České vysoké učení technické v Praze
Fakulta jaderná a fyzikálně inženýrská

Katedra matematiky
Obor: Aplikovaná informatika

Heuristiky v dolování dat z grafů pomocí vnoření uzlů

Heuristics in Graph Data Mining via Node Embeddings

BAKALÁŘSKÁ PRÁCE

Vypracoval:	Adeliia Gataullina
Vedoucí práce:	Ing. Matej Mojzeš, Ph.D.
Rok:	2020

České vysoké učení technické v Praze, Fakulta jaderná a fyzikálně inženýrská

Katedra: matematiky Akademický rok: 2019/2020

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student:	Adeliia Gataullina
Studijní program:	Aplikace přírodních věd
Obor:	Aplikovaná informatika
Název práce (česky):	Heuristiky v dolování dat z grafů pomocí vnoření uzlů
Název práce (anglicky):	Heuristics in Graph Data Mining via Node Embeddings

Pokyny pro vypracování:

1. Rešerše literatury o vnoření slov (word embeddings) a o modelech word2vec a node2vec.
2. Implementace algoritmu node2vec.
3. Analýza použití heuristik v node2vec, např. při vzorkování uzlů grafu. Studie vlivu různých algoritmů a nastavení na kvalitu výsledků a jejich praktické využitelnosti.
4. Aplikace implementovaného algoritmu na reálných datech a analýza výsledků.

Doporučená literatura:

1. A. Grover, J. Leskovec, node2vec: Scalable Feature Learning for Networks. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), 2016.
2. T. Mikolov, et al., Efficient Estimation of Word Representations in Vector Space. arXiv:1301.3781, 2013.
3. V. Kvasnička, J. Pospíchal, P. Tino, Genetic Algorithms. STU Bratislava, 2000.
4. F. S. Rizi, M. Granitzer, Properties of Vector Embeddings in Social Networks. Algorithms 10(4), 2017, 109.
5. J. Han, M. Kamber, J. Pei, Data mining: concepts and techniques. 3rd ed., Waltham: Morgan Kaufmann, 2012.
6. E. A. Hansen, R. Zhou, Anytime Heuristic Search. Journal of Artificial Intelligence Research 28, 2011, 267-297.

Jméno a pracoviště vedoucího bakalářské práce:

Ing. Matěj Mojzeš, Ph.D.
Katedra softwarového inženýrství, FJFI ČVUT v Praze, Trojanova 13 , 120 00 Praha 2

Jméno a pracoviště konzultanta:

Ing. Petr Holík
Jumpshot s.r.o., Pikrtova 1737/1a, 140 00 Praha 4 – Nusle

Datum zadání bakalářské práce: 31.10.2019

Datum odevzdání bakalářské práce: 7.7.2020

Doba platnosti zadání je dva roky od data zadání.

V Praze dne 23. října 2019

[illegible]
. . . .
garant oboru

[illegible]
. . . .
vedoucí katedry

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta jaderná a fyzikálně inženýrská
-1-

[illegible]
. . . .
děkan

Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracovala samostatně a použila jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu.

V Praze dne ....

Adeliia Gataullina

Poděkování

Chtěla bych poděkovat Ing. Mateji Mojzešovi, Ph.D. za vedení mé bakalářské práce, cenné rady, odborný dohled, připomínky a čas, který mi věnoval. Děkuji také Čápové Haně Mgr. za pomoc při gramatické kontrole práce a jazykovou podporu. Mé podekovávání patří též mé rodině a blízkým přátelům za pomoc a podporu během studia, zejména Janu Kořínkovi.

Adeliia Gataullina

Název práce:
Heuristiky v dolování dat z grafů pomocí vnoření uzlů

Autor: Adeliia Gataullina

Studijní program: Aplikace přírodních věd
Obor: Aplikovaná informatika
Druh práce: Bakalářská práce

Vedoucí práce: Ing. Matej Mojzeš, Ph.D.
Katedra softwarového inženýrství, Fakulta jaderná a fyzikálně inženýrská, České vysoké učení technické v Praze
Konzultant: Ing. Petr Holík
Jumpshot s.r.o.,Pikrtova 1737/1a, 140 00 Praha 4 – Nusle

Abstrakt: Tato práce se zabývá využitím heuristických algoritmů v úlohách dolování dat v grafech. Jeden ze způsobů jak pracovat s grafy, je reprezentovat ve formě vektorů, tak zvaných vnořených uzlů - embeddingů. Algoritmy node2vec a word2vec vytvoří takové vektorové representace. Optimalizace těchto algoritmů zahrnuje hledání správného nastavení jejich parametrů. Předpokládá se, že heuristiky mohou toto vyhledávání usnadnit. Tato práce si klade za cíl ukázat použití algoritmu node2vec pro řešení úlohy shulkové analýzy na grafech a ověřit, zda jsou heuristiky vhodné pro optimalizaci modelu node2vec.
Klíčová slova: Dolování dat, vnořené uzly, node2vec, heuristiky, shulková analýza

Title:
Heuristics in Graph Data Mining via Node Embeddings

Author: Adeliia Gataullina

Abstract: This paper considers the usage of heuristic algorithms in data mining tasks on graphs. One of the ways how to work with graphs is to represent them in the form of vectors, i.e. embeddings. Algorithms node2vec and word2vec creates such embeddings. The optimisation of those algorithms involves searching for proper settings of their parameters. Heuristics may facilitate that searching. This work aims to show the usage of the node2vec algorithm for solving of the cluster analysis task on graphs and verify if heuristics are suitable for the optimisation of the node2vec model.
Key words: Data mining, embeddings, node2vec, heuristics, cluster analysis

Contents

Introduction

With the development of technology, the amount of data collected about our activities online and offline has become enormous.

Not everyone can accept and agree with this, but anonymous research of this data can be useful for people in terms of business development and improving the quality of services. For example, data gathered from the Internet of things (IoT) devices in medical and health care spheres can enhance the functionality of quick response medicine, of making diagnoses and preventing diseases. Users online behaviour data study may improve recommendation systems and accelerate the development of artificial intelligence, e.g. image recognition. Thence, data mining, i.e. the process of collecting useful knowledge from the data, and data analysis became one of the most promising and rapidly developing areas of computer science.

There are many ways to study data about objects and their relations. It is convenient and understandable to store them in the form of graphs, where vertices represent the objects, and edges between the vertices represent the relationship between these objects. It is more comfortable to work with vector embeddings, i.e. graph representations in the form of mathematical vectors.

In this paper, the algorithms *node2vec* and *word2vec* are considered, the consistent use of which allows us to reconstruct a graph of data in the form of vector embeddings. As the primary source of information, the following papers describing these two algorithms were used: *node2vec: Scalable Feature Learning for Networks* by Aditya Grover and Jure Leskove [7] and *Efficient Estimation of Word Representations in Vector Space* by Tomas Mikolov, Kai Chen, Greg Corrado and Jeffrey Dean [11].

The objective of this work is to learn the *node2vec* model usage and the principles of its functionality. It also aims to investigate the possibility of optimisation of the *node2vec* algorithm employing heuristics. Heuristics are methods of finding solutions using algorithms, based on natural phenomena and common sense, that may not guarantee the optimal results, but can still be very effective. [1]

As an objective function of this experiment, a task of cluster analysis of graph vertices was used. Optimisation of the objective function consisted of the search for the correct parameters of the *node2vec* algorithms. It is expected to establish that the use of heuristics can help in the relatively fast and efficient selection of parameters optimising the objective function. At the same time, the intention is to show that the performance of different heuristic algorithms may differ.

The paper is divided into four parts. In the first chapter, the reader will find a brief description of the theoretical foundations necessary for understanding the techniques used throughout the paper. It also describes the essence of the functionality of *node2vec* and *word2vec*. The second chapter relates to the implementation of the *node2vec* algorithm. The third chapter contains a description of the heuristics considered in the paper and the results of their analysis of simple artificial data. The last fourth chapter is devoted to the use of information obtained in the study on real-world data research.

Chapter 1

Theoretical Basis

1.1 Neural Networks

Computer programs solve all sorts of problems that could be divided into three groups according to the level of their complexity and existence of the known methods of their solving:

- elementary and middle complicated problems with obtained solutions, which can be solved by known algorithms, such as printing a document on a printer, displaying an application window on a computer screen or numerical operations;
- simple and middle complicated problems with partially found solutions, which can be solved by statistical models, e.g., simple prediction, calculation of errors or approximate solutions of equations;
- severe problems without existing solutions, which can not be solved by general algorithms, for example, image and speech recognition or complex forecasts.

One of the tasks from the latter mentioned group – recognising people's faces – is easily solved unconsciously by the human brain. However, it seems quite challenging to construct a computer algorithm that would cope with such a quandary. One of the techniques that may solve it is neural networks.

Neural network is a simplified model of the biological neural system, which consists of interacting artificial neurons. Comparable to biological neurons, artificial ones receive information, process it in some way and transmit it to other neurons.

Neural networks are composed of three main types of neurons: input, hidden, and output (see Fig. 1.1). A large number of neurons represent a layer. Each neuron holds input and output data. In the case of the input neuron, the input and output data are equal. In other instances, summarised information from the previous layers is received by the neuron, is normalised by the activation function and is sent to the output field.

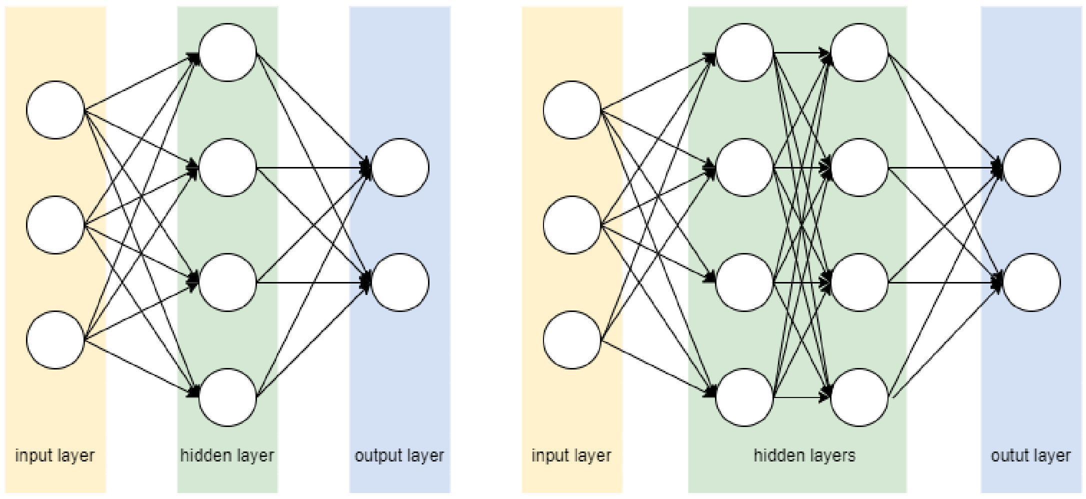

Figure 1.1: Left: a single-layer neural network; right: a multi-layer neural network

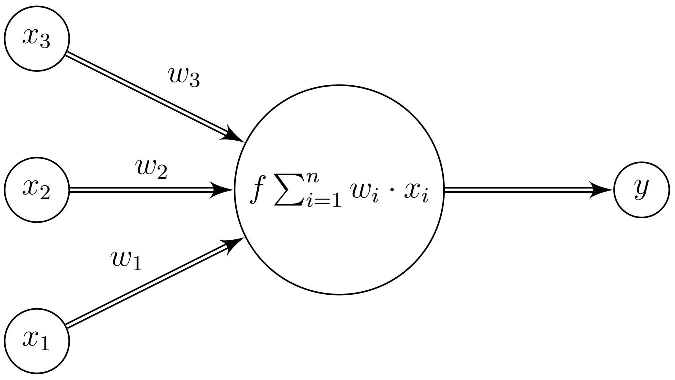

Figure 1.2: x_i - input data with w_i weights of synapses, $f(x)$ - the activation function and y - the output data of a neuron.

