Hybrid Metaheuristics for the Travelling Salesman Problem

Muijsers, Martijn

Award date:
2021

Department of Mathematics and Computer Science
Algorithms, Geometry & Applications Cluster

Hybrid Metaheuristics for the Travelling Salesman Problem

Master Thesis

Martijn Muijsers

Supervisor:

dr. K.A.B. (Kevin) Verbeek

Committee:

dr. K.A.B. (Kevin) Verbeek
dr. V. (Vlado) Menkovski
dr. A.I. (Arthur) van Goethem

Final version

Eindhoven, July 2021

Abstract

With the recent success of machine learning in computer vision and natural language processing, there has been a growing trend towards using deep learning for combinatorial optimization. The Travelling salesman problem is a widely studied NP-hard combinatorial optimization problem with many practical applications. Heuristic algorithms are useful for problem sizes where finding an exact solution becomes infeasible. Hybrid heuristic algorithms where choices are informed by machine learning have not been studied much.

We discuss the creation of hybrid metaheuristics for the travelling salesman problem. We propose a local search metaheuristic that iteratively improves on an existing tour by improving the tours on subgraphs. The subgraph tour optimization is performed by existing machine learning models for the travelling salesman problem. This method can be applied to significantly larger graph sizes than the model that is used.

We perform a hyperparameter analysis and experimentally evaluate our method on TSPLIB instances with up to 10,000 nodes, which is much larger than most previous work. Our results show significant potential for our method, and in some cases competitiveness to the powerful solver OR-Tools, demonstrating that our proposed metaheuristic is an effective approach to the larger-scale travelling salesman problem.

Keywords: hybrid metaheuristics, travelling salesman problem, machine learning, combinatorial optimization, local search

Preface

This thesis is submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science and Engineering. The research was conducted within the Department of Mathematics and Computer Science at Eindhoven University of Technology (TU/e). It was supervised by dr. K.A.B. (Kevin) Verbeek, who is an Associate Professor in the Algorithms, Geometry & Applications Cluster.

From my foremost interest in discrete structures and combinatorial optimization in my bachelor's degree, to my first contact with machine learning during my master's degree, I had not expected to be combining the two in a thesis. I am very grateful for the opportunity to work on a research topic that has so much potential, and where the leading research is characterized by creativity and innovation. The burning questions and problems that arose, from the large to the small, have frequently kept me staring at the night ceiling, for which I am glad. The knowledge and skills I gained while facing these challenges are not only useful, but also fascinating and exciting.

I am very grateful to my supervisor Kevin Verbeek for this opportunity, and for his enthusiasm, continuous support and indispensable guidance. His insights provided direction when it was needed, and freedom when it was possible.

I would also like to extend my gratitude to all the teachers and mentors during my study who encouraged me to try something new, and to all the contributors to open-source software and open-access papers, without whom this field wouldn't be where it is today.

And last but not least, thanks goes out to all those who offered their support, kind words and inspiration, and to my friends and family, who have been there every step of the way.

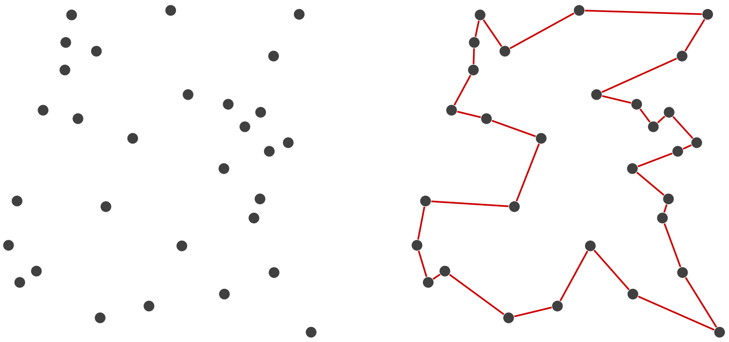
Martijn Muijsers,
July 2021

Contents

Chapter 1

Introduction

The *travelling salesman problem* (*TSP*) is the problem of, given a list of cities, finding the shortest tour that visits all of them. This has many applications, such as route planning and VSLI circuit design. There are many variations of the problem that appear in logistics, such as vehicle routing [1] with time constraints [2] and last-mile delivery with drones [3][4]. A commonly studied variant of TSP is *2D-TSP*, where every point has 2D coordinates, and the distance between two points is given by the Euclidean distance. An example of the 2D travelling salesman problem is shown below in Figure 1.1.



