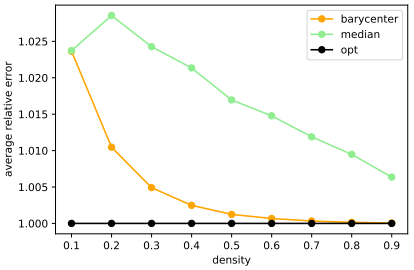

Figure 5.4: performance on the distribution from $G(20, 20, p)$ with $p \sim U(5, 95)$

On our last two datasets with differing node sizes the barycenter heuristic still has the best performance and the median heuristic has the second best performance. The difference between barycenter rises with higher densities which we can see good in figure 5.8. One

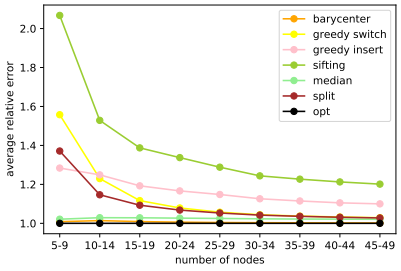

Figure 5.5: performance on the distribution from $G(n, n, 0.25)$ with $n \sim U(5, 49)$

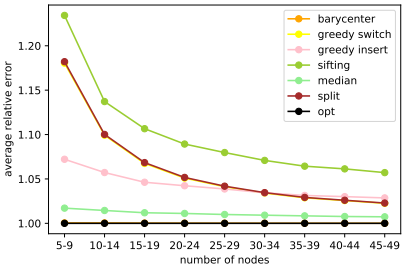

Figure 5.6: performance on the distribution from $G(n, n, 0.75)$ with $n \sim U(5, 49)$

can see that the barycenter heuristic has overall the best performance. On graphs with low densities the heuristic has equaling performance than the median heuristic but with increasing density the barycenter heuristic has by far the best performance.

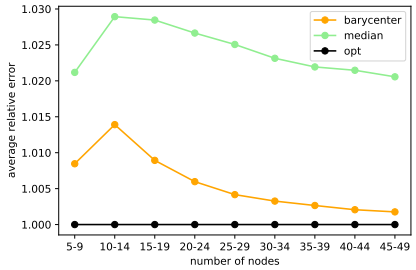

Figure 5.7: performance on the distribution from $G(n, n, 0.25)$ with $n \sim U(5, 49)$

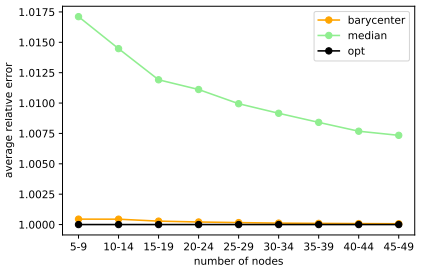

Figure 5.8: performance on the distribution from $G(n, n, 0.75)$ with $n \sim U(5, 49)$

5.3.2 Interpolated Score

In this subsection the models that are trained to predict the interpolated score are tested against each other on the four datasets.

In the figures below we can see the performance of our models on the first two datasets.

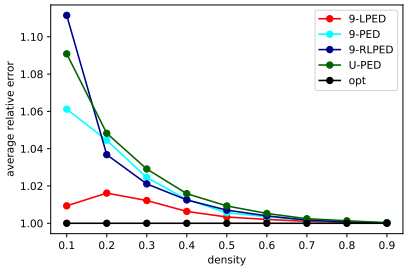

Figure 5.9: performance on the distribution from $G(10, 10, p)$ with $p \sim U(5, 95)$

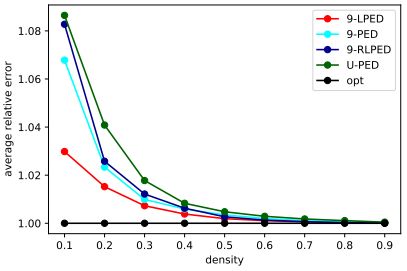

Figure 5.10: performance on the distribution from $G(20, 20, p)$ with $p \sim U(5, 95)$

