Sofia University
"St. Kliment Ohridski"

Faculty of
Mathematics and Informatics

Neural Networks for Facility Location Problems

Vladislav Haralampiev

A thesis presented for the partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Professional field: 4.6 Informatics and Computer Science
Doctoral program: "Computer Science" — Algorithms and Complexity

Scientific supervisor:
Assoc. Prof. PhD Minko Markov

Sofia, 2021

Neural Networks for Facility Location Problems

Vladislav Haralampiev

Abstract

Optimization problems from science and engineering are often modeled mathematically as combinatorial optimization problems. The result is usually an NP-hard problem and this is why heuristic methods are often used to find in reasonable time a good solution.

A wide range of algorithms based on different phenomena have been developed for approximately solving combinatorial optimization problems. Among them, we can find a class of neural network methods. Such methods were studied at least from the 1980s, but they never became a popular choice for solving optimization problems. And there are good reasons for this. Two main types of neural networks for combinatorial optimization exist: Hopfield networks and self-organizing approaches (in our opinion, it is better to call them "template approaches"). Hopfield networks return solutions of poor quality. Furthermore, they have parameters for which it is tricky to find good values. The self-organizing approaches to combinatorial optimization return good solutions, but they can be applied to a very limited set of problems (in the literature, this is almost always the Traveling Salesman Problem). It is clear that with such drawbacks, the existing neural network approaches can not become popular.

This work proposes a new neural network approach to combinatorial optimization. The method is called competition-based neural networks (CBNNs) and can be considered to be a combination of Hopfield networks and the self-organizing approaches. As is often done for neural networks, we can also consider an analogy between the proposed method and the "real" world. Instead of the usual brain analogy, CBNNs can be compared to the economy of an imaginary environment. In this environment, there is a set of competing companies that are divided into

segments. Three rules drive the evolution of the system: the strongest survives, the lucky ones survive, and as time progresses, the total amount of luck in the system decreases. This is an extremely simplified model of our real economy, but the neural networks from machine learning are also an extremely simplified model of the human brain. For solving a combinatorial optimization problem, we can set the economy of this imaginary world of companies to be equal to the objective function of the problem and let the system evolve according to its rules. If we believe that our real economy is tuned to solve the problems that arise in practice, then the imaginary system of companies should also maximize the economy of its imaginary world and, as a result, maximize the value of the objective function of the modeled combinatorial optimization problem.

Of course, there are more concrete reasons to say that competition-based neural networks are able to maximize functions. It can be proven that CBNNs asymptotically converge to an optimal solution of the modeled combinatorial optimization problem. We additionally discuss a way of computing the speed of convergence. The theoretical guarantees provided by CBNNs are either analogous or are stronger than the guarantees of the other well-known metaheuristics for combinatorial optimization.

Asymptotic convergence is a good thing to have but, in practice, we never want to wait that long. To estimate the practical performance of CBNNs, the model is applied to 6 different facility location problems. The field of facility location was chosen because it offers a wide range problems that are at the same time natural and easy to state but are hard to solve. In fact, we originally developed competition-based neural networks for solving p-MiniSum, a classical facility location problem. The results of the neural network approach are very promising. The method is often able to optimally solve the input instance. When the returned solution is not optimal, the difference, even in the worst case, is just several percent.

For historical reasons, there is a lot of skepticism about the field of neural networks for combinatorial optimization. When Hopfield networks were proposed in the 1980s, this generated a lot of enthusiasm because the model is a way of obtain-

ing a meaningful digital result from analog computations. People then started to evaluate the potential of Hopfield networks and soon found out that the model has many problems. This, in turn, generated pessimistic views about the potential of neural networks for solving combinatorial optimization problems and created a "bad name" for the field. We do not want to say that Hopfield networks were bad. In fact, we believe that, as a model, they were an excellent initial step. But people expected too much from them. The goal of this work is to demonstrate that the idea of using massively parallel systems of simple computing units (that is, neural networks) can result in good algorithms for combinatorial optimization and should be considered along with the other metaheuristics. The proposed competition-based neural networks are surely not perfect and may never become an established high quality metaheuristic for solving combinatorial optimization problems. But, at least for facility location, they demonstrate excellent results and can serve as an example of a general neural network algorithm that is very competitive to the established methods of solving combinatorial optimization problems.

Acknowledgements

A lot of people helped me during my study. I want to express my sincere thanks to Minko Markov, Georgi Georgiev and Dobromir Kralchev for their advice and support throughout my PhD program. I am also very grateful to all the people from Sofia University for making my study a pleasant and hassle-free experience, both from scientific and administrative sides.

The Internet is an excellent source of information. It is not exactly clear who to thank for the existence of such a tool because it is the result of the collaborative effort of millions of people. But I was very lucky to do my research in a time when there is an easy and open access to information.

Finally, I want to thank my parents for their support!

Declaration

I declare that this thesis is the result of my own original work that was done during my PhD study at Sofia University between years 2017 and 2021.

Contents

Chapter 1

Introduction

This work describes a neural network approach to combinatorial optimization. More specifically, the approach is designed for and tested on facility location problems. We have chosen this class of problems because they are intuitive, easy to state, and hard to solve. While the approach was designed for facility location, it can be viewed as a general metaheuristic for combinatorial optimization and applied to many other problem classes.

The first chapter of the thesis introduces combinatorial optimization and facility location problems. It also discusses possible ways of tackling a combinatorial optimization problem. Chapter 2 continues the discussion by concentrating on well-known metaheuristics for combinatorial optimization, including neural network methods. Chapter 3 introduces a new neural metaheuristic that we call a competition-based neural network. The theoretical properties of the proposed method are investigated in Chapter 4, where we show that it is guaranteed to asymptotically reach an optimal solution of the modeled problem. Chapter 5 investigates the empirical properties of competition-based neural networks by applying the approach to six classical facility location problems.

1.1 Preliminaries on algorithms and complexity.

This section is a brief introduction to polynomial-time solvability and NP-hardness. Complexity theory is large and is not the topic of this work, so we refer to [4, 36]

for more information.

Informally, a computational problem is an infinite set of input instances together with a description of what needs to be output for each instance. For example, the input may be a map of a city and two points A and B on the map. The problem may ask us to output a solution that is an optimal path from A to B. What is meant by optimal is specified by the problem: maybe a shortest path, or a fastest path, or something else. An algorithm is a fixed finite set of instructions. The set of instructions can be executed on a given input of a problem to produce in a finite number of steps an output. We can think of this as a program in C or any other programming language. The program reads its input from a given file and outputs a solution to another file.

The thesis deals with algorithms that return good but suboptimal solutions. It may not be immediately clear why one would agree to accept a suboptimal solution instead of an optimal one. The reason for this is that some problems are hard and for them finding the exact optimal solution becomes too expensive computationally. In such situations, quickly finding a good suboptimal solution is a valuable result.

An important property of every algorithm is its time complexity: the number of steps it performs as a function of the input size. An algorithm is polynomial-time if there is a constant c such that for every instance of size s, the number of steps the algorithm performs on the instance is bounded by s^c. The constant c is the same for all instances. There are several subtleties with the notion of polynomial-time algorithms. First, the number of steps depends on the computational model. Intuitively, different models allow different sets of instructions. A model X may have more powerful instructions than model Y and simulating them in model Y may require a lot of steps. Second, the size of an instance depends on how it is encoded. As a simple example, we can compress the input and often this will reduce its size. It turns out that for the reasonable computational models and input encodings, the set of problems that are solvable by polynomial-time algorithms is the same [58]. The exact number of steps may be different, but if there is a polynomial algorithm for a problem in one of the models, then there is a polynomial algorithm

in any other model. This allows us to omit the details of the input encoding and the computational model.

Defining what is a hard problem is not straightforward. A major result in computer science from the 1960s is the conjecture that there are problems of such complexity that no polynomial-time algorithm exists for them. This conjecture is still not proven, but it forms the basis of the currently accepted distinction between easy and hard problems. Easy (tractable) problems are the ones for which there is a polynomial algorithm. Hard (intractable) problems are the ones for which there is no such algorithm.

When investigating the complexity of problems, it is convenient to deal with decision problems: ones that have a single bit of output. They are also sometimes called YES / NO problems because their result is often interpreted as YES or NO. For example, does a given graph contain a Hamiltonian cycle, YES or NO? The complexity class P consists of decision problems for which there is a polynomial algorithm. These problems are considered to be easy. The class NP consists of decision problems for which the instances with answer YES have a short "certificate" of correctness that can be verified in polynomial time. To say in another way, there is a polynomial-time algorithm that, when given an instance and its certificate, verifies that the answer to the instance is YES. Consider the problem of detecting a Hamiltonian cycle. Its input is a graph. A certificate can be the Hamiltonian cycle itself (certificates are meaningful only for instances with answer YES). It is easy to write a polynomial algorithm that, given a graph and a cycle, verifies that the cycle is a Hamiltonian cycle in the graph. This way, the problem of deciding if a graph has a Hamiltonian cycle is in the NP class. Notice that the idea of a certificate is related to the object that we try to detect in the decision problem. Often, the certificate is the object itself, but this is not a requirement. Certificates are not unique and, as long as they are short enough, they can be an arbitrary useful thing.

The class NP is large. It is easy to show that P is a subset of NP and our current understanding is that P is a proper subset. We want to stress that this is a conjecture. While it is widely believed to be true, the result is still not proven.

Remember that the class P consists of the easy problems. Intuitively, if we remove these from NP, then we are left with hard problems. As a side note, there is a whole hierarchy of complexity classes and NP is not the end of the hierarchy [4].

We say that a problem is NP-hard if every problem in NP can be reduced to it in polynomial time. There are well-established techniques for creating NP-hardness proofs [26] and such proofs are generally not very complicated. At least, it is relatively easy to understand the proof once somebody invented it.

NP-hard problems appear very often in practice. Hundreds examples of such problems can be found in [36, 55]. NP-hardness implies that the given problem is at least as hard as any problem in NP. If we believe that there are problems in NP for which no polynomial-time algorithm exists, then no polynomial-time algorithm exists for NP-hard problems.

1.2 Combinatorial optimization. Facility location problems.

Combinatorial **O**ptimization **P**roblems (COP) ask for an optimal object among a finite set of objects [88]. The set of objects is sometimes allowed to be countably infinite, but this work only deals with finite sets. Classical examples of COPs are the Travelling Salesman Problem, the Maximum Matching Problem and the Minimum Spanning Tree Problem. The formal definition of a COP is given below and it follows the definition from [81].

Definition 1.2.1. (Combinatorial optimization problem) A combinatorial optimization problem is specified by its instances. An instance is a pair (F, obj). F is called the set of feasible solutions and can be any finite set. obj, the objective function, is a mapping from F to the real numbers. In case of a minimization problem, the goal of a COP is to find a point $s \in F$ for which $obj(s) \leq obj(x)$ for any point $x \in F$. If the problem is a maximization problem, then the condition $obj(s) \leq obj(x)$ is substituted by $obj(s) \geq obj(x)$.

There are two varieties of COPs: for minimization and for maximization. By

inverting the sign of the objective function, a maximization problem becomes a minimization one. This is why, when dealing with algorithms for combinatorial optimization, it is enough to consider only minimization problems.

The main topic of the thesis is a new neural network approach to combinatorial optimization. The approach is evaluated on (and actually designed for) a subset of all COPs that is called facility location problems. These problems form a large class of practically important tasks that are at the same time intuitive and easy to state but are difficult to solve. Facility location problems were first formally introduced by Alfred Weber in 1909 [104]. He was interested in finding an optimal location for a warehouse to minimize the total travel distance between it and a set of customers. Up to the 1960s, the work on location theory consisted of many separate applications that were not tied together by a unified framework. In the 1970s, facility location problems attracted more theoretical interest and classes of common location problems were identified. Solution methods were developed together with results on the computational complexity of the problems.

The unifying idea behind facility location problems is that there are resources that need to be spatially allocated. In general, we have to choose the location of one or more facilities to serve a set of demands. The demands are distributed in space. From then on, there are many varieties. The spatial topology can be, for example, the Euclidean plane or a graph network. The objective function can minimize the sum of distances, or minimize the maximum distance, or something else. The number of facilities to locate may or may not be fixed beforehand. The facilities may interact with each other, there may be competitors, and so on. To get an intuition about the type of problems being solved, we list below several classical facility location problems:

- **p-Hub Problem.** A mail delivery company needs to choose the locations of p processing centers (hubs). Every customer is assigned to a single hub. When customer A sends mail to B, the mail travels through the path $A \to HA \to HB \to B$ (here HA and HB are the hubs of A and B). The reason for such indirect deliveries is economy of scale — the hubs accumulate mail

from all their customers and a single truck delivers all the combined traffic. The expected amount of mail between every pair of customers is known beforehand. The goal of the problem is to route all traffic as cheaply as possible by optimally selecting the locations of the hubs. More information about the p-Hub Problem together with the results of the competition-based neural network solver on a set of instances can be found in Section 5.2.

- **p-DefenseSum Problem.** There are p military facilities that need to be located in a given area. The area may be captured by the enemy and we do not want him to be able to establish stable control over the facilities. To establish stable control, the enemy needs to build the necessary infrastructure. This becomes costly if the facilities are far apart from each other. The goal of the problem is to place the p facilities in such a way that the sum of distances between all pairs of facilities is maximized. In Section 5.3 the competition-based neural network solver is applied to a set of p-DefenseSum instances.

- **MaximalCovering.** A company needs to locate p cell phone towers. The goal is to cover as many populated places as possible. An application of competition-based neural networks to a set of MaximalCovering instances is given in Section 5.4.

- **MaxiSum.** People in a given region produce waste that needs to be recycled somewhere. But people do not want to live near such a recycling facility. This is why it needs to be placed in such a way that the sum of distances from the houses to the recycling facility is maximized.

- **p-Center.** We need to choose the locations of p centers for handling medical emergencies. In such situations, the maximal time of response to an emergency is crucial. This is why we want the maximal distance from a location to the emergency center that services it to be as small as possible.

This work does not aim to provide a detailed review of facility location problems, so we will not spend more time presenting the rich variety of such problems. A

taxonomy and representative tasks from location research can be found in [10, 16, 27]. In Chapter 5, the neural network approach that is developed in this work is applied to six classical facility location problems. There we give more details about the problems, their known variants, and solution methods. Here we just want to mention three points about facility location that we believe are important.

First, facility location problems are not always discrete. For example, the space where we locate facilities can be the Euclidean plane, or we may be allowed to place facilities along the edges of a geometric graph. Continuous problems are more difficult to solve. There are some results for such facility location problems. As an example, there is a method for the Weber Problem [12]. But generally people try to avoid dealing with continuous spaces. The way to avoid them is to either prove that the set of optimal solutions can be found among a hopefully small finite subset of all possible locations, or to discretize / bucketize the solution space and get an approximation of the optimal solution. In the literature, the details of the discretization methods are often omitted and it is stated in the constraints of the problem that facilities can be placed only on a given finite set of locations. In the thesis we do the same and from now on it is assumed that facility location problems are discrete.

The second point is that facility location problems are almost always NP-hard. We know of only one nontrivial facility location problem on a general graph that is not NP-hard, the Discrete Anti-P-Center Problem [62]. As discussed earlier, it may happen that there is an efficient algorithm for NP-hard problems. Researchers have been working for decades on such problems and no one was able to find an efficient polynomial algorithm for even a single one of them. This does not mean that such an algorithm does not exist. But, if our goal is just to solve a single problem and we are not ready to invest a huge amount of effort, then it is reasonable to take as an axiom that no polynomial algorithm exists that optimally solves NP-hard problems. This gives a hint about the type of methods that are necessary when dealing with facility location problems (see the next section).

The last point is that facility location problems often have two different param-

eters: the number of facilities to locate and the size of the network / number of clients. The complexity of facility location problems grows more quickly with the number of facilities than with the size of the network. If the number of facilities is a fixed constant, then facility location problems are usually not NP-hard because iterating through all possible solutions runs in polynomial time. This is good news because the number of facilities to locate is often much smaller than the size of the network. The bad news is that, even for relatively small number of facilities, the problem seems to inherit the "source of difficulty" of the general NP-hard problem. The number of all possible solutions may by polynomial (something like n^{20}), but the time necessary to iterate through all of them is huge and it is not clear how to improve such a "polynomial" algorithm.

When solving a real-world combinatorial optimization problem, it is important to have a mathematical formulation of it. A strict mathematical formulation ensures that we understand correctly what we are asked to do. The problems that appear in practice are usually stated in human language. Their description is vague and missing important details[1]. Someone needs to understand well the essence of the problem and to create a simplified abstract model of it that only keeps the important parts. This requires expert knowledge and the skills that are necessary for creating mathematical formulations seem to be relatively separate from the skills that are necessary for actually performing the optimization. The thesis deals with problems from the point of having a strict mathematical definition.

Below we give the definition of p-MiniSum, one of the best-known facility location problems. This problem is used as a working example throughout the thesis. In human language, it sounds like this: find the optimal locations of p warehouses to serve a set of demands. In the graph variant of the problem, the demands are populated places that are connected by roads. The road network is represented by a weighted undirected graph and the facilities can only be placed in the vertices of the graph.

[1] In [93] the author describes several situations in which he needed to solve problems in practice. It is interesting to read about his experience.

Definition 1.2.2. (p-MiniSum Problem)

- **Input**

  1. A weighted, undirected and connected graph $G(V, E)$ with vertex set V and edge set E. The edges have positive lengths and the shortest distance in G between any pair of vertices $u, w \in V$ is denoted as $dist(u, w)$.
  2. Number p of facilities to place.

- **Solution**

  A subset of p vertices $v_1, ..., v_p$ from V that minimize $\sum_{c \in V} min_{v_i \in v_1 ... v_p} dist(c, v_i)$.

An example p-MiniSum instance is shown in Figure 1.1. Assuming p is part of the input, the problem in general graphs is well-known to be NP-hard (by reduction from the Set Cover Problem [57]). More information about p-MiniSum is given in Section 5.1, where we apply neural networks for solving it.

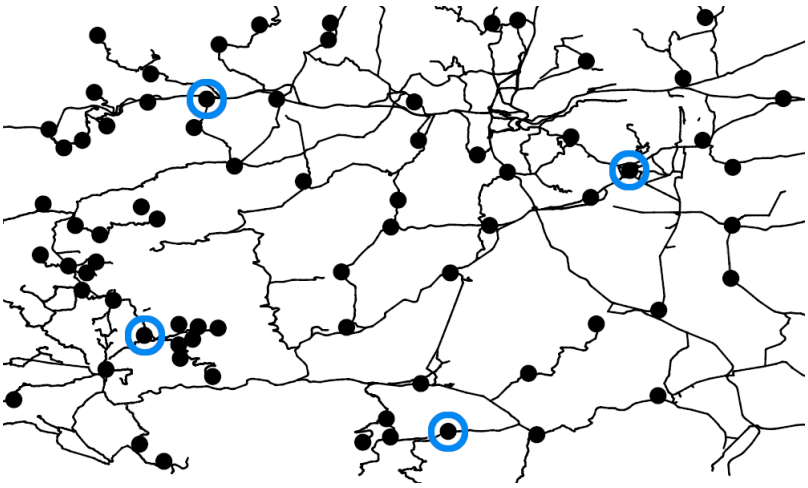

Figure 1.1: Example p-MiniSum instance with 4 warehouse facilities. The dots (vertices) represent populated places and the 4 selected locations are marked with an additional circular outline.

Solution methods for COPs can have specific requirements on how the definition of a problem should look like. For example, Integer Programming [24] is a common way of solving NP-hard problems. It requires the problem to be defined as a set of integer variables that are bound by linear constraints and the function to optimize should also be linear. We call *modeling* the step of converting a general mathematical formulation of a problem to a formulation with specific requirements.

Similarly to Integer Programming, the neural networks from this work also need a modeling step. They require combinatorial optimization problems to be described by a:

- Set of binary variables $\{x_1, ..., x_n\}$.
- Constraints among the variables (see Section 3.1 for the type of constraints).
- Objective function to minimize or maximize.

The *solution space* of the problem consists of all possible ways of assigning values to the variables (2^n ways for n variables). Each assignment is called a point. Some points may not satisfy the constraints of the problem and thus do not represent feasible solutions. The subspace of the solution space that consists of all points that satisfy all constraints of the problem is called the feasible region. This is the same as the set F of feasible solutions from Definition 1.2.1 of a COP. Of course, we seek for an optimum among the set of feasible solutions.

1.3 Solution techniques

The section discusses some possible ways of solving optimization problems. The goal is to see the place of neural networks among the available methods for combinatorial optimization.

Optimization is a problem of *navigating* in the solution space and locating a point with desirable properties. The problems we consider have a finite set of feasible solutions. One way of finding the optimal answer for them is to iterate through every feasible configuration and pick the one that is best. Backtracking is a well-known way of performing such an iteration through the solutions space [93]. The method is guaranteed to optimally solve the problem, but it can consume a huge amount of resources even for relatively small instances.

For most problems, the size of the solution space is at least exponential in the size of the input instance. As discussed in [93], modern processors have a clock rate of several gigahertz, which generally means that we can iterate through several

million solutions in a second, not much more. Several million is roughly the number of permutations of 10 items, or the number of subsets of 22 objects. This gives an idea about the limits of backtracking.

Backtracking demonstrates probably the simplest idea of how to perform the search for an optimum: blindly go through every feasible solution while making sure that you do not visit something twice and do not skip something. Developing better algorithms is related to finding regularities in the solution space. The regularities allow us to efficiently navigate and discover a "path" to the optimum without considering most of the feasible solutions.

A special case is if the problem we are working on can be solved in polynomial time. Intuitively, there is a polynomial algorithm only if the solution space is highly structured. For example, in [88] it is noted that the existence in the solution space of polyhedral characterizations and min-max relations almost always means that a polynomial algorithm can be developed. When solving a COP, it is usually true that either a polynomial-time algorithm exists for it, or the problem is NP-hard [88]. The thesis deals with the second case that seems to be more common. From now on it is assumed that the problem we have is NP-hard and no polynomial algorithm exists for it. As a side note, problems that can be solved in polynomial time are also important and very interesting. The methods for solving them tend to be quite different from the methods for NP-hard problems [26].

The strategy for tackling an NP-hard problem depends on the trade-offs that are acceptable. Assume that an exact optimal solution is absolutely required. We know that no polynomial algorithm exists for the problem. Very intuitively (and not completely correct), this says that an exact algorithm for the problem can not do something "smart" and is like an iteration through a large number of candidate solutions. A very large fraction of the considered configurations are "useless" in the sense that they are neither optimal nor on a "path" to an optimal solution. A good algorithm will try to reduce the amount of "useless" configurations that are considered. The two techniques below implement this idea.

- *Pruning.*

  A backtracking-like algorithm corresponds to a tree in the solution space. The root of the tree contains the set of all possible solutions. The leaves are singleton sets containing individual solutions. Each internal node corresponds to fixing some part of the solution and this creates different branches in the search tree. If at some point the algorithm detects that a whole branch can not possibly contain an optimal solution, then it does not make sense to consider this branch and the procedure can ignore it. This is the general idea of pruning that can be implemented in different ways. For example, one classical design paradigm is called Branch-And-Bound [23]. It uses various bounds to decide that some branches can be discarded because they can not possibly contain a solution that is better than the best solution found so far. Developing good bounds is a hard problem-specific task, so this is not a low-investment way of solving a problem. But it often allows solving instances of moderate size.

- *Instance-specific algorithms.*

  An instance of a problem may have properties that simplify it. For example, there may be symmetries. The input graph may be planar. The maximum degree of a vertex may be small. We can develop a program specifically for solving this single instance. The result is not exactly an algorithm because it does not have an input that changes. But, since we can make use of simplifying properties that do not hold for the general problem, the solution procedure may be substantially faster. The above idea can be generalized to classes of input instances that share the same simplifying properties. In such situations, it may be cleaner to define a special case of the general problem and solve this special case (it can even happen that the special case of the problem is solvable in polynomial time).

If a good suboptimal solution is acceptable or the instance is just too large for exact methods, then we switch to choosing an algorithm that returns approximately optimal solutions. There are two main groups of such methods: approximation algorithms and heuristics.

The first group consists of polynomial-time approximation algorithms with provable approximation bounds. This means that for such methods it can be proven that, for example, the returned solution is at most two times worse than the optimal one. An introduction to approximation algorithms can be found in [100]. The development of such procedures resembles more the development of polynomial-time algorithms.

From the perspective of facility location, up to 1997 no approximation algorithms were described in the literature for this class of problems [68]. Today, approximation algorithms are known for variants of several location theory problems like Uncapacitated Facility Location and p-Median. The main ideas of some of these approaches are outlined in [68]. Approximation algorithms are interesting from theoretical perspective but, from our personal experience, they are not very flexible: if the problem slightly changes, then the algorithm is no longer applicable. Additionally, their guarantees on the solution quality are often not strong enough. As an example, an approximation algorithm may be guaranteed to return a solution that is at most two times worse than the optimal one. In reality, other methods may get us much closer to the optimum than this (in the experiments in Chapter 5, the neural network finds solutions that are at most several percent worse than the optimum).

The other group of methods for finding approximately optimal solutions consists of heuristic approaches. These methods do not provide a guarantee of the quality of the returned solution. Intuitively, here is how the justification of a heuristic algorithm sounds like: we tried it on reasonably similar data and it worked well, so it should work well. This is a bit exaggerated but gives an idea about the drawback of heuristic approaches. Despite the described drawback, heuristics established themselves as a good solution technique and are an important "weapon" in the arsenal of any practical algorist [93].

Common heuristic algorithms are either constructive methods or are based on local search [8]. Constructive algorithms start from an empty partial solution and repeatedly add components until a complete solution is created. They are often

the fastest methods for finding an approximate solution, but the quality is typically inferior to other approaches. Local search starts from a random initial configuration and repeatedly tries to improve it by making small modifications. This turns out to be a very useful idea for solving combinatorial optimization problems and is discussed in more details in Section 2.1.

We need to be careful when developing heuristic algorithms. It is easy to come up with heuristic ideas that look reasonable at first sight but result in terrible solution quality. This is why it is good to base the algorithm on a framework that has proven itself. Metaheuristics are such frameworks. They are not complete algorithms but guidelines on how to develop an algorithm. The metaheuristic provides the skeleton of the search procedure and determines how the procedure navigates towards a good configuration in the solution space. Problem-specific work still needs to be done to fill in the blanks in the procedure. Metaheuristics are discussed in more details in the next chapter.

The neural network approaches to combinatorial optimization fall into the category of metaheuristics. From the above discussion it can be seen that there are methods that are easier to develop but are much slower, like backtracking. There are methods that are faster and return better solutions, like instance-specific algorithms, but their development is much more time-consuming and expensive. Metaheuristics target a sweet spot between returning an optimal result and being fast to execute and cheap to design. They are relatively quick, relatively easy to develop, and relatively reliably return a relatively good solution.

1.4 Contributions of the thesis

The thesis investigates the potential of neural networks for solving combinatorial optimization problems. It introduces a new neural network metaheuristic for combinatorial optimization that we call competition-based neural networks. The metaheuristic is described in the context of facility location problems. The proposed approach is guaranteed to asymptotically reach an optimal solution of the modeled problem and we show how to estimate the speed of convergence. The thesis also de-

scribes the application of competition-based neural networks to six practical facility location problems.

In summary, the main contributions of the thesis are:

- Analysis of the existing neural networks for combinatorial optimization. Insights into the reasons for the poor performance of these methods on facility location problems.
- Introduction of competition-based neural networks, a new neural network metaheuristic for combinatorial optimization, together with asymptotic convergence results and speed of convergence results.
- Selection of six practical facility location problems for evaluating the empirical performance of the introduced neural network method. Generation of more than 500 realistic input instances of the chosen problems, together with the optimal solutions of the instances.
- Demonstration of how to model the six selected facility location problems for solving with competition-based neural networks.
- Extensive empirical evaluation of the proposed neural network metaheuristic on the selected set of facility location problems.

The idea to use neural networks for combinatorial optimization is not something new, it existed at least from the 1980s [48]. But neural networks never became a popular tool for solving combinatorial optimization problems and there is a negative opinion in the community about the entire concept [75, 82, 106]. Chapter 2 of the thesis studies the existing neural networks for combinatorial optimization. We agree that the existing approaches have problems and most probably will never be competitive to other metaheuristics. But we do not agree that the whole concept of neural networks for combinatorial optimization is flawed. The proposed competition-based neural networks can serve as an argument because they are a neural network method that, at least for facility location problems, is very competitive to other metaheuristics, both as theoretical guarantees and as empirical performance. We consider this proof-of-a-concept to be the most important contribution of the thesis.

Chapter 2

Metaheuristics for combinatorial optimization

The term metaheuristic was introduced by Glover in 1986 [37]. As the suffix *meta* suggests, this concept is at a higher level of abstraction than the heuristic methods for optimization. The term metaheuristic still does not seem to have a single universally accepted definition. Several definitions from different researchers are quoted in [8]. The common idea is that metaheuristics are frameworks for developing algorithms. The resulting algorithms are expected to return suboptimal but sufficiently good solutions to optimization problems. As was mentioned at the end of the previous chapter, in the field of combinatorial optimization metaheuristics are positioned as a low-investment way of getting solutions of good quality. This is often exactly what is needed in practice when solving a combinatorial optimization problem, so metaheuristics have become an important tool.

The chapter starts by introducing local search, an idea that is very useful for quickly finding good solutions. We then discuss several established metaheuristics for solving optimization problems. The chapter ends with an overview of the existing neural network methods for combinatorial optimization.

2.1 Local search

Local search is an idea that can be found in many metaheuristics. It is not usually considered to be a separate metaheuristic. Hill climbing is a closely related optimization technique and the two terms are sometimes used interchangeably [93]. Local search starts from some solution and repeatedly performs small modifications that improve the quality of the solution. The procedure stops when no improvement can be made with such modifications. This type of movement in the solution space seems to be important and beneficial for quickly finding good answers to optimization problems that occur in practice.

Local search uses the notion of neighborhood structures in the solution space.

Definition 2.1.1. (neighborhood) Let S be the solution space. The neighborhood function $next : S \to 2^S$ assigns to every point $s \in S$ a set of neighbors $next(s) \in S$.

Definition 2.1.2. (local minimum) Let S be the solution space and f be the objective function of a given problem. The point $s \in S$ is a local minimum if for every $s' \in next(s)$ $f(s) \leq f(s')$.

The neighborhood structures capture the intuition of making small modifications to the current solution. By repeatedly moving to a better neighbor of the current configuration, local search eventually reaches a local minimum. This process is shown in Algorithm 1.

Algorithm 1: Local search that minimizes the objective function F.

```
1 sol ← initial random solution
2 while True do
3 |   candidate ← argmin{F(x) | x ∈ next(sol)}
4 |   if F(candidate) < F(sol) then
5 |   |   sol ← candidate
6 |   else
7 |   |   break
8 end
```

Defining a good neighborhood for local search is a problem-specific task and it can significantly affect the quality of the final solution. A bad neighborhood choice can create many poor local minima or plateau regions that artificially increase the

complexity of the problem. The size of the neighborhood is important for the time complexity of local search approaches. Indeed, the neighborhood of a point can be chosen to be the whole solution space. Then local search is guaranteed to find a global minimum of the objective function, but it will be nothing more than an iteration through the set of all possible solutions. Usually very small (constant size) neighborhoods are used.

In the thesis we often say that an algorithm somehow moves in the solution space. This can cause confusion and we want to clarify the terms that are used to describe the movement. The solution space consists of points. What is a point depends on the problem and is often a concrete assignment of values to the variables of the problem. Once a neighborhood structure is defined in the solution space then we can talk about neighbors of a given point. Local search and methods related to it start from some point and repeatedly move to a neighbor. This creates a chain of transitions that is called a trajectory. The movement consists of proposing a candidate that is a neighbor of the current point and transitioning to it if some algorithm-specific condition is satisfied. This step is sometimes called accepting / rejecting the candidate. Local search is a simple method that accepts the new point if the value of the objective function for it is better than the value for the current point.

The quality of the solution that local search finds highly depends on the initial random configuration. Because of this, the method is often modified to increase the likelihood of reaching a good solution. The modification can be problem-specific but usually a well-known framework (a metaheuristic) is used.

2.2 Overview of established metaheuristics

This section gives a brief overview of some established metaheuristics. While the methods are quite different, we can still see some common ideas that apparently work well for a wide range of problems. Neural metaheuristics are omitted in the list below and are discussed in greater detail in the next section.

- **Repeated local search**

  The idea is to repeatedly run local searches starting from different randomly generated points. This is probably the most basic of the trajectory methods (methods that operate on a single solution at a time). Local minima define basins of attraction in the solution space. A basin on attraction is a region of points from which local search reaches the same local minimum. By randomly generating the starting point, we hope to explore enough such regions to obtain a good solution.

- **Simulated annealing (SA)**

  Simulated annealing was introduced in the paper [61]. The method extends local search by allowing it to accept moves that decrease the quality of the current solution. In this way, simulated annealing has an explicit strategy for escaping from local minima. For a point s in the solution space local search always transitions into a new point $s' \in next(s)$ that has a better value of the objective function. If no such point exists, then local search stops. Simulated annealing also selects a point $s' \in next(s)$ as a candidate for next state. Let F denote the objective function that is minimized. As in local search, if $F(s') < F(s)$, then we make s' the new current solution. Otherwise, we accept the worsening transition to s' with probability that exponentially decreases with the difference between $F(s')$ and $F(s)$. There is also a positive parameter T called *temperature* that scales the difference $F(s') - F(s)$. High values of the parameter make high the probability of accepting even very bad transitions. Values of T close to 0 allow only a small deterioration of the value of the objective function and basically transform the algorithm into local search. SA starts from a high temperature and gradually decreases it. In this way, it gradually shifts from exploration to exploitation. The method is based on an analogy with the thermal process of annealing substances in a heat bath to obtain a low energy state. This is contrary to local search that corresponds to quenching (instantaneously lowering the temperature in a heat bath).

- **Tabu search**

  This metaheuristic introduced the idea of memory structures. It takes basic local search and forces it to always move to the best next solution s' in the neighborhood of the current solution s, even if such a step makes the solution worse. This is a type of exploration strategy but it results in short cycles in the solution space and in reality does not perform meaningful exploration. For this reason, tabu search introduces a tabu list that contains the most recently visited points. When taking the neighborhood of the current solution, the points from this list are excluded. If the list is long enough, then it should break the cycling behavior. Currently, tabu search has become much more advanced. The tabu list usually operates on the level of solution components and not only on complete solutions. The length of the list changes dynamically and there are other memory structures that implement long(er)-term memory. Comparing the implementation of recent tabu search to simulated annealing it is generally true that the tabu search code is "messier", it contains a lot of complex logic implementing the memory structures. More information about the tabu search metaheuristic can be found in [38].

- **GRASP**

  GRASP stands for Greedy Randomized Adaptive Search Procedure. Similarly to repeated local search, it consists of multiple local search runs from different starting points. The starting points are not completely random but are generated by a randomized greedy algorithm. This algorithm iteratively adds solution components to a partial solution. The "quality" of each component is evaluated at every step of the greedy algorithm and a random component is chosen to be added among the k best ones where k is a parameter of the method. The hope is that the greedy algorithm samples the most promising regions of the solution space and that they represent different basins of attraction. More information about GRASP can be found in [83].

- **VNS**

  The idea of Variable Neighborhood Search (VNS) is that a local minimum

for one neighborhood may not be a local minimum for another neighborhood. By dynamically changing the neighborhood, we may be able to escape from poor local minima. VNS uses a sequence of neighborhood functions $next_1, next_2, ..., next_k$. Often, $next_i$ is a subset of $next_{i+1}$ in the sense that $next_i(x)$ is a subset of $next_{i+1}(x)$ for all points x. The metaheuristic performs three steps in cycle: shaking, local search, and transition. In the shaking step, a point s' is chosen among $next_i(s)$. Then a classic local search is performed starting from s'. This local search is not related to the neighborhood sequence and may even use a neighborhood not in it. If the local search reaches a solution s'' that is better than s, then s'' becomes the current solution, the neighborhood is reset to $next_1$ and a new iteration starts. Otherwise, $next_i$ is substituted by $next_{i+1}$ and the process is repeated. More information about the VNS metaheuristic can be found in [80].

- **Guided local search**

  Another possible way of escaping from poor local minima is to dynamically change the objective function. This is the main idea of Guided local search. Of course, the objective function can not be a completely random function. It needs to express the goals of the optimization problem being solved. Guided local search takes the "cost" of the solution and adds to it weighted terms that detect the presence of certain solution features. This features are problem-specific and can be any properties that discriminate between solutions. As the algorithm progresses, it changes the weights of the features in the hope of escaping from poor local minima. More information about Guided local search can be found in [102].

- **Genetic algorithms**

  The metaheuristics described so far are called trajectory methods because they operate on a single solution at a time. Genetic algorithms are population methods. Instead of a single solution, they manage a set of solutions. Genetic algorithms define a set of crossover and mutation operations. The crossover operations take two (or more) "parent" solutions and produce new ones by

combining the features of the "parents". The hope is to select the best features from both "parents" and obtain a stronger solution (that is, a solution with a smaller value of the objective function). Mutations operate over a single solution and randomly modify some part of it. This serves to diversify the population and mutations are also a form of self-adaptation (a way to make the solution stronger). Genetic algorithms also perform a step of the type "the strongest survives". Either the weakest solutions (with the worst value of the objective function) are discarded or the probability for weak solutions to become "parents" is substantially decreased. The process that is described above is inspired by the natural evolution of species. Genetic algorithms start from a random population and let it develop according the described rules. The best solution encountered during the process becomes the answer to the optimization problem. More information about Genetic algorithms and their properties can be found in the book [101].