(a) The input

(b) The shortest possible tour

Figure 1.1: A 2D example instance of the travelling salesman problem.

TSP is a well-known NP-hard combinatorial optimization problem [5], in both its general formulation and its 2D Euclidean variant. As such, it is unlikely that a polynomial time algorithm exactly solving TSP exists. However, there are algorithms that can efficiently compute approximations of the optimal solution. For example, the Christofides approximation algorithm [6] always finds a tour with length within a factor 1.5 of the optimal solution. Other algorithms make use of heuristics to find good quality solutions, while having a low time complexity or practical running time. These algorithms are useful for problem sizes where finding an exact solution becomes infeasible. In particular, a common technique is local search, where improvements to a tour are iteratively applied until a condition for termination, such as a time limit, is met.

TSP is widely studied as a fundamental combinatorial optimization problem. Very strong heuristics are already known, such as the Lin-Kernighan heuristic (LKH) [7]. In addition, exact algorithms with fast practical running time also exist, particularly the software Concorde [8]. The volume of existing research, coupled with these strong existing solutions and the availability of widely used standard benchmarks, makes TSP an excellent problem to test novel heuristic approaches on.

With the recent success of machine learning (ML), especially deep learning, in computer vision and natural language processing, there has been a trend in research towards using deep learning for a greater variety of domains, including combinatorial optimization. Most proposed solutions for TSP that use machine learning use a constructive approach, where a solution tour is constructed using the output of an ML model. Hybrid heuristic algorithms where choices, such as the iterative improvements in local search, are informed by ML have not been studied much.

Further research in this direction may provide new insights into using machine learning on combinatorial optimization problems for which heuristic methods can be formulated. This can lead to novel solutions and shorten the development time of algorithms for novel problems.

Goals of this thesis

In this thesis, we will study hybrid metaheuristics for TSP that use machine learning. The goal is not to achieve state-of-the-art performance, but to consider how such hybrid approaches can work. We examine various characteristics of such hybrid metaheuristics for combinatorial optimization, as well as formulate methods and considerations for their development. We illustrate these ideas in the context of TSP, and perform some exploratory experimental investigation.

Stemming from some of the explored concepts, we develop a metaheuristic approach, *local search using TSP with mandatory edges.* We motivate this approach, and experimentally evaluate its effectiveness.

Outline

First, in Chapter 2, we will define and elaborate on the Travelling Salesman Problem and the algorithms relevant for this thesis. We will also introduce the relevant machine learning concepts and neural network architectures. In Chapter 3, we list related work on machine learning for combinatorial optimization and TSP.

Next, Chapter 4 discusses the characteristics of hybrid metaheuristics for combinatorial optimization guided by machine learning, and provides an overview of the considerations we made in developing such a metaheuristic. Additionally, we roughly describe some of the preliminary experiments carried out to explore some of these concepts.

In Chapter 5, we define the problem of TSP with mandatory edges, where some edges must be included in the tour. We describe its usage and significance in hybrid local search. We show how to apply two types of existing solutions for TSP to solve TSP with mandatory edges: local search algorithms and algorithms that allow negative edges. Based on this concept, we introduce two hybrid metaheuristics: one using beam search, and one using the ML architecture for performing 2-opt moves introduced by Da Costa et al. [9]

In Chapter 6, we experimentally evaluate the effectiveness of our method, specifically the hybrid metaheuristic based on the 2-opt ML architecture. We perform hyperparameter analysis, detailing the results for each analyzed hyperparameter. After this, we evaluate on an instance of size $n = 10^5$, and a subset of instances from TSPLIB [10] of graphs with size $n \leq 10^4$, and examine the results. These results show promising potential of the proposed method, and are in some cases competitive to the powerful existing solver OR-Tools [11].

Finally, in Chapter 7, we summarize the core notions and results presented in this thesis, and present our concluding thoughts and suggestions for future work.