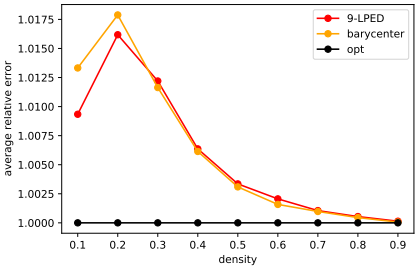

Figure 5.11: performance on the distribution from $G(10, 10, p)$ with $p \sim U(5, 95)$

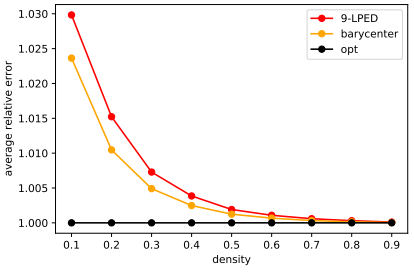

Figure 5.12: performance on the distribution from $G(20, 20, p)$ with $p \sim U(5, 95)$

We can see that in the 9-LPED model has in both cases overall the best performance in comparison to the other models. On smaller node sizes the 9-LPED has even better

performance than the barycenter heuristics for small densities and for higher densities they are about the same. With increasing node sizes it gets a little worse.

In the two figures below we tested our models on the last two datasets. We can see that the 9-LPED model has once more the best performance smaller node sizes and on higher node sizes the 9-RLPED gets a slightly better.

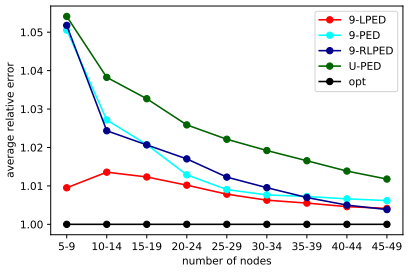

Figure 5.13: performance on the distribution from $G(n, n, 0.25)$ with $n \sim U(5, 49)$

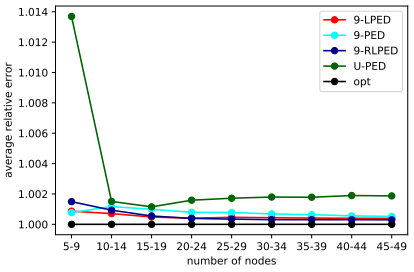

Figure 5.14: performance on the distribution from $G(n, n, 0.75)$ with $n \sim U(5, 49)$

Therefore we displayed the 9-LPED and the 9-RLPED in comparison to the baryenter heuristic in the two figures below. We can see that the 9-LPED is compatible to the barycenter heuristic on small instance sizes and on small densities but in the other cases it is slightly worse.

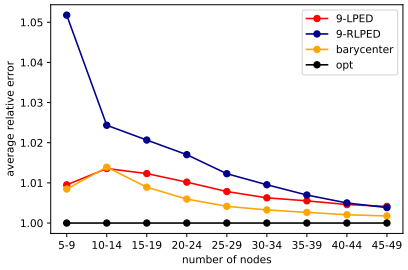

Figure 5.15: performance on the distribution from $G(n, n, 0.25)$ with $n \sim U(5, 49)$

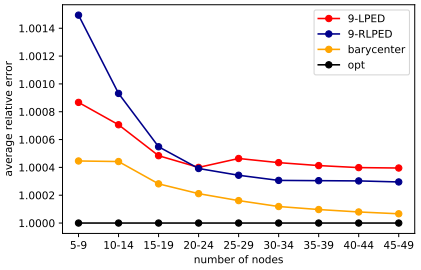

Figure 5.16: performance on the distribution from $G(n, n, 0.75)$ with $n \sim U(5, 49)$

5.3.3 Binary Score

In this subsection the models that are trained to predict the binary score are tested against each other on the four datasets.

In the figures below we can see the performance of our models on the first two datasets.