- **Ant colony optimization**

  Ant colony optimization was introduced in [28]. This metaheuristic, similarly to Genetic algorithms, is a population method. Additionally, Ant colony optimization is not local search based but a constructive method (a solution is produced by adding components to an initially empty solution). The metaheuristic is inspired by the way ants find best paths. Each solution component has a dynamic heuristic evaluation of its quality. It also has a pheromone value (pheromone is the substance that ants leave on a path when they walk on it). This pheromone value expresses how often the component was used and how good the obtained solutions, containing this component, were. A combination of the heuristic evaluation of the component and the pheromone value defines the probability of adding it to the solution that the "artificial ant" is building. After all ants in the population complete their solutions, a pheromone update is performed and a new iteration starts. The best solution obtained during this process is taken as the final answer to the optimization problem.

All metaheuristics from the list above either return or are usually modified to

return a local minimum of the objective function. Finding a local minimum is relatively cheap and it is one thing we can expect from a good metaheuristic. The quality of local minima can vary drastically and just returning the first one found is not enough. Because of this, all metaheuristics struggle to escape from poor local minima. This is done either by accepting worsening moves, or by using memory structures, or by modifying neighbourhoods, or by applying some other idea. A mechanism for escaping from poor local minima is an important property of a good metaheuristic.

Another common idea is the use or randomization. As mentioned earlier, developing an exact efficient algorithm is often related to finding useful structures in the solution space. Intuitively, if a metaheuristic is deterministic, then it defines a relatively simple structure in the solution space. If the metaheuristic gives good solutions, then this structure is useful and most probably an exact efficient algorithm can be developed based on it. If we can not find such an efficient algorithm for the considered problem, then we can not find a good deterministic metaheuristic either. Another way to think about this is that during the execution of a metaheuristic there are places where decisions need to be made. If we knew how to correctly make these decisions, then the metaheuristic would have been an exact algorithm. We do not know, so we make decisions at random. Randomization is a way of implicitly dealing with complexities.

Finally, metaheuristics are strategies for traversing parts of the solution space and aim to find a good balance between exploration of new regions and exploitation (improvement) of the most promising solutions found (the exploration-exploitation trade-off dilemma is often discussed in the multi-armed bandit problem and, in general, in the field of reinforcement learning [97]). Just blindly trying new regions is not enough because, for most problems, the solution space is complex, consisting of a huge number of different basins of attraction, and hoping to blindly hit a good region is a dead end. This is why good metaheuristics introduce bias in the exploration. While they still make random moves and not always greedily prefer the locally best option, the metaheuristics show preference towards good "solution

components", either by recombining good solutions to produce new ones, or by substantially decreasing the probability of bad moves, or by using some other idea. This implicit preference allows the metaheuristics to find a good region without fully exploring the solution space. But it also limits the set of problems to which the corresponding method can be successfully applied. If the properties of the problem being solved do not align with the implicit bias of the metaheuristic, then we can not expect the method to magically find a good solution to the problem. To this end, applying a metaheuristic vaguely resembles the "dark art" of tuning very deep neural networks in machine learning where there are many hyperparameters whose relationships are not fully understood but whose values highly affect the performance of the model [71].

2.3 Neural networks for combinatorial optimization.

Artificial neural networks are an attempt to produce machines with capabilities that are similar to nervous systems of animals. It is believed that the essence of natural nervous systems is "control through communication" [13]. These natural systems are massive networks of simple interconnected cells and it seems that from the cooperation of the cells emerges consciousness and complex behavior [51]. While artificial neural networks are inspired by nature, we should be careful when making biological analogies. As noted in [74], humans tend to compare the most advanced machines of their time to the human brain. Historically brains were compared to pneumatic machines (ancient times) and clocks (the Renaissance). More recently they were compared to telegraph networks. Nowadays we can say that the brain is similar to an artificial neural network, but it is not clear if this is a good (or useful) model. Even if they are not a good model of the brain, artificial neural networks emerged as a useful tool for solving many practical problems. In our opinion, the most valuable conclusion from the biological analogy is that massively parallel hierarchical systems of simple computing units are intelligent in the sense

that they are able to solve problems. Here we will not go into the details of the biological model of nervous systems. In recent years much progress has been made in understanding how the brain works. Nowadays we know very well how individual neurons function and we try to understand how ensembles of such neurons produce complex behavior. A good introduction of the biological model of nervous systems can be found in [85].

Artificial neural networks are often associated with machine learning. They provide a robust and convenient framework for approximating functions. Much of the recent progress in artificial intelligence is one way or another related to neural networks. But the model itself is far from new. Ideas to use something similar to neural networks existed from the beginning of the computer era. Back then, there were not one but many competing definitions of computability [85]. Eventually, the von Neumann model established itself as a winner, but the other models are still present. Among them, for example, are cellular automata that, as a computing model, are related to neural networks.

The neural networks for combinatorial optimization are different from the ones for machine learning and classification. The goal of machine learning is essentially to approximate a function. We have an "empirically" given function and the network needs to "learn" it from examples. In combinatorial optimization we seek an optimal structure. There is no notion of learning or training examples. In this context, it is convenient to think about neural networks as machinery for optimization and not as something that learns.

Existing neural network methods for solving combinatorial optimization problems are divided into two categories: Hopfield networks and self-organizing approaches. They are introduced in the following subsections. It should be mentioned that both these methods are not popular for solving optimization problems. At least, they are not as popular as neural networks in machine learning. There are good reasons for this. As discussed later in the section, the original Hopfield networks produce solutions of poor quality and thus can not compete with other metaheuristics. Self-organizing approaches, on the other hand, produce good solutions, but can

only be applied to a very restricted set of problems.

2.3.1 Hopfield networks

One way to think about **H**opfield **N**etworks (HN) is that they are a special case of **B**idirectional **A**ssociative **M**emories (BAM). A good detailed description of both models can be found in [85]. We start by introducing the BAM model that historically appeared after Hopfield networks.

The goal of a BAM and associative memories in general is to store a set of vectors $\{(in_1, out_1), (in_2, out_2), ..., (in_k, out_k)\}$. When given a vector close enough to in_i, the associative memory should produce out_i. Systems of this type are useful for de-noising input vectors.

BAMs are fully connected bipartite graphs consisting of simple computing units. The the left side of Figure 2.1 shows a BAM network and the right side shows one of the computing units.

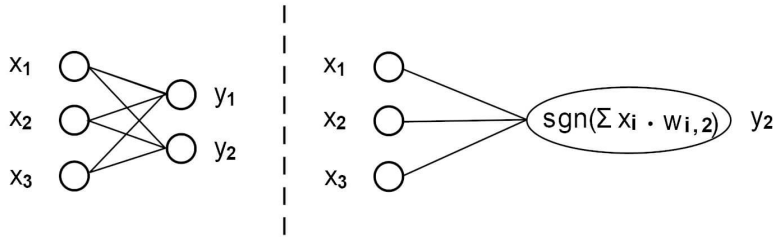

Figure 2.1: A small BAM network. On the left is the network itself with layers of size 3 and 2. The right side shows one of the computing units, unit y_2.

We denote the n units on the left side of a BAM as $\vec{X} = (x_1, x_2, ..., x_n)$ and the m units on the right side as $\vec{Y} = (y_1, y_2, ..., y_m)$. The weight of the connection between x_i and y_j is denoted as w_{ij} and the weights of all connections are written in a single n x m matrix W. Each unit of a BAM has a value: either 1 or -1. The unit can update its value. When performing an update, it computes the weighted sum over its connections and sets its value to the sign of the sum. For example, for unit y_j this can be written as $y_j = sgn(\sum_{i \in \{1..n\}} x_i \cdot w_{ij})$. It is assumed that the sign function for 0 is undefined and in that case, the value of y_j is not changed.

Initially, a BAM is fed from the left side with a vector $\vec{X_0}$. It computes the vector

$\vec{Y_0}$ on the right side as described in the previous paragraph. BAMs are synchronous networks, so all units on the right side compute their values at the same time. The vector $\vec{Y_0}$ can be written as $sgn(\vec{X_0} \cdot W)$. After $\vec{Y_0}$ is computed, the units on the left side compute their values. This operation can be written as $\vec{X_1} = sgn(W \cdot \vec{Y_0}^T)$. The "passing" of values back and forth between the layers continues until a stable state $(\vec{X}, \vec{Y})$ is reached. The vector $\vec{Y}$ is taken as the final result of the network for the input $\vec{X_0}$.

BAMs have learning algorithms that, given a set of input and output vectors, try to memorize this set. For example, Hebbian learning can be used for training a BAM. In Hebbian learning, the weight matrix W is set to the sum of the correlation matrices of the vector pairs that need to be stored. The details of the procedure are described in [85]. In the thesis, we are not interested in the learning capabilities of BAMs but in the existence of stable states $(\vec{X}, \vec{Y})$. For Hopfield networks, these stable states are going to be the solutions of the modeled combinatorial optimization problem.

The notion of an energy function is used for proving that a BAM network reaches a stable state. The same energy function is central in the analysis of Hopfield networks. The pair $(\vec{X}, \vec{Y})$ is a stable state if $\vec{Y} = sgn(\vec{X} \cdot W)$. This happens if $\vec{Y}$ is close enough to $\vec{X} \cdot W$. The scalar product between these two vectors can be though of as a measure of closeness (if not taking into account the lengths of the vectors, then the scalar product is the cosine of the angle between them). Following this line of thought, the energy function is defined as a scalar product of $\vec{X} \cdot W$ and $\vec{Y}$: $E = -\vec{x} \cdot W \cdot \vec{y}^T$. The minus at the front is because it sounds more natural to say that we minimize the energy. If the scalar product was not multiplied by -1, then our goal of making $\vec{X} \cdot W$ and $\vec{Y}$ close to each other would mean that the energy needs to be maximized.

The stable states of a BAM network correspond to local minima of the energy function. The result is stated below.

Theorem 2.3.1. *(Theorem 19 from Section 13.2 of [85]) A BAM with an arbitrary weight matrix W reaches a stable state that corresponds to a local minimum of*

the energy function in a finite number of iterations by using either synchronous or asynchronous updates.

A proof of the theorem can be found in [85]. The essence is that every update that changes the value of a unit decreases the energy function. This is simple to show by subtracting the energy before and after the update. An important point is that the network does not need to be synchronous. The result still holds even if the units update their values in arbitrary order.

Now we introduce the Hopfield network model by describing how it can be derived from a BAM. Assume that there is a BAM with layers $\{x_1, ..., x_n\}$ and $\{y_1, ..., y_n\}$ of equal size. Lets glue together the two layers: x_1 and y_1 are merged, x_2 and y_2 are merged, and so on. The result is a fully connected graph with n vertices. The weights of the edges in it are symmetric. From this graph we remove all edges from a vertex to itself. This is necessary because the presence of such short feedback loops interferes with the convergence properties of the model. The resulting graph is the underlying structure of Hopfield networks.

Similarly to a BAM, a Hopfield network is fed with some initial vector $\vec{X}$ and the computing units are left to update their values. Unlike BAMs, Hopfield networks are asynchronous. Their computing units decide to update at random times and independently from one another. The probability of two units updating simultaneously is zero. Such asynchronous networks biologically are more realistic, but the assumption of zero delay in computation and value updates is, of course, a non-realistic simplification.

The computing units of a BAM use the *sign* function as nonlinearity and when updating, unit x_i sets its value to $sgn(\sum_{j\in\{1..n\}} w_{ij} \cdot y_j)$. Hopfield networks use the step function as nonlinearity. More specifically, unit x_i sets its output to 1 iff $\sum_{j\in\{1..n\}} w_{ij} \cdot x_j > \sigma_i$ where σ_i is a parameter of the unit. When we later discuss the energy function of Hopfield networks, we will see that this is not a major difference.

A very small Hopfield network is shown in Figure 2.2. It has two computing units, x_1 and x_2. Since the value of each unit is either 1 or -1, the network can be in 4 different states. Assume the network is in state $\vec{X} = (1, 1)$. If unit x_2 updates

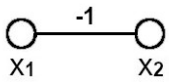

Figure 2.2: A small Hopfield network. The weight of the only edge is -1 and $\sigma_1 = \sigma_2 = 0$.

itself, then it will compute $x_1 \cdot -1 = 1 \cdot -1$ and compare this value to 0. It is less than zero, so the unit sets its state to -1. If now x_1 decides to update, it will compare $-1 \cdot -1$ to 0 and will keep its state 1. From then on it can be verified that not changes will happen and the network will indefinitely stay in state $(1, -1)$.

One additional modification is often applied to Hopfield networks when they are used for solving combinatorial optimization problems. The values of the neurons of BAMs are bipolar, 1 or -1. HNs use the values 1 and 0. The reason for BAMs to use bipolar encoding is that they, as a memory, are better at storing orthogonal or almost orthogonal vectors. Bipolar vectors have a higher probability of being orthogonal than vectors composed of 0s and 1s. Hopfield networks for combinatorial optimization do not care about orthogonal vectors but 0 and 1 are more easily interpreted as OFF / ON when modeling problems. The operation of the individual neurons and of the whole network does not significantly change because of switching to 0/1 vectors. If neuron x_i updates, it sets its value to 1 iff $\sum_j w_{ij} \cdot x_j > \sigma_i$.

As can be seen from the example network in Figure 2.2, Hopfield networks have stable states. This is the property that makes them interesting for solving optimization problems. Remember that BAMs with asynchronous updates are also guaranteed to reach a stable state (see Theorem 2.3.1). The proof that a Hopfield network eventually reaches a stable state is very similar to the one for BAMs and can be found in [85]. Again, an energy function is defined and it is shown that every flip of the state of a unit decreases the energy. The energy function is actually the same as for BAMs. The only detail is that BAMs use the sign function, while HNs use a step function. For modeling this we can notice that the expression $\sum_j w_{ij} \cdot x_j > \sigma_i$ of a Hopfield network corresponds to (and has the same effect as) $sgn(\sum_j w_{ij} \cdot x_j + 1 \cdot (-\sigma_i))$. If we transform the vector $\vec{X}$ to $\vec{X}'$ by adding an additional dimension that is constantly 1 and expand the weight matrix W to

W' with an additional row and column containing the $-\sigma_i$ values, then we get the *sign* function as nonlinearity. BAMs energy function that measures vector closeness becomes $E = -\vec{X'} \cdot W' \cdot \vec{X'}^T$. To work with the original weight matrix, we can substitute X' and W' in the expression above by their definitions and obtain $E = -\vec{X} \cdot W \cdot \vec{X}^T + 2\sigma_i \cdot x_i$. When simplifying the expression, we use the fact that the variables x_i are binary, so $\sigma_i \cdot {x_i}^2 = \sigma_i \cdot x_i$. In most literature on HNs the energy function is further divided by 2 and written as $E = -\frac{1}{2} \sum x_i \cdot x_j \cdot w_{ij} + \sum x_i \cdot \sigma_i$.

The idea to use Hopfield networks for solving combinatorial optimization problems was first described by Hopfield and Tank in the paper [48]. They apply the method to the Traveling Salesman Problem and report very promising results. Moreover, they describe a hardware implementation of the network that uses analogue computing units (neurons). This generated a lot of enthusiasm because the method essentially describes an extremely effective way of delivering meaningful digital result from analogue input. But the enthusiasm was not long and soon it became clear that the approach has many problems. We will talk about the problems a little bit later. First we want to write in pseudocode the Hopfield-Tank method for solving optimization problems. Originally, the approach was described in terms of equations of motion for the neurons (because it was designed to be implemented as a chip). Here the approach is described in pseudocode for conventional computers.

A Hopfield network manages a set of n binary variables that correspond to computing units or neurons. The first step is to express the objective function and the constraints of the problem in a form that is compatible with the energy function E. This may sound restrictive, but a lot of problems can be modeled in such a way. The next step is to actually run the network until a stable state is reached. We already described in text how the network operates. Algorithm 2 shows how this looks like in pseudocode.

The whole algorithm is compact and straightforward. The neurons update asynchronously and independently of one another. Since the probability of two neurons updating at the same time is 0, on a conventional consequential computer we can model the whole process by randomly selecting a neuron and calling its update

Algorithm 2: Combinatorial optimization with a Hopfield network

```
1  Function Update(x_i):
2      if ∑ x_j · w_ij > σ_i then
3          x_i ← 1
4      else if ∑ x_j · w_ij < σ_i then
5          x_i ← 0
6
7  while stable state is not reached do
8      // {x_1, ..., x_n} are the variables of the problem
9      i ← UniformIntRand(1..n)
10     Update(x_i)
11 end
```

procedure. In the update procedure we apply a step function and set x_j to 1 iff $\sum x_j \cdot w_{ij} > \sigma_i$. Assuming $E|_{x_i=v}$ is the value of the energy function with x_i set to v, the expression in the update procedure is actually $(E|_{x_i=0}) - (E|_{x_i=1})$. So, from the two options for x_i, the one that minimizes the energy is chosen.

Now we describe how to apply the method to solve the p-MiniSum Problem introduced in Section 1.2, Definition 1.2.2. The input of the problem is a weighted undirected graph G with vertex set $V = \{v_1, ..., v_m\}$ and the number p of facilities (warehouses) to place. There is a client in every vertex of the graph and the goal is to find a set of p vertices where to locate the facilities so that the cost of servicing the clients is minimized. The cost is defined as $\sum_{c \in V} min_{v_i \in v_1 ... v_p} dist(c, v_i)$. Figure 1.1 from Section 1.2 shows an example instance of the problem.

The idea of how to model p-MiniSum for solving with Hopfield networks is taken from our paper [42]. In the problem, there are two different types of relations: which facility a client uses and where a facility is located. The relations are modeled with two groups of binary variables: CF_{ij} (client-facility relation) and FL_{jk} (facility-location relation). Variable CF_{ij} is 1 iff the client in vertex v_i uses facility number j. Variable FL_{jk} is 1 iff facility number j is placed in vertex v_k. Figure 2.3 illustrates the proposed p-MiniSum model.

The p-MiniSum model has $n = 2 \cdot m \cdot p$ variables. Each variable is naturally mapped to a neuron of the Hopfield network. An assignment of values to the variables represents a feasible solution to p-MiniSum if every client is serviced by

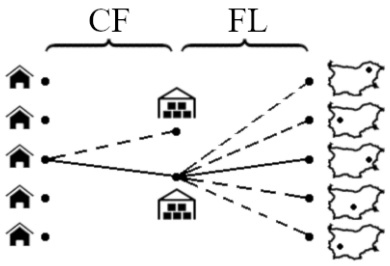

Figure 2.3: p-MiniSum model. The variables of the problem are actually the edges in image (not all edges are shown). The edges on the left side represent the client-facility relations and on the right — the facility-location relations. Solid edges have value 1 and dashed edges have value 0. The path of solid edges in the image can be interpreted as saying that the third client (house, left column) uses the second facility (middle column) that is placed in the third location (right column).

exactly one facility and every facility is placed in exactly one location.

The objective function of p-MiniSum can be represented as:

$$\sum_{i,j,k} CF_{ij} \cdot FL_{jk} \cdot dist(v_i, v_k)$$

Intuitively, the formula says that if the client in vertex v_i uses facility j that is placed in vertex v_k, then the distance from v_i to v_k is added to the value of the objective function. For points that are feasible solutions the described function exactly gives the cost of servicing all clients.

The objective function $\sum_{i,j,k} CF_{ij} \cdot FL_{jk} \cdot dist(v_i, v_k)$ is a quadratic function and is compatible with the notion of energy of a Hopfield network. What is left is to model the feasibility constraints: every client needs to be mapped to exactly one facility and every facility needs to be placed in exactly one location. The way constraints are modeled in Hopfield networks is by adding penalty terms to the objective function. For the constraint that every client i uses exactly one facility we can add the penalty $(1 - \sum_{j \in \{1..p\}} CF_{ij})^2$. The expression gets its minimum value of 0 iff the constraint is satisfied. Similarly, for the facility-location relations we add penalty terms of the form $(1 - \sum_{k \in \{1..m\}} CF_{jk})^2$. Notice that both expressions are quadratic and are thus compatible with the energy function of a Hopfield network (after opening the brackets, the constant 1 can be omitted; terms of the form CF_{ij}^2 can be simplified because all variables are binary, so $CF_{ij}^2 = CF_{ij}$).

The full HN-compatible objective function of p-MiniSum is written as:

$$\sum_{i,j,k} CF_{ij} \cdot FL_{jk} \cdot dist(v_i, v_k) +$$
$$A \cdot \sum_{i \in \{1..m\}} (1 - \sum_{j \in \{1..p\}} CF_{ij})^2 +$$
$$B \cdot \sum_{j \in \{1..p\}} (1 - \sum_{k \in \{1..m\}} CF_{jk})^2$$

Here A and B are parameters that need to be specified. After opening the brackets and simplifying, it can be seen that the expression is of the form of an energy function $E = -\frac{1}{2} \sum x_i \cdot x_j \cdot w_{ij} + \sum x_i \cdot \sigma_i$. By equating the two expressions, the values of w_{ij} and σ_i of the neurons are computed and we are ready to run the network. It was already discussed that a Hopfield network finds a local minimum of the energy E. So, what we get at the end is a local minimum of the function from the start of this paragraph. If the constants A and B are well chosen, this local minimum should give us a good solution to the p-MiniSum instance.

In theory all that looks good. But, sadly, in practice Hopfield networks are not able to find high-quality solutions. Before discussing the reasons for this, we want to stress the strong sides of the Hopfield-Tank approach. First, Hopfield networks minimize a quadratic function and by using the right penalties, many problems can be formulated in a HN-compatible way. Second, the approach is massively parallel and asynchronous. This allows a very simple and efficient parallel implementation. Third, Hopfield networks can be implemented in hardware. Moreover, they can benefit from the development of optical computers and other specialized neural network hardware that can speed them up by orders of magnitude. Discussion about neural network hardware can be found in [85].

The main disadvantage of the Hopfield-Tank approach is that it does not produce good solutions. This, of course, is a very deadly disadvantage. The quality of the solutions that are returned by Hopfield networks is not competitive with the quality of the solutions obtained using other metaheuristics. The problems of the Hopfield-Tank approach are described in the paper [106] where the authors try to reproduce

the initial results of Hopfield and Tank. In the paper, the authors are not able to reproduce such good results and their conclusion is that the HN approach does not scale even for moderately-sized inputs. Not only Hopfield networks are not able to find good solutions to the Travelling Salesman Problem, but they are not even able to find feasible solutions. And the probability of finding a feasible solution decreases with the size of the input.

We performed a number of experiments with Hopfield networks on p-MiniSum instances based on the Bulgarian road network. The experiments are described in the paper [42]. The results were also very discouraging. While our approach to p-MiniSum was designed so that HNs eventually reach a feasible solution, the returned solutions turned out to be not much better than just randomly generated ones. Of course, it is possible that another model of p-MiniSum that gives better solutions exists, but we believe that our results illustrate the main problems of the Hopfield-Tank approach. Below a summary of the conclusions from the experiments is presented.

First, the idea to model constraints with penalty terms seems to be very problematic. This has also been noted in [106] for the Travelling Salesman Problem. In the HN-compatible objective function of our p-MiniSum model there are two parameters called A and B. The parameters scale (weight) the penalty terms and are there to enforce feasibility of the final solution. Making the parameters large guarantees that local minima of the energy represent feasible solutions. But large values for A and B bias the energy function and the penalty terms start to dominate over the real objective of p-MiniSum. So, intuitively, A and be B need to be as close to 0 as possible but still large enough to guarantee that a feasible solution is reached. In our experiments, we performed a type of a grid search over A and B to push them as close to 0 as possible while still having the Hopfield network converge to a feasible solution. In general, finding good values for the weighting parameters of a Hopfield network is complex and instance-specific. Indeed, one of the research directions in the field was to develop a method for choosing these parameters [54, 70]. We do not know of a robust general method for this. Also, we believe that if

the number of parameters is kept constant, as is usually done, and the size of the input is allowed to grow, then from one point onward it becomes impossible to find parameters that provide reasonable balance between the objective of the original problem and the feasibility constraints.

Second, a related problem of Hopfield networks is the shape of the search space. HNs are essentially a local search method. They operate over binary variables and use a single variable flip neighbourhood structure. A search space is defined by the variables of the problem and the chosen neighbourhood. If there are many poor-quality local minima or plateau regions, then local search performs badly. This is exactly what penalty terms "contribute" to the search space. Even if the objective function of the original optimization problem behaves well, when penalty terms are added, they artificially create a huge number of poor local minima (intuitively, the local minina roughly correspond to "every" possible solution). This is again the same problem of correctly weighting the penalty terms. If we are too aggressive to enforce feasibility, then we end up with a bad search space. In the other extreme, if our penalties are too weak, then we end up with infeasible solutions.

Expanding on the comparison to local search, we know from the field of metaheuristics that by itself this method is not enough to obtain a high-quality solution. A modern optimization method needs to incorporate some way of escaping from poor local minima. Apart from the randomness in the order of neuron updates, Hopfield networks are completely deterministic and do not have any of the features of high-quality metaheuristics outlined at the end of Section 2.2. This is why it is expected for Hopfield networks to not be able to compete in solution quality with other metaheuristics. In fact, HNs were not designed for solving combinatorial optimization problems. Hopfield and Tank noticed that they can be used for this, but, originally, Hopfield networks were a model of a memory and not a machinery for optimization.

Finally, a few notes about the original hardware implementation of Hopfield networks. The weights and parameters of the network are implemented with resistances and the proposed chip is instance-specific. This is bad because we want hardware

that is able to solve general problems or at least a given class of problems. Creating a new chip for every instance does not sound realistic. Hopfield networks can be implemented on devices like field-programmable gate arrays that allow the integrated circuit to be reconfigured after manufacturing [96]. We also believe that if Hopfield networks demonstrate good problem solving properties, then a better hardware implementation for them is going to be developed. But for now, the hardware implementation of HNs is more like a proof of a concept than something that can be used in practice.

As a summary, Hopfield networks for combinatorial optimization have problems and probably in their pure form will never be competitive to other metaheuristics. But they drew attention to the fact that massively parallel networks of neurons demonstrate spontaneous computational properties. As we will see later, combining this idea with better ways to express constraints and with methods to escape from local minima leads to a robust metaheuristic for facility location problems that is very competitive to other approaches.

2.3.2 Boltzmann machines

As we saw in the previous section, one of the problems of Hopfield networks is that they do not have a mechanism for escaping from local minima. Looking at other metaheuristics for combinatorial optimization, we see that incorporating such a mechanism means that randomness (noise) needs to be introduced in the operation of the network. There are different places where noise can be introduced. **B**oltzmann **M**achines (denoted BM) [46] introduce randomness to Hopfield networks at the point where we decide the value of a variable based on the difference of the energy function between the two possible values. Other possibilities exist. For example, Gauss machines [2] introduce noise by providing to every neuron an external stochastic input. The input obeys the Gauss distribution, which explains the name of the model. This section describes BMs because they seem to be more popular and are representative of the general idea.

Boltzmann machines have several usages. Here we introduce them from the

perspective of solving combinatorial optimization problems. The mechanism used by BMs for escaping from local minima is basically simulated annealing (see Section 2.2). So they can be both considered to be stochastic Hopfield networks and distributed implementation of simulated annealing.

Similarly to Hopfield networks, BMs operate over a complete undirected weighted graph $G(V, E)$. The vertices of the graph, called neurons or units, represent binary variables and can have two states: 0 (OFF) and 1 (ON). Boltzmann machines can also use bipolar encoding for the variables, but, in our opinion, binary encoding is more natural for combinatorial optimization. Every edge (u_i, u_j) of the graph has a connection strength (weight) $c_{ij} = c_{ji}$. Every vertex u_i has a loop (u_i, u_i) with weight c_{ii}.

On intuitive level, the goal of a BM is to reach consensus over the state of the units. Consensus is measured by the function $\sum_{(u_i,u_j)\in E} u_i \cdot u_j \cdot c_{ij}$ that the Boltzmann machine tries to maximize (u_i denotes both the vertex u_i and the value of the corresponding variable). Notice that in the context of Boltzmann machines we say "consensus" but this in reality is a quadratic function analogous to the energy function of Hopfield networks. We emphasize that the strengths c_{ij} can be both positive and negative, which makes maximization hard.

As in Hopfield networks, every unit of a BM has a local update procedure for its state. Assume unit u_i wants to perform an update. The unit computes ΔC, the difference in consensus if its state is flipped. It is easy to verify that the difference in consensus can be written as:

$$\Delta C = (1 - 2u_i) \cdot (\sum u_j \cdot c_{ij})$$

If $\Delta C > 0$, the unit performs the state flip. Otherwise, the unit performs the state flip with probability $(1 + e^{\frac{-\Delta C}{T}})^{-1}$. This acceptance criterion is directly taken from the simulated annealing metaheuristic. T, the *temperature*, is a positive parameter. It controls the acceptability of state flips that decrease the consensus. During the operation of BMs, the value of T decreases, which makes deteriorating flips of variables less and less probable.

Algorithm 3: Combinatorial optimization with Boltzmann machine

```
1  T ← initial temperature, a real positive number
2  α ← temperature decrease coefficient, a real number between 0 and 1
3  STEPS ← steps per iteration, proportional to the number of neurons
4  ITERS ← number of iterations
5
6  Function Accept(Δ):
7      if Δ ≥ 0 then
8          return True
9      else
10         prob ← 1 / (1 + e^(-Δ/T))
11         return UniformRealRand(0, 1) ≤ prob
12
13 Function Update(u_i):
14     Δ ← (1 − 2u_i) · (Σ u_j · c_ij)
15     if Accept(Δ) then
16         u_i = 1 − u_i
17
18 for iter ← 0 to ITERS do
19     for step ← 0 to STEPS do
20         i ← UniformIntRand(1..n) // n - number of neurons
21         Update(u_i)
22     end
23     T ← T · α
24 end
```

Algorithm 3 shows one possible implementation of a Boltzmann machine for solving combinatorial optimization problems. It is similar to the Hopfield networks from Algorithm 2. The main difference is the usage of the randomized acceptance criterion for state flips. The operation of a Boltzmann machine ends with a temperature T that is close to 0. In this case, since the low temperature pushes the

probability of deteriorating flips very close to 0, the BM behaves like a Hopfield network. This means BMs stop in a local maximum of the consensus function. There is a slight nuance that is usually ignored. The probability of deteriorating moves goes to 0 but it never reaches 0. So BMs actually never settle in a single state. But, in practice, we can ignore this and just stop the optimization after a certain number of iterations.

For BMs it can be proven that they asymptotically reach a global maximum of the consensus function (the proof is analogous to the one for simulated annealing [61]). This is a strong result. It says that BMs, in probabilistic sense, can optimally solve any problem that can be modeled in a way that is compatible with them, including NP-hard problems. And sometimes this result produces more enthusiasm then it should. The result says that, given enough time, BMs reach a global maximum with high probability. But the time necessary for this is huge. Even the best of the known bounds on the number of steps are larger than the size of the whole solution space. In this time we can traverse the whole space and trivially compute the optimum. Of course, the property of asymptotically reaching a global maximum is a desirable one for any metaheuristic, but just this property does not mean that the metaheuristic is good.

As with Hopfield networks, if we want to solve a combinatorial optimization problem with BMs, we first need to model it in a BM-compatible way. For our purposes, a Boltzmann machine manages a set of binary variables and approximately finds the maximum of the consensus function. The first part of the modeling is to represent the object that the combinatorial optimization problem asks for as a binary vector. This binary vector naturally maps to the units of the Boltzmann machine. The second part of the modeling is to specify the connections and their strengths. Boltzmann machines maximize the consensus, so the value of this function needs to be equal or directly related to the quality of the represented solution. Additionally, for most models, not every binary vector corresponds to a valid solution. There is a subset of these vectors that contains the feasible solutions and we must ensure that the Boltzmann machine arrives at a feasible one. It is guaranteed that a BM finds

a local maximum of the consensus function, so the easiest way to ensure feasibility is to make any local maximum a feasible solution. This can be achieved by adding penalty terms to the consensus function.

The described framework is quite flexible and many combinatorial optimization problems can be modeled in this way. Possible modelings of the classical Max Cut, Independent Set and Graph Coloring problems can be found in [67]. The authors solve instances of these problems using Boltzmann machines and obtain very good results. In the end of this subsection, we briefly describe their model of the Independent Set problem as an example of a successful application of a BM to a combinatorial optimization problem. While Boltzmann machines are better than Hopfield networks at maximizing a function, they suffer from the same problem of using penalty terms to enforce feasibility constraints. The penalty terms introduce bias in the consensus function and artificially increase the complexity of the solution space. In such unfavorable situations, Boltzmann machines can hardly improve on the results of Hopfield networks [85]. From the perspective of facility location problems, the constraint to choose one option from many (one facility for every client, for example) produces a difficult solution space. Penalty terms for this constraint partition the space into many regions with poor local maxima from which it is hard to escape.

To get an idea about the performance of Boltzmann machines on facility location problems, we applied them to a set of realistic p-MiniSum instances based on the Bulgarian road network. Our experiments are described in [42]. These instances are actually the same ones that we used for evaluating Hopfield networks. The results of BMs were very disappointing. They produced solutions that were just slightly better than the ones produced by Hopfield networks, but BMs were significantly slower. Our opinion is that Boltzmann machines are competitive to other metaheuristics only if the problem being solved fits well in the consensus model. Once the need to add penalty terms for enforcing feasibility emerges, the Boltzmann machines are no longer competitive. Sadly, this most probably means that BMs are not a good method for solving facility location problems.

We finish this subsection with a classical problem that fits well in the consensus model and, as a consequence, BMs perform very well on it.

Definition 2.3.1. (Independent Set Problem)

- **Input**

  1. An undirected graph G.

- **Solution**

  The size of the largest independent set of G. An independent set of a graph is a subset of its vertices such that no two vertices in the subset are adjacent.

Theorem 2.3.2. *The* Independent Set Problem *can be modeled for solving with a Boltzmann Machine.*

Proof. The construction below is a slight modification of the construction in [67]. Let $G(V, E)$ be the input graph. For each vertex u_i in G, we create a variable x_i in the Boltzmann machine. The value of x_i is 1 iff vertex u_i is taken in the independent set. For every edge $(u_i, u_j) \in E$, the variables x_i and x_j are connected and the connection strength is -2. Additionally, all loops (x_i, x_i) are added with connection strength 1. This completes the construction.

To prove that the described construction represents a modeling of the Independent Set Problem, it needs to be shown that local maxima of the consensus function correspond to independent sets and that the value of the consensus function is related to the size of the set.

For the first part, assume there is a local maximum of the consensus that does not correspond to an independent set. This means that there are two vertices $u_i \neq u_j$ that are connected by an edge in G and for which $x_i = x_j = 1$. Setting x_i to 0 decreases the consensus by 1 because of the loop at x_i and increases it by at least 2 because of the connection between x_i and x_j. Overall, the consensus increases. This is a contradiction with the current state being a local maximum of the consensus function.

For the second part, we already showed that local maxima correspond to independent sets. When the values of the variables of a Boltzmann machine represent an

independent set, the consensus function is equal to the size of the set because only the terms corresponding to loops can be nonzero. By maximizing the consensus function, the Boltzmann machine actually maximizes the size of the independent set. It is worth mentioning that the intermediate states during the operation of the Boltzmann machine may not be local maxima and so they do not necessary represent independent sets. For such states, the value of the consensus function is not equal to the size of the represented set. □

2.3.3 Self-organizing approaches

The neural network methods for combinatorial optimization that were described so far are based on the Hopfield networks model. The self-organizing approach that is presented in this section is based on a different idea. A better name for it may be a "template approach" because the essence of the method is to take a template of the solution and deform it so that it matches the instance being solved. In the literature it was shown that self-organizing approaches produce better solutions to the Travelling Salesman Problem than Hopfield networks [3, 32]. But the drawback of these approaches is that they are not general enough and are not applicable to most problems. We start by introducing self-organizing feature maps and, more specifically, the elastic net method for combinatorial optimization. Then we talk about the drawbacks of this class of methods.

Probably the best-known self-organizing neural network is the topology-preserving map proposed by Kohonen [65, 66]. What the method does is to create a map of the input space in a self-organizing way. The approach is inspired by the process of mapping visual signals to regions of the human brain. By self-organizing it is meant that the "correct" output is not known *a priori* and so no notion of distance to the "correct" output can be defined. Topology-preserving maps produce tiling of the input space into subregions and are not made for combinatorial optimization. But the idea of a self-organizing process can be used for solving optimization problems. The elastic net method [30] is an example of how to use such a process for solving the Travelling Salesman Problem (TSP).

In self-organizing feature maps we are given an input space that provides us with vectors according to its probability distribution (the probability mass may not be uniformly distributed). This space is complex and is not convenient for the application that we have in mind. For example, maybe in a 3-dimensional space there is a 2-dimensional surface with many curvatures and we want to move a robot on this surface but can not analytically describe the surface. We need to simplify the input space by approximately mapping another space that has desirable properties to the input space.

For simplicity, assume that we are charting an n-dimensional space with a chain of units. In the chain each unit has a left and right neighbor (with the exception of the leftmost and the rightmost units). Every unit i has an associated n-dimentional weight vector w_i. The weight vectors are initially random and are adjusted during the training process.

Once the network is trained, when an input vector x is provided, each unit computes the Euclidean distance between its weight vector and x. The unit with the smallest distance outputs 1 and all other units output 0. This way, each input vector is associated with one unit which defines a tiling of the input space. Intuitively, the map is trained in such a way that the neighborhood relation between the units extends to the tiles of the input space.

The training is performed by an iterative procedure. It has two parameters: η (learning rate) and neighborhood function ϕ from a pair of states to a non-negative real number. The value $\phi(i, j)$ represents the strength of the connection between units i and j. A popular choice is setting $\phi(i, j) = 1$ iff $|i - j| \leq R$ for a chosen radius R and 0 otherwise. The training procedure is described below:

1. Initialize the weight vectors w_i to random values.

2. Select an input vector inp.

3. Find the unit k for which the Euclidean distance between w_k and inp is minimal.

4. For each unit i update $w_i = w_i + \eta \cdot \phi(i, k) \cdot (inp - w_i)$.

5. Modify η and ϕ according to the schedule. If the maximum number of iterations has not been reached yet then go to step 2.