Connections (or so-called synapses) between neurons are weighted to show the amount of influence that has one neuron to another. The information received from the most weighted neuron is considered dominant in the receiving node. Precisely due to the set of synaptic weights, input data can be processed and converted into a meaningful result.

On initialisation, weighs of the neural network are distributed randomly. As input data, a neuron receives a weighted sum from the previous neurons, that is, the sum of the input signals multiplied by the corresponding weights:

$$\sum_{i=1}^{n} w_i x_i$$

The activation function of neuron accepts the sum as an argument and normalises it into the output result (see Fig. 1.2).

At the first launch of a network, the answer will be far from correct since the system is not trained. The connection weights need to be strengthened or weakened to approach the right answer. The process of searching the set of weights, which will lead to the desired output, is called the neural network training.

The final set of input signals used for training of the network is denoted as the training set. Besides, there is a testing set of input data used for evaluating the network quality. Both sets may at times contain valid output answers.

1.2 Embeddings

Further in this paper, *node2vec*, i.e. an algorithm representing graphs in the form of vectors, will be considered. However, *node2vec* works based on *word2vec*, i.e. an algorithm that represents words in the form of vectors, so-called word embeddings. An architecture of *word2vec* involves neural networks.

1.2.1 Word Embeddings

For word processing by a computer, it is necessary to solve the problem of representing words in the form that the computer will understand.

One of the simplest ways of representing words can be the idea of simple numbering words in the order they appear in a dictionary. However, this method has several significant problems. Words in most of the dictionaries are in alphabetical order, so on adding a word, it would be necessary to renumber most of the words. However, this is not the main problem. Words with different meanings can be spelt likewise or conversely words that looks differently may be semantically similar. For example, a person who does not know the meaning of the words: "rooster", "hen" and "chicken", may not understand that they belong to the same semantic group, because visually they are very different from each other. Whereas, for a person who is familiar with these words, it is apparent that they denote a male, female and cub of one species of bird. [8]

Thus, to be able to represent the semantic proximity, it was proposed to use word embeddings, i.e., an association of a word with a specific vector, which reflects its meaning in the "space of meanings". It is a set of language modelling and feature learning techniques in Natural Language Processing (NLP), which allows representing words and phrases as vectors of real numbers.

word2vec

In 2013, a Czech graduate student Tomas Mikolov proposed his approach to word embedding, which he called *word2vec*. His approach is based on a hypothesis, which is called the principle of locality - "words that occur in the same environments have similar meanings". [11]

The principle of the *word2vec* model is to find connections between contexts of words. More formally, the task is: to maximise cosine similarity between word vectors appearing next to each other and to minimise cosine similarity between vectors of words that do not appear next to each other. Cosine similarity is a binary measure of similarity, which shows a similarity between two non-zero vectors of inner product space and measures the cosine of the angle between them (see Fig.1.3). Put it differently, if the words are situated closely, it means that they are located in similar contexts.

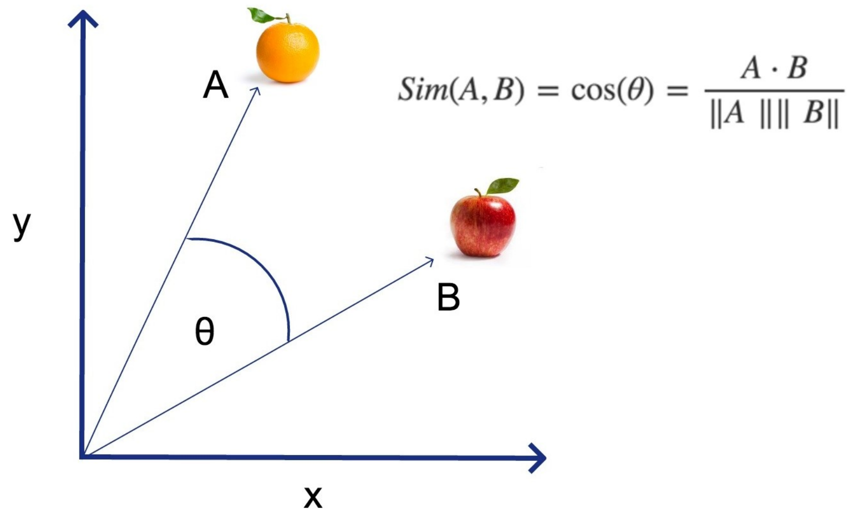

Figure 1.3: Cosine similarity of A: Orange and B: Apple words.

For example, the words "analyse" and "research" can be found in similar contexts, such as "Students have analysed the algorithm" or "Students conducted research on the algorithm". The *word2vec* model analyses these texts and concludes that the words "analyse" and "research" are close in meaning.

The resulting representations of word vectors can be used for calculating the "semantic distance" between words. Precisely based on the contextual proximity of these words, the *word2vec* technology makes its predictions.

The main steps of the *word2vec* model processing are as follows:

1. During a corpus reading a number of each word in the corpus occurrences is calculated. A corpus, here, is denoted as a collection of texts for model training.
2. Words stored in a hash table are sorted by frequency of occurrence, and rare words are deleted.
3. A Huffman tree is built. A Huffman tree is a full binary tree representing a given alphabet, where a leaf corresponds each letter. It is often used to encode a dictionary - this dramatically reduces the computational complexity of the algorithm.

4. The so-called sub-sentences are read from the corpus and sub-sampling of the most frequent words is processed. A sub-sentence is an essential body element, usually just a sentence, but it can also be a paragraph or even a whole article. Sub-sampling is the process of removing the most frequent words from the analysis, which speeds up the learning process and contributes to a significant increase in the quality of the resulting model.

5. The sub-sentences are processed within a window. For instance, if the window size is equal to three, then for the sentence "python interpreted high-level programming language", the analysis will be held inside the blocks: "python interpreted high-level" "interpreted high-level programming", "high-level programming language". The default window size is five, and the recommended value is ten. [7]

6. At last, a Feedforward neural network is used with a Hierarchical Softmax activation function, a Negative Sampling or both. A Feedforward neural network is a deep learning model that do not contain cycled links between neurons. In this network, the information flows in one direction only, from the input neurons, through the hidden layer to the output neurons. In the *word2vec* model information goes through a Softmax activation function, i.e. a function that takes as input a vector of K real numbers and normalises it into a probability distribution consisting of K probabilities proportional to the exponential of the input numbers. If Negative Sampling also involved, weights for only a small amount of random words are updated.

Although no semantics, but only the statistical properties of the corpus of the texts, were taken into account in the model. It appears that the trained *word2vec* model can capture some semantic features of words. So, using a large enough corpus for training may bring quite adequate results.

A classic example of word embeddings is shown in the left part of the figure 1.4. The vectors of the words "man" and "woman" are related to each other, at the similar way as vectors of the words "king" and "queen", which is completely natural and understandable for English speaking people. In addition to semantic links, the syntax is also captured. See the right part of the Figure 1.4, there the relationship between singular and plural forms of the words is shown.

There are two main architectures implemented by *word2vec*: Continuous Bag of Words (CBOW) and Skip-gram (Fig. 1.5). CBOW is a model architecture that predicts the current word based on its surrounding context. The architecture of the Skip-gram type works differently: it uses the current word to predict the words surrounding it. In general, CBOW and Skip-gram are neural network architectures that describe precisely how a neural network "learns" from data and "remembers" representations of words. The user of *word2vec* can switch and choose between algorithms. The word order of the context does not affect the result in any of these algorithms.

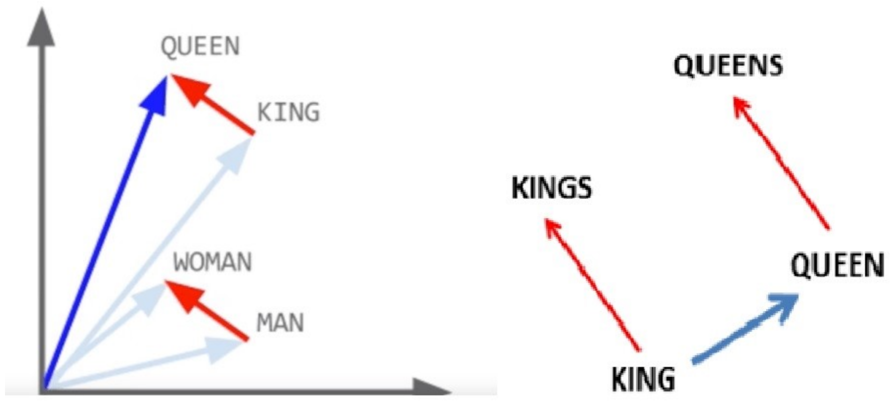

Figure 1.4: **a.** Relation between words; **b.** semantic links between words.

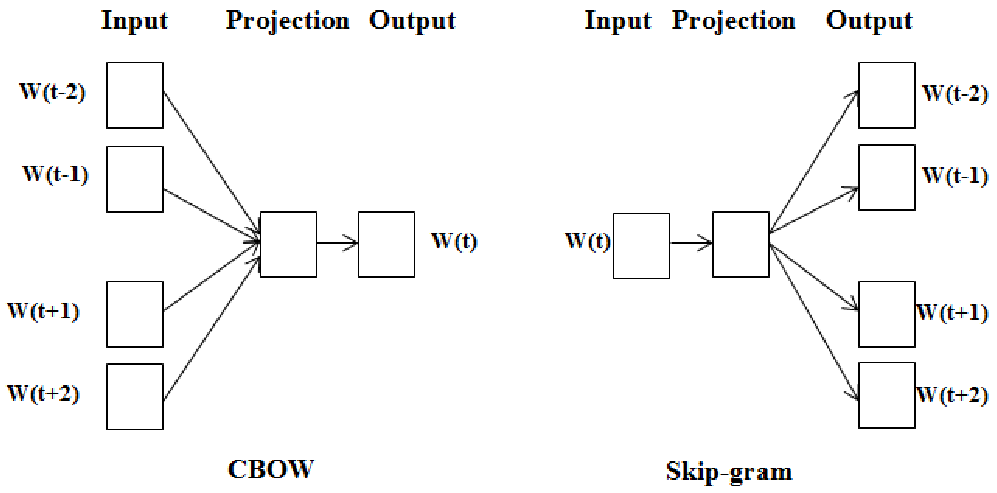

Figure 1.5: the CBOW and Skip-gram models' architecture. CBOW: according to the given context words predicts a probability of a target word; Skip-gram: for the target word given predicts a probability of context words.