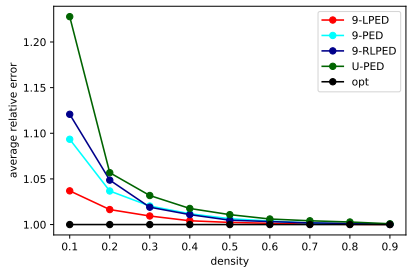

Figure 5.17: performance on the distribution from $G(10, 10, p)$ with $p \sim U(5, 95)$

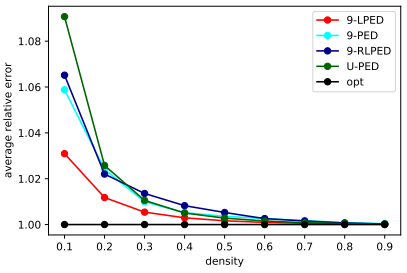

Figure 5.18: performance on the distribution from $G(20, 20, p)$ with $p \sim U(5, 95)$

One can see that also here the 9-LPED has the best performance. In the figure below this model is compared to the barycenter heuristic and on low instance sizes our learned model has almost overall better performance which lowers with higher instance sizes.

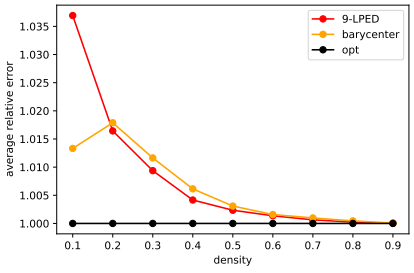

Figure 5.19: performance on the distribution from $G(10, 10, p)$ with $p \sim U(5, 95)$

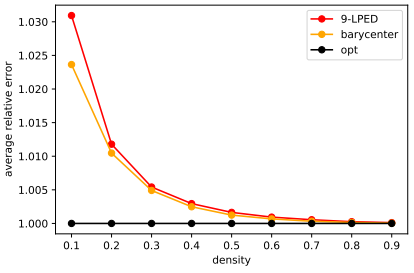

Figure 5.20: performance on the distribution from $G(20, 20, p)$ with $p \sim U(5, 95)$

In the figure below we can see how our algorithm scales to larger inputs. In both plots the 9-LPED has overall the best performance. The U-RLPED seems to have the worst performance on low inputs. The reason for this may be the lower number of GN block iterations. The other models all have 9 GN block iterations and this model has on small instance sizes less.

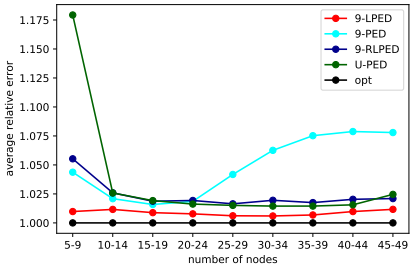

Figure 5.21: performance on the distribution from $G(n, n, 0.25)$ with $n \sim U(5, 49)$

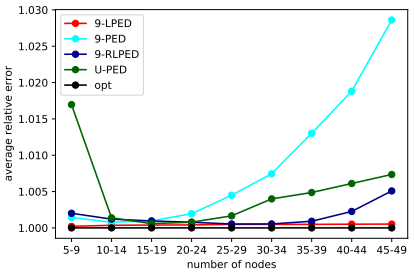

Figure 5.22: performance on the distribution from $G(n, n, 0.75)$ with $n \sim U(5, 49)$

We displayed the 9-LPED in comparison to the baryenter heuristic in the two figures below. We can see that the 9-LPED is slightly better than the barycenter heuristic on small instance sizes and on high densities but in the other cases not.