During the training there is a schedule that says how to gradually decrease the learning rate η and the neighborhood function ϕ. Often both the radius of ϕ and the strength of the connection are decreased. The goal of the decrease schedule is to make the corrections smaller as time progresses so that the whole system can arrive at a stable state at the end of the training.

On intuitive level, each vector of the input space pulls the closest unit from the chain. When a unit is pulled, its neighbors are also pulled in the same direction because of the neighborhood function. The effect is that the chain of units gets positioned uniformly in the subspace defined by the input vectors. The properties of Kohonen networks are still not completely understood. There are several proofs of convergence of one-dimensional networks in one-dimensional space, but there is no general proof of convergence for multidimensional networks [85].

As was said earlier, Kohonen networks do not perform combinatorial optimization. The elastic net method is the best-known example of how to apply ideas from Kohonen networks to combinatorial optimization problems (more specifically, to a variant the TSP problem). Assume that n cities in the Euclidean plane are given. The goal of the TSP is to find the shortest cycle that visits every city exactly once. The elastic net method starts with a template of such a cycle: a ring of units. This template is then elongated until it starts to pass sufficiently close to the input cities. Similarly to Kohonen networks, the deformation is guided by two forces — cities pull the units towards them and units pull their neighboring units. The first force is responsible for mapping the template to the input cities and the second force makes the resulting cycle short.

Lets denote by x_i the coordinates of the i-th input city and by y_j the coordinates of the j-th unit on the ring. The optimization in the elastic net method is an iterative process of changing the y_j. The rule of change is written below.

$$\Delta y_j = \alpha \sum_i w_{ij}(x_i - y_j) + \beta K(y_{j+1} + y_{j-1} - 2y_j)$$

The two terms in this sum correspond to the two forces (their relative importance is determined by the α and β parameters). K is a length parameter that gradually decreases during the optimization. w_{ij} is a decreasing function of the distance between x_i and y_j and expresses how strongly city i attracts unit j in the current configuration. Both terms in the equation depend on distances between points and, as the optimization progresses, larger distances are less and less favored. This somewhat resembles the decrease of the temperature in simulated annealing. At the start of the optimization each point on the ring is almost uniformly pulled by each city. As time progresses, every city "specializes" on its section of the ring.

In [30] it is shown that the iterative process described above finds a local minimum to an energy function for which global minima are optimal solutions. This is the basis of the claim that the elastic net method finds good approximate solutions to the TSP problem. [30] also contains experimental results showing that the elastic net method indeed is able to find good solutions.

While the idea of the elastic net method is very interesting, for the method to work it needs a template of the solution. Moreover, the template object needs to be in the same space as the input objects of the problem. From the perspective of facility location problems this is a serious drawback. While the elastic net method is not limited to just cycles (it can map, for example, star graphs or any other graph), defining a template for the solution of a facility location problem seems to be complicated. Intuitively, we do not even know how many clients each facility will service. If we can not define a template, we can not apply the method. This limitation can be seen in the set of problems to which the elastic net method has been applied. To the best of our knowledge, all applications in the literature are to the TSP problem or some of its variants.

Chapter 3

Competition-Based Neural Networks (CBNNs)

This section introduces **C**ompetition-**B**ased **N**eural **N**etworks (CBNN), a method that we originally developed for solving facility location problems [42]. As is usually done for neural networks, the section starts with an analogy between the method and the real world.

Instead of a human brain analogy we use a business analogy. There is a set of companies that can be active (ON, 1 in binary encoding) or closed (OFF, 0). The companies are partitioned into segments. Inside a segment the competition is severe but there is no competition between companies from different segments. The whole system is guided by three principles. First, the companies are selfish. They care only about themselves and want to be ON (this is the goal of every company). Second, the "law of the jungle" holds: in every segment only the strongest survives. This is the essence of the competition. Every company wants to be ON, but eventually only one company from every segment will be ON (in other words, only one company from a segment survives). There are complex relationships between the companies, including cross-segment relationships, so who is strongest depends on the current global configuration. By configuration it is meant the state, ON or OFF, of the companies. Company A may be stronger than company B in one configuration but weaker in another. Third, there is luck (randomness) in the environment. Occa-

sionally it can happen that a weaker company is lucky and is able to successfully compete with a stronger company and to keep its ON state. The probability of this to happen decreases exponentially with the difference between the strengths of the companies. Such "luck" can have consequences for the global configuration. It can produce enough change to transform the previously weaker company into the current leader in its segment and in this way to secure an ON state for it. We emphasize that the described system has a complex behavior and its final state depends not only on the properties of the companies but also on the initial state and the sequence of (random) events that happen during the evolution of the system.

In addition to the three principles above, there is another mechanism in the environment that is called temperature T (as in Simulated annealing). This temperature controls the amount of "luck". A large value means there is a lot of randomness in the competition. We can think of it as introducing a new invention in the environment. Initially, there is a lot of enthusiasm from the invention. Even a "small" and weak company can be lucky enough to find a good application for it and this increases the total amount of noise in the competition. Eventually, everyone learns the best applications of the invention, and the noise because of it decreases.

The economy of this imaginary environment is a function of the states of the companies. In the real world, competition makes the economy stronger. Additionally, new inventions quickly push forward the economy. If we are able to represent the objective of a combinatorial optimization problem as an economy of such an imaginary world, we can leave the competing companies to push it up to a good state (solution). Of course, the described environment is a very simplified model of the real world, but artificial neural networks are also extremely simplified models of the human brain.

To sum up, the idea of the method is to binary encode the variables of a given combinatorial optimization problem as states (ON / OFF) of companies. Set the economy to be equal to the value of the objective function. Then randomly initialize the system of companies and set the temperature T to a high value. After this, leave the companies to compete with one another according to the three described

principles while slowly decreasing the temperature. It can be shown that after the system is left for long enough to evolve according to its principles, it reaches a stable state. This stable state (hopefully) represents an approximately optimal solution to the combinatorial optimization problem.

The method can be expected to produce good solutions to real problems because our society and economy are tuned to solve such problems. Of course, there are more formal reasons than this to say that CBNNs can be expected to find solutions of high quality. The next chapter is devoted to analyzing the properties of competition-based neural networks. There we show that they asymptotically converge to a global optimum and we discuss the speed of convergence. In the Chapter 5, we apply competition-based neural networks to several facility location problems and empirically demonstrate their excellent performance.

3.1 The problem solved by CBNNs

Competition-based neural networks operate over a set $\{x_1, x_2, ..., x_t\}$ of binary variables. The set of variables is partitioned into r non-empty subsets. In the business analogy from the start of the section we called the subsets segments and from now on we use the name groups for them. The goal of a competition-based neural network is to minimize the value of an objective function $F(x_1, ..., x_t)$. F can be an arbitrary function. We say arbitrary but, as any other method, CBNNs have certain bias and plugging a "badly behaving" function leads to worse solutions. When modeling a given problem for solving with a CBNN, there may be many different possibilities for defining the objective function. One reason for this is that competition-based neural networks automatically enforce feasibility constraints. Over the infeasible region of the solution space, the objective function can be defined any way we like. Of course, we should define it reasonably to make it simpler for the network to find a good solution. This is discussed further in Chapter 5 where the neural network method is applied to several facility location problems. The general form of the CBNN problem is formally stated below.

Definition 3.1.1. (CBNN problem) Assume that the set $\{x_1, ..., x_t\}$ of binary variables is partitioned into subsets $\{G_1, ..., G_r\}$ that are called groups. The goal of the CBNN problem is to minimize a given function $F(x_1, ..., x_t)$ under the constraints that $\sum_{x_i \in G_j} x_i = 1$ for every group G_j.

It is easy to verify that there are combinatorial optimization problems that cannot be formulated as a CBNN problem. The problem is defined in a way that is convenient for modeling facility location. The main idea of using *competing companies* for performing optimization can solve a more general problem. This work uses the problem above because it is enough for facility location and it simplifies the analysis of the method.

The feasibility constraints of the CBNN problem are of the type $\sum_{x_i \in G_j} x_i = 1$ for every group G_j. These are the 1-of-k constraints (choose one option from k) that were hard to model with Hopfield networks and Boltzmann machines. Such constraints can be seen in all facility location problems we know of and competition-based neural networks can naturally express them.

We finish the section by returning to the p-MiniSum facility location problem from Section 1.2. When discussing Hopfield networks, we formulated p-MiniSum as a problem over the binary variables CF_{ij} and FL_{jk}. The problem can be written as below (see Section 5.1):

Minimize:

$$\sum_{i,j,k} CF_{ij} \cdot FL_{jk} \cdot dist(v_i, v_k)$$

Subject to:

$$\forall i \in \{1..n\} \quad \sum_{j \in \{1..p\}} CF_{ij} = 1$$

$$\forall j \in \{1..p\} \quad \sum_{k \in \{1..n\}} FL_{jk} = 1$$

This is a CBNN problem. It has two types of variables, but we can still call all of them x_i if we want without changing anything. For the group constraints, all

CF_{ij} variables with the same i are placed in a single group. Likewise, all FL_{jk} variables with the same j go into one group. This partitions all variables into $n + p$ groups. Looking at Figure 2.3, for every node, all outgoing edges define a group. This guarantees that for every node, exactly one of its outgoing edges is selected to be ON in the final solution.

3.2 The CBNN solver

We present a formal description of the neural network solver for the CBNN problem. In fact, several slightly different variants of the solver are used in the thesis. The section starts with a description of a variant of the solver that is called group-best. This is the solver that is used most of the time in Chapter 5 where we solve several practical facility location problems. The section continues with the description of how to transform the group-best solver into a solver that we call group-average. The difference between the two is very small. In the business analogy, the only change is in the way we compute the strongest competitor for a company. The group-average variant of the CBNN solver is the one that is used when proving convergence of competition-based neural networks. Finally, several possible improvements of the general CBNN solver are described. The goal of these modifications is to reduce the runtime of the method and we sometimes apply them in Chapter 5 when solving large facility location instances. All variants of the CBNN solver are very similar to each other and essentially express the same idea.

According to the business analogy, the optimization process of a CBNN is a simulation of the economy of an imaginary world of variables (companies) of the problem. To that end, the group-best solver needs three main components: a way to evaluate the *strength* of a variable, a competition mechanism, and a luck factor. For a given optimization problem, we know its objective function $F(x_1, ..., x_t)$ where x_i are the variables. We want this function to be our economy and it is clear that the strength of a variable needs to somehow reflect its contribution to the objective function. Assume we have two variables x_i and x_j from the same group. $F[x_i = 1](\vec{x})$ denotes value of the objective function F for the vector $\vec{x}$ in which all variables of

the group of x_i are set to zero except for x_i that is set to 1. If x_i and x_j belong to the same group and $F[x_i = 1](\vec{x}) < F[x_j = 1](\vec{x})$, it makes sense to say that the variable x_i is stronger than x_j. This can be slightly unintuitive because x_i is on the left side of the less comparison. An explanation of why x_i is stronger (better) is that our goal is to minimize F and in the current configuration, setting x_i to 1 makes the value of F smaller. Remember that x_i and x_j are in the same group, so they compete against each other and in the final solution only one of them can be in ON state.

The above definition of strength glues together the mechanism of competing companies and our goal of minimizing the objective function F. Notice that in the definition of strength, whether x_i is stronger than x_j depends on the current global configuration $\vec{x}$ (that is, on the states of the other companies). Algorithm 4 shows in pseudocode the general CBNN solver. So far, we have only introduced the strength evaluation component that is represented by the function *OnCost* in the pseudocode.

The second part of the neural network method is the competition mechanism. In the pseudocode, it is represented by the *Update* function. This function has two arguments: a variable x_i and the current temperature T. The temperature is necessary for the luck mechanism and is discussed later in the section. The variables in the network are independent. When updating x_i, we can check the states of the other variables, but we can only change the state of x_i. The update of x_i starts with the computation of the set of variables from its group that are in ON state. This set is called *active* (line 15 in the pseudocode). Then the strongest among the *active* variables is computed. It is denoted by *best* (line 21). For deciding which variable is the strongest, we use the already described way of measuring variable strength that is represented by the *OnCost* function. Finally, the competition mechanism is implemented between lines 23 and 26: if the current variable x_i is stronger than *best*, then it is ON. Otherwise, it is OFF. The decision follows the simple "rule of the jungle". This concludes the description of the competition mechanism of group-best CBNNs. The variant of the solver is called group-best because the current variable

Algorithm 4: General CBNN solver

$\alpha \leftarrow$ temperature decrease coefficient, a real number between 0 and 1
$STEPS \leftarrow$ steps per iteration, proportional to the number of variables
$ITERS \leftarrow$ number of iterations for the whole optimization

// Value of the objective function if only x_i was ON in its group.
Function *OnCost*($\vec{x}$, i)**:**
    return $F[x_i = 1](\vec{x})$

Function *Accept*(Δ, T)**:**
    $prob \leftarrow \frac{1}{1+e^{\frac{\Delta}{T}}}$
    return $UniformRealRand(0, 1) \leq prob$

Function *Update*(x_i, T)**:**
    $G \leftarrow$ group of x_i
    $active \leftarrow \{x_j \in G \mid i \neq j \text{ and } x_j = 1\}$

    if *active is empty* **then**
        $x_i \leftarrow 1$
        return

    $best \leftarrow min(\{OnCost(\vec{x}, j) \mid x_j \in active\})$
    $me \leftarrow OnCost(\vec{x}, i)$
    if $me < best$ **then**
        $new\_value \leftarrow 1$
    else
        $new\_value \leftarrow 0$

    if $Accept(|best - me|, T)$ **then**
        $new\_value \leftarrow 1 - new\_value$
    $x_i \leftarrow new\_value$

$T \leftarrow$ initial temperature, a real positive number
$\vec{x} \leftarrow$ random initial state
for $iter \leftarrow 0$ **to** $ITERS$ **do**
    for $step \leftarrow 0$ **to** $STEPS$ **do**
        $i \leftarrow UniformIntRand(1..t)$ // t - number of variables
        $Update(x_i, T)$
    end
    $T \leftarrow T \cdot \alpha$
end

is compared against the best variable in its group. There are other possibilities for implementing the competition mechanism. For example, x_i can be compared against a random variable from the *active* set or against the average *OnCost* of the variables. The second option is used in the group-average solver that we introduce later.

One special case of the competition mechanism is handled between lines 17 and 19. For a variable x_i, its set of competitors, denoted *active* in the pseudocode, can be empty. If this happens, then there is no competitor to which to compare the variable x_i and the *Update* procedure simply sets the value of x_i to 1 to satisfy the corresponding group constraint. The described situation can happen for the strongest variables in their corresponding groups during the late stages of the optimization and it sounds reasonable to keep their ON state. Of course, such variables can still be turned OFF with some probability later, when the states of the other variables change. Another possibility for getting an empty *active* set is for this to occur in the initial random solution. It does not matter much what we do in this case and setting the value of x_i to ON again is OK.

The last component is the luck factor. In the *Update* procedure, luck is applied on lines 28-29. At this point, the value of x_i that follows the "rule of the jungle" is already known and what the luck factor does is to flip this value with some probability. In the business analogy, "being lucky" means that the company-variable is still able to compete and keep its ON state despite being weaker. The other side of the luck factor is that the strongest company may with some probability transition from ON to OFF state. The probability of the flip depends on the difference between the strengths of x_i and *best* (the strongest variable among the *active* variables). It also depends on the temperature T, a real positive number. The formula for the probability is $(1 + e^{\frac{\Delta}{T}})^{-1}$ and is implemented by the function $Accept(\Delta, T)$. Here Δ is the absolute value of the difference of strengths of x_i and *best*. The swap probability exponentially decreases with the increase of Δ. This makes sense: if x_i and *best* are very close in quality, then it sounds natural to try to substitute *best* by x_i in an effort to improve the global solution. On the other hand, if x_i is

much worse than *best*, then most probably it does not make sense to try to turn ON x_i instead of *best*. The swap probability also depends of the temperature T that scales Δ. The temperature is very similar to the one in Simulated annealing and is a way of controlling the balance between exploration and exploitation. When it is high, even swaps with large value of Δ are accepted with probability around 0.5 and the method is heavily biased towards exploration. Temperatures close to 0 force the algorithm to reject swaps with very small Δ and the algorithm becomes biased towards exploitation and local search.

The main loop of the CBNN solver is between lines 32 and 40 of Algorithm 4. We call the inner loop between lines 35 and 38 an *epoch*. It consists of a sequence of calls to the *Update* procedure. Every call executes the competition mechanism for a single variable that is chosen uniformly at random among all variables. During an epoch, for all calls to *Update* the temperature T is kept the same. $STEPS$ is a hyperparameter that denotes the number of calls to *Update* that happen in an epoch. It sounds reasonable for $STEPS$ to be at least several times larger than the number of variables. Such a value should guarantee that with high probability every variable is updated at least once for every temperature T. The outer loop of the CBNN solver is on line 34. What it does is to perform $ITERS$ epochs with different temperatures. For every subsequent epoch, the temperature is gradually decreased by the rule $T \leftarrow T \cdot \alpha$. The first epoch is the one with the highest temperature and the last epoch is the one with the lowest temperature. It was already mentioned that by decreasing T, the method shifts from exploration to exploitation. So the main loop gradually directs the CBNN solver from almost random exploration of the search space to local search in the most promising region found so far. The main loop requires several additional parameters: the number of different temperatures for which the loop is executed ($ITERS$), the initial temperature, and the temperature decrease coefficient α. Later in the section, we discuss strategies for selecting the parameters.

So far, the group-best variant of the general CBNN solver was described. In Chapter 4 it is proven that if a competition-based neural networks is given enough

time, then it converges to an optimal solution of the problem being solved. Key to this proof is that we can approximately say what is the probability of the network to be in a given state. Explicitly writing this probability for the group-best CBNN solver is hard. This motivates the introduction of the group-average CBNN solver. For this variant of the neural network it is possible to explicitly write an approximation of the probability of a state. Remember that in the group-best CBNN solver, in the *Update* procedure the current variable x_i competes with the strongest variable among the *active* ones in its group. In the group-average CBNN solver, x_i competes with the "average" variable, that is, with an imaginary variable whose strength, expressed by *OnCost*, is the average *OnCost* of the variables from the *active* set. If there is no luck and the *OnCost* of x_i is larger than this average *OnCost*, then x_i will be ON. Otherwise, it will be OFF. When converting the group-best CBNN solver to the group-average one, the only difference in the pseudocode is that line 21 is changed from $best \leftarrow min(\{OnCost(\vec{x}, j) \mid x_j \in active\})$ to $best \leftarrow avg(\{OnCost(\vec{x}, j) \mid x_j \in active\})$. On later stages of the optimization, when the temperature is relatively low, it is generally true that the *active* set either consists of a single variable or of variables with almost the same *OnCost*. This makes the difference between the group-best CBNN solver and the group-average one small. On earlier stages, when the temperature is high, the difference is more noticeable because there is more variation among the *active* set.

Another variant of the general CBNN solver is the group-random solver. In it, x_i competes with a random variable among the *active* variables instead of competing with the strongest one or the average. In the pseudocode, the modification is again only on line 21. The motivation is to improve the performance. The *active* set can be quite large, especially on earlier stages of the optimization, and the *OnCost* computation can also be expensive. When x_i competes against a random variable, only the *OnCost* of this variable needs to be computed and this saves time. The speed improvement from such a modification looks very substantial, but our experiments show that it is not that big. For obtaining comparable quality to the two other variants of the solver, the number of iterations per epoch needs to be increased and

this reduces the speed gains from the modification.

Several other modifications of the general CBNN solver are sometimes used in Chapter 5 to reduce the runtime of the method. Depending on the objective function, the computation of *OnCost* can be very expensive. As a concrete example, the objective function of p-MiniSum is $F(CF, FL) = \sum_{i,j,k} CF_{ij} \cdot FL_{jk} \cdot dist(v_i, v_k)$. Computing the *OnCost* requires the evaluation of F and takes $O(n^2 p)$ time. But we do not need to know the exact *OnCost*. What we need is the difference between the *OnCost* of the current variable and the strongest variable in its group. Assuming the current variable is CF_{ab}, we can write the expression for the *OnCost* as

$$F[CF_{ab} = 1](\vec{CF}, \vec{FL}) =$$
$$\sum_{k} 1 \cdot FL_{bk} \cdot dist(v_a, v_k) +$$
$$\sum_{i \neq a} \sum_{jk} CF_{ij} \cdot FL_{jk} \cdot dist(v_i, v_k)$$

For all variables in the group of CF_{ab} the second summand is the same and will cancel out when computing the difference in *OnCost*. This way, the second summand can safely be ignored and the computation now takes $O(np)$ time. When solving problems with CBNNs, a similar optimization can often be made and it significantly speeds up the solver without changing the quality of the final solution. The performance of the *OnCost* computation can sometimes be further improved for specific problems by maintaining appropriate data structures.

Another possible way of speeding up the general CBNN solver is to perform all updates *at once*. That is, we execute the *Update* procedure as described in Algorithm 4 but on line 30 instead of performing $x_i \leftarrow new\_value$, we just remember that the variable x_i needs to be set to the value *new_value*. Then, after the epoch is completed, we apply all the saved updates. With such a modification there is no benefit from updating the variables at random (line 36) because all calls to *Update* use the state of the variables at the start of the epoch. This way, during an epoch we can just iterate through the variables from the first one to the last one and call *Update*. The benefit from the modification is that all variables are

changed at once and we can precompute the *OnCost* and other information at the start of an epoch to speed up the *Update* procedure. This can substantially decrease the runtime, especially if the group sizes are large. Notice that the resulting method is not exactly equivalent to Algorithm 4. Performing all updates at once decreases the randomness in the operation of the neural network and can potentially create unwanted regularities. Our experiments show (see Chapter 5) that if the number of epochs is kept the same, then the solutions from the original CBNN solver are slightly better than the solutions from the modified solver described in this paragraph. Increasing the number of epochs for the modified solver is often enough to close the gap in solution quality. The original CBNN solver can be thought of as an approximation of a model of asynchronous parallel computation in which no two processors update at the same time (this is discussed in the next subsection). The described modification resembles a system of synchronous processors. If every variable is a separate processor and *Update* takes the same time for all of them, then starting all processors simultaneously at the beginning of an epoch results in the described modification. This type of multiprocessor systems do exist [103] and one of the reasons for trying out the modified CBNN solver was to see if it still returns solutions of good quality on such systems.

Apart from the possible modifications of Algorithm 4, another important aspect of applying CBNNs in practice is how to choose its parameters. What we need to specify is the initial temperature T, the temperature decrease coefficient α, the number of steps per epoch $STEPS$, and the number of epochs $ITERS$. In our experiments, we set the number of steps per epoch to be several times larger than the number of variables. Choosing the value of $ITERS$ depends on how much computational time we are ready to invest in solving the instance of the combinatorial optimization problem. The initial temperature T needs to be high enough to allow almost any variable flip with probability not much less than 0.5. To that end, either a problem-specific expression for T can be used, or the initial temperature can be empirically computed. For the empirical computation, the neural network is initialized at random and a dummy first epoch is executed that does not modify

the values of the variables (line 30 in Algorithm 4 is not executed). During this first epoch, the largest Δ passed to the *Accept* function is computed and it is used as the initial temperature T. We think that for choosing the value of α it is best to decide what the final temperature T_f of the network should be and compute α based on this (by final temperature it is meant the one of the last epoch). In Chapter 4 it is shown that for any given temperature and any state there is a certain probability for the network to be in this state. Moreover, only states that correspond to optimal solutions have significant probability. But this only holds if the temperature is "low enough". Exactly writing a formula for what is "low enough" is not trivial. For the problems in Chapter 5, a value of T_f equal to 0.1 works good. It can be seen that $T_f = T \cdot \alpha^{ITERS}$, so $\alpha = (\frac{T_f}{T})^{\frac{1}{ITERS}}$. Actually, the temperature decrease process by itself can be considered to be a hyperparameter. In all our experiments we use an exponential decrease: after every epoch the temperature is multiplied by α. Other possibilities exist. For example, in Simulated annealing there is the same idea of a temperature decrease schedule and for this method several different schedules were developed (exponential decrease seems to be the most popular of them, though).

The runtime of the CBNN solver is proportional to $STEPS \cdot ITERS$. In the next chapter, when we prove the convergence properties of CBNNs, we will see how these two hyperparameters relate to the quality of the solution found. At this point it suffices to say that for competition-based neural networks the ideal case is for *alpha* to be very close to 1 (meaning that $ITERS$ is large) and the number of steps per epoch $STEPS$ to be large as well. Remember that $STEPS \cdot ITERS$ is the runtime of the method and it is bounded. There are two main strategies to satisfy the constraint on the runtime: either make $ITERS$ large and $STEPS$ small or the opposite. The first option with many relatively short epochs often results in a better solution to the combinatorial optimization problem. A third option of balancing $STEPS$ and $ITERS$ and making them approximately equal is not a good idea. The reason for this is discussed in the next chapter.

When solving an instance of a problem with a competition-based neural network, it makes sense to perform several runs of the optimization algorithm. This is

because of the randomness in the solver. From empirical tests with real-world problems we noticed that CBNNs are quite robust and the quality of the final solution does not highly depend on the initial (random) solution. Because of this, performing multiple runs of the solver from different initial solutions and using the same hyperparameters is not that beneficial. It is better to perform several runs of the CBNN algorithm with different values of the hyperparameters ($STEPS$, $ITERS$, initial temperature, temperature decrease coefficient). Generally speaking, the best strategy is to perform a single very long run of the competition-based neural network. In practice, this takes too much time and a compromise is needed. Quickly obtaining a good solution requires selecting good values for the hyperparameters. Understanding all "forces" behind this compromise is hard, so performing multiple runs with different reasonable values of the hyperparameters often results in a better overall solution.

3.3 Remarks on the CBNN solver

Competition-based neural networks are a metaheuristic for combinatorial optimization and it is interesting to see where the model is positioned in the metaheuristic classification. The CBNN solver is a trajectory method because it operates on a single solution and not on a population of solutions. The solver performs small modifications of the current solution, so it is a local search based method and not a constructive one. It uses a simple single variable flip neighbourhood and has an explicit strategy for escaping from local mimima. The strategy resembles the one used by simulated annealing. We can see that the CBNN solver possesses all the properties of high-quality metaheuristics that were outlined in Section 2.2 but is a very simple one and more similar to the early work on metaheuristics. The neural network method does not have memory structures like in tabu search. This is expected because the idea of neural networks is to create meaningful global result from local computations, while the concept of memory structures in tabu search is by itself a global construct. Of course, we can find a way to combine CBNNs with ideas from other metaheuristics, but the thesis does not deal with this.

The optimization properties of CBNNs emerge from the joint operation of many independent computing units which is the main characteristic of neural networks. Every unit (neuron) computes the *OnCost* of some of the units in its group and decides its state based on these values (the way the decision is made is called a competition mechanism). The operations are local inside every group and are in a sense local inside every unit.

From all metaheuristics that were outlined in the first chapter, competition-based neural networks are closest as an idea to Hopfield networks, Boltzmann machines, simulated annealing, and the self-organizing approaches. Hopfield networks stand apart from the rest of the methods because, as a machinery for optimization, Hopfield networks are rather primitive. They are like a simple single run of a hill climbing heuristic. But the idea from Hopfield networks to solve combinatorial optimization problems by mapping variables to independent computing units that execute simple update procedures is useful and can be seen both in Boltzmann machines and competition-based neural networks.

Simulated annealing is a very general method. It is a framework for developing algorithms, but when solving a concrete problem, many of the details need to be invented. So it is not completely correct to compare simulated annealing to Boltzmann machines and competition-based neural networks. The last two methods are much more concrete. It is true that Boltzmann machines are basically simulated annealing applied to a specific graph with specific neighborhood structure. But this specific application is valuable, it has good properties and is a convenient model for a class of problems. In a similar way, competition-based neural networks can be considered to be ideas from simulated annealing that are applied to a specific graph with a specific neighborhood structure. Here the application is not as direct. First, simulated annealing always accepts moves that improve the value of the objective function. Competition-based neural networks accept such moves with probability that is smaller than 1. This aligns better with the idea of competing companies that can sometimes be lucky or not lucky. Second, in simulated annealing, Boltzmann machines, and Hopfield networks there is a global function that is used to decide

whether to accept a move. It has different names: energy, consensus. But the idea is the same. The probability of a given transition in these models depends on the difference of this global function between the candidate for next state and the current state. There is no such function in competition-based neural networks. There is a function that measures the solution quality and is related to the energy or consensus functions. But in CBNNs, when making a decision on whether to accept a given transition, the delta in quality is computed between the candidate and the best of its currently active competitors. This is a local property and is not expressed by a single global energy/consensus that is shared by all variables. It is well-known that if given enough time, simulated annealing converges to an optimal solution of the problem being solved. This proof is based on the existence of a single (simple) global function that dictates the probabilities of the state transitions. The same proof holds for Boltzmann machines because they use the same global function. But the proof does not hold for competition-based neural networks. It is true that they asymptotically converge to an optimal solution and we show this in the next chapter. But the way they move and converge to an optimal solution is different, especially in the beginning of the optimization.

CBNNs also have similarities with the self-organizing approaches to combinatorial optimization. A classical example of these approaches is the already introduced elastic net method. When applied to the Traveling Salesman Problem, the method fits around the input points a template that is like a rubber band. The competition mechanism of competition-based neural networks can also be seen as a type of a template. It forces the optimization procedure to choose for every group constraint of the input problem exactly one of the k possible options. This type of a template can be more expressive than the rubber band from the elastic net method. The template that is produced by the competition mechanism of CBNNs can model in a natural way the feasibility constraints of facility location problems. There is no need to introduce penalty terms in the objective function. Notice that penalty terms are necessary when modeling facility location problems for Hopfield networks and Boltzmann machines. As was already discussed, penalty terms should be avoided

because they artificially increase the complexity of the search space. Another similarity between CBNNs and self-organizing approaches is the winner-takes-all (or most) idea. This was called "rule of the jungle" in the description of CBNNs.

One of the strong points of neural networks is that they allow a relatively simple parallel implementation. Algorithm 4 describes competition-based neural networks as a sequential method. This is more convenient for classical computers and simplifies the analysis of the algorithm. Competition-based neural networks, like Hopfield networks, also have a straightforward parallel implementation. Under similar simplifying assumptions (no two units update at the same time) it can be shown that the sequential algorithm performs the same computations as a parallel version. As a consequence, the returned solutions are the same. The main drawback of the existing neural networks for combinatorial optimization is the poor quality of the final solution. This is why in the thesis we concentrate on analyzing the quality of the solutions, produced by competition-based neural networks, and not on the runtime of the method. It is shown that CBNNs find good solutions and, in this respect, are competitive to other metaheuristics. The problems of the existing neural network approaches are not in the idea of using massively parallel ensembles of neurons but in the specifics of their implementation of this idea.

Chapter 4

Analysis of CBNNs

In Chapter 5, competition-based neural networks are applied to a number of facility location problems to demonstrate their good empirical performance. The current chapter gives a theoretical justification of why competition-based neural networks can solve combinatorial optimization problems.

At any point in time, the values of the neurons of a CBNN define a configuration that maps to some (possibly infeasible) solution of the combinatorial optimization problem. During the operation of the neural network, it changes the values of the neurons. In this way, the neural network goes through a chain of states in the solution space. Every slot in the chain contains a configuration: the values of the neurons at this point in time. Transitioning from a slot to the next one corresponds to proposing a variable flip (this is done in the *Update* function of Algorithm 4). The flip may or may not be accepted, so it is possible for two consecutive slots to contain the same configuration. Randomness is involved in the acceptance criterion and the whole operation of the neural network is randomized.

At least two perspectives exist for looking at the operation of a competition-based neural network. We can think of it a single point trajectory method and for every position in the chain of states to consider the exact single solution that sits there. This perspective is convenient for gaining intuition about the explore-exploit strategy of CBNNs. But it is not convenient for proving the convergence properties of the neural network method because of the randomness in its operation. Another

possibility is to think of the neural network operation as of a Markov chain (see Appendix A for the definition of Markov chains). For every slot in the chain, we do not consider which single configuration is there but are interested in the probability distribution of the configurations that can sit there. This perspective is useful for proving that CBNNs converge to optimal solutions. For example, we can prove a result like this: the probability for a slot far enough in the chain to contain an optimal solution converges to 1 as the temperature T decreases.

One thing we want to mention is that, when talking about probabilities, there should be a large number of independent runs of the process. Probabilities are meaningful only when the "experiment" is repeated many times. When solving a problem, a single run of the neural network is performed (maybe 5-10 runs, but not billions for sure). This single run is the important one. It does not matter that from 1000 runs 999 are good if our single run is bad. This is something we should consider when dealing with randomized algorithms. From a positive side, if the probability of bad runs is extremely small, then we can believe that we are lucky and will never hit a bad one.

The chapter starts with two empirical experiments that highlight important properties of CBNNs. After the two experiments, we proceed with the proof of asymptotic convergence of competition-based neural networks.

4.1 Empirical properties as a single point method

For this experiment, the general CBNN solver (Algorithm 4) is applied to a small p-MiniSum instance that is derived from the Bulgarian road network. The instance is shown in Figure 4.1 and is one of the instances that are used in Section 5.1. The p-MiniSum problem has already been introduced in Section 1.2. A detailed explanation of how p-MiniSum is modeled for solving with a CBNN is omitted here. The goal of this section is to provide a high-level intuition about the operation of CBNNs and not to specifically deal with the p-MiniSum problem. The details of how to apply competition-based neural networks to the problem can be found in Section 5.1.

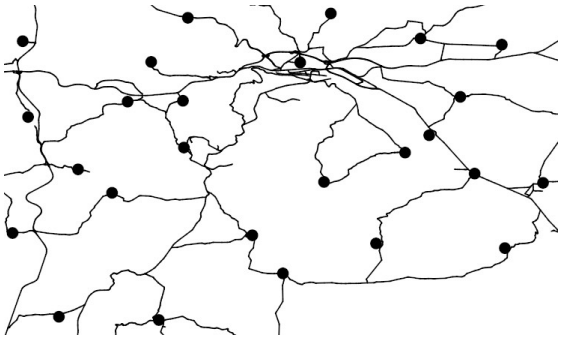

Figure 4.1: Example p-MiniSum instance. The black circles are the populated places.

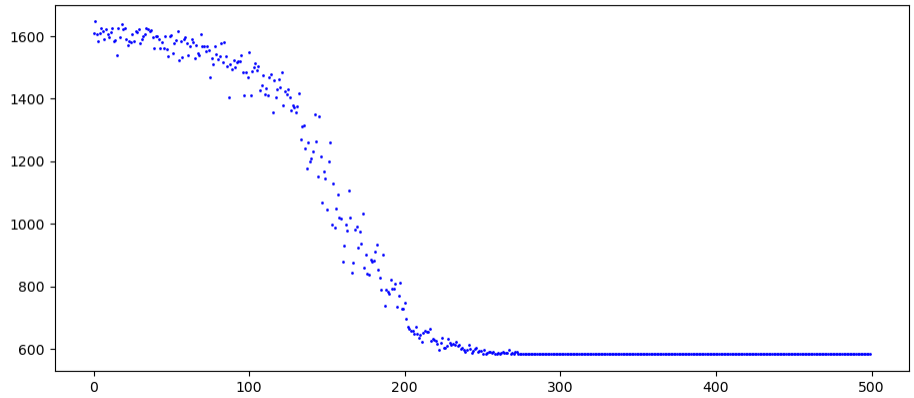

Figure 4.2: Value of the objective function of p-MiniSum during the optimization process. On the X axis is the epoch number. On the Y axis is the value of the objective function.

Figure 4.2 shows the value of the objective function of p-MiniSum after every epoch of the optimization. The general tendency is to decrease the objective function, but there is a lot of variance that gradually narrows as time progresses. This is a characteristic property of CBNNs: they allow not only moves that improve the value of the objective function but also moves that worsen it. The temperature of the neural network gradually decreases with time and the probability of worsening moves also decreases.

The graph in the figure can be divided into three distinctive stages. The first stage, up to approximately iteration 80, is similar to a random exploration of the search space. During this stage, the objective function is not significantly decreased. Next, between approximately iterations 80 and 200, follows a quick improvement of the solution. During this stage, the exploration side of the algorithm is gradually

reduced and the exploitation of the best-performing subcomponents is increased. The meaning of a *solution subcomponent* is problem-specific and a good property of CBNNs is that we do not need to specify what a subcomponent is. The neural network spontaneously finds them during its operation. They are like building blocks from which the whole solution can be built piece by piece. In the p-MiniSum example, one subcomponent can be a cluster of populated places that are serviced by the same facility. As time progresses, it can be noticed that such clusters are naturally formed by the neural network for the p-MiniSum instance and the clusters are relatively stable. In the beginning of the optimization, populated places often change their cluster, but later, the core populated places of a cluster move very rarely. The formation of subcomponents is somewhat similar to representation learning of the neural networks for classification. The third stage starts from approximately iteration 200 and can be thought of as final tuning of the solution. The general structure of the solution is already decided and here only small modifications are made. Maybe a populated place on the border moves to another cluster, or something similar. The stage is dominated by exploitation and is like hill climbing.

It is interesting to compare the operation of the CBNN to a classical local search (hill climbing) heuristic. Figure 4.3 shows how the objective function of p-MiniSum changes during several runs of a variant of hill climbing. The search repeatedly tries to assign a client to another warehouse or to change the location of a warehouse if this decreases the value of the objective function.

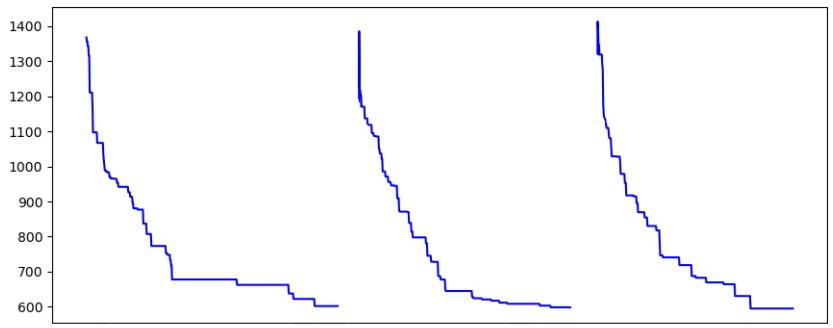

Figure 4.3: Values of the objective function for three runs of a local search heuristic. The X axis shows the iteration number. On the Y axis is the value of the objective function of p-MiniSum.