1.2.2 Graph Embeddings

Graphs help to represent data in a meaningful and understandable way. However, it is not that convenient to work with them. Machine learning on graphs is limited (only a specific subset of mathematics, statistics and machine learning can be used), while vectors have a richer tool-set of approaches. Transformation graphs into vectors or a set of vectors, i.e. graph embeddings, may solve that issue.

Graph embeddings map each node to a low-dimensional feature vector and try to preserve the connection strengths between vertices. Generally, graph embeddings represent graphs by capturing the relevant information about it, such as the graph topology, vertices relations, sub-graphs. The more properties are encoded in embedding, the better results can be retrieved later in machine learning tasks.

Graph embeddings can be roughly divided into two groups: [6]

1. **Vertex embeddings:** Each vertex (node) is encoded with its vector representation. This embedding type can be used when it is necessary to perform visualisation or prediction the level of the nodes, e.g. 2D plane visualisation of the graph, or connections prediction based on nodes' similarities. The *node2vec* algorithm belongs to this group
2. **Graph embeddings**: A single vector represents the whole graph. These embeddings are used for making predictions at the graph level, e.g. comparison of chemical structures.

Initially, the range of methods of graph embeddings was small and included the following methods:

1. **Methods based on neighbours in the graph** – the most obvious of them is the number of common neighbours. It is intuitive that the greater the intersection of the sets of neighbours of two peaks, the more likely there is a connection between them – for example, most new acquaintances are with friends of friends;
2. **Methods based on graph paths** – the idea is that the shortest path between two vertices on a graph is related to the chance of a connection between them – the shorter the path, the higher the chance.

node2vec

The next milestone on the way to graph embeddings was the development of random walk methods. The *DeepWalk* and the *node2vec* algorithms were presented. The random walk, started in a selected node of the graph, moves to the random neighbour from a current node in a defined number of steps. Thus, the algorithm visits each vertex of the graph and collects information about the vertices and edges between them.

Both methods consist of three steps:

1. **sampling:** random walks sample a graph. From each node, a few random walks are performed.
2. **training:** random walks are treated as sentences, and each node is treated as a word in *word2vec* algorithm. The neural network accepts each node of the random walks as input and maximises the probability for predicting neighbour nodes.
3. **computing embeddings:** embeddings are output from a hidden layer of the neural network.

There is a difference between the two algorithms. The *DeepWalk* method performs walks randomly, which means that embeddings may not preserve the local neighbourhood of a node well. The *node2vec* approach corrects that by adding parameters p and q, which allow the researcher to bias transitions between nodes.

The q parameter defines how probable it is that the random walk would discover the undiscovered part of the graph, while the p parameter defines the probability of the random walk to return to the previous node (see Fig.1.6). The Figure illustrates the situation where a step from the red to the green vertex was made. The probability of going back to the visited (red) node is $1/p$, while the probability of going to the node not connected with the previous (red) node $1/q$. The probability of going to the red node's neighbour is one. [7]

The parameter p controls the discovery of the microscopic view around the node. In contrast, the parameter q discovers the broader neighbourhood. Both of them infer communities and complex dependencies.

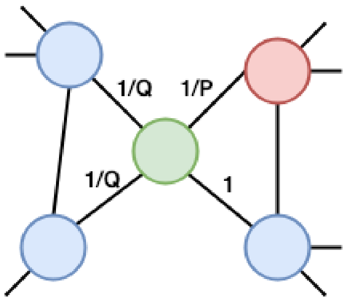

Figure 1.6: The probabilities of a random walk step in *node2vec*.

Other steps of the embedding are the same as in the *DeepWalk* approach. The main goal of *node2vec* is to ensure that vertices closely located in the graph receive a close representation in the vector map. In other words, the angle between the two obtained vectors is minimal.

Chapter 2

Implementation of the *node2vec* algorithm

2.1 Pseudo-code of the *node2vec* algorithm

The original paper [7] gives the following pseudo-code of the *node2vec* algorithm (see Algorithm 1).

Algorithm 1 The *node2vec* algorithm

```
function LearnFeatures(Graph $G = (V, E, W)$, Dimensions $d$, Walk per node $r$, Walk length $l$, Context size $k$, Return $p$, In-out $q$)
    $\pi$ = PreprocessModifiedWeights($G, p, q$)
    $G' = (V, E, \pi)$
    Initialize $walks$ to Empty
    for $iter = 1$ to $r$ do
        for all nodes $u \in V$ do
            $walk$ = node2vecWalk($G', u, l$)
            Append $walk$ to $walkes$
    $f$ = StochasticGradientDescent($k, d, walks$)
    return $f$

function node2vecWalk(Graph $G' = (V, E, \pi)$, Start node $u$, Length $l$)
    Initialize $walk$ to $[u]$
    for $walk_iter = 1$ to $l$ do
        $curr = walk[-1]$
        $V_{curr}$ = GetNeighbors($curr, G'$)
        $s$ = AliasSample($V_{curr}, \pi$)
        Append $s$ to $walk$
    return $walk$
```

As an input the algorithm receives a graph $G = (V, E, W)$, where V is vertices, E is edges and W is edges' weights of the graph G. Beside of that, it receives algorithm parameters defining the random walks(r number of walks made from one node,l length of each random walk, p return probability parameter, q probability parameter of visiting non-visited nodes), outputting embeddings dimension d and the context size k as a parameter of *word2vec*. From each node r biased random walks are constructed, where transitions are done with probabilities π_{vx}. Alias sampling allows to increase the efficiency of the algorithm to $O(1)$ and the optimisation by Stochastic Gradient Descent, i.e. an optimisation method whose output is the partial derivative of its inputs, is used. [7]

There are two official implementations of the *node2vec* algorithm written by the authors of the algorithm in Python and C++ as a part of Stanford Network Analysis Project (SNAP), the reference and the high-performance implementations, respectively [7]. Aside from that, also an implementation in Python 3 exists [3].

2.2 Optimisation of the original algorithm

The original implementation of the algorithm might be thoroughly designed; however, a study of existing unofficial implementations has shown that some parts of it can be improved.

2.2.1 Speed acceleration

The first thing to consider is the size of the graph and the time of the working on it. Indeed, some graphs may become too huge to handle. So, calculation embeddings on it may take a long time, especially when it is needed to make 1000 calculations for one experiment. For example, during the learning of the node2vec parameters sensibility, each iteration of the algorithm run took up to 100 seconds, which in turn led to 36 days long calculation for the experiments. So, research on possible ways of managing it was made.

One of the ways how to solve the speed problem is using multi-core computers and running calculations parallelly. For these purposes may be appropriate to use cloud virtual massively multi-cores machines, e.g., Google Cloud Computer Engines.

Another way of achieving speed acceleration is shown in the implementation of *node2vec* with the usage of sparse matrices [14]. A sparse matrix is a matrix with predominantly zero elements. Constructing random walks on sparse matrices is much faster. Also, it allows reducing the number of jumping from the different parts of memory, unlike while working with graphs, which uses the linked list idea there each node contains pointers to each other [13].

2.2.2 Reproduction of results

The second thing and the critical issue that authors of the original paper concealed is the reproduction of results. The same results reproduce at each iteration is necessary to conduct reliable calculations.

The implementation of *node2vec* with ‘seed‘ parameter was suggested by CSIRO’s Data61 group from Australia in the StellarGraph library for working with graphs and machine learning. [2].

The main difference between the original paper algorithm and the StellarGraph implementation lies in the construction of biased random walks. In latter probabilities of transition are ruled by arbitrary weights. During the graph sampling the transition made to a next neighbour is selected at random, weighted by the iterator *weights* of arbitrary (non-negative) floats. That is, x is returned with probability $weights[x]/sum(weights)$. Doing a single sample with arbitrary weights is much (5x or more) faster than doing it using the *numpy.random.choice* function, because the latter requires much more preprocessing (normalised probabilities), and does a lot of conversions, checks, preprocessing internally. [2]

Chapter 3

Analysis of the heuristics using in *node2vec*

3.1 Objective function

Many machine learning tasks working with graphical data involve predictions on nodes and connections between them. The most typical one is a classification task or cluster analysis, that is, the attempt to differentiate nodes and categorise them into groups. For example, in social networks, it can be useful to find connections between users or predict their interests depending on possible relevant properties of nodes referring to the users.

As an objective function in this work, a cluster analysis of graph vertices task was used. Optimisation of the objective function consisted of the search for the correct parameters of the *node2vec* algorithms. Different ways to solve this problem can be found, but the goal of this work is to study the solutions employing heuristics.

The data in the objective function goes through several stages:

1. input data is converted into a graph;
2. the graph is sampled with random biased walks implemented in the *node2vec* algorithm;
3. the random walks are treated as a corpus by *word2vec* and are transformed into embeddings describing the graph;
4. the embeddings are used as properties for the cluster analysis;
5. the cluster analysis is evaluated using an adjusted rand score, which is maximised.

A pseudo-code describes the objective function (see Algorithm 2).

Algorithm 2 Objective function

```
function EVALUATE( $p$, $q$, $num\_walks$, $len\_walks$, $window$)
    $weighted\_walks$ = RandomWalks.run($graph.nodes, p, q, num\_walks, len\_walks, seed$)
    $weighted\_walks$ = to_str($weighted\_walks$)
    $model$ = word2vec($weighted\_walks, window$)
    set $known\_lables$ to $Empty$
    for all nodes $i$ in $model.indices$ do
        Append $label$ of $i$ to $known\_lables$
    $n\_clusters$ = len($labels$.unique())
    $km$ = KMeans($n\_clusters$).fit_pedict($model.vectors$)
    $score$ = RandIndex($known\_labels, km$)
    return $score$
```

3.2 Methods used

Since the ability to replicate the results was vital, it was decided to work with the StellarGraph implementation of the algorithm. The *RandomWalks.run*() function in the Algorithm 2 creates the instance of *BiasedRandomWalk* class and runs the graph sampling by random walks.

For vertices labelling the k-means clustering algorithm was chosen. This method randomly picks the centres (centroids) of the future clusters and calculates the closest nodes to each centre. On each iteration, the centre of the cluster moves to the actual centre of the nodes batch and rearrange the closest nodes to the new centre. The latter operation repeats until the centres are stabled (see Fig. 3.1). The k-means clustering algorithm allows setting the number of clusters.

For cluster analysis estimation the Rand index was chosen. The index collates an array of cluster numbers with a testing array of pre-defined labels. The closer the value to 1.0, the better the result of clustering. If the metric reaches exactly 1.0, then the compared arrays are identical (up to a permutation). On the contrary, 0.0 means that clusters are different. This experiment aims to maximise this value.

3.3 Parameters sensitivity

The quality of the embeddings, and thus the quality of cluster analysis, heavily depends on the parameters chosen for the model construction. Moreover, it seems that for each data package its parameters should be detected. Thus, finding the optimal parameters can be quite tricky.

Some selected for studying parameters of *node2vec* are as follows:

dimensions (parameter) is a number of dimensions of the embeddings collected from the algorithm. Since the model contains more information about a network, the better classification results might be expected. Though, calculations on higher dimensions request more resources. The default value is 128.

There is a difference between the two algorithms.