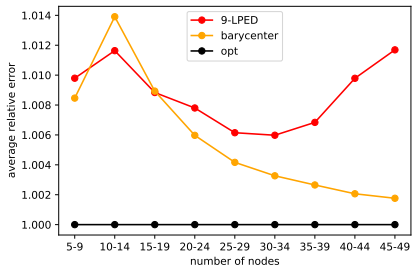

Figure 5.23: performance on the distribution from $G(n, n, 0.25)$ with $n \sim U(5, 49)$

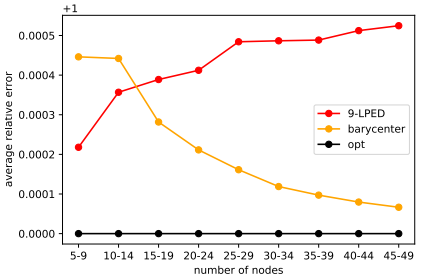

Figure 5.24: performance on the distribution from $G(n, n, 0.75)$ with $n \sim U(5, 49)$

5.3.4 Binary Switch

In this subsection the models that are trained to predict the binary switch are tested against each other on the four datasets.

In the figures below we can see the performance of our models on the first two datasets.

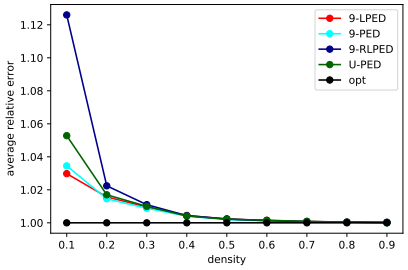

Figure 5.25: performance on the distribution from $G(10, 10, p)$ with $p \sim U(5, 95)$

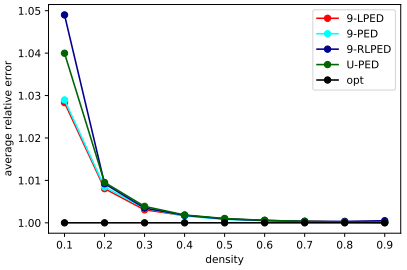

Figure 5.26: performance on the distribution from $G(20, 20, p)$ with $p \sim U(5, 95)$

One can see that the learned switch approaches have a better performance overall but on low densities. This time the 9-LPED has almost the same performance than the 9-PED and the 9-RLPED has the worst.

In the figures below the performance on different node sizes is pictured. Figure 5.31 shows the 9-LPED in comparison to the barycenter heuristic and this model has except for very low instance sizes better performance for two layered networks with even much higher sizes than the trainingdata.

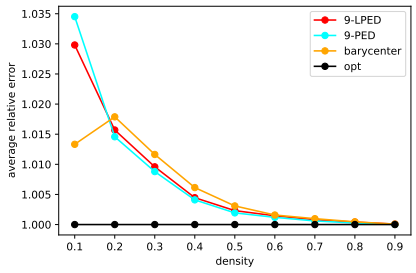

Figure 5.27: performance on the distribution from $G(10, 10, p)$ with $p \sim U(5, 95)$

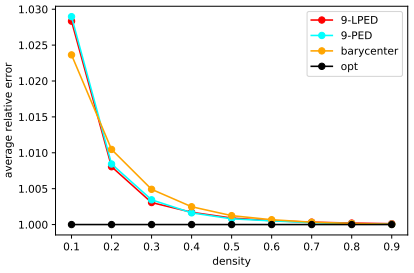

Figure 5.28: performance on the distribution from $G(20, 20, p)$ with $p \sim U(5, 95)$

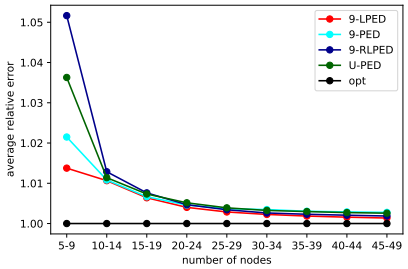

Figure 5.29: performance on the distribution from $G(n, n, 0.25)$ with $n \sim U(5, 49)$

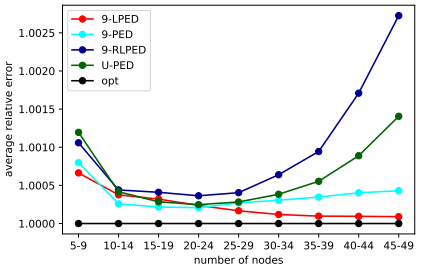

Figure 5.30: performance on the distribution from $G(n, n, 0.75)$ with $n \sim U(5, 49)$