Clearly, the shape of the curves is very different from the one of the neural network. In Figure 4.3, we see immediate rapid decrease of the objective function that ends in a local minimum. There are no stages as in the CBNN graph from Figure 4.2. Hill climbing is "pure exploitation" of the region from which the optimization starts. It is difficult to see this from the graphs, but all three local search runs end up in solutions for which the value of the objective function is slightly worse than the optimal value of 583.344. The competition-based neural network finds the optimal solution. The results of the hill climbing heuristic are expected because hill climbing does not perform exploration and highly depends on the initial random solution. Every local optimum — good or bad, forms a basin of attraction. If the search starts in this basin, then it finds the corresponding local minimum. The test p-MiniSum instance is relatively simple, but for a complex problem, the quality of the local minima varies widely, and many of them are poor. No matter what the quality is, every local optimum forms a basin of attraction and there is no guarantee that the "good" ones cover a significant "area" of the search space. Hill climbing starts from a random point that falls in a random basin of attraction and this directly affects the quality of the solution found by the method. On the other hand, CBNNs perform exploration by occasionally accepting locally "bad" moves. This allows the CBNN to escape at least from some of the poor local optima and the neural network method does not depend that much on the initial solution. The cost of this is a much larger computational time in comparison to hill climbing. Additionally, as can be seen from Figure 4.1, during the first and last stages of a CBNN run, the improvement of the value of the objective function is small. But these two stages consume a significant amount of time. In practice, especially the first one, because in it many variables have value 1, which slows down the computation of *OnCost* in Algorithm 4.

The CBNN solver starts from a high temperature and gradually decreases it. It is interesting to see what happens if the solver is started with a low temperature. Its operation in this case is similar to hill climbing, but there are nuances. Figure 4.4 shows how the value of the objective function of p-MiniSum changes for one

run of a low temperature CBNN.

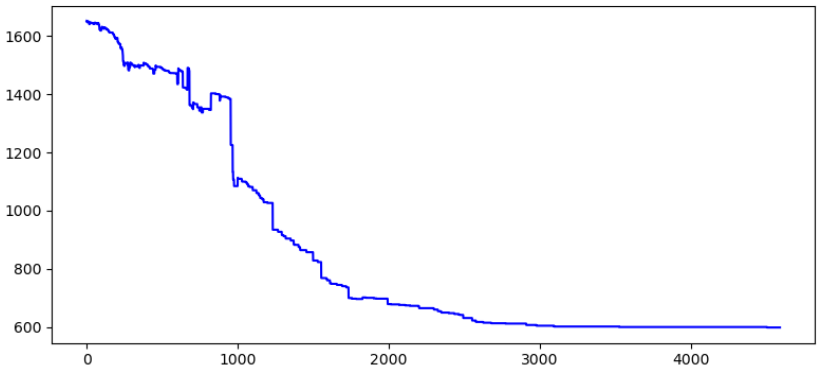

Figure 4.4: Value of the objective function of p-MiniSum for a low temperature CBNN run. The X axis shows the iteration number. On the Y axis is the value of the objective function.

It can be seen that the value of the objective function is not always decreasing. This is different from hill climbing and is a consequence of the competition mechanism and the independent local operation of the neurons. As an example, consider a client that is connected to three facilities. This represents an infeasible configuration, but it can occur during the operation of the neural network. Let the *OnCost* of the three connections be 1, 2, and 4. All of them are in a single group and the best one has value 1. If we now update the neuron with value 2, then, for a low temperature, it will become OFF (the neuron is not the best in its group and a low temperature means there is no luck). This results in an increase of the value of the objective function because now the contribution of this group of neurons is $\frac{1+4}{2}$ instead of $\frac{1+2+4}{3}$. This is a simple example. More complex relationships between neurons and groups can exist and they can cause growth of the objective function. The independence of the neurons is beneficial for the parallel implementation of the neural network. In comparison to a hill climbing heuristic, it additionally introduces more randomness in the solver. This may be beneficial for escaping from poor local optima. The last statement can cause controversy and it obviously depends on the problem. As some justification, for the test p-MiniSum instance, the low temperature CBNN finds a solution of value 598.323. This is slightly worse than the optimal value of 583.344 and it is better than the solutions from all three hill climbing runs.

Looking from left to right in Figure 4.2, it seems that the variance of the objective function of p-MiniSum first grows and then shrinks to 0. Figure 4.5 shows the standard deviation of the objective function on a separate graph.

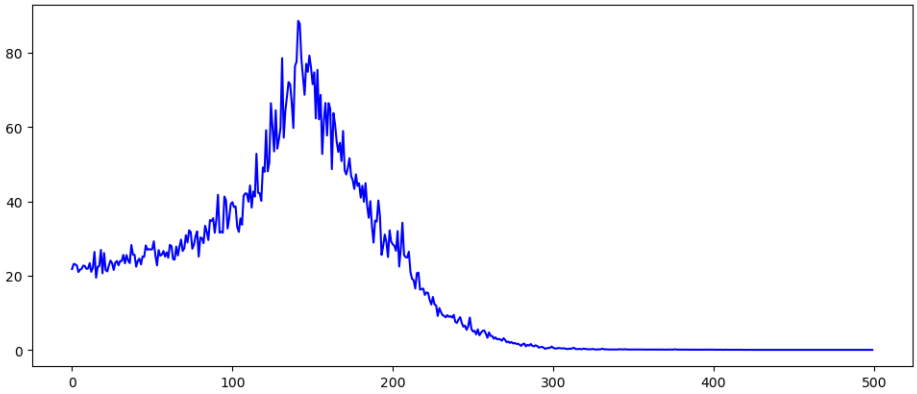

Figure 4.5: Standard deviation of the objective function of p-MiniSum during each epoch of the optimization. The X axis shows the epoch number. On the Y axis is the standard deviation.

Indeed, there is a spike at approximately epoch 150. This may look surprising. During the operation of the neural network, the temperature monotonically decreases. Intuitively, the standard deviation should be the highest in the first stage of the optimization. This is because there the temperature is the highest and the neural network is most likely to accept "risky" moves (it almost randomly explores the solution space). In later stages, as the temperature is lowered, the optimization shifts towards exploitation and the standard deviation should decrease. The explanation of why there is a spike is connected to the formation of solution subcomponents. Initially, the solution is random. The temperature is high, so bad moves are accepted with high probability, but very bad moves almost do not exist. When the solution is random, there is no structure and all variable flips are almost the same. No single one of them affects significantly the quality of the solution. But once good solution subcomponents start to form, then now there are very bad moves (variable flips) because breaking a subcomponent can drastically worsen the current solution. Up to around epoch 150, the temperature is still high enough to allow pretty bad transitions. The combination of the high temperature and the formation

of solution subcomponents causes the standard deviation of the objective function to grow. At around epoch 150, the temperature becomes low enough to prohibit most of the very bad variable flips. From then on, the standard deviation almost monotonically decreases.

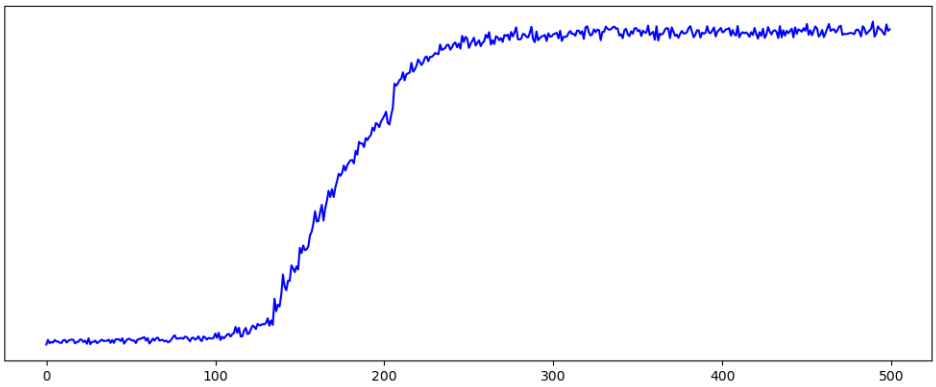

Figure 4.6: Average value of the proposed Δ in the *Accept* function of Algorithm 4. The Y axis shows the epoch number.

Figure 4.5 shows the standard deviation of the objective function during each epoch. This value is created by two forces: the proposed Δ in the *Accept* function of Algorithm 4 and the accept probability that depends on the current temperature. It is interesting to separately see what Δ values are proposed in the *Accept* function. This is illustrated in Figure 4.6. Instead of showing the Δ value for every call to *Accept*, in the figure, the average Δ per epoch is given. This is done to reduce the noise. There is a lot of variance in the individual Δ values, which makes it hard to see the general tendency in the graph. In the right end of the graph, the Δ values are high and flat. This is because there the competition-based neural network has already decided the general structure of the final solution. The middle part shows rapid growth of the delta: good solution components are formed and kept. The left part may look flat, but it actually shows a slow growth. This part is dominated by exploration and while some subcomponents start to form, most of the solution is still random. The process described in this paragraph illustrates the explore-exploit strategy of competition-based neural networks. They initially explore promising solution sub-configurations and, as time progresses, tend to keep the ones that combine to the best possible solution. This is different from the memory structures

of Tabu search, but can be thought of as a kind of a probabilistic memory. The graphs in this section also demonstrate the importance of the temperature schedule. After every epoch, the temperature changes just slightly. But it can be seen how drastically such a change affects the behavior of competition-based neural networks, especially around epochs 100 - 150.

As noted several times, a competition-based neural network occasionally accepts transitions that reduce the quality of the current solution. The acceptance probability depends on two factors, the temperature and the proposed reduction of the quality of the solution. The function *Accept* from Algorithm 4 is the one that tells whether to perform a worsening transition. Figure 4.7 shows what percentage of the proposed worsening transitions are actually accepted.

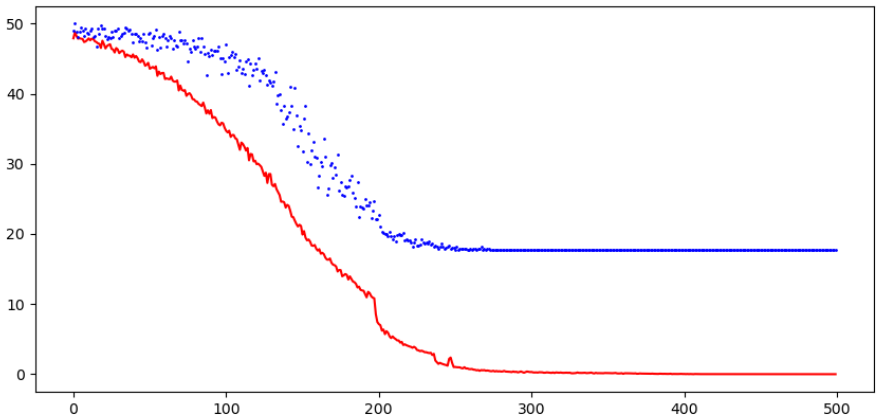

Figure 4.7: The red line shows the percentage of the proposed worsening transitions that are accepted. The X axis is the epoch number, the Y axis is the percentage. The blue line in the figure gives the value of the objective function of p-MiniSum. The blue line is there for convenience, its Y axis is just scaled to fit and has no meaning as percentage.

Initially, when the temperature is high, around 50% of the proposed transitions are accepted. The acceptance probability is $\frac{1}{1+e^{\frac{\Delta}{T}}}$ and it is easy to verify that its limit as T grows is $\frac{1}{2}$. In the right side of the graph, the probability decreases to 0. This is the limit of the acceptance probability when T decreases to 0. Apart from that, the graph does not seem to be very informative: the probability decreases with time, but is not clearly mapped to the 3 stages that are present in the objective function graph.

4.2 Empirical properties as Markov chains

A competition-based neural network manages a set of n binary variables (equivalently, n neurons, or units). Lets denote by $\vec{x_i}$ the binary vector that contains the values of the variables at a given point in time. The operation of the competition-based neural network can be described as a sequence of such vectors $\vec{x_0}, \vec{x_1}, ...$. The initial vector in the sequence is randomly chosen. After that, every new vector is computed by applying the *Update* procedure to one of the variables of the previous vector (Algorithm 4 from Section 3.2 shows in pseudocode the CBNN solver).

Because of the randomness in the network operation, if we concentrate on the exact value of every individual vector $\vec{x_i}$, then it is hard to analyze the global performance. Instead, the probability distribution of every $\vec{x_i}$ over its 2^n possible values can be considered. This distribution is denoted as X_i. If we substitute every $\vec{x_i}$ by such a probability distribution, then the randomness goes away. Every X_i can be uniquely computed from X_{i-1}. What we will empirically see here and prove later is that this sequence of probability distributions converges to a single probability distribution π. If the network runs long enough, the vector $\vec{x_i}$ at time point i is drawn from the distribution π. Further, in this distribution, the vectors that correspond to suboptimal solutions have almost zero probability. So, if we pick a vector $\vec{x_i}$ far enough in the sequence, then it will correspond to an optimal solution because only such vectors have nonzero probability in π. This is the intuition behind the claim that, in probabilistic sense, competition-based neural networks solve combinatorial optimization problems.

Assume for now that the temperature T of the competition-based neural network is fixed. When looking at the operation of the network as a sequence of vectors, it is easy to see that this is a Markov chain: the vector x_i depends only on the vector x_{i-1} (information about Markov chains and their properties can be found in Appendix A). Formally, $P(x_i|x_{i-1}, x_{i-2}, ..., x_0) = P(x_i|x_{i-1})$ and the process is memoryless. Markov chains are well-studied and known results about them can be used to analyze CBNNs.

In Markov chains, the usual terminology is to use the name states for our set

of 2^n possible values of $\vec{x_i}$. It is convenient to naturally map this set of possible values to the integers from 0 to $2^n - 1$ so that we can assume $\vec{x_i}$ is an integer whose binary encoding gives the values of the neurons of the network. In Markov chains, there is also a state transition probability matrix A. Each element $A[\vec{a}][\vec{b}]$ of the matrix contains $P(\vec{b}|\vec{a})$, the probability to transition from $\vec{a}$ to $\vec{b}$ in the chain. For competition-based neural networks, the matrix A is implicitly given by the *Update* procedure from Algorithm 4. For a state $\vec{a}$ and a variable, the *Update* procedure flips the value of the variable with certain probability. By writing these probabilities in A (and dividing by the total number of variables, because in Algorithm 4 the variable itself is randomly chosen), we get the state transition probability matrix. Notice that this matrix is huge. Its size is $2^n \times 2^n$ where n is the number of variables. Additionally, the matrix is very sparse. Nonzero values in it correspond to variable flips (or staying in the same state), so the number of nonzero values is on the order of $n \cdot 2^n$.

In this section, we perform three experiments. The goal of the first one is to empirically see that the sequence $X_0, X_1, ...$ of probability distributions converges and that, assuming the temperature is low, in the limit distribution π only optimal solutions have significant probability. The second experiment investigates how quickly the sequence of X_i converges. It may not be obvious why we work with sequences of probability distributions if what we care about is a single run of the competition-based neural network. The goal of the third experiment is to provide additional insight into this.

In the experiments from this section, the p-Defense-Sum Problem is used instead of the usual p-MiniSum Problem. This is because the size of the state transition probability matrix grows very quickly and we were not able to create a meaningful p-MiniSum instance for which the size of the matrix is manageable. The p-Defense-Sum problem is introduced in Section 5.3. In the problem, we are given a graph representing a road network. The goal is to position p facilities in the vertices while maximizing the sum of pairwise distances between the facilities. One way to think of this is that we are locating military objects. Spreading them

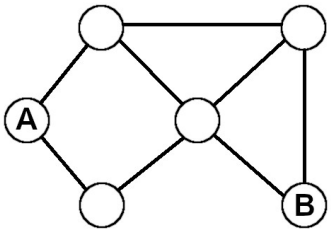

Figure 4.8: Example p-Defense-Sum instance in which 2 facilities need to be located. It is optimal to place the two facilities in vertices A and B. The distance between them is 3. For any other pair of vertices, the distance is less than 3.

in space makes it hard for the enemy to capture and establish stable control over the facilities. Figure 4.8 shows the p-Defense-Sum instance that is used for the experiments. The exact CBNN-compatible model of the problem is not needed for this section. It is described in detail in Section 5.3 that is devoted to p-Defense-Sum. We only want to say that the model of the instance from Figure 4.8 has $2 \cdot 6$ variables, resulting in a state transition probability matrix of size 4096 x 4096.

Using a matrix notation, the probability distribution X_i, a row vector, can be written as $X_{i-1} \cdot A$. By unrolling this, we get $X_i = X_0 \cdot A^i$. CBNNs can be made to choose one specific hardcoded value for the initial state $\vec{x_0}$, so the initial probability distribution X_0 is almost all-zero vector except for having 1 at index $\vec{x_i}$. The first experiment is to take such a vector X_0 and verify that $X_0 \cdot A^i$ converges.

Table 4.1 shows the probability distributions $X_i = X_0 \cdot A^i$ for several different i and temperatures T (i has at least three equivalent interpretations: length of the chain, number of calls to *Update*, or runtime of the neural network). In all cases, the initial probability distribution X_0 is set to a vector that has 1 at index 4095 and 0 at all other indexes. As expected, for any temperature, the last two columns are very similar. The larger i is, the more the corresponding column is similar to the last column. This hints that X_i converges when increasing i. Later in this chapter it is proven that X_i always converges to the same vector π no matter what the initial distribution X_0 is. The fact that the neural network converges is important because it gives a clue that the algorithm does something meaningful. Convergence to the same vector π is also important because it means that the quality of the solution, returned by a CBNN, ideally is independent of the initial random solution. This is in

T = 10					
$\vec{x}$	X_{10}	X_{40}	X_{100}	X_{200}	X_{2000}
2049	0.000001	0.000477	0.000565	0.000565	0.000565
96	0.000001	0.000477	0.000565	0.000565	0.000565
2053	0.000005	0.000485	0.000550	0.000550	0.000550
224	0.000005	0.000484	0.000548	0.000548	0.000548
2055	0.000022	0.000509	0.000550	0.000550	0.000550
3585	0.000022	0.000505	0.000547	0.000547	0.000547
368	0.000020	0.000451	0.000491	0.000491	0.000491
T = 2					
$\vec{x}$	X_{10}	X_{40}	X_{100}	X_{200}	X_{2000}
2049	0.000008	0.006807	0.008222	0.008227	0.008227
96	0.000008	0.006807	0.008222	0.008227	0.008227
2053	0.000052	0.005250	0.005904	0.005906	0.005906
224	0.000052	0.005196	0.005870	0.005872	0.005872
2055	0.000217	0.004238	0.004483	0.004483	0.004483
3585	0.000195	0.003941	0.004290	0.004292	0.004292
368	0.000173	0.002643	0.002819	0.002820	0.002820
T = 0.5					
$\vec{x}$	X_{10}	X_{40}	X_{100}	X_{200}	X_{2000}
2049	0.000085	0.149850	0.251901	0.261692	0.262021
96	0.000085	0.149850	0.251901	0.261692	0.262021
2053	0.000396	0.034137	0.036915	0.036735	0.036724
224	0.000384	0.032522	0.036176	0.036417	0.036428
2055	0.001080	0.007670	0.005849	0.005641	0.005634
3585	0.000903	0.006081	0.005450	0.005491	0.005634
368	0.000878	0.001671	0.000956	0.000900	0.000898
T = 0.2					
$\vec{x}$	X_{10}	X_{40}	X_{100}	X_{200}	X_{2000}
2049	0.000123	0.263950	0.466142	0.485568	0.486573
96	0.000123	0.263950	0.466142	0.485568	0.486573
2053	0.000520	0.026631	0.006222	0.003436	0.003285
224	0.000497	0.023608	0.005175	0.003352	0.003282
2055	0.001261	0.002777	0.000238	0.000035	0.000026
3585	0.001059	0.001728	0.000099	0.000028	0.000026
368	0.001082	0.000544	0.000007	0.000001	0.000000

Table 4.1: Convergence of CBNNs. $X_i = X_0 \cdot A^i$ is computed for $i \in \{10, 40, 100, 200, 2000\}$. The initial distribution X_0 is an all-zero vector except for index $4095 = 2^{12} - 1$ where there is 1. This corresponds to starting the competition-based neural network from a very symmetric state in which all variables have value 1. The dimensionality of X_i is 4096. For readability, the probabilities of only 7 of the 4096 states are given in the table (the column $\vec{x}$ shows which ones). Among them are the two optimal solutions, 2049 and 96. In fact, there is a single optimal solution if we look at it as a set. But the neural network returns an ordered pair and this is why there are two optimal states. The state transition probability matrix A depends of the temperature T of the CBNN. In the table, $X_i = X_0 \cdot A^i$ is computed for 4 different values of T.

contrast to local search, for which the quality of the final solution strongly depends on the initial one. Of course, when solving a problem, we do not want to depend on things like being lucky enough to start from a "good" point. From table 4.1 it can be guessed that when the temperature T is 0.2, in π the probability of state 96 is approximately 0.49. This means that if the CBNN is stopped after performing 10^{12} iterations, with probability 0.49 it will be in state 96 representing an optimal solution. 10^{12} is just a very large number and it is used here because we do not want to talk about the speed of convergence yet. Running the neural network for so long to solve such a simple instance is not practical, neither is it guaranteed that 10^{12} is enough time for the CBNN to converge on large instances.

Apart from the fact that the sequence $X_0, X_1, ...$ of probability distributions converges, it is interesting to see what is its limit distribution. For temperature $T = 10$, a high temperature, the limit distribution π is close to uniform. From the perspective of solving combinatorial optimization problems, such a distribution gives us nothing. It is like randomly generating a solution and hoping to hit an optimal one. The situation is different when $T = 0.2$. For this temperature, in π only states 2049 and 96 have a significant probability and both of them represent optimal solutions. It can be guessed that for high temperatures π is uniform and as the temperature is lowered, the probability concentrates on states representing optimal solutions. In the limit case when T approaches 0, only optimal solutions have non-zero probability. From these considerations it can be seen how we will later prove asymptotic convergence to an optimal solution: first prove convergence of CBNNs to some distribution π, then prove that in π the probability of non-optimal solutions is close to 0.

The speed of convergence is another property of the CBNN solver that is important in practice. Sadly, the bounds for the speed are not satisfactory (they are larger than the size of the solution space). Competition-based neural networks use the mechanism of gradually lowering the temperature exactly because of this. There are heuristic reasons to believe that such a mechanism speeds up the process of finding a good solution. In addition, we should remember that the goal of

competition-based neural networks is not to find the exact optimum but is to find a good approximate solution. Convergence to the optimum is not necessary for this.

The state transition matrix A is called a stochastic matrix: a matrix in which every row represents a probability distribution. From Markov chain theory it is known that chains with certain properties converge to a unique distribution (vector) π. Later, these properties are defined and we show that they hold in our case. For this vector π it is true that $\pi \cdot A = \pi$. Such vectors are called eigenvectors of the matrix A corresponding to eigenvalue 1. The set of eigenvalues of A is called spectrum and it can be used to bound the speed of convergence. Appendix A introduces the terminology and results that are mentioned in this paragraph. For Markov chains, the speed of convergence is formalized by the notion of mixing time. Intuitively, the mixing time is a power s for which every component of $X_0 \cdot A^s$ differs from the corresponding component of π by at most $\frac{1}{4}$ for any initial distribution X_0 (see Definition A.0.7). The constant $\frac{1}{4}$ is an arbitrary one. It can be $\frac{1}{10}$, for example. This only scales the mixing time. There is a theorem saying that the spectrum of the stochastic matrix A contains the eigenvalue 1 and all other eigenvalues in the spectrum are smaller than 1 by absolute value (the result is a consequence of the Perron-Frobenius theorem; see Theorem A.0.2). This is actually the reason why Markov chains converge to π: its eigenvalue is 1 and it dominates all other. If we denote by μ the absolute value of the second-largest eigenvalue, it is known that the mixing time is proportional to $\frac{1}{log(\frac{1}{\mu})}$. Intuitively, this is because the s-th power of the matrix A can be written as a sum of terms of the form: eigenvalue of the matrix to the s-th power multiplied by something. The largest eigenvalue 1 to any power is 1, but the powers of eigenvalues that are less than 1 by absolute value diminish when s grows. Formal description of the above results is in Appendix A. For now, what we care about is that the speed of convergence depends on the second-largest by absolute value eigenvalue of A, denoted as μ. If the CBNN performs around $\frac{1}{log(\frac{1}{\mu})}$ updates, this should get it pretty close to the stationary distribution π. It should be mentioned that eigenvalues are generally complex numbers and eigenvectors consist of complex numbers. For a stochastic matrix, the largest eigenvalue by absolute

value is 1, but the other eigenvalues can be complex numbers. In this section, we always talk about (and write) their absolute values because this is what matters for the speed of convergence of Markov chains.

T = 10		
1.000000	0.932283	0.926618
T = 2		
1.000000	0.969312	0.94976
T = 0.5		
1.000000	0.998949	0.967023
T = 0.2		
1.000000	0.999998	0.971709

Table 4.2: The three largest by absolute value eigenvalues of the matrix A for different temperatures T (the largest eigenvalue is always 1). The second-largest eigenvalue gives the speed of convergence of the CBNN.

Table 4.2 gives the three largest by absolute value eigenvalues of the state transition probability matrix A for different temperatures. It can be seen that the second-largest eigenvalue grows when the temperature is decreased. For example, the second-largest eigenvalue for $T = 10$ is approximately $\mu = 0.932283$, giving a mixing time of $\frac{1}{log(\frac{1}{\mu})} \approx 33$. For $T = 0.2$ the same computation shows that the mixing time is ≈ 1150000. High temperatures T correspond to almost uniform distribution π over the solutions that gives us no information about the optimal solution of the problem. But the low value of the mixing time says that the competition-based neural network "finishes" quickly. On the other hand, low temperatures correspond to distributions in which only optimal solutions have significant probability. In this way, a CBNN running with a low temperature optimally solves the combinatorial optimization problem. But for low temperatures, the mixing time becomes huge. This can be called a *temperature trade-off* of a CBNN: high temperatures are not informative, but the computation is quick. Low temperatures give the optimal solution, but the computation is slow. The goal of gradually decreasing the temperature of a competition-based neural network from a high value to a low value is to combine the good properties of both worlds and reach a nearly-optimal solution in reasonable time.

The last experiment for this section illustrates the link between the CBNN algo-

rithm and the probability distributions that we talk about. The CBNN algorithm is run with temperature $T = 0.8$. For this temperature, the mixing time is ≈ 390. The algorithm is left to perform 3000 updates, several times more than the mixing time. After this, the CBNN is left to perform additional 20000 updates that are stored to compute the empirical probabilities of the states. As mentioned several times, probabilities make sense when something is performed repeatedly. The 20000 updates "materialize" the distribution to which the neural network converges.

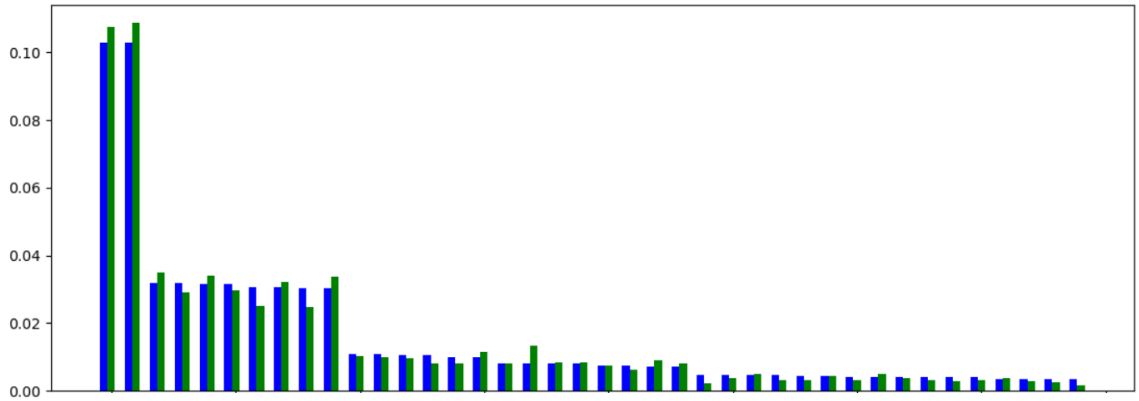

Figure 4.9: Expected and empirical distribution over the states of the CBNN for temperature $T = 0.8$. The blue bars on the left represent the expected distribution (eigenvector for eigenvalue 1). The green bars on the right are the seen empirical probabilities. To make the graph readable, only the 40 most probable states from the expected distribution are taken and their empirical and expected probabilities are shown (in decreasing order of the expected probability).

Figure 4.9 shows the results of the experiment. The empirical distribution that we get and the expected distribution π obtained from solving $\pi \cdot A = \pi$ are quite close. The two tallest bars are the two optimal states, 96 and 2049. The probability of each one of them if a little bit more than 10%. From the 20000 iterations of the CBNN it ends up in one of these two states in approximately 4400 of the iterations. This means that the CBNN passes through an optimal solution of the instance 4400 times during the 20000 iterations. As the temperature decreases, the probability mass concentrates on the two optimal states and the competition-based neural network will stay even more often in them. In this sense, the network solves the combinatorial optimization problem.

4.3 CBNNs with restarts

We already discussed in the previous section that the speed of convergence of a CBNN depends on the second-largest by absolute value eigenvalue of the state transition probability matrix. The value is instance-specific and is outside of our control. This section describes a modification of the general CBNN solver for which we can control the second-largest by absolute value eigenvalue.

During the operation of a CBNN, a chain of states is created. As we empirically saw in the previous section, the probability distribution of the elements of this chain converges to a distribution π. This is true no matter what the initial state is. But how quickly the sequence of distributions converges to π depends on the initial state. Intuitively, the neural network can get stuck in a bad region of the solution space and a huge amount of time will be necessary to escape from this region. In such a case, it is better to just randomly jump to a new point. The described intuition is the motivation for CBNNs with restarts. The method performs identical operations as regular competition-based neural networks with a fixed temperature, but at each step, with a certain small probability p, it can jump to an arbitrary state. Ideally, we want to jump only when the neural network is stuck in a bad region. The solver does not know for sure if the current region is bad, so it always jumps with some probability. This somewhat resembles performing multiple runs of a hill climbing heuristic.

Using the terminology from the previous section, the state transition probability matrix A_r of CBNNs with restarts can be written as $(1-p) \cdot A + p \cdot E$. The matrix A is the state transition matrix of regular CBNNs and E is a matrix of all $1/|S|$ where $|S|$ is the size of the state space. The matrix E says that we can jump with equal probability from any state to any other state.

In Section 4.5, it is proven that the second-largest by absolute value eigenvalue of A_r is at most $1-p$. This means that the mixing time of CBNNs with restarts is $O(1/log(\frac{1}{(1-p)}))$. Large value for the probability p makes the mixing time small and the neural network finishes quickly. On the other hand, a large p makes the stationary distribution close to a uniform one. Remember that a uniform distribution

is useless for solving combinatorial optimization problems. Finding an acceptable value for p is not trivial: it needs to give a good enough for our purposes mixing time and still to guarantee that the neural networks arrives at a nearly optimal state with a high probability.

Tables 4.3 and 4.4 show the results of two experiments that are similar to the ones from the previous section. In the experiments, the CBNN with restarts is run with a temperature $T = 0.2$ and restart probability p equal to 0.1 and $\frac{1}{|S|}$. The p-Defense-Sum instance is the same as the one from the previous section. Its model has 12 variables, so $|S| = 4096$.

$\mathbf{p = 0.1}$		
1.000000	0.900000	0.874538
$\mathbf{p = \frac{1}{\lvert S\rvert}}$		
1.000000	0.999755	0.971472

Table 4.3: The three largest by absolute value eigenvalues of the state transition probability matrix A_r of a CBNN with restarts for restart probabilities 0.1 and $\frac{1}{|S|}$.

$\vec{x}$	π for $p = 0.1$	π for $p = \frac{1}{\lvert S\rvert}$
2049	0.035110	0.481761
96	0.035110	0.481761
2053	0.008039	0.003590
224	0.007482	0.003520
2055	0.002646	0.000076
3585	0.002253	0.000061
368	0.001821	0.000020

Table 4.4: The stationary distribution π of a CBNN with restarts for restart probabilities 0.1 and $\frac{1}{|S|}$. For readability, the probabilities of only 7 of the 4096 states are shown. Among them are 96 and 2049, the two optimal solutions.

From the tables it can be seen that the second-largest eigenvalue in both cases is equal to $1-p$. In the $p = 0.1$ case, in the stationary distribution π, the probability of the two optimal states 96 and 2049 is not substantially larger than the probability of the other states. This means that the CBNN with $p = 0.1$ is not useful for solving the p-Defense-Sum instance. On the other hand, when $p = \frac{1}{|S|}$, the total probability of ending up in one of the two optimal states is close to 97%, so it can be said that the CBNN optimally solves the given instance. It is interesting to compute

the mixing time for $p = \frac{1}{|S|}$. In this case, $\mu = 1 - \frac{1}{|S|}$ and we get $\frac{1}{log(\frac{1}{\mu})} \approx |S|$. If the CBNN with restarts with $p = \frac{1}{|S|}$ performs $|S|$ steps, then it will find with high probability the optimal solution.

CBNNs with restarts are attractive because they are relatively simple: they run at a constant temperature and we can control the speed of convergence. The caveat is that the stationary distribution can be very sensitive to the value of p. By carefully balancing the temperature and the value of p, we can prove results of the type: if $|S|$ is the size of the state space, then in time $O(|S|^2)$ a CBNN with restarts will find an optimal solution with high probability. We do not consider such results to be useful: in time $|S|$ we can trivially solve the problem by iterating through every possible solution. CBNNs with restarts are convenient for some proofs, but do not seem to be beneficial when solving real problems.

4.4 Proof of asymptotic convergence

The goal of this section is to show that if a competition-based neural network is given enough time, then it finds an optimal solution with probability 1. As mentioned multiple times in the chapter, during its operation, the CBNN solver (Algorithm 4) creates a Markov chain. Markov chains are a very convenient tool for analyzing processes. Our plan is to use well-known facts about them to first show that the chain of a CBNN converges and then to show that in its stationary distribution only optimal states have significant probability. Appendix A introduces the properties of Markov chains that are necessary for the proofs.

CBNNs have two properties that complicate the proof of convergence. The first one is created by groups in which all variables (neurons) have value 0. If a state contains such a group, then it represents an infeasible solution. The problem is that in the CBNN solver, we never turn OFF all variables in a group. This can be seen from the if statement on line 17 of the general CBNN solver: if the *active* set is empty, then the variable is always set to 1. Consider what happens to a state with a group that is completely OFF. We can get out of this state with nonzero probability by turning ON one of the variables in the group. But we can not get back into

this state because we never turn OFF all variables. If the CBNN starts in such a violating state, then the neural network will get out of it and never return back. Instead of dealing with this type of cases, it is simpler to remove all states that contain a group that is completely OFF. From now on, we assume that the state space S of the CBNN Markov chain does not contain such states. Notice that once we get inside this cleaned space S, then we always travel inside it. And it is trivial to find an initial state that does not contain a completely OFF group (turn everything ON, for example). So cutting out the states with a completely OFF group does not change at all the CBNN solver (assuming we do not initialize the neural network with such an infeasible solution).

The Markov chain of a CBNN can be described by its state transition probability matrix A (see Definition A.0.2). This is the same matrix as the one from Section 4.2 and we use the same encoding of the binary vectors of the state space S as natural numbers (the numbers need to be shifted to account for the removal of states that contain completely OFF groups; this is a technical detail). The element $A[\vec{v_i}, \vec{v_j}]$ contains the probability of transitioning from $\vec{v_i}$ to $\vec{v_j}$. This probability is implicitly given by the *Update* procedure of the solver (Algorithm 4). The second property that complicates the proof of convergence is that the transition probability depends of the temperature T. In terms of Markov chains, the chain is not homogeneous (see Definition A.0.1). From now on, we assume that the temperature T is fixed and so the Markov chain becomes homogeneous (the matrix A is constant). Notice that in the real CBNN solver, the matrix A is piecewise constant. The proof for constant temperatures can be extended to piecewise constant temperatures and the convergence results for the fixed temperature case also hold for the complete CBNN solver that gradually decreases T. Dealing with piecewise constant temperatures introduces technical details that just make it harder to see the essence of the convergence proof. This is why here only the proof for the fixed temperature case is given.

Explicitly writing the state transition probability matrix A of the CBNN Markov chain is tedious. Instead of writing formulas for the entries of the matrix, the

content of the matrix is described below in text by tracing the code from the *Update* procedure of the general CBNN solver (Algorithm 4). The important property to remember is that a competition-based neural network performs single variable flips. Intuitively, the probability of a flip is large if it improves the solution. Otherwise, the probability is small. Assume $\vec{v_i}$ and $\vec{v_j}$ are two states. For the probability $A[\vec{v_i}, \vec{v_j}]$ it holds:

- If $\vec{v_i}$ and $\vec{v_j}$ differ in the values of at least two variables, then $A[\vec{v_i}, \vec{v_j}] = 0$. This is because the general CBNN solver only performs single variable flips.

- If $\vec{v_i} = \vec{v_j}$, then $A[\vec{v_i}, \vec{v_j}] = 1 - \sum_{\vec{v} \neq \vec{v_j}} A[\vec{v_i}, \vec{v}]$. That is, if the solver did not decide to transition to another state, then it stays in the current state.