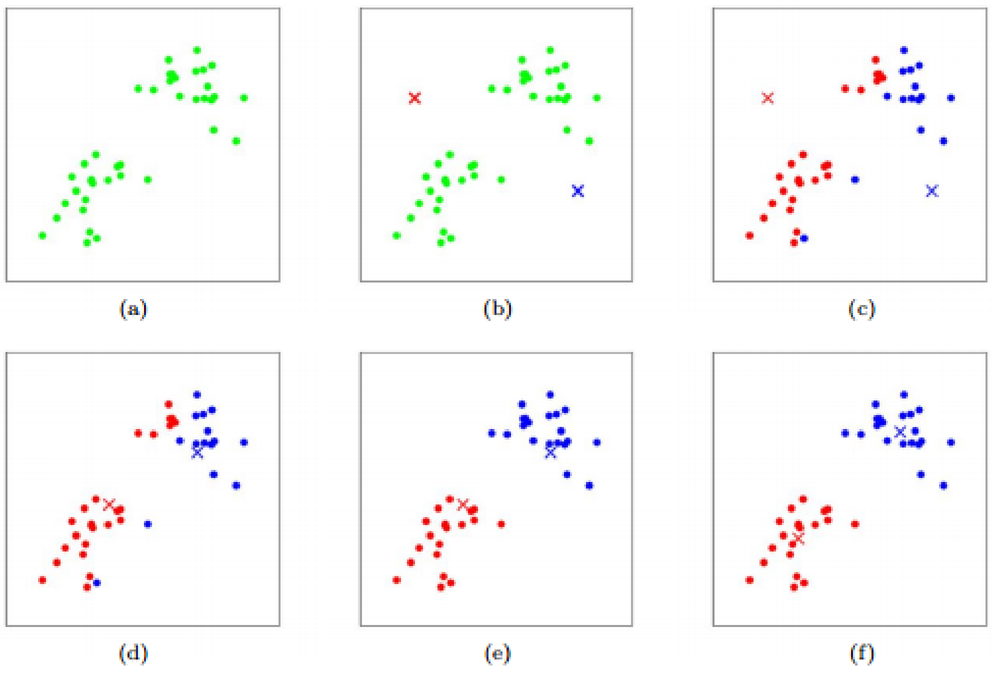

Figure 3.1: The k-means algorithm. (a) Original dataset. (b) Random initial cluster centroids. (c-f) Illustration of running two iterations of k-means.

walk_length is a number of nodes in each random walk. The sequences of nodes are treated as the sentences and are fed to the *word2vec* algorithm. For smaller graphs, it appears to be more feasible to decrease the length of walks. The default value is 80.

num_walks is a number of walks made from each node. Larger values of this parameter theoretically lead to the more detailed embeddings. The default value is 10.

p is a return parameter that specifies the probability of the moving back to the already visited node. Bigger values provide more local search (BFS) and describe the local relationships between the nodes. The default value is 1.0.

q is a neighbour parameter defining the probability of visiting the earlier unvisited nodes. The bigger value provides the depth first search technique and learns the external relationships in a data set. Depending on the characteristics of the graph, it may be more appropriate to commit more breadth-first (BFS) like or depth-first (DFS) like search. The default value is 1.0.

Some selected parameters of *word2vec* are as follows:

window defines the number of nodes before and after the node given treated as a context of the node, i.e. nodes that may appear together and have some relations. The recommended value is 10.

min_count sets a minimal frequency of the node's occurrence in the training model. The nodes with a total frequency lower than the value set are ignored.

Here, only a particular part of the parameters was considered. There are even more possibilities to manage the work of algorithms. Heuristics can help us in choosing the right combinations.

3.4 Heuristics

The term "heuristic" originates from the Ancient Greek word meaning "find" or "discover". A heuristics technique is a method of problem-solving based on common logic, subconscious thinking or rules of nature. Even though such methods do not guarantee the most optimal, rational or perfect results, they can be surprisingly efficient in finding quick solutions. That makes heuristics widely used in various spheres of our lives. Examples of heuristics employment are a rule of thumb, trial and error, educated guess, brainstorm, and lots of others. [1]

In a situation of ever-increasing data sizes, scientists are in search of methods of analysis that will perform effectively and fast on big data sets. Under such conditions often only the good enough solution, or so-called satisfactory solution, is searched.

3.4.1 Hill climbing (or Shoot and go)

A Hill climbing algorithm is a mathematical optimisation technique that belongs to the family of local search algorithms. The algorithm starts with an arbitrary solution to a problem and then tries to improve it by gradual changing of one of its elements. If the new solution gives a better result, the change is accepted. The elements of the solution continue to be changed until no improvements can be reached. At this point, the algorithm finishes.

The weak point of this algorithm is a risk that the solution will become trapped at a local extremum of the objective function, or in the case of the plateau, no changes will lead to any improvement.

The relative simplicity of the algorithm makes it popular among optimisation algorithms. Although more advanced algorithms such as an annealing simulation or a tabu search may give better results, hill climbing works just as well in some situations (especially when the time for searching is limited). That is important in real-time systems, provided that a small number of steps converges to a good-enough solution (optimal or close to optimal ones). [1]

3.4.2 Simulated annealing

This method has been observed in nature. It is based on substances crystallisation process increasing the uniformity of metals.

All materials have a crystalline lattice, which describes the geometrical position of atoms. Each state of the system, the set of the atoms' positions, corresponds to a certain level of energy. The purpose of metal annealing is to bring the system into a state with lower energy. The lower the energy level, the "better" the crystal lattice, i.e. the metal has fewer defects and higher strength.

During heating and subsequent slow and controlled cooling, the atoms change their initial positions. Atoms tend to get into a state with less energy; however, with a certain probability, they may appear in a position with higher energy. This probability decreases with temperature. The process ends when the temperature drops to a pre-determined value.

In this process, energy is minimised. At higher temperatures, the algorithm acts as a random search, accepting any steps even though they do not lead to a better result. That allows us to avoid trapping in a local minimum with a high probability. With the cooling, the algorithm becomes more greedy and accepts only improving changes in the system, i.e. it acts like the Hill climbing algorithm. [1]

3.4.3 Genetic algorithms

Genetic or evolutionary algorithms were inspired by nature as well and are based on Darwin's principle of natural selection. There are three stages of the method, as follows: selection, crossover, mutation. The algorithm randomly creates the populations of the most promising solutions.

The process of creating a population from individuals of the previous generation is called the reproduction strategy. Some of their types are canonical, simple and breedN [1]. In a canonical approach, all parents are replaced by new offsprings, i.e. the crossover probability equals one. In the simple method, the crossover probability decreases and some parents may be just cloned. In a BreedN strategy, the imaginary breeder is applied. That breeder chooses the parents with the best qualities and uses them to create the next generation.

A crossover is carried out in several ways. For easier explanation, let us assume parents $x = (x_1, x_2, \ldots, x_n)$ and $y_2 = (y_1, y_2, \ldots, y_n)$. Then the children will be created, as follows:

- one-point crossover (1-point): a cut point $p_1 \in [1, n]$ defines the point, where the exchange between parents happens. For instance: $p_1 = 2$ formulates the children $c_1 = (x_1, x_2, y_3, \ldots, y_n)$ and $c_2 = (y_1, y_2, x_3, \ldots, x_n)$;
- two-points crossover (2-point): two cut points, $p_1 < p_2$, from the same interval $[1, n]$ define the part of the parents' chromosomes interchange. So, for $p_1 = 2$ and $p_2 = 4$ children are formulated as $c_1 = (x_1, x_2, y_3, y_4, x_5, \ldots, x_n)$ and $c_2 = (y_1, y_2, x_3, x_4, y_5, \ldots, y_n)$;
- uniform: each parents' chromosome exchanges with a 50/50 probability;
- parent chromosomes alternate in every child, so, $c_1 = (x_1, y_2, x_3, \ldots, x_n)$ and $c_2 = (y_1, x_2, y_3, \ldots, y_n)$, assuming n is even.

If the result is unsatisfactory, these steps are repeated until the result starts to be suited, or one of the following conditions occurs: the number of generations (cycles) reaches a pre-selected maximum or the mutation time is up. [1]

3.5 Experiments

3.5.1 The functionality of instruments chosen

After a rather long series of unsuccessful attempts to work with medium-sized (with 1005 and 10312 vertices) and small graphs (just 24 vertices), it was decided to test the functionality of the methods on elementary data. So, it is easy to interpret the data and results achieved by the method.

An artificial data set was created to simplify and speed calculations. The graph consists of 25 vertices divided into five equally large groups. Inside the group, all the vertices are connected. Also, the 'leader' node of each group is connected to the 'leader' nodes of the other groups, and one-eighth of all edges in the graph are random (see Fig. 3.2).

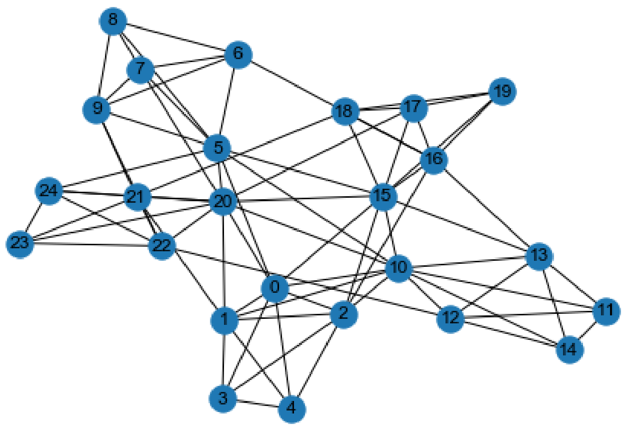

Figure 3.2: The test data graph.

Only for the very first experiment, even simpler data was used, which should have led to a maximum value of the objective function with a hundred per cent probability. This data is similar to the one-eighth data, but it has edges only within each vertex group. Thus, nodes from different groups were not connected at all, and it would be bizarre if, as a result of the experiment, they were distributed among the foreign groups. Due to this step, it was possible to detect errors in the calculations, namely during the cluster analysis, and correct them.