Chapter 2

Background

In this chapter, we define the travelling salesman problem, and then describe various relevant solution approaches: exact algorithms, beam search, and finally local search. Next, we describe the relevant ideas in machine learning that will be used in this thesis, beginning with general concepts and continuing with specific neural network architectures.

2.1 Travelling salesman problem

2.1.1 Definition

We will define the travelling salesman problem, clarify the scope of this research and specify some notation used in this thesis.

Graph In the *travelling salesman problem* (*TSP*), we are given a graph $G = (V, E)$ with nodes $V = \{1, \ldots, n\}$ and $n \times n$ distance matrix c with $c_{ij}, i, j \in V$ the distance from node i to j (and $c_{ij} = \infty$ if $(i, j) \notin E$). To prevent the need for self-loops and simplify the given definitions, we require $n \geq 3$.

Following some previous work, for simplicity we limit our scope to that of the *2D Euclidean travelling salesman problem* (*2D-TSP*): where every node i has a point $u_i = (x_i, y_i)$ on the 2D plane, and the distances between them are

$$c_{ij} = \|u_j - u_i\|_2 = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2}.$$

Parts of the discussion in this thesis nevertheless may not require these assumptions and the reader may find them more generally applicable.

2D-TSP is a case of the *symmetric travelling salesman problem*, where $c_{ij} = c_{ji}$, where we can regard G as an undirected graph. Any instance of asymmetric TSP with n nodes can be solved by solving an instance of symmetric TSP with at most $2n$ nodes [12].

Tour A *tour* is a permutation π on the indices $(1, ..., n)$ so that $\pi(i)$ is the i'th node in the tour.

For notation, we define the set E_π of directed edges in the tour π as

$$E_\pi : \{(\pi(i), \pi(i+1)) \mid i \in \{1, ..., n\}\}$$

Additionally, we define a function encoding x_{ij} to indicate whether the edge (i, j) is in the tour:

$$x_{ij} = \begin{cases} 1, & \text{if } (i, j) \in E_\pi \\ 0, & \text{otherwise} \end{cases}$$

A *partial tour* τ is a sequence that contains at least 0 and at most n distinct integers in the range $[1, n]$. We can append indices to a partial tour to construct a tour.

Length of tour We define the *cost* $l(\pi)$, also simply called *length*, of the tour π as the sum of distances of all edges in the tour, i.e.

$$l(\pi) = \sum_{(i,j)\in E_\pi} c_{ij}.$$

In the travelling salesman problem, we wish to find a tour with minimal cost. An optimal solution tour is referred to as OPT, with the minimum cost for a given problem being $l(\mathrm{OPT})$.

2.1.2 Exact solutions

TSP is NP-hard [5], and no polynomial-time exact solutions are known. However, there are exact algorithms that are fast in practice and have solved nontrivial instances with $n > 10^4$.

Finding the optimal solution to TSP in the most naive way can be done by simply evaluating the cost of each Hamiltonian cycle on the graph, and taking the one with the minimum cost. The running time of this algorithm is $O(n!)$. This strategy becomes infeasible quickly for higher n.

Using *dynamic programming*, the travelling salesman problem can be solved with a time complexity of $O(2^n n^2)$ [13]. While in practice much faster than the naive solution, this approach runs into a similar degree of infeasibility at slightly higher n.

While techniques arguably better than dynamic programming exist for TSP, these techniques may not be as applicable to or optimized for variations on TSP, which means dynamic programming remains a useful algorithm to consider when solving these variations. An example of a TSP variation that dynamic programming has been adapted for is TSP with drone [14].

Concorde

Considered one of the best practical solutions to TSP is the software *Concorde* [8, 15]. Concorde has lead to exactly solving a TSP problem with integral distances from TSPLIB called `pla85900`, with $n = 85900$. Being feasible for such large instances makes Concorde a good tool for determining optimal solutions to instances of TSP when evaluating a different algorithm. Figure 2.1 shows an instance of 2D-TSP that was quickly solved by Concorde, but would take infeasibly long to solve using dynamic programming.

Concorde only works if the distances are integral, because it uses the *cutting-plane method* [16] which relies on *Integer Linear Programming*. Note that although we can approximate the solution to a general TSP problem by first scaling the problem arbitrarily, rounding to integers and then solving the resulting problem, Concorde limits the maximum distance to prevent integer overflow.