- If $\vec{v_i}$ and $\vec{v_j}$ differ in the value of exactly one variable x, then the probability $A[\vec{v_i}, \vec{v_j}]$ can be written as $\beta \cdot \widetilde{A}[\vec{v_i}, \vec{v_j}]$. $\widetilde{A}[\vec{v_i}, \vec{v_j}]$ is the transition probability from the *Update* procedure of Algorithm 4. β is 1 divided by the number of variables and it multiplies $\widetilde{A}[\vec{v_i}, \vec{v_j}]$ because the CBNN solver randomly selects the variable to update (see line 36 of Algorithm 4). For $\widetilde{A}[\vec{v_i}, \vec{v_j}]$ there are two similar cases depending on whether $\vec{v_i}[x]$ is 0 or 1.

  1. *Transitioning from* $\vec{v_i}[x] = 0$ *to* $\vec{v_j}[x] = 1$*:*

     For the current configuration $\vec{v_i}$, let *me* be the *OnCost* of the variable x and let *active* be the set of variables from the group of x that are in ON state. The names *me* and *active* are directly taken from Algorithm 4. Notice that *active* can not be empty because we explicitly removed from the set of states the configurations that have a completely OFF group. Following the naming from Algorithm 4, let *best* be the *OnCost* of the variable that competes against x. How exactly *best* is computed depends on which variant of the CBNN solver is used. For the group-best CBNN solver, *best* is the smallest *OnCost* among the variables in the *active* set (remember that the CBNN solver minimizes the objective function). For the group-average CBNN solver, *best* is the average *OnCost* among the variables in the *active* set and corresponds to an imaginary variable.

If $me > best$, then $\widetilde{A}[\vec{v_i}, \vec{v_j}] = \frac{1}{1+e^{\frac{\Delta}{T}}}$ where $\Delta = |me - best|$. That is, the current variable x is worse and the only reason for x to transition to ON state is the luck factor. If the temperature T is close to 0, then $\widetilde{A}[\vec{v_i}, \vec{v_j}] = \frac{1}{1+e^{\frac{\Delta}{T}}}$ is also close to 0. In the other case, $\widetilde{A}[\vec{v_i}, \vec{v_j}] = 1 - \frac{1}{1+e^{\frac{\Delta}{T}}}$.

2. *Transitioning from $\vec{v_i}[x] = 1$ to $\vec{v_j}[x] = 0$:*

   This case is completely analogous to the previous one and a detailed description of it is omitted.

For showing that a Markov chain has a stationary distribution (see Definition A.0.3), it is enough to prove that the chain is irreducible and aperiodic (see Theorem A.0.1).

Lemma 4.4.1. *The Markov chain of a CBNN is irreducible.*

Proof. A Markov chain is irreducible if any state can be reached with nonzero probability from any other state in a finite number of transitions (see Definition A.0.4). Competition-based neural networks allow with nonzero probability single variable flips. It is clear that by using such flips, any state can be reached from any other in at most $2 \cdot n$ transitions where n is the number of variables. One way to achieve this is to turn ON all variables in the current state and then to turn OFF the variables that need to be OFF in the final state.

Notice that states that contain completely OFF groups were explicitly removed from the state space. Such states are not reachable from others and if we did not remove them, then the Markov chain formally would not have been irreducible. □

Lemma 4.4.2. *The Markov chain of a CBNN is aperiodic.*

Proof. Definition A.0.5 states what is an aperiodic Markov chain. Intuitively, this property means that the chain does not alternate between several groups of states. As an example, consider the chain from Figure 4.10. Starting from state A, we can return back to it after $2, 4, 6, ...$ transitions but not after an odd number of transitions. Such a chain cannot converge to a single distribution because it infinitely alternates between the states A and B. For every even index in the chain, the probability to be in A is nonzero. For every odd index, the probability drops to 0.

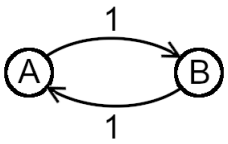

Figure 4.10: A Markov chain with a period of 2. In the chain, with probability 1 we move from A to B and from B to A. Notice that if we start from A, then we return to this state only on even moves.

Lemma 4.4.1 shows that the Markov chain of a CBNN is irreducible. The Markov chain additionally has a state $\vec{v_i}$ for which $A[\vec{v_i}, \vec{v_i}] > 0$ (for example, this obviously holds for a state that is an optimal solution). The fact that the CBNN Markov chain is aperiodic follows as a consequence of these two properties (see Lemma A.0.1). □

Lemma 4.4.3. *The Markov chain, corresponding to a competition-based neural network, has a stationary distribution.*

Proof. The definition of a stationary distribution is given in Definition A.0.3. It intuitively says that if a Markov chain has a stationary distribution π, then $ini \cdot A^k$ converges to π no matter what the initial distribution ini is. Said in another way, for every state $\vec{v_i}$ and index idx that is large enough it is true that the probability for the neural network to be in state $\vec{v_i}$ at index idx in the chain does not depend on the index and on the configuration that is used for initializing the CBNN. The unique vector π of state probabilities (the stationary distribution) can be computed by solving the system $\pi \cdot A = \pi$ and so it is an eigenvector of A that corresponds to eigenvalue 1 (see Definition A.0.6 for information about eigenvalues / eigenvectors)

The result that we want to prove is actually a direct consequence of Theorem A.0.1. The Markov chain of a CBNN is aperiodic and irreducible (Lemmas 4.4.1 and 4.4.2) and from the theorem this means that the chain has a stationary distribution.

□

Lemma 4.4.3 is the first part of the proof that CBNNs asymptotically converge to optimal solutions. The lemma shows that if the neural network runs for long enough, then the states through which it moves start to follow some probability distribution π. The second part of the proof is to show that in π the probability of suboptimal solutions can be made arbitrary small. There is a lot of freedom when

choosing the function F that is minimized by the competition-based neural network. This is good because it allows many problems to be modeled in a CBNN-compatible way. The bad side of the freedom is that we can specifically construct unnatural objective functions that do not align with the expectations of the neural network. As a result, the neural network does not minimize them. Intuitively, the problem stems from the fact that the goal of a CBNN is to minimize a global function F, but it makes decisions based on local differences of *OnCost* (see Algorithm 4). The *OnCost* estimates the strength of a variable by measuring its contribution to the objective function F. It is natural to say that if one variable is better than another one, then enabling the better variable decreases the value of the objective function. But unnatural objective functions F can be constructed for which this does not hold. We decided to only prove here that for the p-MiniSum Problem, the group-average CBNN solver converges to an optimal solution. This proof can be used as a template for creating other proofs showing that CBNNs converge to optimal solutions for the problems from Section 5. Another option was to restrict our attention to some subset of all possible objective functions, like the quadratic functions of Hopfield networks, and to show the result for them. We decided to not do this because, first, the result seems to hold for natural objective functions that come from real-world problems and we did not want to artificially prohibit some of these functions. Intuitively speaking, if the objective function has a probabilistic interpretation and is the expected value of something, then the neural network successfully minimizes it. Second, plugging a different problem in the p-MiniSum proof requires in most cases only slight modification.

Analytically finding the stationary distribution of an arbitrary Markov chain is challenging. The property that a stationary distribution satisfies is that it is the unique probability vector π for which $\pi \cdot A = \pi$ (see Theorem A.0.1). In some cases, we can "guess" π. Lemma 4.4.4 is one possible way of ensuring that our guess is indeed the stationary distribution of the Markov chain.

Lemma 4.4.4. *Assume we have an irreducible aperiodic Markov chain with state transition probability matrix A together with a probability distribution π' such that*

$\pi'[\vec{v_i}] \cdot A[\vec{v_i}, \vec{v_j}] = \pi'[\vec{v_j}] \cdot A[\vec{v_j}, \vec{v_i}]$ *for every pair of states. Then* π' *is the stationary distribution of the Markov chain.*

Proof. From Theorem A.0.1 we know that an irreducible aperiodic Markov chain has a unique stationary distribution π. If π' satisfies $\pi' \cdot A = \pi'$, then it is this stationary distribution. If $\pi' \cdot A = r$, then $r[\vec{v_i}] = \sum_j \pi'[\vec{v_j}] \cdot A[\vec{v_j}, \vec{v_i}] = \sum_j \pi'[\vec{v_i}] \cdot A[\vec{v_i}, \vec{v_j}] = \pi'[\vec{v_i}] \cdot \sum_j A[\vec{v_i}, \vec{v_j}] = \pi'[\vec{v_i}]$ meaning that $\pi' = r$. This shows that π' satisfies $\pi' \cdot A = \pi'$ and is the stationary distribution of the Markov chain. □

The p-MiniSum problem is discussed in detail in Section 5.1. Its objective function is written as $F = \sum \frac{CF_{ij}}{CD_i} \cdot \frac{FL_{jk}}{FD_j} \cdot dist(i,k)$. The binary variables CF_{ij} express whether client i is services by facility j. Similarly, the binary variables FL_{jk} express whether facility j is placed in location k. $CD_i = \sum_j CF_{ij}$ is the degree of client i (that is, to how many facilities it is connected). Eventually, CD_i becomes 1 for every client because of the group constraints. $FD_j = \sum_k FL_{jk}$ is the degree of facility j. As discussed in Section 5.1, the division by CD_i and FD_j gives a probabilistic interpretation to the objective function and helps the neural network to quickly find a good solution.

Theorem 4.4.1. *Assume that the group-average CBNN solver with low constant temperature* T *is applied to a* p-MiniSum *instance. Then, by selecting a low enough temperature* T*, in the stationary distribution* π *of the resulting Markov chain the probability of the suboptimal solutions can be made arbitrary small.*

Proof. A Markov chain is characterized by its state transition probability matrix A. Lemma 4.4.3 shows that the Markov chain of a CBNN converges to a stationary distribution π. If we could "guess" what π is, then everything is easy. The problem is that π most probably can not be written as a simple formula. A workaround is to slightly change the elements of A so that for the resulting matrix we can "guess" π. Then use the fact that such small perturbations of the elements of A only slightly change the stationary distribution (see Theorem A.0.4).

The entries of the matrix A were described in the bullet list at the start of the section. Consider an entry $A[\vec{v_i}, \vec{v_j}]$ for two states $\vec{v_i}$ and $\vec{v_j}$ that differ in the value of

exactly one variable x (the other entries are either on the diagonal or are 0). $A[\vec{v_i}, \vec{v_j}]$ is of the form $\beta \cdot \widetilde{A}[\vec{v_i}, \vec{v_j}]$. β is a normalization constant that appears because the CBNN algorithm randomly chooses the variable to update. $\widetilde{A}[\vec{v_i}, \vec{v_j}]$ is either $\frac{1}{1+e^{\frac{\Delta}{T}}}$ or $1 - \frac{1}{1+e^{\frac{\Delta}{T}}}$. Notice that when the temperature T is close to 0, then $\widetilde{A}[\vec{v_i}, \vec{v_j}]$ is either very close to 0 or it is very close to 1. This is because $\Delta = |me - best|$ is always positive in Algorithm 4 and by pushing T closer to 0, the value of $e^{\frac{\Delta}{T}}$ can be made arbitrary large. It can be verified for the p-MiniSum Problem and a temperature T close to 0 that if enabling the variable x makes the value of the objective function worse, then the probability of enabling the variable is close to 0. If enabling the variable improves the objective function, that the probability is close to 1.

Now we define another Markov chain over the same state space and with a state transition probability matrix B. Lets take two states $\vec{v_i}$ and $\vec{v_j}$ that differ in the value of exactly one variable and for which $F(\vec{v_i}) > F(\vec{v_j})$. Lets denote by Δ_1 the difference $|F(\vec{v_i}) - F(\vec{v_j})|$ and lets set the entry $B[\vec{v_i}, \vec{v_j}]$ of B to be equal to $\beta \cdot \frac{1}{1+e^{\frac{\Delta_1}{T}}}$. For the other direction — the $\vec{v_j}$ to $\vec{v_i}$ transition, $B[\vec{v_j}, \vec{v_i}]$ is set to $\beta \cdot (1 - \frac{1}{1+e^{\frac{\Delta_1}{T}}})$. β is the same normalization constant as the one from the previous paragraph. Similarly to the matrix A, when the states $\vec{v_i}$ and $\vec{v_j}$ differ in the value of more than one variable, the probability $B[\vec{v_i}, \vec{v_j}]$ is set to 0. The diagonal entries $B[\vec{v_i}, \vec{v_i}]$ are set to $1 - \sum_{\vec{v} \neq \vec{v_i}} B[\vec{v_i}, \vec{v}]$ (if we do not explicitly transition to a new state, then we stay in the same state). When the temperature T is close to 0, matrices A and B are very similar. This is because if $B[\vec{v_i}, \vec{v_j}]$ is β multiplied by something that is almost 1, then the $\vec{v_i}$ to $\vec{v_j}$ transition improves the value of the objective function. As discussed in the previous paragraph, in this case $A[\vec{v_i}, \vec{v_j}]$ is also β multiplied by almost 1. When the $\vec{v_i}$ to $\vec{v_j}$ transition makes the value of the objective function worse, then the situation is the analogous: in both A and B the $(\vec{v_i}, \vec{v_j})$ entry is close to 0. From this reasoning we see that the difference between the matrices A and B can be made arbitrary small by decreasing T.

There is one degenerate case that we skipped. It is possible for the value of the objective function to not change when flipping the value of a variable x. This can happen, for example, if we duplicate a vertex in the input graph. In such a case,

the corresponding entries for the x flip in the matrices A and B are going to be $\beta \cdot \frac{1}{2}$ and so the matrices can still be made arbitrary close to each other. It is also not unreasonable to exclude such degenerate cases from consideration: in practice, we can always add small noise to the input that will not change substantially the solution but will break the symmetries.

For the matrix B Lemma 4.4.4 can be applied to show that in the stationary distribution π, the probability $\pi[\vec{v_i}]$ of any state $\vec{v_i}$ is $\gamma \cdot e^{\frac{-F(\vec{v_i})}{T}}$. The value γ is a normalization constant that makes $\sum_{\vec{v_i}} \pi[\vec{v_i}] = 1$. For applying Lemma 4.4.4, it is necessary to show that $\pi[\vec{v_i}] \cdot B[\vec{v_i}, \vec{v_j}] = \pi[\vec{v_j}] \cdot B[\vec{v_j}, \vec{v_i}]$. This is trivially true if $\vec{v_i} = \vec{v_j}$. When $\vec{v_i}$ and $\vec{v_j}$ differ in the values of more than one variable, then $B[\vec{v_i}, \vec{v_j}] = B[\vec{v_j}, \vec{v_i}] = 0$ and the equation is also true. The remaining case is a pair of states that differ in exactly one variable. Take two such states $\vec{v_i}$ and $\vec{v_j}$ for which $F(\vec{v_i}) > F(\vec{v_j})$. It needs to be shown that:

$$\pi[\vec{v_i}] \cdot B[\vec{v_i}, \vec{v_j}] = \pi[\vec{v_j}] \cdot B[\vec{v_j}, \vec{v_i}] \iff \frac{\pi[\vec{v_i}]}{\pi[\vec{v_j}]} = \frac{B[\vec{v_j}, \vec{v_i}]}{B[\vec{v_i}, \vec{v_j}]}$$

By substituting the "guessed" distribution π, for the left side we get:

$$\frac{\pi[\vec{v_i}]}{\pi[\vec{v_j}]} = \frac{\gamma \cdot e^{\frac{-F(\vec{v_i})}{T}}}{\gamma \cdot e^{\frac{-F(\vec{v_j})}{T}}} = e^{\frac{-\Delta_1}{T}}$$

In the above equation, Δ_1 denotes $|F(\vec{v_i}) - F(\vec{v_j})|$. By substituting the transition probabilities from B, for the right side we have:

$$\frac{B[\vec{v_j}, \vec{v_i}]}{B[\vec{v_i}, \vec{v_j}]} = \frac{\frac{1}{1+e^{\Delta_1/T}}}{1 - \frac{1}{1+e^{\Delta_1/T}}} = \frac{1}{e^{\frac{\Delta_1}{T}}} = e^{\frac{-\Delta_1}{T}}$$

This shows that the left and right sides are equal. The symmetric case with $F(\vec{v_i}) < F(\vec{v_j})$ can be verified in the same way.

So far, we have shown that $\pi[\vec{v_i}] \cdot B[\vec{v_i}, \vec{v_j}] = \pi[\vec{v_j}] \cdot B[\vec{v_j}, \vec{v_i}]$ for any pair of states. From Lemma 4.4.4 it follows that π from the previous paragraph is the stationary distribution of the Markov chain with state transition probability matrix B. The matrix A of the CBNN chain can be made arbitrary close to B by decreasing

the temperature T. This means that by decreasing T, the stationary distribution of the CBNN chain with matrix A can be made arbitrary close to the stationary distribution π of the Markov chain with matrix B (see Theorem A.0.4). In π, the optimal states are exponentially more probable than the other states. As a consequence, in the stationary distribution of the competition-based neural network optimal states are also exponentially more probable. Moreover, if Δ_1 is the difference in the value of the objective function F for an optimal and a suboptimal solution, then the optimal solution is $e^{\frac{\Delta_1}{T}}$ times more probable. By decreasing the temperature T, this "gap" in probability can be made arbitrary large. This, combined with the fact that the sum of the probabilities of all states is 1, means that the probability of the suboptimal states can be made arbitrary close to 0. □

Combined, Lemma 4.4.3 and Theorem 4.4.1 say that for the p-MiniSum Problem, competition-based neural networks that operate under a low temperature have a stationary distribution in which the probability of suboptimal states can be made arbitrary close to 0. If the CBNN is left to run for long enough, then it arrives at an optimal solution. This completes the proof of asymptotic convergence.

4.5 Speed of convergence

As discussed in Section 4.2, the worst-case time that is necessary for a CBNN with a fixed temperature to converge to its stationary distribution depends on the second-largest by absolute value eigenvalue of the state transition probability matrix A. This quantity can be computed and, for example, given two instances, we can say which one is simpler for the CBNN solver. The described procedure is not practical, though. Computing the eigenvalues is expensive because the matrix A is huge. Its size is the square of the size of the solution space.

Consider how the matrix A looks like. When the temperature is low, the CBNN accepts with high probability transitions that improve the solution. Transitions that decrease the quality of the solution are accepted with low probability. This is quite similar to the hill climbing heuristic. Separate basins of attraction are formed in

the solution space and they are almost disconnected. In hill climbing, the basins of attraction are completely disconnected and this is why it is not guaranteed to find a global optimum (in terms of Markov chains, the hill climbing Markov chain is not irreducible). In the CBNN case, the basins of attraction are weakly connected. While this allows the method to find a global optimum, the convergence will be slow. Markov chains of the described type are called *nearly uncoupled chains* [45]. It is well-known that for such chains, the second largest by absolute value eigenvalue is very close to 1.

Section 4.3 introduced a modification of the CBNN solver that is called CBNN with restarts. The state transition probability matrix of it is of the form $A_r = (1-p) \cdot A + p \cdot E$ for some restart probability p. The matrix E is a matrix of all $1/|S|$ where $|S|$ is the size of the state space. A CBNN with restarts behaves almost in the same way as a regular CBNN, but at every iteration with probability p it resets the states of its neurons to random values. This operation improves the connectivity between the different basins of attraction. For such neural networks, the result below holds:

Theorem 4.5.1. *For the second-largest by absolute value eigenvalue μ of the state transition probability matrix A_r of a CBNN with restarts it is true that $|\mu| \leq 1 - p$.*

Proof. We care about the absolute value of eigenvalues. To avoid adding everywhere *by absolute value*, when we say that an eigenvalue is the largest or is less than something, it is assumed that the statement is about the absolute value of the eigenvalue.

Lets denote by $\vec{v}$ an eigenvector that corresponds to μ, the second-largest eigenvalue of the matrix A_r. From the definition of eigenvectors and eigenvalues (see Definition A.0.6) it holds that $\vec{v} \cdot A_r = \mu \cdot \vec{v}$. The matrix A_r is a stochastic matrix and its rows sum up to 1. This means that $A_r \cdot \vec{1} = \vec{1}$. Lets compute the expression $\vec{v} \cdot A_r \cdot \vec{1}$. If we compute it in the order $(\vec{v} \cdot A_r) \cdot \vec{1} = (\mu \cdot \vec{v}) \cdot \vec{1}$, we get $\mu \cdot \vec{v} \cdot \vec{1}$. If the expression is computed in the order $\vec{v} \cdot (A_r \cdot \vec{1}) = \vec{v} \cdot (\vec{1})$, we get $\vec{v} \cdot \vec{1}$. As a result, $\mu \cdot \vec{v} \cdot \vec{1} = \vec{v} \cdot \vec{1}$ and $(\mu - 1) \cdot (\vec{v} \cdot \vec{1}) = 0$. From Theorem A.0.2 we know that $|\mu| < 1$, so $(\mu - 1) \neq 0$. Then $\vec{v} \cdot \vec{1}$ needs to be 0.

The matrix A_r is $(1-p)\cdot A + p\cdot E$. All entries of the matrix E are equal to $\frac{1}{|S|}$. We already showed that $\vec{v}\cdot\vec{1} = 0$, so $\vec{v}\cdot E = 0$. $\vec{v}$ is an eigenvector of A_r. Because of this, $\vec{v}\cdot A_r = \mu\cdot\vec{v}$. The expression $\vec{v}\cdot A_r$ can be written as $\vec{v}\cdot((1-p)\cdot A + p\cdot E) = (1-p)\cdot\vec{v}\cdot A + p\cdot\vec{v}\cdot E = (1-p)\cdot\vec{v}\cdot A$. The last equality is because $\vec{v}\cdot E = 0$. Now we have that $(1-p)\cdot\vec{v}\cdot A = \mu\cdot\vec{v} \iff \vec{v}\cdot A = \frac{\mu}{1-p}\cdot\vec{v}$. The same vector $\vec{v}$ is an eigenvector of the matrix A and the corresponding eigenvalue is $\frac{\mu}{1-p}$. The matrix A is also a stochastic matrix, so the absolute value of all of its eigenvalues is less than or equal to 1 (see Theorem A.0.2). This means that $\frac{|\mu|}{1-p} \leq 1 \iff |\mu| \leq 1-p$.

As a summary of the result, the largest eigenvalue of A_r is 1 and the absolute value of the second-largest eigenvalue is at most $(1-p)$. □

We see that we can control very well the second-largest by absolute value eigenvalue of a CBNN with restarts. The cost of this is that the restart probability p biases the stationary distribution of the method towards a uniform distribution and the stationary distribution can be very sensitive to the value of p. The paper [20] shows one possible way of measuring the sensitivity of a Markov chain to small perturbations using the notion of mean first passage time. Assume that $\vec{v_i}$ and $\vec{v_j}$ are two states. The mean first passage time is the expected number of steps to reach for the first time $\vec{v_j}$ from $\vec{v_i}$. The probability of state $\vec{v_j}$ in the stationary distribution is very sensitive to small perturbations when the maximum mean first passage time from some state to $\vec{v_j}$ is large. Intuitively, if $\vec{v_j}$ is an optimal solution, then the mean first passage time for it measures how long it takes to reach $\vec{v_j}$ from some initial configuration. This is related to the time that is necessary for a regular CBNN to reach $\vec{v_j}$ from a random point. So, if a regular CBNN converges slowly, then a CBNN with restarts is very sensitive to the value of p.

Remember that CBNNs can solve NP-hard problems. Unless $P = NP$, we should not be able to prove that they converge to an optimal solution in polynomial time. Indeed, the above reasoning shows that the described methods do not possess "special powers": they can adapt to the complexity of the input instance, but there are instances for which the methods are slow. Bounds on the time can be proven,

but the bounds are larger than the size of the solution space and we do not consider such results useful. In reality, we do not apply fixed temperature CBNNs or CBNNs with restarts for solving problems. What we apply is the general CBNN solver from Algorithm 4. This solver gradually decreases the temperature from a high value towards 0. As discussed in the next section, there are heuristic reasons to believe that this helps the method to quickly find a good solution.

4.6 CBNNs in practice

The asymptotic convergence of CBNNs to an optimal solution is a good property. It says that the method is not internally biased and indeed is a procedure for solving combinatorial optimization problems. Otherwise, CBNNs would have been a method that returns for unknown reason a solution of unknown quality. But the runtime requirements for asymptotic convergence of CBNNs are too bad to be useful in practice. We want to be able to solve problems in polynomial time. In this section, it is discussed why CBNNs can still be expected to return good solutions even when restricted to polynomial time in the size of the input instance. The reasoning is an intuition, not a proof. Since CBNNs solve NP-hard problems, a real proof for polynomial time CBNNs should not exist unless $P = NP$.

From the convergence proof, it can be seen that solving a problem with a competition-based neural network is equivalent to reaching the stationary distribution of the corresponding low temperature Markov chain. Computers can simulate CBNNs with temperatures very close to 0 and the low temperature constraint is not a major problem in practice. The trouble comes from the time that is necessary to reach a stationary distribution. It is true that CBNNs reach this distribution no matter what the initial solution is. But the time that is necessary for this significantly depends on the initial solution. Assume that the initial solution of a CBNN is drawn from a distribution *ini*. If *ini* is close to the stationary distribution, then the neural network converges quickly. If it is not close, then the convergence is slow. Restricting a CBNN to polynomial time means that the epoch length is also polynomial. The epoch length is the length of the Markov chain. Forcing the length to

be polynomial and, at the same time, requiring that the chain reaches its stationary distribution means that the initial distribution ini needs to be quite close to the stationary distribution. Assume T_1 is the "low enough temperature" for solving the input instance and π_1 is the stationary distribution of the CBNN for this temperature. How to find ini_1 from which to start the neural network? It needs to be close to π_1, a distribution in which only optimal states have nonzero probability. If we knew what the optimal states are, then we would not have used the neural network to find them. A workaround is to take a temperature T_2 that is just slightly larger than T_1. Compute the stationary distribution π_2 of the CBNN for T_2 and set ini_1 to be equal to π_2. This works because T_1 and T_2 differ just slightly and so the state transition probability matrices for the two temperatures are almost equal. As a result, π_1 and π_2 are also very close (see Theorem A.0.4). Now one problem has been solved, but we face another one: how to choose the initial distribution ini_2 for the temperature T_2. For this, a third temperature T_3 is taken that is slightly larger than T_2 and the stationary distribution for T_3 is used as the initial distribution for T_2. This process can be repeated and, as a result, we get a sequence $T_1, T_2, T_3, ..., T_m$. On every step, the stationary distribution for T_i is used as the initial distribution for T_{i-1}. Notice that on every step, the temperature grows, and T_m is not a small temperature. For large temperatures, the stationary distribution of a CBNN is close to a uniform one. If we make the sequence long enough, then T_m will be large and we can use the uniform distribution as the initial distribution for T_m. A uniform distribution is easy to create. This is the essence of the complete CBNN solver from Algorithm 4 that decreases the temperature after every epoch. The sequence $T_m, T_{m-1}, T_{m-2}, ..., T_1$ was called there a temperature decrease schedule.

We want to repeat again the same idea but from a slightly different perspective. The explanation is an expanded version of the observations in our paper [44]. The CBNN solver from algorithm Algorithm 4 can be though of as creating a sequence of Markov chains for different temperatures. The last chain with the lowest temperature is the one that actually "solves" the problem. The other chains are a type of preprocessing. Assume that the network has already reached its stationary distri-

bution for a given temperature. If we only slightly decrease the temperature, then the stationary distribution will also only slightly change. Which should mean that only a small number of transitions is necessary to restore it for the new temperature. Intuitively, the number of steps now should be much smaller than the number of steps needed to reach the stationary distribution from a random state (solving the problem from scratch). When the temperature is high, the stationary distribution is close to a uniform distribution (all transitions are almost equally probable). So we can assume that for high temperatures, the neural network starts in its equilibrium state. We only care about the last Markov chain with the lowest temperature. Our hope is that the epoch length L is long enough to maintain the network close to its stationary distribution for all temperatures (as a special case, for the lowest temperature). We want to again stress the difference with directly "solving" the given problem. When "solving" a problem, we intuitively want to reach the stationary distribution from a random state, while here we already start in a stationary distribution and only need to maintain it. This should be simpler.

Chapter 5

Applications of CBNNs

The first four chapters are devoted to introducing competition-based neural networks and to proving their properties. We have already discussed why the model is expected to find good solutions to combinatorial optimization problems. In this chapter, we apply CBNNs to a number of facility location problems to demonstrate their good empirical performance. The problems are listed below.

- **The p-MiniSum Problem** (Section 5.1).

  This maybe is the first problem that comes to mind when talking about facility location. We have a number of clients and need to position p warehouses so as to minimize the transportation costs. In the thesis, when we needed an example problem, the p-MiniSum Problem was used most of the time. In Section 5.1, the CBNN solver is evaluated on p-MiniSum instances from the Bulgarian roads data set. The data set was created specifically for the experiments in the thesis.

- **The p-Hub Problem** (Section 5.2).

  The goal here is to design a hub network. In this type of networks, there is a set of locations and hubs. The traffic between the locations is routed through the hubs. Postal delivery networks and airplane passenger networks are two examples of systems of this type. The goal of the p-Hub problem is to position the available hubs in such a way that the total cost of routing the traffic is minimized. Usually, for facility location problems locating the facilities is

hard. But once the locations are selected, it is easy to assign the clients to them. The p-Hub problem is unusual because for it both the location step and the assignment step are hard. In Section 5.2, the CBNN solver is tested on the Australia Post data set. This is a well-known data set for evaluating algorithms for the p-Hub problem.

- **The p-Defense-Sum Problem** (Section 5.3).

  In this problem, the goal is to position p facilities so as to maximize the sum of pairwise distances between the facilities. The p-Defense-Sum Problem is an example of an obnoxious facility location problem. In obnoxious facility location, the goal is to maximize distances instead of minimizing them. Another problem of this type is locating a garbage dump site: people want to push such facilities further away from their home. The p-Defense-Sum Problem is also an example of a facility location problem without clients. The CBNN solver is evaluated on p-Defense-Sum instances that were derived from the road network inside Bulgarian cities. The data set was specifically created for the experiments in Section 5.3.

- **The Maximal Covering Location Problem (MCLP)** (Section 5.4).

  MCLP is from the class of covering problems and is related to the classic Set cover problem (one of Karp's 21 NP-complete problems that were shown to be NP-complete in 1972 [57]). We have a set of populated places and the goal is to position p cell phone towers so that they cover the maximal number of populated places. The CBNN solver for the MCLP Problem is evaluated on two sets of instances. The first one is derived from the map of Bulgaria and is created specifically for the experiments in Section 5.4. The second set of instances is created from Steiner Triple Systems. They are a well-known source of hard instances for set covering problems.

- **Flow Intercepting Facility Location** (Section 5.5).

  In the Flow Intercepting Facility Location Problem, we know the daily commute routes of people and want to position advertising boards so that

the boards cover as much of the roads as possible. An obvious difference with the other facility location problems is that here the clients are not individual points but are roads. Apart from that, the problem is very similar to the other set covering problems. The CBNN solver is tested on instances that are derived from the road network of Sofia. The instances were specifically created for the experiments in the thesis.

- **The Assignment Problem** (Section 5.6).

  The Assignment Problem is different from the rest of the problems in this section. It is not usually considered to be a facility location problem and a polynomial-time algorithm is known for it. We decided to evaluate competition-based neural networks on the Assignment Problem because this application illustrates one way of dealing with overlapping group constraints. The CBNN solver is applied to randomly generated instances of the problem.

The field of facility location offers a wide range of problems. We selected the six problems above because they highlight different aspects of facility location. Overall, competition-based neural networks demonstrate excellent performance on the test problems. They are often able to quickly find the optimal solution to the input instance. When the returned solution is not optimal, it is at most several percent worse than the optimal one. This is not bad for a method that works out-of-the-box. The results of the CBNN solver on the test facility location problems are very competitive to the results of the other high-quality metaheuristics.

5.1 The p-MiniSum problem

The p-MiniSum Problem, also known as p-Median, is one of the best-known problems of discrete location theory. The goal of the problem is to locate p facilities while minimizing the sum of distances between demand points and the closest facility. *MiniSum* problems originated in the 17th century when Fermat posed the problem of finding a median point of a triangle (a point that minimizes the sum of distances to the triangle vertices). In the early 20th century, the problem was generalized to the case of finding more than one median point among more than three points (this is called the Multifacility Weber Problem).

The section deals with the graph variant of p-MiniSum. Intuitively, there is a set of populated places and a road network connecting them. The goal is to locate p warehouses so as to minimize the sum of distances from every populated place to the closest warehouse. The problem was studied in the 1960s by Hakimi [40, 39] in terms of communication networks. Originally, Hakimi allowed locating warehouses not only in populated places (the vertices of the graph) but also along the edges. He showed that there is an optimal solution to the problem in which warehouses are placed only in vertices. The reason is that if a warehouse is in the middle of an edge, then moving it in one of the two directions along the edge will not decrease the quality of the solution. This result leads to the modern formulation of p-MiniSum in which warehouses can only be placed in the vertices of the input graph.

The content of the section is based on our paper [42] that compares different neural network approaches for the p-MiniSum Problem.

5.1.1 Mathematical definition and known results

The p-MiniSum Problem was formally stated in Chapter 1, Definition 1.2.2. The input of the problem is a weighted, undirected and connected graph $G(V, E)$ with positive edge lengths. The number p of facilities is also given. The goal of the problem is to find a subset of p vertices $u_1, ..., u_p$ that minimizes $\sum_{c \in V} \min_{i \in \{1..p\}} dist(c, u_i)$ where $dist(c, u_i)$ is the distance in G from c to u_i. This subset of vertices represents the locations of the facilities. A weighted variant of the problem exists in which the

distance from every vertex to the closest facility is multiplied by the weight of the vertex.

Kariv and Hakimi showed that the p-MiniSum Problem in general graphs is NP-hard [56]. In trees, the problem can be solved in polynomial time via dynamic programming on subtrees [56, 98]. It should be noted that if the number of facilities p is a fixed constant, then the p-MiniSum Problem (and many other facility location problems) can trivially be solved in polynomial time by iterating through every possible configuration (in this context, polynomial does not mean fast). In real-world applications of p-MiniSum, the value of p is usually much smaller than the total number of vertices. Of course, even for such p, iterating through all possible ways of placing the facilities quickly becomes too slow. But special algorithms can be developed for the small p case that are quite fast. For example, in [43] we show that the 1-MiniSum Problem in k-trees, a generalization of trees, can be solved in $O^*(n \cdot lg(n))$ time. This is a subquadratic complexity and is less than the time that is necessary to compute all pairwise distances between the vertices of the input graph. From our experience, the complexity of p-MiniSum and most other facility location problems grows very quickly with the increase of p. Even for relatively small values of p, exactly solving the problem becomes expensive.

The p-MiniSum Problem in general graphs can be formulated as an Integer Programming (IP) problem and solved using any of the available IP solvers. One possible formulation is given below. n in the formulation denotes the total number of vertices in the graph. The distance between any two vertices u_i and u_j is denoted as $dist(u_i, u_j)$. The binary variables $x_{i,j}$ in the formulation express whether the client in vertex i is serviced by a facility in vertex j. The binary variables y_j express whether a facility is placed in vertex j. The first group of constraints guarantees that if a client is serviced by a facility in vertex j, then this facility is indeed opened. The second group guarantees that every client is serviced by exactly one facility. The third group says that in total p facilities need to be opened.

Minimize:

$$\sum_{i,j \in \{1..n\}} x_{i,j} \cdot dist(u_i, u_j)$$

Subject to:

$$x_{i,j} \leq y_j \quad \forall i, j \in \{1..n\}$$

$$\sum_{j \in \{1..n\}} x_{i,j} = 1 \quad \forall i \in \{1..n\}$$

$$\sum_{j \in \{1..n\}} y_j = p$$

$$x_{i,j} \in \{0, 1\} \quad \forall i, j \in \{1..n\}$$

$$y_j \in \{0, 1\} \quad \forall j \in \{1..n\}$$

Even for relatively small instances of p-MiniSum, exactly solving the IP problem above becomes too slow. Many approximate approaches were developed for the p-MiniSum Problem. Iteration optimization and heuristics were the earliest proposed techniques. From these approaches, modifications of vertex substitution [99] seem to be the most popular. Methods that are based on LP relaxations of IP formulations of p-MiniSum have also been investigated. More recently, many metaheuristics like Genetic algorithms, Tabu search and Simulated annealing have been applied to the p-MiniSum Problem. A recent annotated bibliography of the literature on solution methods for the p-MiniSum Problem can be found in [84].

5.1.2 CBNN model

The p-MiniSum Problem is used as a working example throughout the thesis. A model of the problem that is compatible with Hopfield networks was presented in Section 2.3. This subsection presents a CBNN model of p-MiniSum that uses the same set of variables. The CBNN-compatible formulation of a problem is not unique. We present one possible model that seems to be good and that was used

for the experiments in this section.

For defining a model, we need to specify the set of variables, the set of group constraints, and the objective function. In the CBNN formulation of p-MiniSum there are two families of binary variables: CF_{ij} and FL_{jk}. CF stands for client-facility relation and CF_{ij} is 1 if and only if the client in vertex i uses facility j. The FL family of variables models the facility-location relation and FL_{jk} is 1 if and only if the facility j is placed in vertex k. Figure 5.1 visually illustrates the variables of the encoding.

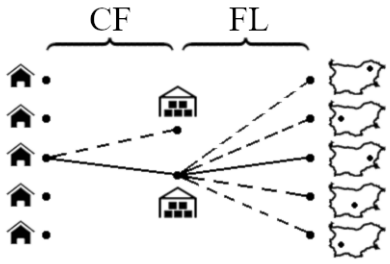

Figure 5.1: p-MiniSum. The variables of the CBNN model are actually the edges in image (not all edges are shown). The edges on the left side represent the client-facility relations and on the right — the facility-location relations. Solid edges have value 1 and dashed edges have value 0. The path of solid edges in the image can be interpreted as saying that the third client (house, left column) uses the second facility (middle column) that is placed in the third location (right column).