Figure 3.3 shows the result of corrected clusterisation. Nodes of the pure data set are perfectly divided into five clusters. The objective function on default parameters is maximised, i.e. its values equal to 1.0. For comparison, the value of the objective function with the same default parameters but calculated for the one-eighth data is

0.625. These and the following calculations can be found in the repository dedicated to the bachelor project on github. [5]

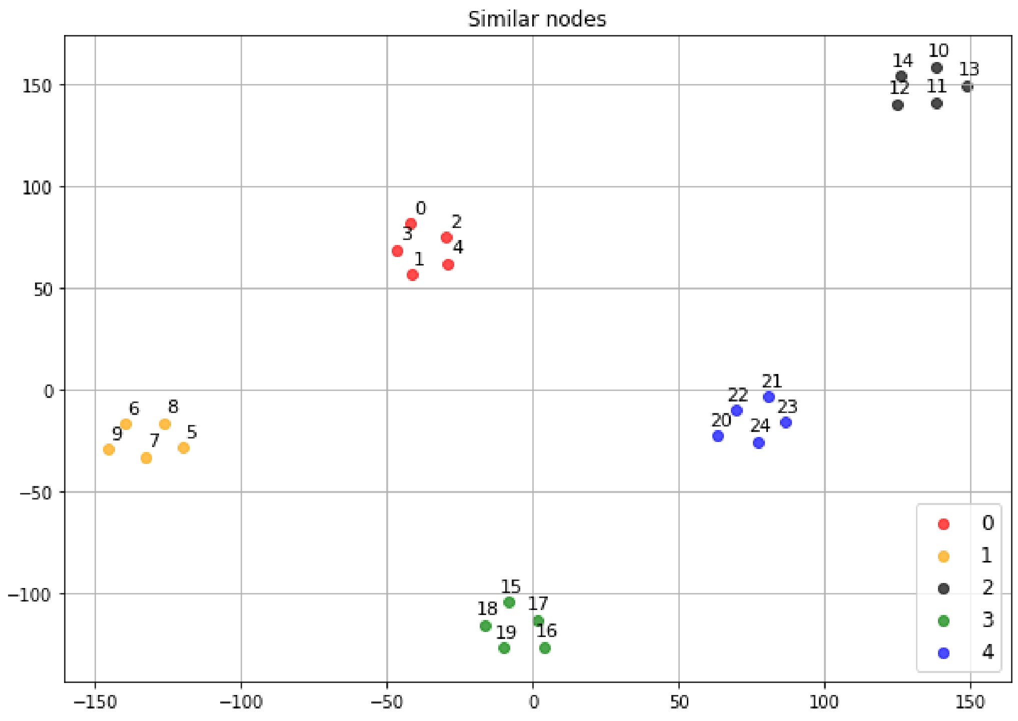

Figure 3.3: Clusters analysis of the pure data. Colours stand for the groups of nodes.

3.5.2 Parameters sensitivity

To find the optimum combination of settings, one should examine the influence of individual parameters on the final result. In theory, knowledge of the individual parameters extrema can help in finding a combination of optimal parameters.

The original work shows the opposite relation between the quality of embeddings and the p and q parameter. That means that with an increase in the value of these two parameters, the quality of the embeddings decreases. The bond between the result and the dimension, number of walks and length of walks, on the contrary, is direct. That is, an increase in the value of the parameters leads to an improvement in the result. [7]

Dependencies between parameters and the Rand index score was calculated for the one-eight data as well. According to the schematic drawing 3.4, the parameters p and q influence the result in the opposite direction. This result does not confirm the description of the influence of parameters in the original work. Therefore, it can be assumed that for each data graph, we can find the optimal values of the parameters p and q, which may not coincide with the optimal values for some other data packet.

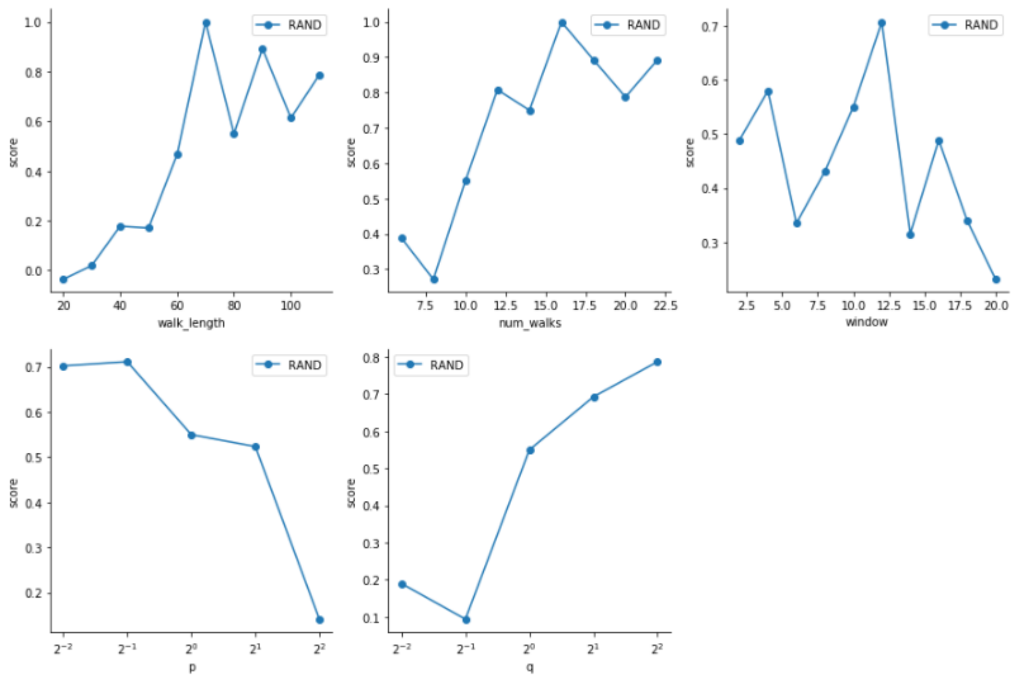

Figure 3.4: Parameters sensitivity for the one-eighth graph.

Moreover, the effect of settings on the system may differ. Indeed, the connections between the points in the graph are affected by those parameters at most.

The drawing also clearly confirms the assumption that the greater the length of the walks and their number, the more qualitatively the algorithm describes the graph.

It is also interesting to note that, by simple enumeration, two optimal parameter values were found here. It was the sixth iteration of inspecting the length of walks with a value of 70 and also the sixth inspection iteration of a number of walks with a value of 16. Other parameters, in this case, kept default.

Such a simple example was specifically used here so that finding its optimal solution does not take much time. The next chapter section deals with the ability of heuristics improving that result.

3.5.3 Experiments with heuristics

The experimental framework was borrowed from the Heuristic course at FNSPE, Czech Technical University, with some changes. Two super-classes are implemented in the framework: ObjFun and Heuristic. ObjFun stores evaluation values, parameter boundaries, a graph, a random walks generator instance. The Heuristic class stores the best-found parameters and the corresponding objective function value. It is responsible for the experiment to stop, in case the better solution was found, or the maximum number of iterations was reached. [12]

In each heuristic the p parameter values were examined. Other four parameters were unchanged, their default values were taken. The maximum number of evaluations and the number of experiments runs were set at 50.

Hill climbing (or Shoot and go)

The Shoot and Go algorithm starts with a random point, for the neighbours of which the value of the objective function is calculated. The "best neighbour" is accepted, and the algorithm begins to evaluate its neighbouring points. If no improvements are possible at the point, the algorithm starts from the random point again, i.e. makes a jump from the local search. The *hmax* parameter sets the maximum of local searches. Zero value of that parameter means that the algorithm acts as a Random shoot, i.e. never considers the neighbour and tries to hit the optima randomly.

Algorithm 3 shows the pseudo-code of Hill climbing.

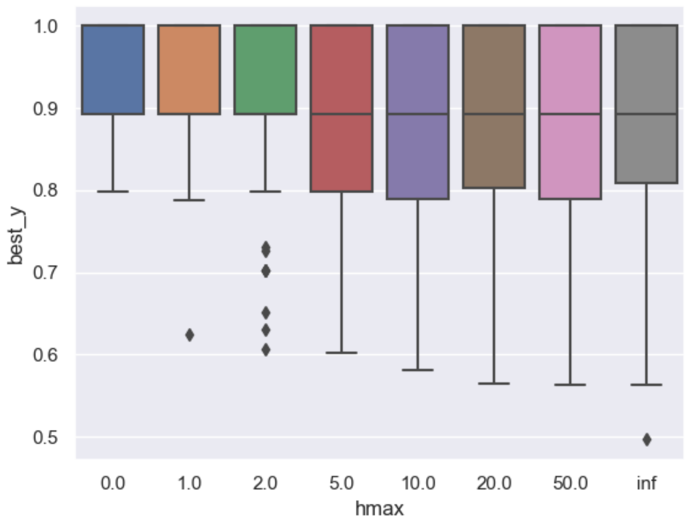

Figure 3.5: Examination of the *hmax* parameter of the Shoot and go heuristic. All parameters except p are default.

The *hmax* parameter examination is illustrated in Fig. 3.5. The data is shown in the form of a Box and whisker plot. It is a convenient way to map statistical information. Whiskers show the spread of the data. For instance, for $hmax = 0$ the maximum value is 1.0, and the minimum is 0.8. A line within boxes gives a medium, and it

Algorithm 3 Shoot and go

```
function SEARCH
    while true do
        $x = of$.generate_point()
        $parameters = [x, 1.0, 10, 80, 10\ ]$
        evaluate($parameters$)
        if $hmax > 0$ then
            steepest_descent($x$)
```

```
function STEEPEST_DESCENT($x$)
    $desc\_best\_y = 0$
    $desc\_best\_x = x$
    $h = 0$
    $go = True$
    while $go$ and $h < hmax$ do
        $go = False$
        $h$ += 1
        $neighborhood = of$.get_neighborhood($desc\_best\_x, 0.125 \cdot 2$)
        if $random\_descent$ then
            random.shuffle($neighborhood$)
        for $xn$ in $neighborhood$ do
            $parameters = [xn, 1.0, 10, 80, 10\ ]$
            $yn$ = evaluate($parameters$)
            if $yn > desc\_best\_y$ then
                $desc\_best\_y = yn$
                $desc\_best\_x = xn$
                $go = True$
                if random_descent then
                    break
```

```
function GET_NEIGHBORHOOD($x, d$)
    $left = [x$ for $x$ in linspace$(x - 0.125, x - d, 2)$ if $x >= left\_border]$
    $right = [x$ for $x$ in linspace$(x + 0.125, x + d, 2)$ if $x < right\_border]$
    if size($left$) == 0 then
        return $right$
    else
        if size($right$) == 0 then
            return $left$
        else
            return concatenate($left, right$)
```

```
function GENERATE_POINT
    return random.uniform$(0.125, 4.125)$
```

means that half of the values are less than the median. The borders of the boxes are the medium of the half, i.e. lower and upper quartiles. Diamonds outside the box and whiskers indicate statistically not important abnormal values. According to the experiment, it seems that the algorithm acted most successfully at values $hmax = 0.0$, $hmax = 1.0$ and $hmax = 2.0$ because 75% of their values are bigger than 0.9.

Figure 3.6 shows the statistical data about the number of evaluation to find the optimal solution. It is hard to tell if there is any system in this result. The maximum value of evaluation was set to 50, and apparently, it was enough to get the right parameter's value. It seems that the most felicitous value of $hmax$ was 5.0 since most of the optimal values were found on 12th-23th steps.

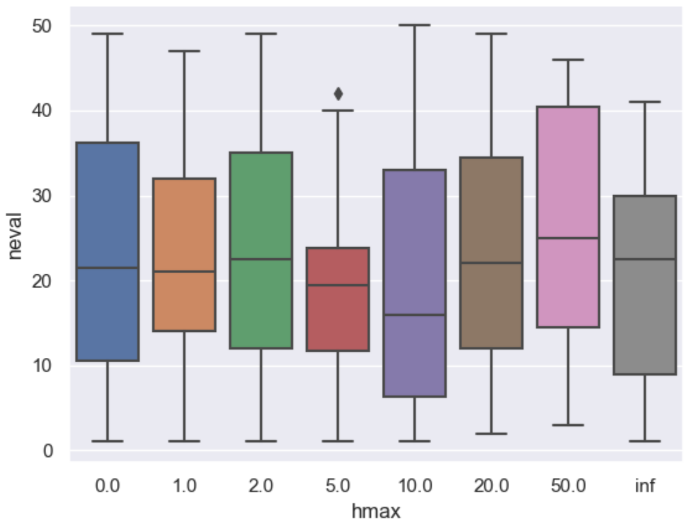

Figure 3.6: The number of iteration of the algorithm to find the optimal solution.