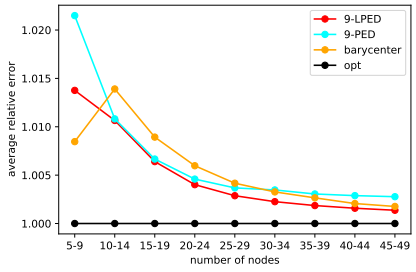

Figure 5.31: performance on the distribution from $G(n, n, 0.25)$ with $n \sim U(5, 49)$

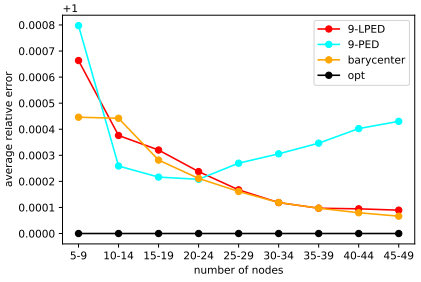

Figure 5.32: performance on the distribution from $G(n, n, 0.75)$ with $n \sim U(5, 49)$

5.3.5 Model Comparison

The 9-LPED has overall the best performance from all trained models. We can see that our algorithms have a good performance on our trainingdata and the 9-LPED for the learned leftmost approach showed for the instance sizes it was trained on performances compatible to the barycenter heuristic. However this algorithm didn't scale well to larger inputs.

The learned switch algorithm with the 9-LPED did scale very well and even had for node sizes up to 50 better results than the barycenter heuristic. The 9-PED also had a

performance that was almost as good as the 9-LPED. The other models were worse and did not scale good as one can see in figure 5.30.

6. Conclusion

The One-SidedCrossingMinimizationProblem is $\mathcal{NP}$-hard [Wor94] and therefore there probably won't exist an efficient algorithm solving this. There exist some well-studie heuristics for approximating this problem. In tests from Michael Jünger and Petra Mutzel [MJ97] the barycenter heuristic lead to the best results in terms of low computation time and quality. They came to the conclusion that on node sizes of up to 60 nodes one can calculate the optimal solution and on higher instance sizes the barycenter heuristic should be used.

We were able to create two heuristics. The learned leftmost approach determines the node that has to be placed in the leftest position and by iterating one can calculate a whole order for the nodes. The learned switch approach starts in a random order and then tries to switch neighbouring nodes with the goal of minimizing the number of crosses.

We compared our appraches using machine learning with existing heuristics. In our test the barycenter heuristic also has the best among the other existing ones like in this paper[MJ97]. We created an algorithm with the learned switch approach that does scale well and has better performance than the barycenter heuristic on most inputs.

6.1 Future Work

We saw that machine learning is a powerful tool that could be used for crossing minimization and think we created heuristics with good performances. We saw that training to predict the interpolated score was in some aspects superior to the binary score. One idea could be to define the interpolated switch and train the models to predict this target function.

Michael Jünger and Petra Mutzel [MJ97] think that for node sizes under 60 the optimal solution can be computed efficiently and for node sizes above the barycenter heuristic can be used. Since our algorithm seems to have very low difference to the barycenter heuristic on this big instance sizes we don't think that our models will improve the state of the art. However if we train models on bigger instance sizes the algorithms may have better performance on very big instances.

Bibliography

[ACNS82] M. Ajtai, V. Chvátal, M.M. Newborn, and E. Szemerédi. Crossing-free subgraphs. In Peter L. Hammer, Alexander Rosa, Gert Sabidussi, and Jean Turgeon, editors, *Theory and Practice of Combinatorics*, volume 60 of *North-Holland Mathematics Studies*, pages 9 – 12. North-Holland, 1982.

[AG16] Jure Leskovec Aditya Grover. node2vec: Scalable feature learning for networks. In *International Conference on Knowledge Discovery and Data Mining*, 2016.