Setting the value of every binary variable of the model to 0 or 1 results in a configuration in the search space. The configurations do not always represent feasible solutions. For example, it is possible for both FL_{01} and FL_{02} to have value 1. This means that facility 0 is placed in two locations at the same time and the configuration is not a feasible solution. The set of group constraints of the CBNN model guarantees that the neural network eventually finds a feasible solution. The group constraints for the p-MiniSum Problem are written below.

$$\forall i \in \{1..n\} \sum_{j \in \{1..p\}} CF_{ij} = 1$$

$$\forall j \in \{1..p\} \sum_{k \in \{1..n\}} FL_{jk} = 1$$

n denotes the number of vertices. The first set of constraints guarantees that every

client is connected to exactly one facility. The second set of constraints guarantees that every facility is placed in exactly one location.

So far, the CBNN model of p-MiniSum tells the neural network to find a valid solution. The objective function F is what guides the neural network towards good solutions. Assuming $dist(i,k)$ denotes the distance between vertices i and k in the graph, the objective function of p-MiniSum can be written as $\sum_{i,j,k} CF_{ij} \cdot FL_{jk} \cdot dist(i,k)$. The term $CF_{ij} \cdot FL_{jk} \cdot dist(i,k)$ is nonzero only if both CF_{ij} and FL_{jk} have value 1. In this case, the interpretation of the variables is that client i is serviced by facility j that is placed in location k. When this happens, the distance between i and k is added to the value of F.

The function F from the previous paragraph directly expresses the goal of the p-MiniSum Problem. But we can define a better objective function. Remember that during the operation of a CBNN, the neural network passes through infeasible configurations. In the final solution, all group constraints are enforced, but this may not be true for intermediate configurations. The objective function still needs to be defined for them. In an intermediate configuration, it can happen that client A is serviced by facility B that is placed in both locations C and D. Adding both the distance from A to C and from A to D to the value of F seems unnatural. Moreover, it makes the job of the neural network harder.

A better objective function for the p-MiniSum Problem can be created by making sure that the function has a meaningful interpretation for the infeasible solutions. A good way to achieve this is to have a probabilistic interpretation of the CBNN objective function. When a facility is placed in two locations, it is assumed that the facility is placed in the first one with probability 0.5 and in the second one with the same probability. A client that is connected to multiple facilities is handled similarly. The objective function is defined as the expected value of the sum of distances from the clients to the facilities that service them. It can be written as $\sum_{i,j,k} \frac{CF_{ij}}{CD_i} \cdot \frac{FL_{jk}}{FD_j} \cdot dist(i,k)$. $CD_i = \sum_j CF_{ij}$ is the degree of client i. FD_j is the degree of facility j. In this way, $\frac{CF_{ij}}{CD_i}$ is the probability for client i to be serviced by facility j and $\frac{FL_{jk}}{FD_j}$ is the probability for facility j to be placed in location k.

The whole term in the sum is assumed to be zero if there is a division by zero. This probabilistic objective function is the one that is used in the CBNN model of p-MiniSum. Notice that when all feasibility constraints are satisfied, then the CD_i and FD_j values are all 1 and the value of the objective function is exactly the sum of the distances from every client to the facility that services it.

5.1.3 CBNN solver

For the experiments with the p-MiniSum instances, the group-average variant of the general CBNN solver (Algorithm 4) is used. This is the same solver as the one from the proof of asymptotic convergence of CBNNs from Section 4.4. The solver uses the p-MiniSum model that was described in the previous subsection and is applied without modifications.

The CBNN solver is randomized, so it makes sense to perform multiple runs per p-MiniSum instance and pick the best of the returned solutions. Additionally, CBNNs have relatively low variance. When performing multiple runs, it is best to change every time the hyperparameters of the neural network. We change the number of steps per epoch and the temperature decrease coefficient. Two general strategies for the hyperparameters were mentioned in Section 3.2: small temperature decrease with smaller number of steps per epoch or larger temperature decrease with large number of steps per epoch. For every p-MiniSum instance, we execute the CBNN solver three times: two times following the first strategy and once following the second strategy.

The same sets of hyperparameters are used for all instances. The hyperparameters were chosen based on general observations about all data and no instance-specific tuning was performed. We want to stress here one difference between the neural networks for classification and the ones for combinatorial optimization. In classification problems, the neural network needs to generalize information from a training set and when given a new instance, the network can not automatically understand how good its prediction is. When solving combinatorial optimization problems, CBNNs known very well how good their solution is because they can eas-

ily compute the value of the objective function for the solution. While CBNNs do not know for sure what is the optimal value of the objective function, they can easily say whether solution A is better than solution B. This allows instance-specific tuning to be performed: try many combinations of the hyperparameters and pick the best solution found for the instance. Such tuning can not be done in a simple way for classification problems because on new data we generally can not automatically infer if one solution (a label for the input) is more correct than another one.

5.1.4 Test data

The Bulgarian road network is used to generate realistic p-MiniSum instances. For creating an instance, a rectangular area on the map is selected. All populated places and roads in this area are extracted and used as an underlying graph of the instance. The areas are chosen so that they contain between 20 and 90 populated places. The number of facilities is set to a value between 2 and 6. The resulting p-MiniSum instances are small enough to be able to exactly compute the optimal solution in reasonable time using an IP solver. OpenStreetMap [25] is used as a source of geographic data. The data is extracted with Overpass XML queries as described in Appendix B. For computing the optimal solution, each instance is modeled as an integer programming problem and solved using the Cbc mixed integer programming solver [17]. In total, 50 different rectangular areas were selected. This gives $50 \cdot 5 = 250$ p-MiniSum instances. Four of them are shown in Figure 5.2.

As can be seen from the definition of p-MiniSum, the solution to the problem is a list of p vertices of the input graph. Once this list is known, assigning clients to facilities is trivial: every client is assigned to the closest facility. Figure 5.3 shows an example instance together with one possible solution.

5.1.5 Results

Even without tuning, the results of the CBNN solver on the Bulgarian roads data set are very promising. The average error on all instances is 0.2% and 87% of all instances are solved optimally. Here, by error we mean the difference between

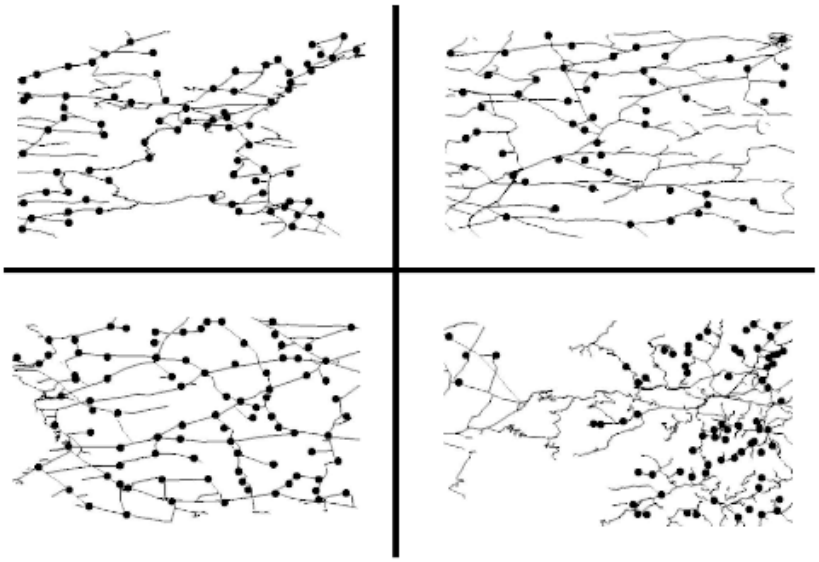

Figure 5.2: Four example p-MiniSum instances that were derived from the road network of Bulgaria. The dots denote populated places and the lines represent the road network.

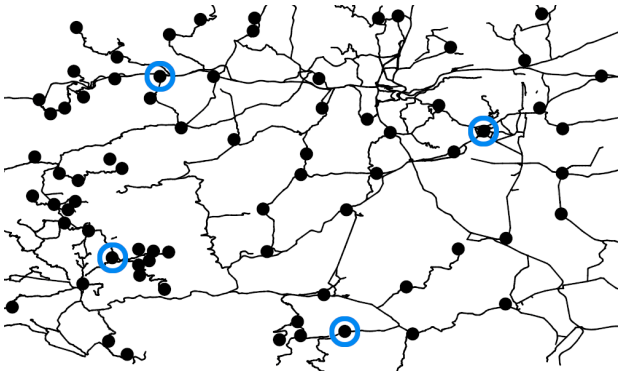

Figure 5.3: An example p-MiniSum instance in which 4 facilities need to be placed. The dots (vertices) represent the populated places and the 4 selected locations are marked with an additional circular outline.

the value of the objective function for the optimal solution and for the one that is returned by the CBNN solver (the difference is written as a percentage of the value of the optimal solution). Table 5.1 shows the results of the CBNN solver that are split by the number of facilities p. As expected, $p = 2$ is the simplest case and the neural network optimally solves all such instances. $p = 6$ gives the hardest instances. The same tendency can be seen for most facility location problems. Their complexity grows quickly with the increase of p.

p	Error	Exact	Max. error
2	0%	100%	0%
3	0.2%	87.9%	2.68%
4	0.06%	90.9%	1.45%
5	0.14%	90.9%	2.88%
6	0.57%	66.7%	4.48%

Table 5.1: Results of the CBNN solver on the Bulgarian roads data set. The data set is split into five parts by the number of facilities to be placed (the **p** column). The **Error** column gives the average error for the instances where the error is the difference between the optimal solution and the solution that is returned by the neural network. The difference is written as a percentage of the value of the optimal solution. The **Max. error** column gives the maximum error for the group of instances. The **Exact** column gives the percentage of the instances in the group that are optimally solved by the neural network.

5.1.6 Final notes

In our paper [42], we compared three neural network approaches for the p-MiniSum problem on the Bulgarian roads data set: Hopfield networks (see Section 2.3.1), Boltzmann machines (see Section 2.3.2) and the group-best variant the CBNN solver. The results of the group-best CBNN are very close to the ones obtained in this section using the group-average variant. We want to mention here the difference between group-best and group-average CBNNs. The first type compares the *OnCost* of the current variable to the *OnCost* of the best active variable in its group. The second type compares the *OnCost* of the current variable to the average *OnCost* of the active variables in its group. On earlier stages of the optimization, the result of the comparison can be quite different between the two variants. On later stages of the optimization, the difference between the two CBNN variants becomes relatively small and should not significantly affect the operation of the neural network.

The results of both Hopfield networks (HN) and Boltzmann machines (BM) on the Bulgarian roads data set are very discouraging. While both methods were always able to find a valid configuration, on average they find solutions that are more than two times more expensive than the optimal solution. The results of BMs are slightly better than the results of HNs, but the runtime is significantly longer. Also, the value of the objective function for the returned solutions just slightly decreases when

increasing the number of facilities p. And the quality of the solutions is not much better than the quality of a random solution. This hints that both HNs and BMs put emphasis on finding valid configurations and almost do not optimize for their quality. The full experiment with these two types of neural networks is described in [42]. We just want to mention our conclusions on why they do not perform well. The poor performance of HNs and BMs seems to follow from a combination of the usage of penalty terms as a mechanism for enforcing solution validity and the inflexible neighborhood definition. For Hopfield networks, there is an additional drawback of complete determinism in the sense that the model only allows moves that decrease the value of the energy function.

After comparing the results of CBNNs to the results of the two other neural network approaches, we can conclude that competition-based neural networks significantly outperform in solution quality both HNs and BMs. CBNNs are slower than Hopfiled networks but are much faster than Boltzmann machines.

5.2 The p-Hub problem

Assume there are N locations and there is traffic between every pair of locations. For example, in a postal delivery system there may be N clients who send mail to each other. Ideally, there will be a direct link between each pair of clients. In practice, deploying such an infrastructure is too expensive. Therefore, usually several hub locations are selected through which all traffic is routed. Every client is serviced by exactly one hub. When client A wants to send a package to client B, the package travels from A to the hub that services A, from there to the hub that services B, and finally to B (see Figure 5.4). If the number of hubs is P, then the number of direct links in such a system is around $P^2 + P \cdot N$. This is much less than the approximately N^2 links that are necessary if every pair of clients was directly connected. The drawback of the hub system is that the distances, traveled by the packages, increase (which also increases costs). The p-Hub Problem, sometimes referred to as USApHMP, asks for the optimal placement of the P hubs so as to minimize the cost of servicing all traffic. Apart from postal delivery networks, the problem arises when designing telecommunications networks and airline passenger networks.

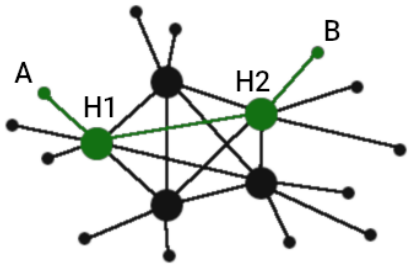

Figure 5.4: Example of a hub network. The larger circles represent hubs. If client A wants to send a package to client B, then the package goes through the hubs H_1 and H_2.

For evaluating the CBNN solver on the p-Hub Problem, the Australia Post (AP) data set [31] is used. Next, the p-Hub Problem is described in terms of postal delivery. There are N locations in the plane that are numbered with the integers $1..N$. The amount of mail traffic from location i to location j is denoted as W_{ij}. This amount may not be symmetric. Exactly P of the locations need

to be designated as hubs and every location needs to be mapped to exactly one hub. When sending a package from location A to location B, the package is routed through the hubs of A and B. The cost of delivering one unit of mail is proportional to the distance traveled. The route of a package consists of three distinct stages: collection from a source to a hub, transfer between hubs, and delivery from a hub to a destination. These three stages have different costs per unit of distance. For example, the transfer between hubs is the cheapest because of economies of scale. To account for this, in the AP data set there are three multipliers α, β, and γ. They are used to multiply the distance when computing the cost of a stage. For a pair of locations i and j, the distance between them is denoted as $dist(i, j)$. If the route of a package is $A \to H_1 \to H_2 \to B$, then the cost per unit of mail is $\alpha \cdot dist(A, H_1) + \beta \cdot dist(H_1, H_2) + \gamma \cdot dist(H_2, B)$. Notice that it is possible for H_1 to be equal to H_2. In this case, the transfer cost is 0 and only the collection and delivery steps need to be performed. As mentioned in the previous paragraph, the goal of the p-Hub Problem is to decide which P locations to designate as hubs and how to assign the other locations to the hubs so that the total cost of servicing the mail traffic is minimized. The next section describes a quadratic programming model of this intuitive definition of the p-Hub Problem.

Several variations of the p-Hub Problem exist that are not considered in this chapter. For example, instead of creating a link between every pair of hubs, it may be more appropriate to order them in a ring. A truck may cycle through this ring and deliver all cargo more efficiently. There may be capacity constraints on the amount of mail that each hub can service. The number of hubs may not be specified beforehand. Instead, there may be a cost of opening a new hub. In such a scenario, the p-Hub Problem asks to minimize the cost of opening the hubs plus the cost of delivering mail traffic for a given period of time. There is also a variant of the problem in which we want to deliver as much traffic as possible within a given budget. These problems can still be solved by competition-based neural networks or their modifications.

5.2.1 Mathematical definition and known results

Below, one possible quadratic programming formulation of the p-Hub Problem is given. It has N^2 binary variables. The meaning of the variable Z_{ij} is that the client in location i is serviced by a hub in location j. The variables Z_{jj} have special interpretation: Z_{jj} is 1 iff there is a hub in location j.

Minimize:

$$\sum_{i,j} \alpha \cdot Z_{ij} \cdot dist(i,j) \cdot out_i +$$

$$\sum_{i,j} \gamma \cdot Z_{ij} \cdot dist(i,j) \cdot in_i +$$

$$\sum_{i,j} \sum_{k,l} \beta \cdot Z_{ij} \cdot Z_{kl} \cdot dist(j,l) \cdot W_{ik}$$

Subject to:

$$\forall i \in \{1..N\} \sum_{j \in \{1..N\}} Z_{ij} = 1$$

$$\forall i,j \in \{1..N\}\ Z_{ij} \leq Z_{jj}$$

$$\sum_{j} Z_{jj} = p$$

$$\forall i,j \in \{1..N\}\ Z_{ij} \in 0,1$$

The first set of constraints guarantees that every client is serviced by exactly one hub. The second set of constraints guarantees that if a client is serviced by a hub in location j, then there is indeed an open hub in this location. The third constraint says that exactly p hubs are open. In [77], a very similar formulation to the one above is given. Several other formulations of p-Hub can be found in the literature. For example, Campbell gives an integer programming formulation of the problem with $O(N^4)$ variables [15]. The variables are of the form X_{ijkl} with a meaning that the traffic from i to j is routed through hubs k and l. While solving integer programming problems is in general easier than solving quadratic programming problems, the large increase in the number of variables probably makes the integer

programming formulation less attractive.

As many other facility location problems, the P-Hub Problem in general graphs is known to be NP-hard [72]. Even if the hubs are already selected, the allocation step of assigning clients to hubs is still NP-hard [72]. This is in contrast to most other facility location problems in which the optimal allocation can be found efficiently.

General integer programming and quadratic programming solvers can be used to exactly solve a p-Hub instance. Specialized exact solving procedures have also been developed for the problem. For example, in [77] an LP-based branch-and-bound method is described. Solving large p-Hub instances is expensive, so many approximate approaches for the problem have been developed. These include problem-specific heuristics [77, 64], Tabu search [63, 92], Genetic algorithms [1, 73], Simulated annealing [31], and Variable neighbourhood search [49]. Neural network approaches have also been applied to the problem. For example, the paper [94] describes a mapping of p-Hub instances onto a Hopfield network.

5.2.2 CBNN model

This section describes a formulation of the p-Hub problem that is compatible with competition-based neural networks. Additionally, a modification of the general CBNN solver is described that reduces the runtime of the method. The modification is purely for speeding up the *Update* function of Algorithm 4 and does not essentially change the operation of the neural network or the quality of the final solution. We try to avoid tuning the solver for speed, but in this case, the problem instances are quite large and the runtime becomes significant.

In the CBNN formulation of p-Hub, there are two types of binary variables: CH_{ij} and HL_{jk}. The first type of variables represent the client-hub relation. In the final solution, CH_{ij} is 1 iff client i is serviced by hub j. The second type of variables represent the hub-location relation. HL_{jk} is 1 iff hub j is placed in location k. It is convenient to think of the CH and HL variables as of oriented edges going from a client to a hub and from a hub to a location (see Figure 5.5). As usual,

in the intermediate solutions during the operation of the CBNN it is possible for one client to be serviced by multiple hubs and for one hub to be placed in multiple locations. That is, intermediate states of the neural network may not represent feasible solutions. In the final solution, it is guaranteed that there will be no such violations of the feasibility constraints.

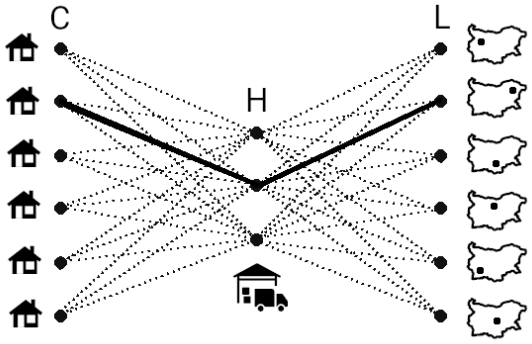

Figure 5.5: CBNN model of p-Hub. On the left are the clients, in the middle are the hubs, and on the right are the locations (in the p-Hub Problem, the set of clients coincides with the set of possible locations). The binary variables of the formulation are the edges in the figure. The two solid edges are an example of variables with value 1. Intuitively, the solid edges say that the client on the left is serviced by the hub in the middle that is placed in the location on the right.

The constraints that guarantee the feasibility of the solution are written as follows:

$$\forall i \in \{1..N\} \sum_{j \in \{1..P\}} CH_{ij} = 1$$

$$\forall j \in \{1..P\} \sum_{k \in \{1..N\}} HL_{jk} = 1$$

The first set of constraints guarantees that every client is serviced by exactly one hub. The second set guarantees that every hub is placed in exactly one location. In terms of CBNNs, these are called group constraints. For every client, its outgoing edges are placed in one group. Similarly, the outgoing edges of every hub are placed in one group. The CBNN solver enforces the constraint that in the final solution, exactly one edge from a group has value 1.

When defining the objective function, it needs to be taken into account that group constraints from the previous paragraph may be violated in intermediate

solutions. This is because of the way competition-based neural networks operate. A possible interpretation of such violations is that a client is serviced by several hubs and that a hub is placed in several locations. The objective function still needs to be defined for such states and it needs to be smooth. When a client is serviced by several hubs, we assume that the traffic from the client splits evenly between the hubs. Branching hub-location edges are handled in a similar way. Figure 5.6 gives an example. In the figure, the edges are the variables with value 1. Client C_1 is

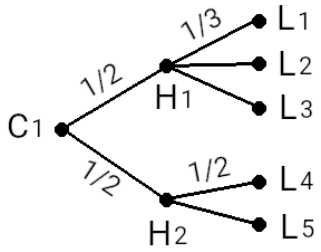

Figure 5.6: Example of an intermediate state in which a client is serviced by multiple hubs that are placed in multiple locations. The traffic splits evenly at each branching point.

serviced by hubs H_1 and H_2. Hub H_1 is placed in locations L_1, L_2 and L_3. Hub H_2 is placed in L_4 and L_5. At each branching point, the traffic splits evenly. For example, the $C_1 \to H_1 \to L_1$ path services $\frac{1}{2} \cdot \frac{1}{3} = \frac{1}{6}$ of the traffic from C_1, so $\frac{1}{6}$ of the traffic from C_1 goes through L_1. In the same way, we can compute for every client C_i and every location L_j what fraction of the traffic from C_i goes through L_j. This fraction is denoted as $part(C_i, L_j)$. For computing the value of the objective function, we iterate through every pair of clients C_i, $C_{i'}$ and every pair of locations L_j and $L_{j'}$. If C_i is serviced through a hub in L_j and $C_{i'}$ — through $L_{j'}$, then the cost is $W_{ii'} \cdot (\alpha \cdot dist(C_i, L_j) + \beta \cdot dist(L_j, L_{j'}) + \gamma \cdot dist(L_{j'}, C_{i'}))$. Note that this is traffic from C_i to $C_{i'}$. The cost of the traffic in the opposite direction is written similarly. This cost is further multiplied by $part(C_i, L_j) \cdot part(C_{i'}, L_{j'})$ because of the split of traffic at branching points. Finally, the objective function is defined as the sum of all these multiplied costs. There is one special case, the traffic from a client to itself. While in this case there is still a collection step and a delivery step (potentially split between several hubs), the transfer cost is set to 0. It is easy to verify that for states of the neural network that represent feasible solutions, the

value of the objective function, defined in this paragraph, is equal to the value of the function that the p-Hub Problem asks us to minimize. The objective function from this paragraph also has a meaningful interpretation for intermediate states that violate the feasibility constraints and is relatively smooth when flipping the values of the variables.

The described constraints and objective function can already be plugged into the general CBNN solver from Section 3.2. The resulting algorithm for the p-Hub Problem is rather slow. One way to speed it up is to notice that the bottleneck of the algorithm is the computation of the objective function. As described, it requires N^4 time. The CBNN solver performs a sequence of variable updates and each one requires the objective function to be computed multiple times for slightly modified states. Say we are performing a variable update for the variable x. Let g be the group of x and $\overline{g}$ be the variables from g that are currently ON (have value 1). We need to compute the objective function for modified states in which all variables from g are OFF except for a single variable: either x itself or a variable from $\overline{g}$. Moreover, the solver does not care about the exact value of the objective function F. What it actually needs is the difference between the value of F with $x = 1$ and the smallest value of F with a variable from $\overline{g}$ set to 1. By maintaining certain additional information, this difference can be computed much faster than in N^4 time. The computation is different for client-hub and for hub-location variables, so the two cases are described separately. The modification of the solver allows it to perform the same operation with a smaller number of computations and does not change the logic of the CBNN algorithm or the quality of the final solution.

Assume we need to update the hub-location variable HL_{jk}. Let I be the set of indexes for which CH_{ij} is 1. Changes to the objective function come from pairs of clients for which at least one is in I (all other costs are the same and will cancel out when computing the difference in the value of the objective function). The collection and distribution costs are easily and efficiently handled by precomputing for every client how much traffic goes out of it and how much traffic goes into it (these values are known from the beginning). The harder part is the transfer cost.

For computing it, we maintain the amount of traffic between every pair of hubs. The CBNN performs variable flips and the traffic between hubs can be efficiently maintained under such changes. Once this traffic is known, we can iterate through every hub j' and its location k' and add the cost of all the transfer traffic between j and j'. This completes the computation of the objective function. As described, the procedure takes at most $N \cdot P$ time. We want to stress that the *Update* function of the general CBNN solver only cares about differences of the value of the objective function and so we compute only the part of the objective function that is not equal in the two states.

The final part is updating a client-hub variable. Say we want to update the variable CH_{ij}. Again, changes to the objective function come from traffic to / from client C_i. Computing the collection and delivery costs is done in a similar way as for hub-location variables. For computing the transfer costs, we first calculate for every hub the amount of traffic to / from it. This is done in a straightforward way by iterating through all pairs of a client and a hub. Once the traffic to hubs is known, we compute the traffic to locations by iterating through hub-location pairs and branching the hub traffic appropriately. We then iterate through every location that is connected to hub H_j and add the cost of the traffic to every other location (the amount of traffic is already computed). The evaluation takes at most N^2 time.

This completes the description of the modified CBNN solver for the p-Hub Problem. In comparison to the general CBNN solver, the only non-standard part is the more efficient computation of the value of the objective function. Solving large instances is still quite slow and we put a hard limit of 20 seconds for solving an instance. For the largest instances, this limits the optimization to 15 - 20 epochs. Better results can probably be achieved by giving the optimization more time (or by further tuning the speed of the solver).

5.2.3 Test data and results

The described approach is tested on the Australia Post (AP) data set. This data set consists 200 locations (postcode districts) together with the mail flow between them.

The collection cost α is set to 3, the transfer cost $\beta = 0.75$, and the distribution cost γ is 2. This data set was created by the authors of [31]. In the paper, they additionally describe a procedure for generating smaller instances from the one with 200 locations. Figure 5.7 shows two example inputs. Table 5.2 summarizes the results of the CBNN solver on the 28 instances from the AP data set.

Figure 5.7: Two example instances from the Australia Post data set.

N	P	Optimal sol.	Error	N	P	Optimal sol.	Error
10	2	167 493.06	0	40	4	143 968.88	0
10	3	136 008.13	0	40	5	134 264.97	0.13%
10	4	112 396.07	0	50	2	178 484.29	0
10	5	91 105.37	0	50	3	158 569.93	0
20	2	172 816.69	0	50	4	143 378.05	0
20	3	151 533.08	0	50	5	132 366.95	0
20	4	135 624.88	0	100	5	136 929.44	2.27%
20	5	123 130.09	0	100	10	106 469.57	1.28%
25	2	175 541.98	0	100	15	90 605.10	0.71%
25	3	155 256.32	0	100	20	80 682.71	3.29%
25	4	139 197.17	0.04%	200	5	140 409.41	0.06%
25	5	123 574.29	0	200	10	111 088.33	0.43%
40	2	177 471.67	0	200	15	95 460.54	4.68%
40	3	158 830.54	0	200	20	85 560.39	1.95%

Table 5.2: Results of the competition-based neural network on the AP data set. The Error column shows by how much the solution, found by the CBNN, is worse than the optimal solution for the instance. 0 in this column means that the neural network has found the optimal solution.

As in [31], the test instances can be divided into two groups: small ($N \leq 50, P \in \{2, 3, 4, 5\}$) and large ($N \geq 100, P \in \{5, 10, 15, 20\}$). The neural network approach solves exactly all but two of the small instances. For these two instances, the error is so small that probably it does not matter in practice. For the large instances the average error is 1.8%. While the results on the large instances are good, they

are worse than the results on the small ones. An explanation for this is that the complexity of the instances grows quickly. One way to measure it is to count the number of terms in the objective function. This count is proportional to the square of the number of client-location pairs. For the largest of the small instances, this value is approximately $(50 \cdot 5 \cdot 50)^2 \approx 150 \cdot 10^6$. For the largest instance from the whole data set, this count is approximately $(200 \cdot 20 \cdot 200)^2 \approx 640 \cdot 10^9$. While most of the time a large fraction of the terms is zero, still, the massive increase shows that the complexity grows quickly. We put a hard limit of 20 seconds on the execution time of the CBNN, so it is expected for the results to become worse as the size of the input increases.

In comparison to the results of other approaches on the AP data set, the performance of the CBNN solver is similar to the performance of the Simulated annealing algorithm from [31]. This algorithms is an improvement of a method that was developed for Australia Post to decide the locations of mail sorting centers. The method was used by Australia Post and we can expect that its solutions are good enough for practical applications. The best results on the AP data set seem to be produced using Genetic algorithms. The algorithm from [1] is able to exactly solve all of the AP instances. In comparison to the results of Hopfield networks, the performance of the competition-based neural network solver is significantly better. To summarize the above, the performance of CBNNs is good and seems to be acceptable for applications in practice. But recent algorithms for the p-Hub Problem outperform the CBNN solver. Maybe there is a better CBNN-compatible model of the p-Hub Problem that is smaller in size. This most probably will lead to a better performance in the allocated time.

5.3 The p-Defense-Sum problem

The p-Defense-Sum Problem is an example of an obnoxious facility location problem. These problems are characterized by a push objective. The goal of p-Defense-Sum is to locate p facilities on a network without clients so as to maximize the sum of pairwise distances between the facilities. Civil and military applications are known for the model [16].

For an intuitive justification of the p-Defense-Sum Problem, assume that p military facilities need to be positioned. If an enemy is ever able to conquer them, then we want to make it as hard as possible for the enemy to establish stable control over the facilities. By maximizing the sum of distances between the facilities, we make it more time-consuming and expensive for the enemy to deploy the required infrastructure for ensuring stable control.

Another closely related model can also be found in the literature, the p-Defense-Min Problem. This problem asks us to maximize the minimum interfacility distance. Intuitively, maximizing this distance increases the cost of destroying two facilities in a single attack. If the enemy just wants to destroy the facilities instead of controlling them, then the objective function of p-Defense-Min may be more adequate.

5.3.1 Mathematical definition and known results

The p-Defense-Sum Problem has several slightly different definitions. We use the definition of unweighted p-Defense-Sum from [16].

Definition 5.3.1. (p-Defense-Sum) Find a list of vertices $u_1, u_2, ..., u_p$ that maximizes $\sum_{i<j} dist(u_i, u_j)$.

The p-Defense-Sum Problem in general graphs was shown to be NP-hard by Hansen [41]. He also shows that p-Defense-Sum can be solved in polynomial time when restricted to tree graphs.

The p-Defense-Sum Problem is a special case of the Generic Multifacility Location Problem (GMLP) considered by Chhajed and Lowe [18]. They

present an algorithm for GMLP that is designed for a special case of the problem in which the dependency graph, defined by interactions between pairs of facilities, is a k-tree. This algorithm can be used for solving p-Defense-Sum instances, but the time complexity is at least $\Omega(|V|^k)$.

Kincaid [60] applies two metaheuristics — Simulated annealing and Tabu search, to small randomly generated instances of p-Defense-Sum. The two metaheuristics are directly applied in their classical form. A 2-interchange neighborhood is used. In this neighborhood, a random vertex from the chosen list of vertices is swapped with a random vertex not in the list. Kincaid concludes that both metaheuristics produce surprisingly good results and that Tabu search performs better than Simulated annealing.

The p-Defense-Sum Problem can also be formulated as a 0-1 integer programming problem (see, for example, Kuby [69]). One possible formulation is given below. In this formulation, it is not possible to locate two facilities in the same vertex.

Maximize:

$$\sum_{a<b} y_{ab} \cdot dist(a,b)$$

Subject to:

$$\forall_{a\in\{1..n\}}\ x_a \in \{0,1\}$$

$$\forall_{a\in\{1..n\}}\forall_{b\in\{(a+1)..n\}}\ y_{ab} \in \{0,1\}$$

$$\forall_{a\in\{1..n\}}\forall_{b\in\{(a+1)..n\}}\ 2 \cdot y_{ab} \leq x_a + x_b$$

$$\sum_{a\in\{1..n\}} x_a = p$$

Each binary variable x_a indicates whether a facility is located in the corresponding vertex a. The sum constraint guarantees that exactly p facilities are placed in the graph. The binary variables y_{ab} indicate whether the distance between vertices a and b needs to be added to the value of the objective function. The distance needs to be added when there are facilities in both vertices a and b. This is represented

by the $2 \cdot y_{ab} \leq x_a + x_b$ constraints.

5.3.2 CBNN model

Assume we need to solve the p-Defense-Sum Problem in a directed weighted graph G with vertex set $v_1, v_2, ..., v_n$. One possible CBNN-compatible model of the problem is given below.

Maximize:

$$\sum x_{ia} \cdot x_{jb} \cdot dist(a, b)$$

Subject to:

$$\forall_{i \in \{1..p\}} \forall_{a \in \{1..n\}} \; x_{ia} \in \{0, 1\}$$

$$\forall_{i \in \{1..p\}} \sum_{a \in \{1..n\}} x_{ia} = 1$$

There are $n \cdot p$ binary variables. Variable x_{ia} has value 1 iff facility i is placed in vertex a. The sum constraints guarantee that every facility is placed in exactly one vertex. The objective function is the sum of terms of the form $x_{ia} \cdot x_{jb} \cdot dist(a, b)$. The meaning of each term is that if facility i is placed in vertex a and facility j is placed in vertex b, then the distance between a and b needs to be added to the value of the objective function. In this way, the distance between every pair of facilities is actually added twice, but scaling the value of the objective function does not matter when maximizing it.

The described CBNN-compatible model of p-Defense-Sum assumes that multiple facilities can be placed in the same vertex. Depending on the structure of the input graph, this can be beneficial for maximizing the sum of pairwise distances between the facilities. Locating several facilities in the same vertex sounds unnatural, so we additionally add the constraint that this should not happen. For enforcing the constraint, a small modification is made to the general CBNN solver from Section 3.2. Whenever it decides the value of variable x_{ia}, the solver first checks if some other variable x_{ja} is 1 (which means that another facility is already placed in vertex

a). If this happens, then the solver aborts the value update of x_{ia} and continues to the next variable update. The modification guarantees that there is no vertex in which two or more facilities are located.

5.3.3 CBNN solver

For the experiments in this section, the group-best variant of the general CBNN solver (Algorithm 4) is used. The solver is modified as described in the previous subsection to guarantee that no two facilities are placed in the same vertex.

The quality of the returned solutions was excellent from the first attempt, so the parameters of the solver were not tuned at all. For all instances, we run the solver just once. The initial temperature is set automatically to the maximum *OnCost* of a variable in the initial random solution.

5.3.4 Test data

The test p-Defense-Sum instances are derived from the road network of several Bulgarian cities (Appendix B describes the process of extracting geodata). To create an instance, a rectangular area inside a city is selected and all roads are extracted. In this way, the underlying graph of the instance is created. The roads are the edges of the graph and the vertices are the intersections of the roads. We additionally remove vertices of degree one. Figure 5.8 shows the graphs of three of the test instances.

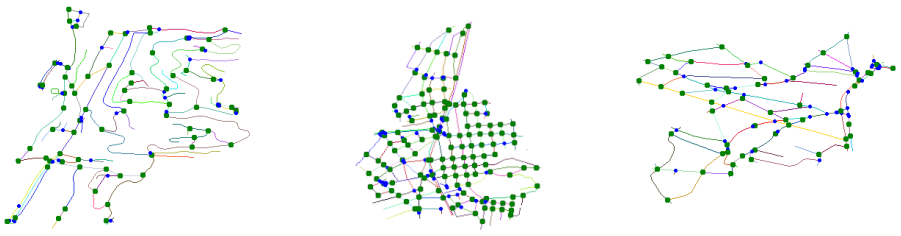

Figure 5.8: Three example p-Defense-Sum instances.

We choose 32 different areas and for each area set the number of facilities to a value between 2 and 5. This gives 128 p-Defense-Sum instances. The areas are

chosen so that the resulting graphs consist of between 40 and 200 vertices. The instances are relatively small. Competition-based neural networks can quickly solve larger problems. The reason for restricting to small instances is to be able to compute the exact solution with a method that is different from CBNNs. Surprisingly, the integer programming formulation from the previous section performs very badly even for the smallest instance (we used the CBC mixed integer programming solver [17]; it was too slow even for 40 vertices and 4 facilities). In the end, the optimal solutions for the instances were computed by an optimized backtracking algorithm.

5.3.5 Results

The results of competition-based neural networks on the Bulgarian cities data set are excellent. The described CBNN solver quickly finds an optimal solution to 127 of the 128 instances. Figure 5.9 shows the only instance that is not solved optimally. For this instance, the CBNN solver finds a solution of value 3098.676, while the optimal solution has value 3115.645.

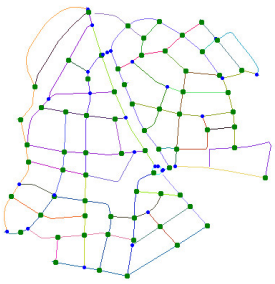

Figure 5.9: The only instance from the Bulgarian cities data set that is not solved optimally by the CBNN solver. In the instance, 3 facilities need to be positioned.

5.4 The Maximal Covering Location Problem

When locating private facilities, the decision maker usually wants to maximize the profit. Because of this, the resulting abstract model is of *MiniSum*-type, in which the goal is to minimize the sum of distances or something similar. Unlike private facility location, defining an acceptable objective function for public facility location is more complicated. The cost of a public facility is paid by every taxpayer and the chosen location of the facility needs to balance the interests of everyone. According to [22], two types of objective functions are usually employed: functions that are based on the total weighted distance and functions that minimize the distance from the furthest user of a facility. The second option is the basis of the class of covering facility location problems.