The reliability was measured by the Feoktistov criterion (FEO), which is calculated by the formula:

$$FEO = \frac{MNE}{REL}$$

$$MNE = \frac{1}{m}\sum_{i=1}^{m} neval_i$$

$$REL = \frac{m}{q}$$

where MNE is a mean number of objective function evaluations, REL is a reliability, m is a number of successful iterations and q is a total number of iterations.

According to the Feoktistov criterion, the most reliable in this case was the Random shooting algorithm (see Tab. 3.1), i.e. $hmax = 0$ setting with $FEO = 42.75$.

Repeating of the algorithm with Random Descent parameter improved reliability of the result by almost 10 (see Tab. 3.2). By the neighbourhood diameter examination no improvements were achieved.

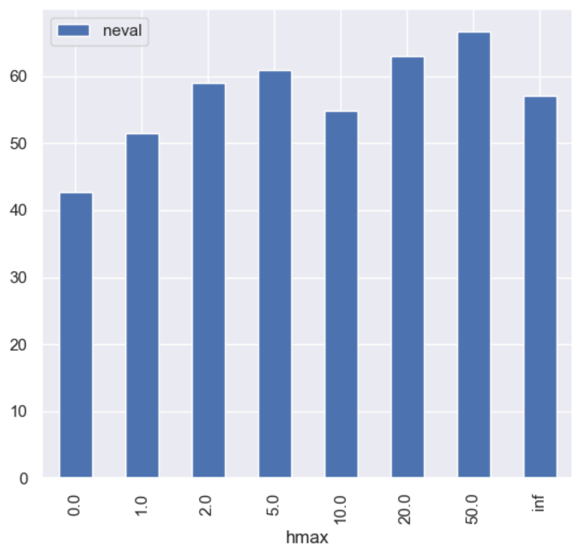

Figure 3.7: Reliability of the Shoot and go algorithm according to the Feoktistov criterion by hmax.

Heuristic	**hmax**	**feo**	**mne**	**rel**
SG_0	0.0	42.75	22.23	0.52
SG_1	1.0	51.42	23.65	0.46
SG_2	2.0	59.00	23.60	0.40
SG_5	5.0	60.93	19.50	0.32
SG_10	10.0	54.78	19.72	0.36
SG_20	20.0	63.02	23.94	0.38
SG_50	50.0	66.62	25.31	0.38
SG_inf	inf	57.09	20.55	0.36

Table 3.1: Reliability of the Shoot and go heuristic.

Heuristic	**hmax**	**feo**	**mne**	**rel**
SG_RD_1	1.0	32.93	20.42	0.62
SG_0	0.0	42.75	22.23	0.52
SG_RD_0	0.0	44.16	22.08	0.50
SG_RD_2	2.0	46.02	24.85	0.54
SG_RD_inf	inf	50.17	17.05	0.34

Table 3.2: Reliability of the Random Descent Shoot and go heuristic.

Fast simulated annealing

Fast simulated annealing (FSA) is a variation of the Simulated annealing algorithm. FSA is a semi-local search consisting of occasional long jumps. Classical simulated annealing (CSA) is strictly local, so it is slower compared to FSA. The difference is also in the cooling schedule, which is inversely linear in time in FSA and inversely proportional to the logarithmic function of time in CSA.

The main parameters of the FSA algorithm are:

- $T_0 \in \mathbb{R}$ as an initial temperature,
- $n_0 \in \mathbb{N}$ and $\alpha \in \mathbb{R}$ as a cooling strategy parameters.

The current temperature at k-th step of the algorithm is:

$$T = \begin{cases} \frac{T_0}{1+(\frac{k}{n_0})^{\alpha}}, & \text{if } \alpha > 0 \\ T_0 \cdot \exp(-(\frac{k}{n_0})^{-\alpha}) & \text{otherwise} \end{cases} \tag{3.1}$$

The current solution x is mutated using a Cauchy mutation for u random number if the following condition is true:

$$u < \frac{1}{2} + \frac{\arctan(s)}{\pi}$$

$$s = \frac{f_y - f_x}{T}$$

The Cauchy mutation operator with perimeter controlled by parameter $r \in \mathbb{R}$, where r is a random number, is

$$y = x + r \cdot \tan(\pi(r - \frac{1}{2}))$$

Algorithm 4 shows the pseudo-code of the algorithm.

First, an experiment with different initial temperatures was carried out. In this experiment, only the p parameter was again sought; the remaining settings were default ones. The optimal value of the objective function was reached only occasionally. Figure 3.8 shows that most of the best values of the experiment are less than 0.62.

According to Tab. 3.3, FSA performed similarly at the initial temperature $T_0 = 1e{-}2$ and $T_0 = 1$.

An attempt to increase the objective function value by changing the r mutation parameter also was not successful (see Figure 3.9).

FSA did not reach the optimal value in 683 experiments, and in 17 cases it achieved the optima at the first run. It can be assumed that this behaviour is not trustable, and the optimal values are just random. So, the MNE values are not credible in that case (see Tab.3.4).

Algorithm 4 Fast simulated annealing

```
function SEARCH
    $x = of$.generate_point()
    $params = [x, 1.0, 10, 80, 10]$
    $f_x$ = evaluate($params$)
    while $True$ do
        $k = neval - 1$
        $T = T_0/(1 + (k/n_0)^\alpha)$ if $\alpha > 0$ else $T_0 \cdot exp(-(k/n_0)^{-\alpha})$
        $y$ = CauchyMutation($x$)
        $params\_mut = [y, 1.0, 10, 80, 10]$
        $f_y$ = evaluate($params\_mut$)
        $s = (f_x - f_y)/T$
        $swap$ = random.uniform() $< 1/2 + \arctan(s)/\pi$
        if $swap$ then
            $x = y$
            $f_x = f_y$

function CauchyMutation($x$)
    $n$ = size($x$)
    $u$ = random.uniform(low= 0.0, high= 1.0, size= $n$)
    $x_{new} = x + r \cdot \tan(\pi \cdot (u - 1/2))$
    $x_{new}\_corrected$ = correct($x_{new}$)
    return $x_{new}\_corrected$
```

Heuristic	**T_0**	**feo**	**mne**	**rel**
FSA_1e-10_1_2_0.5	$1e-10$	50.0	1.0	0.02
FSA_0.01_1_2_0.5	$1e-2$	NaN	NaN	0.00
FSA_1_1_2_0.5	$1e-0$	50.0	1.0	0.02
FSA_inf_1_2_0.5	inf	NaN	NaN	0.00

Table 3.3: Reliability of FSA by the initial temperature T_0.

Heuristic	**T_0**	**n_0**	**r**	**feo**	**mne**	**rel**
FSA_1e-10_5_2_0.1	$1e-10$	5	0.10	12.50	1.0	0.08
FSA_1e-10_5_2_0.01	$1e-10$	5	0.01	16.66	1.0	0.06
FSA_1e-10_10_2_0.5	$1e-10$	10	0.50	25.00	1.0	0.04
FSA_1e-10_3_2_0.5	$1e-10$	3	0.50	25.00	1.0	0.04
FSA_1e-10_5_2_0.5	$1e-10$	5	0.50	25.00	1.0	0.04

Table 3.4: Reliability of FSA after all improvements.

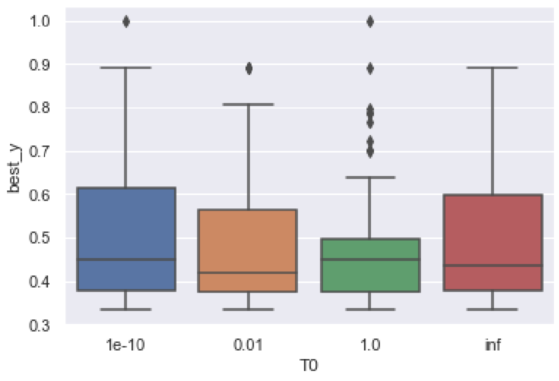

Figure 3.8: FSA by initial temperature T_0.

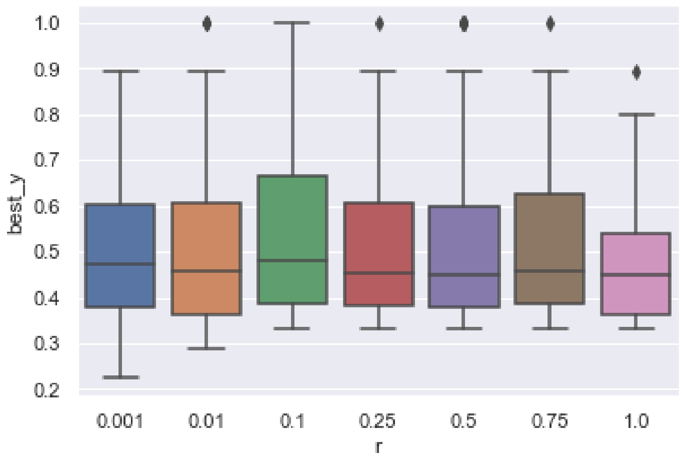

Figure 3.9: FSA by r mutation parameter.

Genetic algorithms

Genetic heuristics are based on the principle of Darwin's evolutional theory. The work of the algorithm begins with the creation of a random generation of solutions, which subsequently undergoes selection, crossbreeding, and mutation. From the generation of solutions obtained, the best representatives are selected, and the cycle begins again. A wide variation of breeding strategies is presented in the literature. In the experiment, a 1-point crossover and a uniform mutation were used. [4]

Algorithm 5 shows the pseudo-code of the algorithm.

In the experiment, the effect of the number of generations on the search speed of optimal parameter values was considered. It was expected that the average value should give the best results for the following reasons. When using a small population, there is a high probability of delaying the search process. It may become that an initial population does not contain fitting parameters, and even a mutation would not save the situation. Variation of chromosomes can be solved by increasing the population. However, this can significantly increase the calculation time.

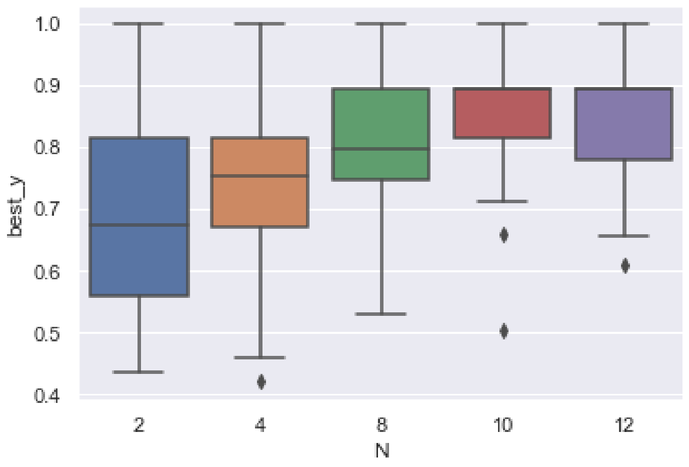

Figure 3.10: GO by N as size of a population.

Figure 3.11 shows the results of the experiment. According to the statistics the algorithm worked fast with population size of 2 and 12.