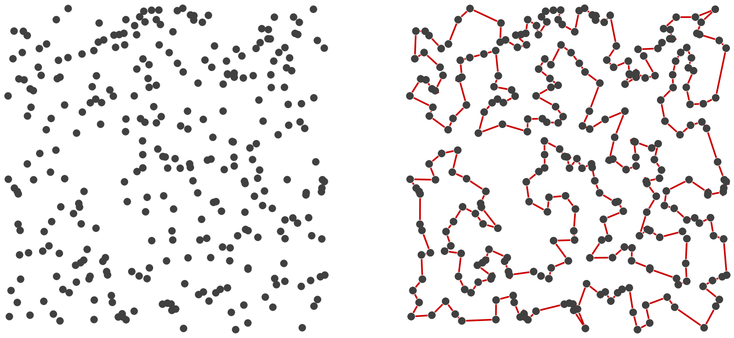


Figure 2.1: An instance of 2D-TSP with $n = 300$ solved to optimality by Concorde in only 24.5 seconds.

Integer Linear Programming

Integer Linear Programming (*ILP*) is a technique that can be used to solve many combinatorial optimization problems by defining the problems as an objective function to minimize and set of linear constraints on integral variables that must be satisfied. Integer Linear Programming is practical because ILP solvers such as QSopt [17] that are very fast in practice exist, and constructing an ILP formulation for a problem and then using a fast ILP solver can sometimes yield fast solutions more easily than formulating a separate algorithm for that problem.

For TSP, if additional constraints are present in a variation, we can adapt an existing TSP ILP formulation. An easily expandable ILP formulation is the Dantzig–Fulkerson–Johnson formulation (for complete TSP with positive edge lengths and no self-loops) [18]:

$$
\begin{aligned}
&\min \sum_{i=1}^{n} \sum_{j \neq i, j=1}^{n} c_{ij} x_{ij} : && \\
&x_{ij} \in \{0,1\}; && \\
&\sum_{i=1, i \neq j}^{n} x_{ij} = 1 && j = 1, \ldots n; \\
&\sum_{j=1, j \neq i}^{n} x_{ij} = 1 && i = 1, \ldots n; \\
&\sum_{i \in Q} \sum_{j \neq i, j \in Q} x_{ij} \leq |Q| - 1 && \forall Q \subsetneq \{1, \ldots, n\}, |Q| \geq 2
\end{aligned}
$$

The cutting-plane method for TSP is a method of iteratively refining a partial solution (such as probabilities over edges), by initializing an empty partial solution and ILP formulation with no (or incomplete) constraints, and repeating the procedure:

- find patterns in the partial solution that violate TSP requirements (such as nodes with sum of probabilities of incident edges greater than 2, or pairs of unconnected nodes) and add these violations to the LP formulation as individual constraints, called *cuts*,

- run the formulation through an LP solver and update the partial tour based on the given LP solution,

until no more violations are present.

2.1.3 Beam search

Definition

Beam search is a technique that can be used to find a tour based on a given probability distribution over all directed edges, denoting the likelihood that the edge's second endpoint should follow its first endpoint in the final tour. This technique is shown in Figure 2.2. This is useful, because it may be applied to translate the output of another preceding step, for instance a machine learning model, to a concrete tour. The resulting solution is not necessarily optimal.

It is an extension of *greedy search*, which constructs a solution tour τ by iteratively adding nodes to the tour:

1. Start with an empty partial tour $\tau = ()$

2. Choose a node $j \in V \setminus \tau$ and append it to τ

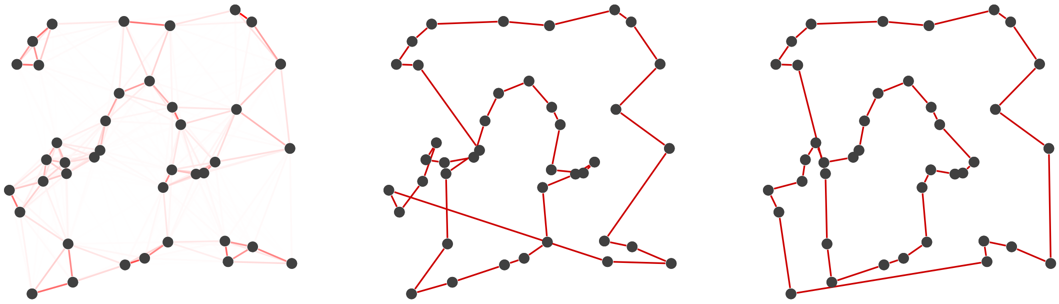
3. If $|\tau| < n$, go to Step 2, otherwise return $\pi := \tau$