The Maximum Covering Location Problem (further called p-MCLP) is one of the first studied covering problems. It was introduced in 1974 by Church and ReVelle [22]. In the same year, a similar problem was also studied by White and Case [105]. The goal of p-MCLP is to locate p facilities while maximizing the covered population. To get an intuition about the problem, we describe an imaginary situation where it may naturally appear. Assume that medical service needs to be provided to a rural area. The area consists of a network of populated places. To provide the service, we need to choose a subset of the populated places where to build the necessary infrastructure. Each of the facilities that we build can only provide service to populated places within a given distance. We want to provide medical service to everyone while spending as little as possible. The number of facilities usually is a good measure of the expenses, so we want minimize the number of opened facilities. Sadly, it is often the case that, for example, 5 facilities are enough to cover 98% of the population, but 10 facilities are necessary to cover everyone. In this case, when the budget is fixed, it makes sense to open 5 facilities and provide alternative specialized access for the rest of the population. We arrived at the p-MCLP Problem. Our budget allows us to open p facilities and the goal is to locate them in such a way that the number of covered populated places is maximized. Review of several other applications of p-MCLP can be found

in [21] and [87]. Apart from obvious applications like locating cell phone towers, in practice, p-MCLP can be useful in conservation biology [95], for airplane crew scheduling, and for some other scheduling problems.

In this section, we study the unweighted p-MCLP Problem. By unweighted it is meant that the goal is to maximize the number of covered populated places and not the total covered population. There is a closely related weighted version of the problem and the difficulty of both of them seems to be comparable. The classical set cover problem in which we want to find the minimal number of facilities that are necessary to cover all populated placed is also closely related to p-MCLP. Many other related covering problems exist. We sometimes want to cover as many populated places as possible within a distance S, but there is an additional requirement that every populated place needs to be within distance $T > S$ from a facility. Intuitively, we want to provide good service to as many people as possible. But we need to provide some (potentially bad) service to everyone. This problem is called MCLP with Mandatory Closeness Constraints [22]. In another class of problems, the goal is to locate p facilities so as to minimize the maximum distance from a populated place to a facility. Such problems are called p-Center. Notice that in p-MCLP there is a hard limit on the distance from a facility, while in p-Center we want to minimize this maximum distance. The second case is appropriate for situations in which the quality of the service decreases with distance, but long distances are still acceptable. The first case is for situations like responding to medical emergencies where responding withing a given time limit is crucial. There are many more related covering problems, some of them are described in [107].

5.4.1 Mathematical definition and known results

In the formal definition of unweighted MCLP, we are given n sets and a number p. We need to choose p of the sets so that the size of the union of the chosen sets is maximized. The problem from the previous section can be transformed into such a formulation by generating, for every possible facility location, the set of populated places that are covered by the facility.

One possible integer programming formulation of MCLP is given below. Assume there are n sets $S_1, S_2, ..., S_n$. Each set consists of some of the the integers from 1 to m. For every integer $i \in \{1..m\}$, we define C_i to be the indexes of the sets from $S_1, ..., S_n$ that contain i. The IP formulation has two families of binary variables: $x_1, x_2, ..., x_n$ and $y_1, y_2, ..., y_m$. The variable x_i is 1 iff the set S_i is selected. The variable y_i is 1 iff the element i is covered by a selected set.

Maximize:

$$\sum\nolimits_{i\in\{1..m\}} y_i$$

Subject to:

$$\sum\nolimits_{i\in\{1..n\}} x_i = p$$

$$\forall_{i\in\{1..m\}}\ y_i \leq \sum\nolimits_{j\in C_i} x_j$$

$$\forall_{i\in\{1..n\}}\ x_i \in \{0,1\}$$

$$\forall_{j\in\{1..m\}}\ y_j \in \{0,1\}$$

The first constraint guarantees that exactly p facilities are placed. Each of the constraints in the second group can be rewritten as $y_i => \bigcup_{j\in C_i} x_j$. They say that if we think an element i is covered, then there is a selected set that covers it. The last two sets of constraints mean that all variables are binary.

The IP formulation above can be directly used for solving P-MCLP instances. According to [22], in their experiments, in 80% of the cases a Linear Programming (LP) relaxation of the problem returned a solution that consisted of only zeroes and ones (as a consequence, the same solution is optimal for the original IP problem). The remaining instances in their experiments were resolved manually by inspection. LP solvers are efficient and can be used for solving quite large instances. When the resulting solution is not all 0-1, different Branch and Bound schemes and other branching procedures can be used. For example, the paper [35] describes a Lagrangian heuristic for solving P-MCLP instances. It should be noted that it often makes sense to preprocess the given instance by applying reduction rules. One sim-

ple reduction rule is to remove a candidate location if what it covers is a subset of the elements that are covered by another location (so the other location is always a better option). More complex reduction rules can also be defined. Such rules can significantly decrease the size of the instance being solved. Additionally, heuristic methods exist for decomposing an instance into several smaller and easier ones [91].

P-MCLP instances in practice tend to be relatively simple. When generating test instances for this section, we noticed that a straightforward greedy algorithm is often able to find a very good solution (even an optimal one in many cases). The algorithm is to add every time the set that covers the largest number of uncovered elements. It can be proven that the algorithm has an approximation ratio of $1 - \frac{1}{e}$ [47]. While in practice this algorithm may perform quite well, it is known that the bound is essentially the best possible unless $P = NP$ [33]. Several extensions of the algorithm above exist that improve the solution quality. For example, an interchange heuristic can be used. In this heuristic, we try to substitute a chosen set with another one if this increases the number of covered elements. Subsequently, a local search procedure can be applied to further improve the solution.

Many of the well-known metaheuristics for combinatorial optimization have been applied to the p-MCLP problem or some of its variants. The paper [5] describes a Genetic algorithm approach for the problem. A Simulated annealing method is described in [52]. Tabu search [59] and interchange heuristics [86] have also been applied.

5.4.2 CBNN model

There are different ways to model the p-MCLP Problem for solving with a CBNN. One of them is described in this subsection. The previous subsection defines p-MCLP as a problem on sets. For convenience, here we call clients the m elements of the union of all sets. Each set is called a possible location of a facility. The p-MCLP Problem is to choose p of the locations so that the union of the clients that are covered by them is maximized. The CBNN model of the p-MCLP Problem has two families of binary variables: cf_{ij} and fl_{jk}. Intuitively, each client chooses

which facility to use. The variable cf_{ij} is 1 iff client i uses facility j. The fact that client i has chosen facility j does not mean that the client is serviced. If the facility is too far from the client, then he will receive no service. The variables fl_{jk} express whether facility j is placed in location k. The constraints below enforce the feasibility of the solution.

$$\forall_{i\in\{1..m\}} \sum\nolimits_{j\in\{1..p\}} cf_{ij} = 1$$
$$\forall_{j\in\{1..p\}} \sum\nolimits_{k\in\{1..n\}} fl_{jk} = 1$$

The first family of constraints guarantees that every client has chosen exactly one facility. The second family of constraints guarantees that every facility is placed in exactly one location.

Consider a pair of variables cf_{ij} and fl_{jk} such that the value of both of them is 1. This configuration means that client i has chosen facility j that is placed in location k. Client i is covered if location k covers him. If all constraints from the previous paragraph are satisfied, then the number of covered clients is $\sum_{i,j,k} cf_{ij} \cdot fl_{jk} \cdot |i \in S_k|$. Here $|i \in S_k|$ is 1 iff $i \in S_k$, otherwise it is 0. The goal of the p-MCLP Problem is to maximize this sum and so it looks like a good objective function for the CBNN model. The problem is that during the operation of the neural network, the constraints from the previous paragraph may not be satisfied. In the final solution all constraints are going to be enforced, but in intermediate states during the optimization it is possible for a client to choose multiple facilities and for a facility to be placed in multiple locations. Similarly to the p-MiniSum Problem, it seems reasonable to have a probabilistic interpretation of such infeasible configurations. When a client is connected to multiple facilities, the interpretation is that the client uses each one of them with equal probability. If a facility is connected to multiple locations, then it is placed in each one of the locations with equal probability. The objective function is then the expected number of covered clients. This can be written as $\sum_{i,j,k} \frac{cf_{ij}}{deg_i} \cdot \frac{fl_{jk}}{deg_j} \cdot |i \in S_k|$. deg_i and deg_j denote the number of

facilities to which client i is connected and the number of locations to which facility j is connected. When all constraints are satisfied, this sum is exactly the number of covered clients.

The described model can be directly plugged into the general CBNN solver (Algorithm 4). In our experiments, we use a modified version of the solver because some of the test instances are quite large and the general solver becomes too slow. The modification is the *update all variables at once* one that was was discussed in Section 3.2. The general solver consists of a sequence of variable updates. Every variable update needs to compute the *OnCost* of all variables in the corresponding group. The *OnCost* intuitively measures the contribution of the variable to the value of the objective function and computing it is slow. The modification of the solver is to performs all variable updates *at once*. To achieve this, we first compute the *OnCost* of every variable. We then iterate through all variables, perform the update logic and remember for which variables we need to change their value, but we do not modify anything. Since we are not changing anything, getting the *OnCost* of every variable is just a table lookup. After the update logic is executed for all variables, we perform at once all the stored changes of variable values. The described operation substitutes the sequence of variable updates that happen during one iteration of the general CBNN solver. Nothing else is changed. Apart from being faster, the described procedure is closer to the operation of a parallel implementation of competition-based neural networks on certain hardware. In our experiments with p-MCLP instances, there is no significant difference in solution quality between the general and the modified solvers.

5.4.3 Test data and results

The CBNN model from the previous subsection is evaluated on two sets of p-MCLP instances: Bulgarian cities and Steiner Triple Systems. The first set of instances is generated specifically for the experiments in this section. The second set is a well-known source of hard instances for evaluating set covering algorithms. The instances in the first set are relatively small and the general CBNN solver is used for them

without modifications. The instances in the second set are larger and we use the modified CBNN solver that was described in the previous subsection. No reduction rules or other preprocessing steps were applied to the instances.

Bulgarian cities

The first set of test instances is generated using data about the populated places in Bulgaria. Intuitively, we are placing facilities that provide service within some distance D (aerial distance, not road distance). For generating an instance, we select a rectangular geographic area and extract all populated places within it. After this, a number of facilities p is chosen and the distance limit D is computed so that the described greedy algorithm for p-MCLP covers a specified percentage C of the populated places. Once D is selected, for every populated place, we compute the set of covered populated places if a facility is located in it. This results in a p-MCLP instance. Additionally, the instance is discarded if the greedy algorithm is able to optimally solve it. The areas are chosen so that the number of populated places is around 100. The value for C is chosen to be around 80% - 95%. The number of facilities p is set to a value between 4 and 6. The sizes of the instances are small enough to be able to find the exact solution in reasonable time by an optimized iteration through all solutions. The CBNN solver can handle much larger instances. Figure 5.10 shows one example p-MCLP instance.

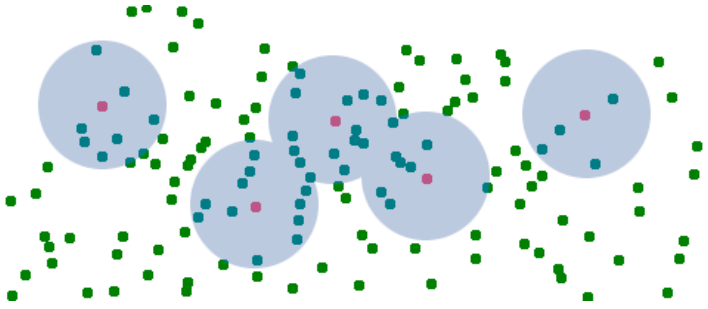

Figure 5.10: A p-MCLP instance. The green and red dots are the populated places. The red dots are the locations where facilities are placed. The large circles show the areas that are covered by the facilities.

Table 5.3 shows the results of the CBNN solver on the Bulgarian cities instances. The CBNN solver is able to solve optimally 67% of the instances and the quality

of the solutions is reasonable for all instances. On instance 7, the CBNN solver performs worse than the greedy algorithm. For all other instances, it outperforms the greedy algorithm. It is interesting that for the Bulgarian cities instances, the original CBNN solver produces better solutions than the modified solver. When the number of epochs for the modified solver is increased, then the gap in solution quality starts to narrow. It seems that on the Bulgarian cities data set the modified solver needs a more gradual temperature decrease schedule to achieve the same solution quality as the general solver.

Test	**Greedy**	**CBNN**	**Opt.**	**Test**	**Greedy**	**CBNN**	**Opt.**
1	93	95	95	14	89	93	93
2	120	124	124	15	75	75	79
3	59	61	61	16	71	73	73
4	57	58	58	17	65	65	67
5	78	78	82	18	76	76	77
6	93	96	96	19	87	91	91
7	55	54	57	20	88	92	93
8	40	41	41	21	61	63	64
9	58	60	60	22	56	57	57
10	85	88	88	23	51	53	53
11	87	92	92	24	93	93	98
12	87	93	93	25	107	113	113
13	99	99	103				

Table 5.3: Results on the Bulgarian cities data set. The numbers in the **Greedy**, **CBNN** and **Opt.** columns show how many populated places the corresponding algorithm was able to cover.

Steiner Triple Systems

A Steiner Triple System (STS) is a set A together with a family B of subsets of size 3 from A with the property that every pair of elements from A appears in a unique triple from B. The problem, posed on a STS, is to find the smallest subset of A that covers all elements from B. In terms of the p-MCLP Problem, the elements of A are the possible locations. The subsets from B are the clients. Each client is covered by exactly 3 of the locations: the ones that form the client-subset. The STS covering problem has many symmetries and was specially chosen to be hard for set covering algorithms. More information about the problem and a description of

one possible solving procedure can be found in [78]. A Steiner Triple System exists iff $|A| \equiv 1$ or 3 (mod 6). It is interesting that up to around year 2000, the exact solution of STS covering instances with $|A|$ greater than 81 was not known. Even now, we do not know the exact solutions for instances of size that is more than several hundred.

The STS covering problem asks for the minimal number of facilities that are necessary to cover all clients, while p-MCLP seeks for the maximal number of covered clients using a given number p of facilities. To convert an STS covering instance to a p-MCLP instance, we set p to be equal to the answer of the STS covering problem (the answer is computed by another method). For the resulting p-MCLP instance, we want the CBNN solver to cover all clients. STS covering instances together with their optimal solutions are taken from [6]. Table 5.4 shows an STS instance with $|A| = 9$ and the corresponding instance of the p-MCLP problem.

STS Covering instance	**p-MCLP instance**
2 3 4	2 3 7 10
1 3 5	1 3 8 11
1 2 6	1 2 9 12
5 6 7	1 5 6 10
4 6 8	2 4 6 11
4 5 9	3 4 5 12
1 8 9	4 8 9 10
2 7 9	5 7 9 11
3 7 8	6 7 8 12
1 4 7	
2 5 8	
3 6 9	

Table 5.4: An STS covering instance with $|A| = 9$ and the corresponding p-MCLP instance. Each row of the STS instance on the left corresponds to a client and the three numbers give the identifiers of the facilities that cover this client. On the right is the p-MCLP instance. Each row of it corresponds to a facility and gives the identifiers of the clients that are covered by the facility. The optimal solution of the STS instance is 5, so in the p-MCLP instance, p is set to this value.

The results of the modified CBNN solver are shown in Table 5.5. The STS instances are quite large. For example, the last one has $61 \cdot (81 + 1080) \approx 70\,000$ variables. To keep the runtime of the neural network small, the modified CBNN

solver is used in the experiments. For the instances from the STS data set, it seems that this does not degrade the quality of the returned solutions.

Instance	**CBNN**	**Optimal**	**Error**
9_12_5	12	12	0
15_35_9	35	35	0
27_117_18	117	117	0
45_330_30	323	330	2.1%
81_1080_61	1066	1080	1.3%

Table 5.5: Results of the modified CBNN solver on the STS instances. The name X_Y_Z of an instance means that there are X facilities, Y clients and Z facilities are enough to cover all clients.

5.5 Flow Intercepting Facility Location

In the p-MiniSum Problem and in many other facility location problems, the clients specially travel to a facility for receiving some service. The setting is different in flow interception problems. Intuitively, there is a city in which people travel to work. Along their paths we can open facilities. The main goal of every client (person) is to get to work, but if he passes by a facility, then he will receive its service. For example, the facilities may be advertising boards. We want to position some number of them along the roads so that as many people as possible see the advertisement. As another example, we want to position gas stations or restaurants along the roads so as to maximize the number of potential clients. This type of problems are called Flow Intercepting Facility Location (FIFL) problems.

In the problem that is considered in this section, we are given a graph (city) together with a set of paths in this graph. The paths correspond to the daily commute routes of the people that live in this city. There is additionally a budget of p advertising boards. Our goal is to optimally position these boards so that as many people as possible see the advertisement. Seeing an advertisement is binary, either yes or no. It does not matter if you saw the advertisement only once or you saw it ten times along your route.

Flow Intercepting Facility Location problems have several variants. Obviously, the paths can be weighted or unweighted (this section deals with unweighted paths). The number of facilities to open can be pre-specified or we may want to minimize the number of facilities that are necessary to cover all traffic. There may be a cost associated with positioning a facility and the goal may be to maximize the revenue that is defined as income from the traffic minus the cost of opening the facilities. Review of some of the FIFL problems and their solution methods can be found in [9].

In Section 5.4, the Maximal Covering Location Problem was presented. It can be noticed that this problem is very similar to the FIFL problems. In fact, the problem is the same: cover as many points as possible. But in flow intercepting problems, the points are actually paths and for covering a path, a facility needs

to be placed on it. It makes sense to consider flow intercepting facility location problems as a separate class of problems because the paths give special structure to the instances that is at the same time a challenge and an opportunity for obtaining better solutions.

5.5.1 Mathematical definition and known results

Assume there is a graph with vertices $v_1, v_2, ..., v_m$ together with a set of n paths $\pi_1, \pi_2, ..., \pi_n$. The goal of the FIFL Problem is to locate p facilities on the graph so as to maximize the number of covered paths. A path is covered if at least one facility is located on it. Generally speaking, a facility can be placed both in a vertex or along an edge of the graph. It is easy to prove that there is an optimal solution in which facilities are placed only in vertices [7]. This is because if a facility is located along an edge, then it can be moved in one of the two directions until it hits a vertex and the movement will not change the quality of the solution. The set of locations can further be restricted to the starting points of the paths together with their intersection vertices. For simplicity, in this subsection we allow placing facilities only in the vertices of the input graph and do not use other ways of reducing the size of the set of possible locations.

An Integer Programming formulation of the FIFL problem is given below.

Maximize:

$$\sum x_j$$

Subject to:

$$\forall_{j \in \{1..n\}} \; x_j \leq \sum_{v_i \in \pi_j} y_i$$

$$\sum_{i \in \{1..m\}} y_i = p$$

$$\forall_{i \in \{1..m\}} \; y_i \in \{0, 1\}$$

$$\forall_{j \in \{1..n\}} \; x_j \in \{0, 1\}$$

There are two set of variables, x_j and y_i. The variable x_j is 1 iff the path π_j is covered.

The variable y_i is 1 iff there is a facility in vertex v_i. The first set of constraints guarantees that if a path is considered to be covered, then there is at least one open facility on the vertices that comprise the path. The second constraint guarantees that exactly p facilities are placed in the graph. The rest of the constraints say that all variables are binary.

According to [9], the FIFL Problem is NP-hard. This is expected because set covering is NP-hard and is closely related to FIFL. For exactly solving an instance of the problem, the integer programming formulation above can be used together with any available IP solver. In [7], a specialized branch-and-bound method for the problem is described. Greedy algorithms also exist. Their main idea is to choose vertices that cover the largest number of still uncovered paths. Such algorithms have surprisingly good performance on real-world instances [7, 9].

5.5.2 CBNN model

As already mentioned, the Flow Intercepting Facility Location Problem is actually the Maximal Covering Location Problem from Section 5.4 that is stated over a special graph. We can use the described CBNN model of the Maximal Covering Location Problem from Section 5.4 to model FIFL instances. For this, we call clients the n input paths. The locations are the m nodes along the paths. Using this analogy, the CBNN formulation from Section 5.4 can be written as below.

Maximize:

$$\sum_{i,j,k} \frac{cf_{ij}}{deg_i} \cdot \frac{fl_{jk}}{deg'_j} \cdot |Node_k \in Path_i|$$

Subject to:

$$\forall_{i \in \{1..n\}} \sum\nolimits_{j \in \{1..p\}} cf_{ij} = 1$$

$$\forall_{j \in \{1..p\}} \sum\nolimits_{k \in \{1..m\}} fl_{jk} = 1$$

The interpretation of the variables is the same as the interpretation for the Maximal Covering Location Problem. Having $cf_{ij} = 1$ and $fl_{jk} = 1$ means that if client (path) i is covered at all, then it is covered by facility j that is placed in location k. The final decision of whether the path is covered is pushed to the objective function: the term $|Node_k \in Path_i|$ is 1 if and only if $Node_k$ is indeed along $Path_i$. Notice that there may be several facilities along a given path and we should not count the path to be covered multiple times. This is achieved by the first set of group constraints guaranteeing that a path is connected to exactly one facility. The second set of group constraints guarantees that a facility is placed in exactly one location. The deg_i and deg'_j in the objective function stand for the number of facilities that are connected to client i and for the number of locations that are connected to facility j (client i is connected to facility j iff $cf_{ij} = 1$). In the final solution, all these values are 1, but in intermediate solutions, the group constraints may be violated and these values may not be 1. The form of the objective function is specially chosen so that it has good behavior for such intermediate solutions (the same objective function is used in Section 5.4, where the reasoning behind it is explained).

The CBNN model of a problem is not unique. Because the model above coincides with the one for the Maximal Covering Location Problem, we give another possible CBNN formulation of the FIFL Problem. There are two sets of binary variables, x_{ij} and y_{kj}. Every pair of a path i and a vertex j along it corresponds to a variable x_{ij}. The value 1 of a variable x_{ij} says that if the path i is covered at all, then it is covered by a facility in vertex j. The variables y_{kj} correspond to pairs of a facility k and a vertex j. y_{kj} is 1 iff facility k is placed in vertex j. The solution feasibility constraints are written as below.

$$\forall_{i\in\{1..n\}} \sum\nolimits_{j\in\{1..m\}} x_{ij} = 1$$
$$\forall_{k\in\{1..p\}} \sum\nolimits_{j\in\{1..m\}} y_{kj} = 1$$

These group constraints guarantee that every path is connected to exactly one vertex

and every facility is connected to exactly one vertex. They do not remove the possibility for two facilities to be placed in the same location. In this case, we should be careful to not overcount the number of covered paths. The easiest way is to ensure that in the final configuration no two facilities are placed in the same vertex. Guaranteeing this results in overlapping group constraints: every facility needs to be placed in exactly one vertex and every vertex needs to be connected to at most one facility. Overlapping group constraints are discussed in Section 5.6, where we deal with the Assignment Problem. The same idea of pushing one of the constraints to the objective function can be applied here. The objective function to maximize becomes $\sum \frac{x_{ij}}{deg_i} \cdot \frac{y_{kj}}{deg'_k} \cdot \frac{1}{deg''_j}$. When both $x_{ij} = 1$ and $y_{kj} = 1$, then the path i is serviced through a facility in vertex j that is indeed opened. This means that 1 needs to be added to the number of covered paths. This value 1 is scaled so that the objective function has good behavior for infeasible intermediate states in which a facility is placed in multiple vertices or a path is serviced through multiple vertices. The scaling coefficients are deg_i, deg'_k and deg''_j. deg_i in the objective function is the degree of path i (to how many locations it is connected in the current configuration). Similarly, deg'_k is the number of locations to which facility k is connected. deg''_j is the number of facilities to which location j is connected. If any of the deg_i, deg'_k, and deg''_j values is zero, then the corresponding term in the sum is set to 0. In the final solution, the feasibility constraints are satisfied and so deg_i and deg'_k have value 1. The division by deg''_j is the way of enforcing the overlapping group constraints. It makes it unattractive to place multiple facilities in the same vertex and, in the local minima of the objective function, there will be no such violations. In fact, there could be such violations if the optimal solution is the same for $p - 1$ and p facilities (that is, we can open more facilities than are needed to cover all paths). But in such a case, we can discard any of the violating facilities without changing the quality of the solution. Overlapping constraints are discussed in more detail in Section 5.6.

5.5.3 CBNN solver

For the experiments with the FIFL instances, we use the group-best variant of the general CBNN solver (Algorithm 4). For the first formulation that is based on the model of the maximal covering location problem, we additionally apply the *update all at once* modification from Section 3.2. That is, for all updates during an epoch, we compute the updated values of the variables but do not apply them immediately. The new values are stored an applied at once at the end of the epoch. This allows the solver to precompute *OnCost*s and other information to speed up the updates.

5.5.4 Test data and results

For generating test instances, we selected 10 rectangular areas from Sofia and extracted the road network in them as described in Appendix B. After that, for every rectangular area, we generated 400 random non-self-intersecting paths and iteratively selected 80 of them. Every path that is added to an instance is chosen so that it has several intersections with the previously added paths. The idea is to make the generated instances dense enough to be interesting. Random paths are used instead of shortest ones because shortest paths often seem to have a structure of the following type: go as quickly as possible to a main street, use the main street for as much as possible, and then turn to the final destination. This creates high-traffic junctions and in turn simplifies the instance because covering such junctions will probably give an optimal solution. Random paths make the traffic more even and deciding where to place facilities becomes harder. An example FIFL instance is shown in Figure 5.11.

For getting a complete FIFL instance, we also need to specify the number of facilities p. This number is set to a value between 2 and 6. The result is a set of 50 instances on which the CBNN solver is evaluated. The optimal solutions for the instances are computed using the described IP formulation and the Cbc mixed integer programming solver [17].

The CBNN solver that uses the first of the two described models of the FIFL Problem (the model that is taken from the Maximal Covering Location

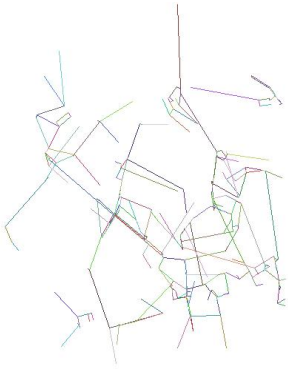

Figure 5.11: An example instance of the Flow Intercepting Facility Location Problem. The 80 paths in the instance are drawn using different colors. The coordinates of the junctions are kept, but the exact shape of the streets between the junctions is simplified to straight lines.

Problem) shows good performance on the test instances. From the set of 50 instances, it optimally solves 47. The remaining 3 instances are all with a number of facilities p equal to 6 and for them the CBNN solver is able to cover one path less than in the optimal solution. This is not bad, but we expected that the solver will optimally solve all instances. The reason is that the greedy algorithm that selects locations covering the maximal number of still uncovered paths performs extremely well of the FIFL instances, so they should be easy.

The results of CBNN solver applied with the second model of the FIFL Problem are worse. It optimally solves approximately half of the instances. For both models, the runtime of the solver is restricted to 10 seconds. We additionally performed an experiment in which the CBNN solver with the second model was executed with a time limit of 2 minutes (in our opinion, this is too much for such small instances). In this case, the solver optimally solved all 50 inputs. It seems that the second model requires more time to converge than the first one. This is somewhat unexpected because the second model is smaller in size than the first one.

5.6 The Assignment problem

The Assignment Problem is a classical combinatorial optimization problem. It has several different equivalent formulations. In one of them, there are n workers and n jobs. Every worker needs to be assigned to exactly one job so that no two workers are assigned to the same job. For every pair of worker i and job j, there is a profit $p[i, j]$ of assigning i to j. The goal of the problem is to find a valid assignment that maximizes the sum of profits. This can be modeled as a bipartite graph. If the workers are placed in the left side of the graph and the jobs in the right side, then in graph theory terms a valid assignment is called a perfect matching. The assignment problem asks for a perfect matching that has the maximum profit. An example instance of the problem is shown in Figure 5.12.

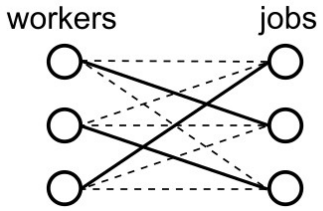

Figure 5.12: Example instance of the Assignment Problem. The nodes on the left are the workers and the nodes on the right are the jobs. Assuming the solid edges have profit 2 and the dashed edges have profit 1, the optimal solution to the instance has profit 6 and consists of the solid edges.

The assignment problem is usually not considered to be a facility location problem. There are some similarities, but the essence of the assignment problem is different. We decided to evaluate competition-based neural networks on it because it looks like an NP-hard problem but can be solved efficiently. The second reason is that it requires overlapping group constraints. This is discussed later, when the CBNN formulation of the problem is given. Here we just mention that in the Assignment Problem, every worker needs to be assigned to exactly one job and every job needs to be assigned to exactly one worker. This is a two-way constraint. In facility location problems, the constraints are usually one-way: every client needs to be assigned to exactly one facility, but a single facility can be assigned to multiple clients.

The Assignment Problem is clearly related to finding matchings. The Maximal Matching Problem is another classical combinatorial optimization problem. It asks for the matching of maximal size in a given bipartite graph. This is a special case of the Assignment Problem. Related to the topic of matchings is the topic of network flows. The Assignment Problem by itself is a special case of the Max-Flow-Min-Cost Problem. So far, we considered only matchings in bipartite graphs. They can also be defined in general undirected graphs. The problem of finding a maximal matching with minimal cost in a general graph is an extension of the assignment problem. The problems, mentioned in this paragraph, may seem to be NP-hard, but all of them can be solved in polynomial time. Except for the algorithm for maximal matching with minimal cost in general graphs, the algorithms for these problems additionally are relatively simple to implement.

5.6.1 Mathematical definition and known results

The Assignment Problem predates the computer era. Recently, it was discovered that an algorithm for it was invented by Carl Gustav Jacobi and was described in his work "About the research of the order of a system of arbitrary ordinary differential equations" that was published around year 1890. The knowledge was lost for more than a century until an algorithm for the problem was reinvented in the 1950s by Kuhn.

The assignment problem can be written as an integer programming problem as follows:

Maximize:

$$\sum x_{ij} \cdot p[i,j]$$

Subject to:

$$\forall_{i \in \{1..n\}} \sum_{j \in \{1..n\}} x_{ij} = 1$$

$$\forall_{j \in \{1..n\}} \sum_{i \in \{1..n\}} x_{ij} = 1$$

$$\forall_{i \in \{1..n\}} \forall_{j \in \{1..n\}} \; x_{ij} \in \{0,1\}$$

The variable x_{ij} is 1 iff worker i is assigned to job j. The first group of constraints guarantees that every worker is assigned to exactly one job. The second group of constraints guarantees that every job is assigned to exactly one worker. The last group says that all variables are binary. While general IP solvers can be used for finding an optimal solution, the Assignment Problem has special properties that simplify it. It can be shown that the last group of constraints that forces all variables to be binary can be dropped and the resulting linear programming problem still has an optimal solution in which all variables are binary. This directly means that the Assignment Problem can be solved in polynomial time because what is left is a linear programming problem and such problems can be solved efficiently. There are also specialized algorithms for the Assignment Problem that are simple and fast. For example, the Hungarian algorithm [76]. When implemented using Fibonacci heaps, its runtime is $O(n^2 lg(n))$. A simple $O(n^3)$ implementation of the Hungarian algorithm is fast enough to solve instances with 1000 workers and jobs in several seconds on a regular computer. The paper [29] reviews several other approaches to the problem. The authors of papers also describe an approximation algorithm that runs in time $O(n^2)$ for any fixed error bound.

5.6.2 CBNN model

In the description below, it is assumed that all profits are positive. If this is not the case, a large enough value C can be added to every element of the profit matrix without changing the optimal assignment.

The Assignment Problem has two types of feasibility constraints: every worker needs to be assigned to exactly one job and every job needs to be assigned to exactly one worker. Each one of the two types of constrains can easily be modeled by group constraints of competition-based neural networks. But modeling both of them at the same time is problematic for classical CBNNs. This is why we use group constraints to model the requirement that every worker is assigned to exactly one job and use a specially chosen objective function to enforce the other set of constraints.

Below, our CBNN-compatible formulation of the Assignment Problem is given.

Maximize:

$$\sum \frac{x_{ij} \cdot p[i,j]}{deg_j}$$

Subject to:

$$\forall_{i \in \{1..n\}} \sum_{j \in \{1..n\}} x_{ij} = 1$$

$$\forall_{i \in \{1..n\}} \forall_{j \in \{1..n\}} \; x_{ij} \in \{0, 1\}$$

As in the IP formulation, x_{ij} are binary variables that indicate whether worker i is assigned to job j. The first group of constraints are classical CBNN group constraints that guarantee that every worker is assigned to exactly one job. The deg_j values in the objective function stand for the degree of job j ($deg_j = \sum_{i \in \{1..n\}} x_{ij}$). It can be seen that for local maxima of the objective function, it is not possible to have two workers assigned to the same job (assuming there are no jobs of equal cost; if such jobs exist, then after the optimization ends, the possible violations can be resolved by a simple procedure). This is because if both a and b are assigned to the same job, then if the profit from a is larger than the profit from b, assigning b to any other free job increases the total profit. It was shown that CBNNs find local maxima of the objective function and, as a result of this, the solver will automatically enforce the constraint that every job is assigned to exactly one worker.

5.6.3 CBNN solver

For the experiments, we use the group-best variant of the general CBNN solver (Algorithm 4). The solver is applied without modifications. It is worth mentioning that the computation of *OnCost* here is slightly more complicated. This is because assigning worker i to job j changes deg_j and this affects the profits of all other workers that are assigned to j (such violating configurations in which more than one worker is assigned to a job can appear during the optimization process). Because of this, when computing the *OnCost* of an $i \to j$ assignment, we should also account for

the (decrease of) profit from the other workers that are assigned to j. Luckily, this can be done efficiently by maintaining for every job the sum of profits of all workers assigned to it together with the deg_j values.

As mentioned several times, there are two general strategies for the hyperparameters of the solver: large number of epochs with smaller number of iterations per epoch or smaller number of epochs with large number of iterations. For every test instance, we perform 4 runs of the solver: two with the first strategy and two with the second one. The same hyperparameters are used for all instances and we did not try to significantly tune them. Generally, the first strategy performs better and applying the second one is not necessary.

5.6.4 Test data and results

For testing the CBNN solver, we generated 30 random instances of the textscAssignment Problem of size 50. For every instance, we first uniformly at random choose an upper bound of the profit between 200 and 500 and then randomly fill in the profit matrix of the instance with values between 100 and the chosen upper bound. The values of the bounds in the generation procedure are empirically selected so that the resulting instances are relatively hard to solve. If the upper bound of the profit is very small, then there is very little variation in the profit matrix and the instance is simple. On the other hand, a very large upper bound also results in an easy instance because then iteratively picking the largest values in the worker - job profit matrix gives an optimal solution.

The optimal solution for every instance is computed using the Hungarian algorithm. We also run a greedy and local search algorithms to have something to compare with the results of the CBNN solver. The greedy algorithm finds the largest element $p[i, j]$ in the profit matrix, assigns worker i to job j, and repeats the same process for the rest of the matrix. The local search algorithm starts from a random assignment, repeatedly chooses two workers i_1 and i_2, and swaps their assigned jobs if this improves the solution. Because there is a lot of randomness in this process, 100 independent runs of the local search algorithm are performed for every instance.

Additionally, a run of the local search algorithm is performed starting from the solution returned by the greedy algorithm. The results of the solution methods are summarized in Table 5.6.

Solution method	**Result**
Greedy	97.81%
Local Search	99.07%
Local Search (G)	99.02%
CBNN	99.70%

Table 5.6: Results on the ASSIGNMENT PROBLEM instances. The solution methods are described in the paragraph above. The Local Search (G) method is local search starting from the solution that is returned by the Greedy algorithm. The Result column gives the quality of the solution for the corresponding method averaged over the 30 test instances. The quality of a solution is its profit divided by the profit of the optimal solution for the instance and multiplied by 100 to have an interpretation as percentage.

As can be seen from Table 5.6, the results of the solution methods are good. The CBNN solver obtains the best results. It also outperforms all other methods on all 30 test instances (but sometimes the difference is very small). None of the methods is able to find the optimal solution for any of the test instances. We also want to mention that our sequential implementation of the CBNN solver is much slower than the other considered methods.

Chapter 6

Conclusion

The need to solve combinatorial optimization problems arises very often. Crew scheduling, facility location, vehicle routing, assembly line balancing, and frequency assignment are just several examples of the wide range of optimization problems that appear in practice. If we need to solve such a problem and have an unlimited budget, then it makes sense to investigate into the special properties of the problem at hand and to develop a highly customized algorithm for it. This is widely believed to result in the best solution. But developing a highly customized algorithm is time-consuming and expensive. And even if we have the resources for this, it is still good to base the algorithm on an idea that has proven itself for other problems. When the budget is small, we look for a method that returns very good solutions but works out-of-the-box and with little problem-specific tuning. Metaheuristics for combinatorial optimization offer exactly this. They are useful ideas, or templates, for building algorithms.

Many metaheuristcs are known. Some of the most popular ones are Simulated annealing, Tabu search, and Genetic algorithms. Section 2.2 describes the main ideas of 8 established metaheuristics. It is noted that the metaheuristics have different strong sides and have bias towards different properties of the optimization problem being solved. Among all approaches to combinatorial optimization, there is a class of neural network methods. Their strong side is the existence of a natural and efficient parallel implementation and the possibility to use special hardware like

optical computers, systolic arrays, and field-programmable gate arrays. This should not be underestimated because multiprocessor systems and parallel algorithms can significantly speed up the computation and allow complex problems to be solved in a reasonable amount of time. But the existing neural networks for combinatorial optimization have very serious problems. The methods can be classified into two groups: Hopfield networks and self-organizing approaches. Hopfield networks do not produce good solutions. In this respect, they are easily outperformed even by very simple algorithms. The self-organizing approaches to combinatorial optimization can be applied to a very limited set of problems. It is clear that with such downsides, the existing neural network methods can not become an established tool for solving optimization problems in practice.