According to FEO, two solutions in one population give the most reliable results in the Genetic heuristics, as illustrated in Tab. 3.5.

Heuristic	**N**	**feo**	**mne**	**rel**
GO_2	2	126.00	12.60	0.10
GO_4	4	129.17	15.50	0.12
GO_8	8	65.63	10.50	0.16
GO_10	10	73.00	14.60	0.20
GO_12	12	73.97	16.27	0.22

Table 3.5: Reliability of the Genetic heuristics with different population sizes.

Algorithm 5 Genetic algorithm

```
function SEARCH
    $pop\_size = (N, 5)$
    $new\_population$ = create_population($pop\_size$)
    while $True$ do
        for $generation$ in range($N$) do
            $fitness$ = fitness_calc($new\_population$)
            $parents$ = select_parents($new\_population, fitness, p\_num$)
            $offspring\_crossover$ = crossover($parents, offspring\_size$)
            $offspring\_mutation$ = mutation($offspring\_crossover$)
            $new\_population = parents + offspring\_mutation$

function CREATE_POPULATION($pop\_size$)
    $population$ = zeros($shape = pop\_size$)
    for $individ$ in $population$ do
        $individ[0]$ = random.uniform($low = 0.125, high = 4.125$)
        $individ[1-4] = 1.0, 10, 80, 10$
    return $population$

function SELECT_PARENTS($new\_population, fitness, parents\_num$)
    for $num$ in range($parents\_num$) do
        $max\_fitness\_idx$ =max($fitness$))
        $max\_fitness\_idx = max\_fitness\_idx[0][0]$
        $parents[num, :] = new\_population[max\_fitness\_idx, :]$
        $fitness[max\_fitness\_idx] = 0$
    return $parents$

function FITNESS_CALC($new\_population$)
    for $ind$ in $new\_population$ do
        $fitness$.append($of$.evaluate($ind$))
    return $fitness$

function CROSSOVER($parents, offspring\_size$)
    $point = offspring\_size[1]/2$
    for $k$ in range($offspring\_size[0]$) do
        $parent1\_idx = k$ % $parents$.shape[0]
        $parent2\_idx = (k + 1)$ % $parents$.shape[0]
        $offspring[k$, before $point] = parents[parent1\_idx$, before $point]$
        $offspring[k$, after $point] = parents[parent2\_idx$, after $point]$
    return $offspring$
```

Algorithm 6 Genetic algorithm: mutation

```
function MUTATION($offspring\_cross, num\_mutations = 1$)
    for $idx$ in range($offspring\_cross$.shape[0]) do
        for $num$ in range($num\_mutations$) do
            $rand\_value$ = random.uniform($-0.1, 1.0, 1$)
            $offspring\_cross[idx, 0] = offspring\_cross[idx, 0] + rand\_value$
            if $offspring\_cross[idx, 0] <= 0$ then
                $offspring\_cross[idx, 0] = 0.1$
    return $offspring\_crossover$
```

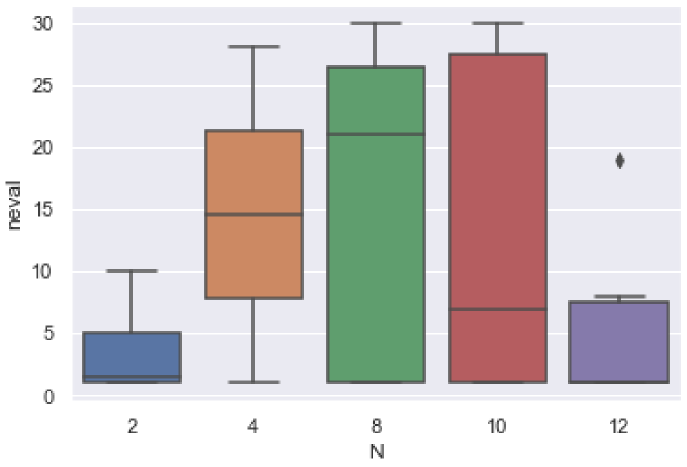

Figure 3.11: The effect of the generations size on the number of evaluations.

Comparison of the used algorithms reliability

The table 3.6 contains the best FEO values for the used heuristics. The smaller the values of the Feoktistov criterion, the more reliable and faster the heuristic. According to the carried out experiments, the most reliable one is the Random descent Shoot and Go algorithm with $hmax = 1$.

Heuristic	**FEO**	**MNE**	**REL**
SG_RD_1	32.93	20.42	0.62
FSA_1e-10_1_2_0.5	50.00	1.00	0.02
GO_8	65.63	10.50	0.16

Table 3.6: The best performance values of the heuristics.

Heuristic	**mean**	**median**	**mode**	**max**
SG	0.90	0.89	1.00	1.00
FSA	0.51	0.45	0.33	1.0
GO	0.80	0.80	0.89	1.0

Table 3.7: Statistics of the heuristics performance by $best\_y$ value.

Chapter 4

Application of the algorithm on real data and analysis of results

4.1 Data used

It was decided to use the email-Eu-core data set proposed by the SNAP group at Stanford University. The network was generated using email data from a large European research institution. So, presumably, they can be considered as data from the real world. [9]

The network contains 1005 nodes and 25571 edges, where each node represents one member of the institution. There is an edge (u, v) between two vertices if a person sends at least one email to another person. It contains communication between the institution members only, incoming and outgoing messages to the rest of the world are ignored.

Furthermore, the network contains labels that define the community membership of each node. Each institution employee belongs to one of 42 departments. The number of people in each department is different, and groups with less than ten people may make the data noisy. To expedite the calculations, the ten largest departments were taken into consideration (see Figure 4.1).

4.2 Data visualisation

Figure 4.2 illustrates the graph representation of the data used, which was created in Gephi. Gephi is an open-source application for visualisation and exploration of various kinds of graphs and networks.

After loading an extensive data set into the software, the user can only see a cloud of black dots displaying vertices of the graph, since their position is random at first. Thus some additional manipulations with data are needed.

In Figure 4.2, each node is coloured under its membership in the particular de-

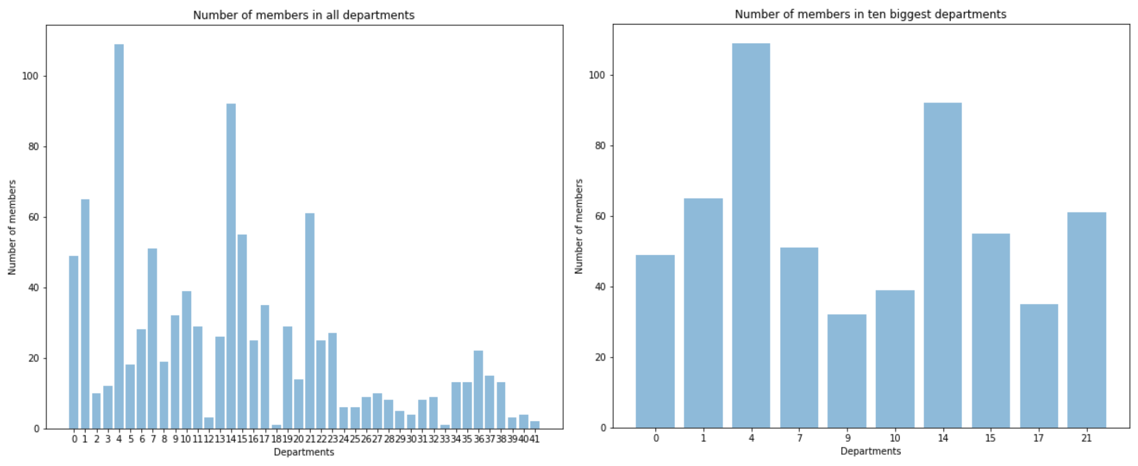

Figure 4.1:

partment. The biggest departments were filtered out to make the graph clearer and visually understandable. So nodes that belong to smaller groups were hidden. The built-in Force Atlas2 algorithm set the graph shape. The "force-based" algorithm works as follows: connected nodes attract one another, and disconnected nodes are pushed apart [10]. It can be seen that the working data was grouped into homogeneous clusters, and vertices from each department were positioned close to each other.

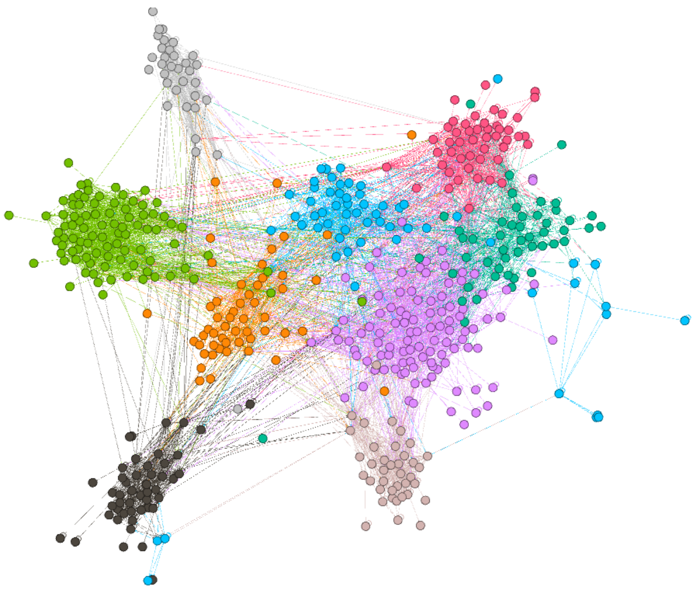

Figure 4.2: The emails network representation in Gephi. The ten prominent department nodes are illustrated, the colours represent different departments.

4.3 Cluster analysis

According to the technology described in Chapter 3, a cluster analysis was carried out. Figure 4.3 shows a new representation of the graph. Colours still refer to the departments, but the layout is different. The newly discovered clusters group nodes and the nodes with the same label number are situated close to each other. It seems that clusters are less homogeneous than in the previous visualisation.

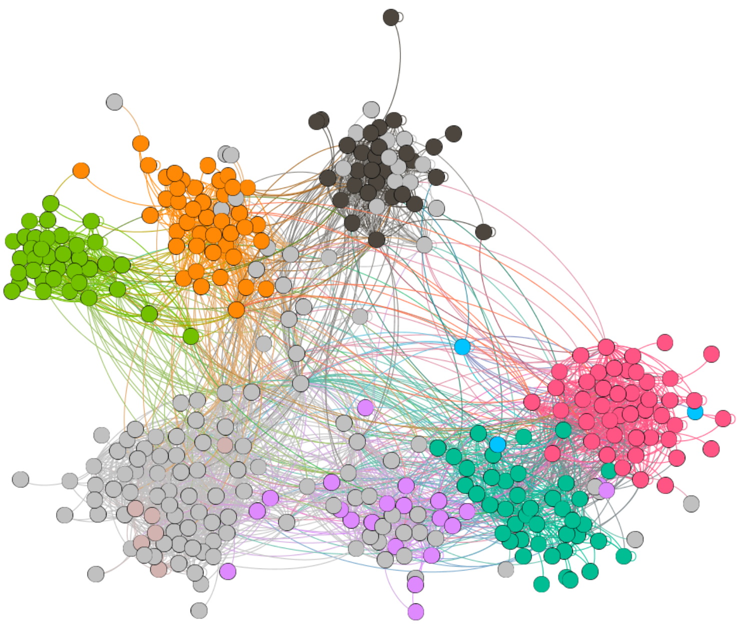

Figure 4.3: Visualisation of cluster obtained from the *node2vec* model and pre-known departments.