[BHB+18] Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Caglar Gulcehre, Francis Song, Andrew Ballard, Justin Gilmer, George Dahl, Ashish Vaswani, Kelsey Allen, Charles Nash, Victoria Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matt Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. Relational inductive biases, deep learning, and graph networks, 2018.

[Ced66] A. Lempel ; I. Cederbaum. Minimum feedback arc and vertex sets of a directed graph. *IEEE Transactions on Circuit Theory*, 1966.

[DS19] Benedikt B¨unz Percy Liang David L. Dill Daniel Selsam, Matthew Lamm. Learning a sat solver from single-bit supervision. 2019.

[FH01] Rudolf Fleischer and Colin Hirsch. *Graph Drawing and Its Applications*, pages 1–22. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[HD17] Yuyu Zhang Bistra Dilkina Le Song Hanjun Dai, Elias B. Khalil. Learning combinatorial optimization algorithms over graphsx. *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017.

[Ism12] Alaa A. K. Ismaeel. *Dynamic Hierarchical GraphDrawing*. PhD thesis, Karlsruher Institut für Technologie, 2012.

[KK90] Manfred Koebe and Jens Knöchel. On the block alignment problem. *Elektronische Informationsverarbeitung und Kybernetik*, 26:377–387, 01 1990.

[Kol19] Yani Kolev. Solving geometric optimizationproblems by reinforcement learning:crossing minimization. Master's thesis, KIT Department of Informatics, 2019.

[Lei83] Frank Thomson Leighton. *Complexity Issues in VLSI: Optimal Layouts for the Shuffle-Exchange Graph and Other Networks*. MIT Press, Cambridge, MA, USA, 1983.

[LFRR17] Daniel R. Figueiredo Leonardo F. R. Ribeiro, Pedro H. P. Saverese. struc2vec: Learning node representations from structuralidentity. In *International Conference on Knowledge Discovery and Data Mining*, 2017.

[MCCD13] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[MJ97] Petra Mutzel Michael Jünger. 2-layer straightline crossing minimization: Performance of exact and heuristic algorithms. *Journal of Graph Algorithms and Applications 1*, 1997.

[MRG83] D. S. Johnson M. R. Garey. Crossing number is np-complete. *SIAM Journal on Algebraic Discrete Methods*, 1983.

[MS05] Erkki Mäkinen and Harri Siirtola. The barycenter heuristic and the reorderable matrix. *Informatica (Slovenia)*, 29:357–364, 10 2005.

[MSM98] Christian Matuszewski, Robby Schönfeld, and Paul Molitor. Using sifting for k-layer straightline crossing minimization. volume 1731, pages 217–224, 12 1998.

[NC85] Shigenobu Abe Takao Ozawa Norishige Chiba, Takao Nishizeki. A linear algorithm for embedding planar graphs using pq-trees. *Journal of Computer and System Sciences*, 1985.

[PE91] Kozo Sugiyama Peter Eades. How to draw a directed graph. *Journal of Information Processing*, 1991.

[Pur02] Helen Purchase. Metrics for graph drawing aesthetics. *Journal of Visual Languages and Computing*, 13:501–516, 10 2002.

[STT81] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical systems structure. *Systems, Man and Cybernetics, IEEE Transactions on*, 11:109 – 125, 03 1981.

[Thr52] R. M. Thrall. A combinatorial problem. *Michigan Math. J.*, 1(1):81–88, 01 1952.

[Vrt01] Xavier MuñozW. UngerImrich Vrt'o. One sided crossing minimization is np-hard for sparse graphs. *International Symposium on Graph Drawing*, 2001.

[Wor94] Peter Eades; Nicholas C. Wormald. Edge crossings in drawings of bipartite graphs. *Algorithmica volume 11*, 1994.

[Zar55] Casimir Zarankiewicz. On a problem of p. turan concerning graphs. *Fundamenta Mathematicae*, 41(1):137–145, 1955.