We believe that the idea of using massively parallel systems of simple computing units (that is, neural networks) to solve combinatorial optimization problems is good. The skepticism surrounding the field is largely because of historical reasons. Hopfield networks were the first proposed neural network method for combinatorial optimization. But they were not designed as a machinery for optimization. They were initially designed as an associative memory. Hopfield and Tank noticed that the model can solve instances the Travelling Salesman Problem and is a way of obtaining a meaningful digital result from analog computations. This produced a lot of enthusiasm in the optimization community. People started to ask themselves if neural networks are better suited for solving NP-hard problems than standard digital computers. The results soon followed: Hopfield networks do not scale. For instances of moderate size, they are not only unable to find good solutions, but they can not even find feasible ones. No Hopfield network of polynomial size can solve the Travelling Salesman Problem to a desired accuracy. And so on. Some of the negative results can be found in [14, 53, 106]. The set of negative results created doubts about whether neural networks are suitable at all for combinatorial optimization. And this stopped the progress in the field.

Finding a good way of using neural networks for solving combinatorial optimization problems is still an open question. This work proposes to base the neural

networks for combinatorial optimization on a competition mechanism between the neurons. The mechanism allows the model to maximize functions and, at the same time, to enforce certain feasibility constraints. Section 3.2 describes a general CBNN solver for optimization problems that uses the mechanism of competing neurons. In Chapter 4, several desirable properties of the proposed algorithm are proven: it finds local optima of the objective function, it asymptotically converges to an optimal solution, and its speed of convergence can be estimated. From theoretical perspective, the guarantees of the CBNN solver are either equivalent or are stronger than the known guarantees of the established metaheuristics for combinatorial optimization.

Theoretical guarantees are a good thing. They give a hint that the competition-based neural network is doing something reasonable. It is not biased in the sense that we think the neural network optimizes function A but, in reality, it optimizes a completely different function B. Just theoretical guarantees are not enough, though. What matters a lot for metaheuristics is their empirical performance on the problems that we need to solve in practice. Metaheuristics need to be able to automatically adjust to the input instances and to quickly find very good solutions without the tedious work for the developer of manually inspecting the instances.

In Section 5, the proposed competition-based neural networks are evaluated on 6 facility location problems. We selected this class of problems because it offers a wide range of practically important tasks that are at the same time intuitive and easy to state but are hard to solve. Competition-based neural networks demonstrate very promising performance on the test instances. They are often able to optimally solve the input. When the returned solution is not optimal, it is always at most several percent worse than the optimal one. This level of performance seems to be enough for the practical applications of the facility location problems that were used in the experiments. The performance on the test problems is also very competitive to that of other metaheuristics.

6.1 Future work

A lot of work is yet to be done for the neural networks to establish themselves as a good and reliable metaheuristic. As we already said, our opinion is that neural networks have the potential to rapidly find good solutions to combinatorial optimization problems. We also think that the competition mechanism between neurons is beneficial for this. The CBNN solver provides an example of a neural network algorithm that quickly finds excellent solutions to a range of facility location problems. Having such an example is good, it acts as a proof of a concept.

An important step for future developments is to apply neural networks to more combinatorial optimization problems. We are not saying that the CBNN solver is ideal or that it is the only possible neural network method. The solver is just an initial step. In fact, we designed it with a bias towards facility location. The idea of using competing neurons has a more general implementation than the CBNN solver from Section 3.2. We created the CBNN problem from Section 3.1 and the restricted CBNN solver because it is enough for the problems that we targeted and we wanted to keep the amount of work manageable so that we can prove the asymptotic convergence and discuss the speed of convergence. For other problems, maybe it is good to have a slightly different implementation of the mechanism of competing neurons and their groups. Maybe it is also beneficial to combine it with something else. The mechanism seems to work for combinatorial optimization and can act as a basis for neural network algorithms.

Developing a method is one thing. A different thing is making it popular. We really like integer programming because it offers an intuitive *language* for defining problems and has available solvers. You define the problem, pick a solver, and receive a solution. If we want neural networks for combinatorial optimization to become a popular tool, then something similar needs to be done. Either a new language for defining problems that is better suited for neural networks needs to be developed, or an existing one can be used. Then a reasonably efficient and stable solver needs to be written. This is a large effort that is often underestimated.

Appendix A

Markov chains

Markov chains are a convenient model for analyzing competition-based neural networks. Here we state several well-known facts about them that are used for proving asymptotic convergence of CBNNs and for estimating the speed of convergence. Markov chains are well studied. Good overview of their properties can be found in [11, 34, 50].

In the Markov chains we are interested in, there is a finite state space S. The chain itself is a collection of random variables $\{X_1, X_2, ..., X_i, ...\}$. Each one of the variables is drawn from S. The chain satisfies the Markov property:

$$Prob(X_i = v_i \mid X_1 = v_1, X_2 = v_2, ..., X_{i-1} = v_{i-1}) = Prob(X_i = v_i \mid X_{i-1} = v_{i-1})$$

This property is sometimes called lack of memory: the probability of the current state depends only on the previous state and not on how we got to the previous state. $Prob(X_i = v_i \mid X_{i-1} = v_{i-1})$ are usually called transition probabilities.

An intuitive way to think about Markov chains is that we have a robot. In its hand there is a ball of a certain color. In total, there are S possible colors that are called a state space. For the robot, the time is discrete and can be written as the numbers 0, 1, 2, 3, ... At every point in the time sequence, the robot looks at the ball in its hand. Based on its color, the robot selects a new color. It then removes the current ball from its hand and puts there a new one with the newly chosen color. The robot then continues to the next time step. If we write down the sequence of

colors $X_0, X_1, X_2, ...$, we end up with the sequence of random variables from the previous section. The robot is not deterministic and it can perform a very complex procedure for choosing the next color. What is important is that the robot only knows the color of the ball in its hand. It does not have a working memory where it can store all the colors of the balls so far. Generally speaking, when deciding the color of the next ball, the robot can check the current time. This complicates the analysis of the resulting Markov chain. For simplicity, we do not allow the robot to perform such checks. The operation of the robot becomes straightforward: it has a table in its "head" that tells, given the color of the ball in its hand, what is the probability distribution for the next color. What the robot does is to select a color following this probability distribution. This type of Markov chains are called homogeneous and are defined below.

Definition A.0.1. (Homogeneous Markov chain) A Markov chain $\{X_n \mid n \in \mathbb{N}\}$ is called homogeneous if for all $a, d \in \mathbb{N}$ and $v_i, v_j \in S$ it holds that $Prob(X_{a+d} = v_i \mid X_a = v_j) = Prob(X_d = v_i \mid X_0 = v_j)$.

The definition basically says that in homogeneous chains, the probability of the next state does not depend on the current time.

Definition A.0.2. (Transition probability matrix) A state transition probability matrix for a homogeneous Markov chain with a discrete state space S of size n is a $n \times n$ matrix P for which $P_{ij} = Prob(X_2 = v_j \mid X_1 = v_i)$.

For homogeneous Markov chains, the transition matrix P together with the probability distribution ini of the initial state X_0 is enough to define the whole process. It is easy to see that the probability distribution of X_k at any time step k can be written as $ini \cdot P^k$ (ini here is a row vector and P^k is the k-th power of P). A special property of the matrix P is that every row of it is a probability distribution: the entries of every row are real numbers between 0 and 1 that sum up to 1. Such matrices are called stochastic matrices.

An important property of Markov chains is that they (in certain cases) converge. This means that if the chain is long enough, then every next element in it is drawn

from the same probability distribution π no matter what the initial distribution *ini* is. In the robot analogy, if the machine runs long enough, then it starts to produce balls of color that follows the probability distribution π (called stationary distribution). The following definition formalizes this.

Definition A.0.3. (Stationary distribution) Assume there is a Markov chain $X_1, X_2, ...$ with state space $S = \{1, 2, ..., n\}$ and transition matrix P. The stationary distribution of the Markov chain (if it exists) is defined as a stochastic vector π whose i-th component is given by $\lim_{k\to\infty} Prob(X_k = i \mid X_1 = j)$ for all j.

Two properties of Markov chains that hold for the chains of competition-based neural networks are defined below. These properties guarantee that a stationary distribution exists for the Markov chain.

Definition A.0.4. (Irreducible chain) A Markov chain is irreducible if for any pair of states v_1 and v_2, the probability of reaching v_2 from v_1 in a finite number of transitions is nonzero.

Definition A.0.5. (Aperiodic chain) For a state v_i, let D_i consist of all integers $k > 0$ for which $P^k[v_i, v_i] > 0$. The Markov chain is aperiodic if $gcd(D_i) = 1$ for every state v_i.

The following straightforward lemma is convenient for showing that an irreducible Markov chain is aperiodic.

Lemma A.0.1. *If for an irreducible Markov chain with transition matrix P there is a state i with $P[i, i] > 0$, then the Markov chain is aperiodic.*

If a Markov chain is aperiodic and irreducible, then it converges to a unique stationary distribution. Theorem A.0.1 states this result. A proof of the theorem can be found in [34].

Theorem A.0.1. *Let P be the transition matrix of an irreducible and aperiodic Markov chain. Then there exists a unique stationary distribution π the components of which are determined by the following equation:*

$$\sum_i \pi_i \cdot P[i, j] = \pi_j \text{ for all } j$$

Theorem A.0.1 says that $ini \cdot P^k$ converges to π for any initial distribution ini (P is the transition matrix of the Markov chain). But it does not say anything about the speed of convergence. Results from linear algebra about eigenvectors and eigenvalues can be used for estimating the speed of convergence.

Definition A.0.6. (Eigenvector / eigenvalue) A nonzero vector v is called a left eigenvector for a matrix M if $v \cdot M = \alpha \cdot v$ for a number α. α is called the corresponding eigenvalue.

Notice that the stationary distribution π is an eigenvector of the state transition matrix P with eigenvalue 1 (eigenvectors can be arbitrary scaled; π is scaled so that its components sum up to 1 because it needs to be a probability distribution). The definition above talks about left eigenvectors. Similarly, right eigenvectors can be defined if we substitute the left matrix multiplication $v \cdot M$ with right multiplication $M \cdot v$. It is well-known that the multiset of left eigenvalues is the same as the multiset of right eigenvalues.

It was already mentioned that the matrix P of a Markov chain, a stochastic matrix, has special properties. A simple one is that $P \cdot \vec{1} = \vec{1}$ where $\vec{1}$ is a vector of ones. This is because every row of P sums to 1 and it shows that 1 is an eigenvalue of P. The following result is a consequence of the Perron-Frobenius theorem and says that 1 is the largest by absolute value eigenvalue of P.

Theorem A.0.2. *Let P be the transition matrix of an aperiodic irreducible Markov chain. If $\alpha_1, \alpha_2, ..., \alpha_n$ is the list of eigenvalues of P, ordered by absolute value, then $\alpha_1 = 1$ and $\alpha_1 > |\alpha_i|$ for any other eigenvalue α_i of P.*

The proof of the theorem can be found in [90]. A simple intuition about one possible proof is the following. It is obvious that $P \cdot \vec{1} = \vec{1}$ because the rows of P sum up to 1. This means that 1 is an eigenvalue of the matrix P. Assume that α is an eigenvalue for which $|\alpha| > 1$ and that an eigenvector $\vec{v}$ corresponds to α. We know that $P \cdot \vec{v} = \alpha \cdot \vec{v}$. This means that $P^k \cdot \vec{v} = \alpha^k \cdot \vec{v}$. If $|\alpha| > 1$, then the length of $\alpha^k \cdot \vec{v}$ grows infinitely as k increases. So the length of $P^k \cdot \vec{v}$ should grow infinitely as k increases. This in not possible. P to any integer power k remains a stochastic

matrix and so all elements of P^k are at most 1. For $P^k \cdot \vec{v}$ to grow infinitely, it should be true that at least one element of P^k grows infinitely. This shows that all eigenvalues of P are at most 1.

The theorem can be used to prove that Markov chains converge. Intuitively, if all eigenvalues are different, then the spectral decomposition of $P^k = \sum {\alpha_i}^k \cdot u_i \cdot w_i$. The α_i are the eigenvalues of P, w_i and u_i are the corresponding left and right eigenvectors. For every eigenvalue that is less than 1 by absolute value, the summand diminishes as k grows. In the end, only the summand of $\alpha_i = 1$ is left. Similar reasoning shows that $P^k = \vec{1} \cdot \pi + O(k^{deg-1} \cdot |\alpha_2|^k)$ in the general case. The matrix P can have several equal eigenvalues (the property is called algebraic multiplicity of the eigenvalue) and this is why there is a term of the form k^{deg-1} (deg is the multiplicity of α_2). A strict description of the reasoning from this paragraph can be found in the beginning of Chapter 6 of [11]. The important conclusion from the formulas is that the speed of convergence of the Markov chain is geometric and depends on the absolute value of the second-largest by absolute value eigenvalue. This is formalized below in the definition of mixing time.

Definition A.0.7. (Mixing time) For an aperiodic irreducible Markov chain $X_1, X_2, ...$ with state transition matrix P, the mixing time is the smallest t for which $|Prob(X_t \in A) - \pi(A)| \leq \frac{1}{4}$ for all subsets A of the state space and all initial distributions ini of the Markov chain. Here $Prob(X_t \in A)$ denotes the probability of X_t being in the subset A and π is the stationary distribution of the Markov chain.

Theorem A.0.3. ***(Speed of convergence)*** *Let $\mu = |\alpha_2|$, where α_2 is the second-largest by absolute value eigenvalue of the transition matrix P. Then the mixing time of the Markov chain is $O(\frac{1}{log(\frac{1}{\mu})})$.*

One more property of Markov chains is useful for analyzing competition-based neural networks. Assume that P is the state transition probability matrix of an aperiodic irreducible Markov chain. We know that the chain has a stationary distribution π and it is interesting to see how this stationary distribution changes when the matrix P is slightly changed. Results concerning this are called perturbation

bounds. A survey of known bounds can be found in [19]. The bound below is taken from [89].

Theorem A.0.4. ***(Perturbation bound)*** *Let P and $\widetilde{P}$ be the state transition probability matrices of two aperiodic irreducible Markov chains with stationary distributions π and $\widetilde{\pi}$. The two chains are defined over the same set of states. Assuming $E = P - \widetilde{P}$, it holds that $\|\pi - \widetilde{\pi}\|_1 \leq \|Z\|_\infty \|E\|_\infty$. $Z = (I - P + \vec{1} \cdot \pi)^{-1}$ is called the fundamental matrix of P. The 1-norm of a vector $\|v\|_1$ is the absolute sum of its entries. The ∞-norm of a matrix $\|M\|_\infty$ is the maximum absolute row sum.*

Notice that the matrix Z depends only on P and when P is fixed, $\|Z\|_\infty$ is a constant. This means that we can make the difference between π and $\widetilde{\pi}$ arbitrary small by pushing $\widetilde{P}$ closer to P. In this sense, very small perturbations of the matrix P change the stationary distribution just slightly.

Appendix B

Datasets based on geographic data

Many facility location problems have a very natural interpretation in terms of road networks. Such interpretations are often used in this work to introduce the problems. For example, the p-MiniSum Problem (Section 5.1), the p-Hub Problem (Section 5.2) etc. When choosing the data on which to evaluate competition-based neural networks, we wanted to select test instances resembling facility location problems that are solved in practice. This is why we decided to generate test instances based on a real-world road network and selected the Bulgarian road network as an underlying graph. OpenStreetMap [25] was chosen as a source of geographic data. Here it is briefly described how data from this resource can be used to generate test instances for facility location problems.

OpenStreetMap (OSM) is a collaborative project to create a map of the world. The geodata of the project can be queried in different ways, one of which is by using Overpass XML queries [79]. One of the endpoints for executing such queries is `http://overpass-api.de/api/interpreter`.

Listing B.1 below shows two possible queries for retrieving information about the populated places and the roads in a given region.

Listing B.1: Overpass XML queries for fetching populated places and roads

```
1 <!-- Query for retrieving populated places -->
2 <osm-script output="json" timeout="25">
3   <union>
4     <query type="node">
```

```
5         <has-kv k="place" regv="village|town|city"/>
6           <has-kv k="is_in:country" v="COUNTRY"/>
7           <bbox-query w="WW" s="SS" e="EE" n="NN"/>
8       </query>
9     </union>
10    <print mode="body"/>
11    <recurse type="down"/>
12    <print mode="skeleton" order="quadtile"/>
13 </osm-script>
14
15 <!-- Query for retrieving roads -->
16 <osm-script output="json" timeout="25">
17    <union>
18      <query type="way">
19        <has-kv k="highway"/>
20        <has-kv k="highway" regv="motorway|trunk|primary|secondary|
              tertiary|unclassified"/>
21        <bbox-query w="WW" s="SS" e="EE" n="NN"/>
22      </query>
23    </union>
24    <print mode="body"/>
25    <recurse type="down"/>
26    <print mode="skeleton" order="quadtile"/>
27 </osm-script>
```

In the example queries, $COUNTRY$ is a placeholder for the country name for which we want to fetch data. WW, SS, EE, NN are placeholders for the coordinates of the bounding box for which to return the populated places and roads. The *regv* parameters in the queries specify what type of roads and populated places we are interested in. For example, if we do not want "unclassified" roads to be included in the response (these are generally small roads), then this option should be omitted in the *regv* parameter. The service `http://lxbarth.com/bbox` can be used for visually obtaining the bounding box coordinates WW, SS, EE, NN.

Once the query is executed, it returns a set of entries for the objects that are inside the specified region. When the query is about populated places, then each entry

contains the coordinates of a populated place together with additional information like its name. For roads, each entry in the response specifies a section of a road and contains a list of coordinates of points along the section and additional information. The returned segments need to be further glued together to obtain the complete road map. The coordinates are real numbers and there can be small rounding errors in them. When gluing road segments, we should not check for exact match of endpoints but for small enough distance between the endpoints. If the number of entries in the response is small, then finding the approximately equal endpoints can be done naively by iterating through all pairs of road segments. If the response if large, the space can be divided into buckets and then we should only check the distances between the endpoints in the same bucket or in neighboring buckets.

The result of gluing road segments together and positioning populated places is a graph representation of the road network inside the specified region. This representation is used for generating test instances for facility location problems. Depending on the problem, we may want to compute the shortest distances between all pairs of populated places, or to generate random paths, or something else. Creating this type of data from graphs is relatively easy and can be achieved using well-known algorithms. It should be noted that the road network can be used at different levels of precision. For example, we may work on a city level and ignore the roads inside the populated places. This results in a sparser graph. We may also concentrate entirely on the roads inside a single city. The result is a denser graph with more redundancy. Such test instances may be more challenging for some facility location problems.

The whole process of creating test instances is relatively simple and consists of several straightforward steps. Our implementation is a pipeline of Python scripts that process the data one after another until a test instance is generated. The details are omitted here. We only mention how the approximate distance between a pair of points is computed (Listing B.2) and give one example visual representation of the result of an Overpass XML query (Figure B.1).

Listing B.2: Computation of the (approximate) distance between two points specified by their coordinates

```
def Distance(p1, p2):
  R = 6371
  fi_1 = DegreesToRadians(p1.lat)
  fi_2 = DegreesToRadians(p2.lat)
  delta_fi = fi_1 - fi_2
  delta_lambda = DegreesToRadians(p1.lon - p2.lon)
  a = (math.sin(delta_fi / 2) * math.sin(delta_fi / 2) +
       math.cos(fi_1) * math.cos(fi_2) *
       math.sin(delta_lambda / 2) * math.sin(delta_lambda / 2))
  c = 2 * math.atan2(math.sqrt(a), math.sqrt(1 - a))
  return R * c
```

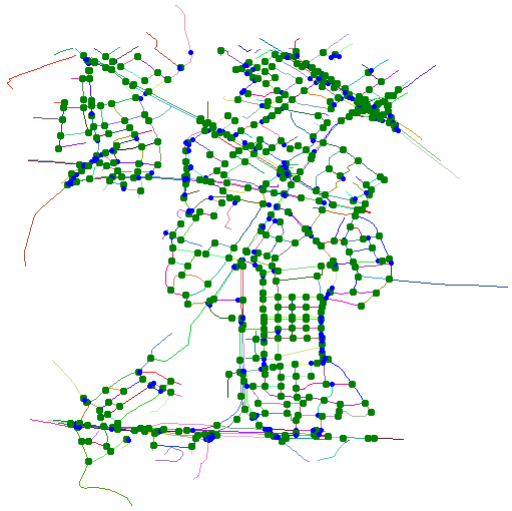

Figure B.1: Visualization of the result of an Overpass XML query for the area around Mladost 2 in Sofia, Bulgaria. The returned road segments are colored in different colors.

Bibliography

[1] Benaini A. et al. "Solving the Uncapacitated Single Allocation p-Hub Median Problem on GPU". In: *Bioinspired Heuristics for Optimization. Studies in Computational Intelligence.* 774 (2019).

[2] Akiyama et al. "Combinatorial optimization with Gaussian machines". In: *International 1989 Joint Conference on Neural Networks.* Vol. 1. 1989, pp. 533–540.

[3] B. Angéniol, G. De La Croix, and J-Y. Le Texier. "Self-organizing feature maps and the travelling salesman problem". In: *Neural Networks* 1.4 (1988), pp. 289–293. ISSN: 0893-6080.

[4] S. Arora and B. Barak. *Computational Complexity: A Modern Approach.* USA: Cambridge University Press, 2009.

[5] S. Atta, P. R. S Mahapatra, and A. Mukhopadhyay. "Solving maximal covering location problem using genetic algorithm with local refinement". In: *Soft Computing - A Fusion of Foundations, Methodologies and Applications* 22 (12 2018).

[6] J. E. Beasley. "OR-Library: Distributing Test Problems by Electronic Mail". In: *Journal of the Operational Research Society* 41 (11 1990).

[7] O. Berman, R. Larson, and N. Fouska. "Optimal Location of Discretionary Service Facilities". In: *Transportation Science* 26 (3 1992).

[8] Christian Blum and Andrea Roli. "Meta-heuristics in combinatorial optimisation: Overview and conceptual comparison". In: *ACM Computing Survey* 35(3) (Jan. 2003), pp. 268–308.

[9] M. Boccia, A. Sforza, and C. Sterle. “Flow Intercepting Facility Location: Problems, Models and Heuristics.” In: *J Math Model Algor* 8 (2009), 35–79.

[10] Margaret L. Brandeau and Samuel S. Chiu. “An Overview of Representative Problems in Location Research”. In: *Management Science* 35.6 (1989), pp. 645–674.

[11] P. Bremaud. *Markov Chains: Gibbs Fields, Monte Carlo Simulation, and Queues*. Springer-Verlag, New York, 1999.

[12] J. Brimberg. “The Fermat—Weber location problem revisited”. In: *Mathematical Programming* 71 (1995), 71–76.

[13] A. G. Brown. *Nerve Cells and Nervous Systems. An Introduction to Neuroscience*. Springer-Verlag London, 2001.

[14] J. Bruck and J. Goodman. “On the power of neural networks for solving hard problems”. In: *Journal of Complexity* 6 (2 1990), pp. 129–135.

[15] J. Campbell. “Integer programming formulations of discrete hub location problems”. In: *European Journal of Operational Research* 72 (2 1994).

[16] Paola Cappanera. *A Survey on Obnoxious Facility Location Problems*. 1999.

[17] *Cbc (Coin-or branch and cut) mixed integer linear programming solver*. `https://github.com/coin-or/Cbc`. Accessed: 23.10.2020.

[18] D. Chhajed and T.J. Lowe. “Solving Structured Multifacility Location Problems Efficiently”. In: *Transportation Science* 28 (1994), pp. 104–115.

[19] G. Cho and C. Meyer. “Comparison of perturbation bounds for the stationary distribution of a Markov chain”. In: *Linear Algebra and its Applications* 335.1 (2001), pp. 137–150.

[20] G. Cho and C. Meyer. “Markov chain sensitivity measured by mean first passage times”. In: *Linear Algebra and its Applications* 316 (Sept. 2000), pp. 21–28.

[21] C. Chung. "Recent Applications of the Maximal Covering Location Planning (M.C.L.P.) Model". In: *The Journal of the Operational Research Society* 37 (8 1986), pp. 735–746.

[22] R. Church and C. ReVelle. "The maximal covering location problem". In: *Papers of the Regional Science Association* 32 (1974), 101–118.

[23] J. Clausen. "Branch and Bound Algorithms-Principles and Examples". In: 2003.

[24] M. Conforti, G. Cornuéjols, and G. Zambelli. *Integer Programming*. Springer, Cham, 2014.

[25] OpenStreetMap contributors. *Planet dump [Data file from 27/12/2019]. Retrieved from* `https://planet.openstreetmap.org`. 2015.

[26] T. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd Edition. The MIT Press, 2009. ISBN: 0262033844.

[27] Mark Daskin. "Network and Discrete Location: Models, Algorithms and Applications". In: *Journal of the Operational Research Society* 48 (Jan. 1996). DOI: 10.1057/palgrave.jors.2600828.

[28] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. "Ant System: Optimization by a colony of cooperating agents." In: *IEEE transactions on systems, man, and cybernetics. Part B, Cybernetics : a publication of the IEEE Systems, Man, and Cybernetics Society* 26 (Feb. 1996), pp. 29–41. DOI: 10.1109/3477.484436.

[29] R. Duan and S. Pettie. "Linear-Time Approximation for Maximum Weight Matching". In: *Journal of the ACM* 61 (2014).

[30] R. Durbin and D. Willshaw. "An analogue approach to the travelling salesman problem using an elastic net method". In: *Nature* 326 (1987), 689–691.

[31] A. Ernst and M. Krishnamoorthy. "Efficient algorithms for the uncapacitated single allocation p-hub median problem". In: *Location Science* 4 (3 1996), pp. 139–154.

[32] F. Favata and R. Walker. “A study of the application of Kohonen-type neural networks to the Travelling Salesman Problem”. In: *Biol. Cybern.* (1991).

[33] U. Feige. “A threshold of ln n for approximating set cover”. In: *Journal of the ACM* 45 (4 1998).

[34] W. Feller. *An Introduction to Probability Theory and Its Applications*. New York: Wiley, 1950.

[35] R. D. Galvão and C. ReVelle. “A Lagrangean heuristic for the maximal covering location problem”. In: *European Journal of Operational Research* 88 (1 1996), pp. 114–123.

[36] M. Garey and D. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[37] F. Glover. “Future paths for integer programming and links to artificial intelligence”. In: *Comput. Oper. Res.* (1986), 533–549.

[38] Fred W. Glover and Manuel Laguna. *Tabu Search*. Springer Science+Business Media New York, 1997.

[39] S.L. Hakimi. “Optimum distribution of switching centers in a communication network and some related graph theoretic problems”. In: *Operations Research* 13 (1965), 462–475.

[40] S.L. Hakimi. “Optimum locations of switching centers and the absolute centers and medians of a graph”. In: *Operations Research* 12 (1964), pp. 450–459.

[41] P. Hansen and D. Moon. *Dispersing Facilities on a Network*. Rutgers University. Rutgers Center for Operations Research, 1988.

[42] V. Haralampiev. “Neural network approaches for a facility location problem”. In: *International Scientific Journal Mathematical Modeling* 4 (2020).

[43] V. Haralampiev. “Single facility location problems in k-trees”. In: *58th Annual Science Conference of Ruse University and Union of Scientists - Ruse*. 2019.

[44] V. Haralampiev. “Theoretical Justification of a Neural Network Approach to Combinatorial Optimization”. In: *Proceedings of the 21st International Conference on Computer Systems and Technologies*. 2020, pp. 74–77.

[45] D.J. Hartfiel and C. Meyer. “On the structure of stochastic matrices with a subdominant eigenvalue near 1”. In: *Linear Algebra and its Applications* 272.1 (1998), pp. 193–203.

[46] G. Hinton and T. Sejnowski. “Boltzmann Machines: Constraint Satisfaction Networks that Learn”. In: *Carnegie Mellon University Technical Report* (1984).

[47] D. Hochbaum. “Approximating covering and packing problems: set cover, vertex cover, independent set, and related problems”. In: 1996.

[48] J.J. Hopfield and D.W. Tank. ““Neural” computation of decisions in optimization problems”. In: *Biol. Cybern.* 52 (1985), 141–152. DOI: 10.1007/BF00339943.

[49] A. Ilic et al. “A general variable neighbourhood search for solving the uncapacitated single allocation p-hub median problem”. In: *European Journal of Operational Research* 206 (2010), 289–300.

[50] D. Isaacson and R. Madsen. *Markov Chains: Theory and Applications*. Wiley, 1976.

[51] Horgan J. “Can science explain consciousness?” In: *Sci Am.* 271(1) (1994), pp. 268–308.

[52] M. Jabalameli, B. Tabrizi, and M. Javadi. “A Simulated Annealing method to solve a generalized maximal covering location problem”. In: *International Journal of Industrial Engineering Computations* 2 (2 2011), pp. 439–448.

[53] D. Johnson. “More Approaches to the Traveling Salesmann Guide”. In: *Nature* 330 (1987).

[54] B. Kamgar-Parsi. “Dynamic Stability and Parameter Selection in Neural Optimization”. In: *Proceedings International Joint Conference on Neural Networks*. Vol. 4. 1992, pp. 566–571.

[55] M. Kao. *Encyclopedia of Algorithms*. 2nd Edition. Springer Publishing Company, 2016.

[56] O. Kariv and S. L. Hakimi. "An Algorithmic Approach to Network Location Problems. II: The p-Medians". In: *SIAM Journal on Applied Mathematics* 37 (1979), pp. 539–560.

[57] R. Karp. "Reducibility among combinatorial problems". In: *Complexity of Computer Computations*. Ed. by R. Miller and J. Thatcher. Plenum Press, 1972, pp. 85–103.

[58] P. Kaye, R. Laflamme, and M. Mosca. *An introduction to quantum computing*. Oxford U. Press, New York, 2007.

[59] R. Kincaid, C. Easterling, and M. Jeske. "Computational experiments with heuristics for two nature reserve site selection problems". In: *Comput. Oper. Res.* 35 (2 2008), pp. 499–512.

[60] R.K. Kincaid. "Good solutions to discrete noxious location problems via metaheuristics". In: *Ann Oper Res* 40 (1992), 265–281.

[61] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. "Optimization by Simulated Annealing". In: *Science* 220.4598 (1983), pp. 671–680.

[62] C. Klein and R. Kincaid. "The Discrete Anti-P-Center Problem". In: *Transportation Science* 28.1 (1994), pp. 77–79.

[63] J.G. Klincewicz. "Avoiding local optima in the p-hub location problem using tabu search and GRASP". In: *Annals of Operations Research* 40 (1992), 283–302.

[64] J.G. Klincewicz. "Heuristics for the p-hub location problem". In: *European Journal of Operational Research* 53 (1 1991).

[65] T. Kohonen. *Self-organization and associative memory*. Springer-Verlag Berlin Heidelberg, 1989.

[66] T. Kohonen. "Self-organized formation of topologically correct feature maps". In: *Biol. Cybern.* 43 (1982), 59–69.

[67] J. Korst and E. Aarts. “Combinatorial optimization on a Boltzmann machine”. In: *Journal of Parallel and Distributed Computing* 6.2 (1989), pp. 331 –357.

[68] Bernhard Korte and Jens Vygen. *Combinatorial Optimization. Theory and Algorithms*. Springer-Verlag Berlin Heidelberg, 2006.

[69] M. Kuby. “Programming models for facility dispersion: the p-dispersion and maxisum dispersion problems”. In: *Mathematical and Computer Modelling* 10 (10 1988), p. 792.

[70] W. Lai and G. Coghill. “Genetic Breeding of Control Parameters for the Hopfield/Tank Neural Net”. In: *Proceedings International Joint Conference on Neural Networks*. Vol. 4. 1992, pp. 618–623.

[71] D. Lakhmiri, S. Digabel, and C. Tribes. “HyperNOMAD: Hyperparameter optimization of deep neural networks using mesh adaptive direct search”. In: *Technical Report G-2019-46, HEC Montreal* (2019).

[72] R. F. Love and J. G. Morris. “Facility Location: Models and Methods”. In: *Publications in Operations Research* 7 (1996).

[73] M. Marić, Z. Stanimirović, and P. Stanojević. “An efficient memetic algorithm for the uncapacitated single allocation hub location problem”. In: *Soft Computing* 17 (3 2012).

[74] P. McCorduck. *Machines Who Think: A Personal Inquiry into the History and Prospects of Artificial Intelligence*. W. H. Freeman, 1979.

[75] B. Müller, J. Reinhardt, and M. Strickland. *Neural Networks: An Introduction*. Springer-Verlag Berlin Heidelberg, 1995.

[76] J. Munkres. “Algorithms for the Assignment and Transportation Problems”. In: *Journal of the Society for Industrial and Applied Mathematics* 5 (1 1957).

[77] M. O’Kelly. “A Quadratic Integer Program for the Location of Interacting Hub Facilities”. In: *European Journal of Operational Research* 32 (3 1987).

[78] J. Ostrowski et al. “Solving large Steiner Triple Covering Problems”. In: *Operations Research Letters* 39 (2 2011).

[79] *Overpass API/Overpass QL*. `https://wiki.openstreetmap.org/wiki/Overpass_API/Overpass_QL`. Accessed: 29.10.2020.

[80] Hansen P. and Mladenović N. “An Introduction to Variable Neighborhood Search.” In: *Voß S., Martello S., Osman I.H., Roucairol C. (eds) Meta-Heuristics.* (1999).

[81] C. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. USA: Prentice-Hall, Inc., 1982.

[82] C. Peterson. “Parallel Distributed Approaches to Combinatorial Optimization: Benchmark Studies on Traveling Salesman Problem”. In: *Neural Computation* 2.3 (1990), pp. 261–269.

[83] Leonidas Pitsoulis and Mauricio Resende. “Greedy Randomized Adaptive Search Procedures”. In: (Feb. 2001).

[84] J. Reese. “Solution methods for the p-median problem: An annotated bibliography”. In: *Wiley Periodicals, Inc. NETWORKS* 48 (3 2006), 125–142.

[85] R. Rojas. *Neural Networks - A Systematic Introduction*. Springer-Verlag Berlin New-York, 1996.

[86] K. Rosing, C. ReVelle, and J. Williams. “Maximizing Species Representation under Limited Resources: A New and Efficient Heuristic”. In: *Environmental Modeling & Assessment* 7 (2002), 91–98.

[87] D. Schilling, V. Jayaraman, and R. Barkhi. “A review of covering problems in facility location”. In: *Computers & Operations Research* (1993).

[88] Alexander Schrijver. *Combinatorial Optimization. Polyhedra and Efficiency*. Springer-Verlag Berlin Heidelberg, 2003.

[89] P. Schweitzer. “Perturbation Theory and Finite Markov Chains”. In: *Journal of Applied Probability* 5.2 (1968), pp. 401–413.

[90] E. Seneta. *Non-negative Matrices and Markov Chains*. Springer-Verlag, New York, 1981.

[91] E. Senne, M. Pereira, and L. Lorena. "A Decomposition Heuristic for the Maximal Covering Location Problem". In: *Advances in Operations Research* (2010).

[92] M. R. Silva and C. B. Cunha. "New simple and efficient heuristics for the uncapacitated single allocation hub location problem". In: *Computers & Operations Research* 36 (12 2009), pp. 3152–3165.

[93] S. Skiena. *The Algorithm Design Manual*. Springer, 2008. ISBN: 9781848000704 1848000707 9781848000698 1848000693.

[94] K. Smith, M. Krishnamoorthy, and M. Palaniswami. "Neural versus traditional approaches to the location of interacting hub facilities". In: *Location Science* 4 (3 1996), pp. 155–171.

[95] S. Snyder and R. Haight. "Application of the Maximal Covering Location Problem to Habitat Reserve Site Selection: A Review". In: *International Regional Science Review* 39 (1 2014).

[96] M. Sousa et al. "Architecture Analysis of an FPGA-Based Hopfield Neural Network". In: *Advances in Artificial Neural Systems* (2014).

[97] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 2015.

[98] A. Tamir. "An $O(pn^2)$ algorithm for p-median and related problems on tree graphs". In: *Operations Research Letters* 19 (1996), pp. 59–64.

[99] M. B. Teitx and P. Bart. "Heuristic Methods for Estimating the Generalized Vertex Median of a Weighted Graph". In: *Operations Research* 16 (5 1968), 955–961.

[100] V. Vazirani. *Approximation Algorithms*. Springer-Verlag Berlin Heidelberg, 2003.

[101] M. Vose. *The Simple Genetic Algorithm: Foundations and Theory. Complex Adaptive Systems*. MIT Press, 1999.

[102] Christos Voudouris, Edward Tsang, and Abdullah Alsheddy. "Guided Local Search". In: Sept. 2010, pp. 321–361. DOI: 10.1007/978-1-4419-1665-5_11.

[103] M. Wakamura and Y. Maeda. "FPGA implementation of bidirectional associative memory via simultaneous perturbation rule". In: *Proceedings of the 41st SICE Annual Conference. SICE 2002.* Vol. 3. 2002, pp. 1631–1632.

[104] Alfred Weber. *Ueber den Standort der Industrien*. 1909.

[105] J. White and K. Case. "On Covering Problems and the Central Facilities Location Problem". In: *Geographical Analysis* 6 (3 1974), pp. 281–294.

[106] G.V. Wilson and G.S. Pawley. "On the stability of the Travelling Salesman Problem algorithm of Hopfield and Tank". In: *Biol. Cybern.* 58 (1988), 63–70. DOI: 10.1007/BF00363956.

[107] R. Zanjirani Farahani and M. Hekmatfar. *Facility Location: Concepts, Models, Algorithms and Case Studies*. Physica-Verlag HD, 2009.