The Rand index of this result is 0.46.

4.4 Parameters sensibility

Parameters sensibility for the emails data set was also calculated. (see Fig. 4.4) Not all the charts look like it was expected, but some relationships still can be seen.

According to the first chart, it is evident that an increase in the value of the dimension improves the quality of the model and shows the best result at dimensions values from $2^4 = 16$ to $2^5 = 32$.

An increasing trend was awaited for the *walk_length* and *num_walks* parameters. Though in both cases, chart lines rose to the particular value and start to fell after

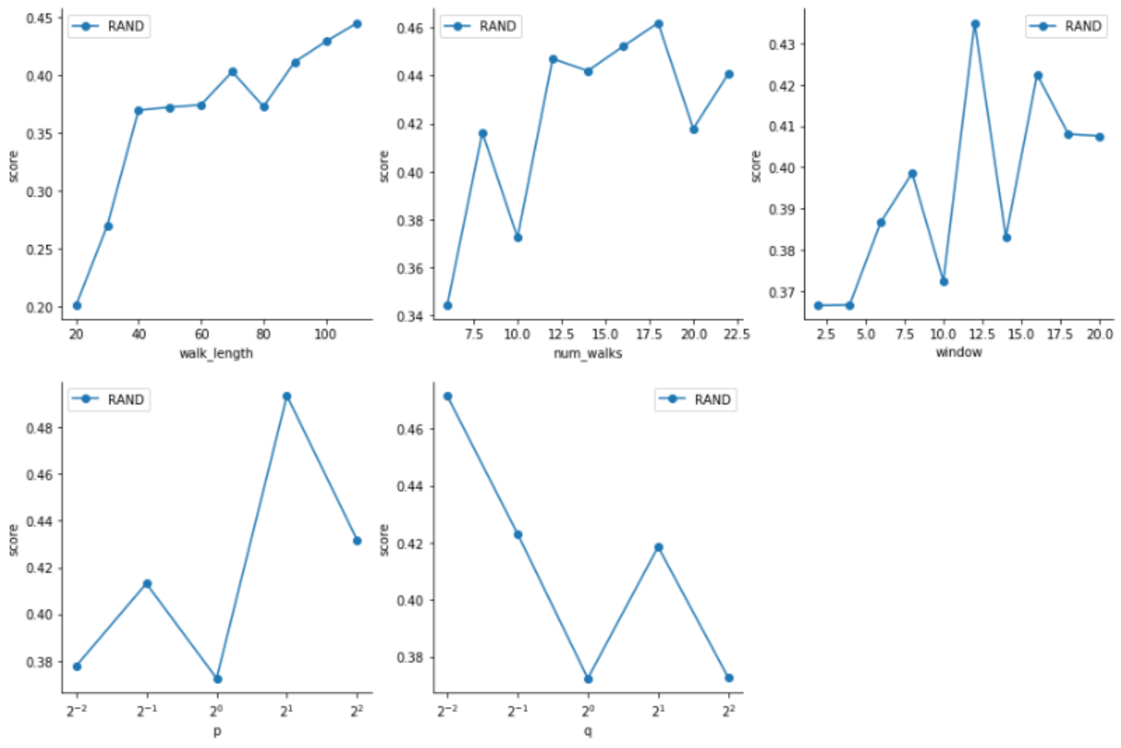

Figure 4.4: The parameters sensitivity for the emails data set. Default parameters values beside of the studying one are used.

reaching the peak. Thus, it can be assumed that the best values of these parameters are 60 and 6, respectively.

Chart lines for p and q parameters are fluctuating, so that could mean that for this particular data changes of these parameters do not have much impact on the results. Indeed, the data is designed in such a way that it is difficult to claim that the search must be done locally or more globally. People send emails within and outside their departments. Probably, if the data also contained the information of the frequency of the emails send, it would be easier to classify nodes representing members into departments.

The window parameter's chart is also fickle. The good results are reached at window equals two and eight. What is worth noticing is that at the recommended value of ten, the model dropped and has its local minimum. This fact once again proves that for each data packet its combination of parameters should be selected.

4.5 Optimisation by heuristics

Table 4.1 shows the five best Rand index scores obtained during the parameters sensibility examining. The best value equals to 0.49. Clearly, with the help of heuristics the improvement of the result was managed (see Tab. 4.2).

walk_length	**num_walks**	**p**	**q**	**window**	**adj_rand**
80.0	10.0	2.00	1.00	10.00	0.49
80.0	10.0	1.00	0.25	10.00	0.47
80.0	10.0	1.00	1.00	10.00	0.46
80.0	10.0	1.00	1.00	10.00	0.45
80.0	10.0	1.00	1.00	10.00	0.44

Table 4.1: The best scores achieved during the parameters sensibility experiment on emails data.

Heuristic	**hmax**	**best_p**	**best_y**
SG_1	1.0	1.72	0.75
SG_inf	*inf*	1.83	0.75
SG_10	10.0	3.48	0.75
SG_1	1.0	1.85	0.75
SG_5	5.0	1.78	0.74

Table 4.2: The best scores achieved by Shoot and Go on emails data.

Conclusions

In this paper, node2vec, an algorithm for creating embeddings, and the use of heuristics for its optimisation were considered. In its implementation, it was essential to ensure the reproducibility of the results. The implementation of the algorithm in the StellarGraph library coped with this.

The optimal value of the p parameter was searched utilising heuristic algorithms. The experiments have shown that the use of heuristics makes sense in this task. The Shoot and go algorithm was recognised as the fastest and most reliable.

Definitely, the *node2vec* algorithm deserves attention, but a review of its implementation may be required. It would be beneficial to work on optimising the process of calculations, data storage and reproduction of results. Improving and unifying existing implementations could be the goal of subsequent publications and Master's thesis, or both.

In this work, an acquaintance with the problems was carried out. Three types of heuristics have been tested. More in-depth study and testing of other heuristics could be exciting and may reveal their potential even more.

The corresponding notebooks and source code can be found at github. [5]

Acronyms

BFS breadth first search. 24

breadth-first is an algorithm for traversing tree or graph data structures moving in depth prior. 24

CBOW Continuous Bag of Words. 15

CBOW is a model architecture that predicts the current word based on its surrounding context. 15

centroids are centres of future clusters. 23

classification is a attempt to differentiate nodes and categorise them into groups. 22

corpus is a collection of texts on which the model is trained. 14

cosine similarity is a binary measure of similarity, which shows a similarity between two non-zero vectors of inner product space and measures the cosine of the angle between them. 14

DeepWalk is an approach for learning latent representations of vertices in a network, which uses unbiased random walks. 17

depth-first is an algorithm for traversing tree or graph data structures explores as far as possible along each branch. 24

DFS depth first search. 24

dimensions is a number of dimensions of the embeddings collected from the algorithm. 23

Feedforward neural network is a deep learning model, that do not contain cycled links between neurons. In this network, the information flows in one direction only, from the input neurons, through the hidden layer to the output neurons. 15

Force Atlas2 is a "force-based" algorithms of data visualisation, where connected nodes attract one another, and disconnected nodes are pushed apart. 42

Genetic algorithm is a mathematical optimisation technique based Darwin's principle of natural selection. 26

Gephi is an open-source application for visualisation and exploration of various kinds of graphs and networks. 41

graph is a mathematical structure constructed from a set of objects having some relations between each other. The objects are called vertices (or nodes), and each relation of vertices is denoted as an edge (or connection). 16

graph embedding is the transformation of graphs to a vector or a set of vectors. 16

heuristics are a problem-solving method that produces good-enough solutions given a limited time frame. It is mainly used working with complex data. 7, 22, 25

Hill climbing is a mathematical optimisation technique that belongs to the family of local search algorithms. 25

Huffman tree is a full binary tree representing a given alphabet, where each letter is corresponded by a leaf. 14

k-means clustering is a clusterisation method based on k-means principle. 23

min_count is a minimal frequency of the node's occurrence in the training model. 25

Negative Sampling is a model used in word2vec algorithm, where weights for only a small number of random words are updated. 15

network (same as a graph). 41

neural network training is the process of searching the set of weights, which will lead to the desired output. 13

neural network is a simplified model of the biological neural system, which consists of interacting artificial neurons. Similar to biological neurons, artificial ones receive information, process it in some way and transmit it to other neurons. 11

NLP Natural Language Processing. 13

node2vec is an algorithm representing graphs in the form of vectors. 17, 23

num_walks is a number of walks made from each node. 24, 43

p is a return parameter that specifies the probability of the moving back to the already visited node. 17, 18, 24, 44

q is a neighbour parameter defining the probability of visiting the earlier unvisited nodes. 17, 18, 24, 44

Rand index is a index that collates an array of cluster numbers with a testing array of pre-defined labels. 23

random walk is a technique of a graph sampling. 24

Simulated annealing is a mathematical optimisation technique based on substances crystallization process increasing the uniformity of metals. 7, 25

Skip-gram is a model architecture that uses the current word to predict the words surrounding it. 15

SNAP Stanford Network Analysis Project. 20

Softmax is a function that takes as input a vector of K real numbers and normalises it into a probability distribution consisting of K probabilities proportional to the exponentials of the input numbers. 15

sparse matrix is a matrix with predominantly zero elements. 20

walk_length is a number of nodes in each random walk. 24, 43

window is a parameter of the word2vec algorithm that sets the maximum distance between the current and the predicted word in the sentence. 15, 24, 44

word2vec is an algorithm that represents words in the form of vectors, so-called word embeddings. 24

word embeddings is a set of language modelling and feature learning techniques in Natural Language Processing, where vectors of real numbers map words or phrases. 13, 15

Bibliography

[1] Hussein Abbass, Ruhul Sarker, and Charles Newton. Data mining: A heuristic approach, 2002.

[2] CSIRO's Data61. Stellargraph machine learning library, 2018.

[3] Eliorc. Python3 implementation of the node2vec algorithm. `https://github.com/eliorc/node2vec`, 2018.

[4] Ahmed Gad. Genetic algorithm implementation in python, 2018.

[5] Adeliia Gataullina. Notebooks for the bachelor thesis. `https://github.com/adeliia/BP`, 2020.

[6] Primož Godec. Graph embeddings — the summary, 2018.

[7] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. 2016.

[8] Dhruvil Karani. Introduction to word embedding and word2vec, 2018.

[9] Jure Leskovec. Snap: Network datasets - email-eu-core network, 2018.

[10] Jacomy M, Venturini T, Heymann S, and Bastian M. Forceatlas2, a continuous graph layout algorithm for handy network visualisation designed for the gephi software. *PLoS ONE*, 9(6), 2014.

[11] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.

[12] Matej Mojzeš. Heuristics 2020, course at fnspe, ctu. `https://github.com/matejmojzes/18heur-2020`, 2020.

[13] Matt Ranger. 700x faster node2vec models: fastest random walks on a graph, 2019.

[14] Matt Ranger. Nodevectors - node2vec with usage of csr graph representations, 2019.