We can choose the node j to add based on the previously added node $i = \tau(|\tau|)$. To do so, first we choose a node $j_1 \in V$ in the first iteration. Then we define a function $L \in V \times V \to \mathbb{R}$ so that in each iteration we choose the next node as the one with the lowest value: $j = \text{argmin}_{j' \in V \setminus \pi} L(i, j')$.

Rather than constructing a single partial tour, as done by the greedy approach, beam search iteratively adds nodes to at most w (the *beam width*) partial tours:

1. For some initial beam width $1 \leq w_1 \leq w$, choose an initial node j_b for each $b = 1, \ldots, w_1$, and create a partial tour set $B = \{\tau_b \mid b = 1, \ldots, w_1\}, \tau_b = (j_{1,b})$
2. Create a partial tour set $B' = \emptyset$
3. For each partial tour $\tau_b \in B$:
   (a) Choose a set of potential next nodes $J_b \subseteq V \setminus \tau_b$
   (there must be at least one non-empty J_b)
   (b) For each potential next node $j_b \in J_b$, add the resulting partial tour $\tau_b +\!\!+ (j_b)$ to B'
4. Set B to a subset of B', choosing at most w tours from B'
5. If the partial tours in B (which all have the same length) do no have length n, go to Step 2
6. Otherwise, return the tour with minimal length $\pi = \text{argmin}_{\tau_b \in B} l(\tau_b)$

Which partial tours to keep in each step is determined by minimizing the sum of the values L of chosen edges along each partial tour. The most basic choice of $L(i, j) = c_{ij}$ minimizes the tour length at each step.



(a) The edge probabilities as a heatmap (normalized over all edges)

(b) $w = 5$, $l(\pi) = 6.69$

(c) $w = 10^4$, $l(\pi) = 6.19$

Figure 2.2: An instance of 2D-TSP with edge probabilities, and the tours constructed by beam search.

In beam search, we often wish to evaluate a probability that denotes the likelihood of a given partial tour being the most desired out of all the possible partial tours starting at the same initial node. We can choose L so that there is a transformation from L to probabilities $P(i, j) \in V \times J \to [0, 1]$, based on the set J of potential next nodes, using *softmax* on negative L:

$$P(i, j) = \frac{e^{-L(i,j)}}{\sum\limits_{j' \in J} e^{-L(i,j')}}$$

Which partial tours to keep in each step can then also be determined by maximizing the product of the probabilities P of chosen edges along each partial tour. Because this product becomes very small and is difficult to compute, we can instead maximize the sum of logits of these probabilities ($logit(p) = log(\frac{p}{1-p})$).

The inverse of the logit function is the logistic function, and softmax is a generalization of the logistic function to multiple dimensions. Therefore, to save on computation, the sum of L of chosen edges along a partial tour is often minimized, with the conceptual goal of maximizing the probability of that partial tour compared to other partial tours.

The time and space complexity of beam search for TSP is unlike other forms of beam search, and depends, amongst others, on:

- the way τ_b for all beams are stored, and the time complexity of checking for some $i \in V, \tau_b \in B$ whether $i \in \tau_b$
- the size and method of computing for J_b (e.g. all nodes $V \setminus \tau_b$, or only neighbors of the node last added to τ_b in the case of a non-complete graph)

2.1.4 Local search

In *local search*, changes are iteratively applied to a given initial tour. We will first define some notation and note some properties of local search, and then discuss two relevant instances of local search: k-opt and LKH.

In a step $t \geq 1$ some edges X_t are removed from, and edges Y_t are added to, the tour. The pair X_t, Y_t defines an *edge swap* or *move*, shown in Figure 2.3.

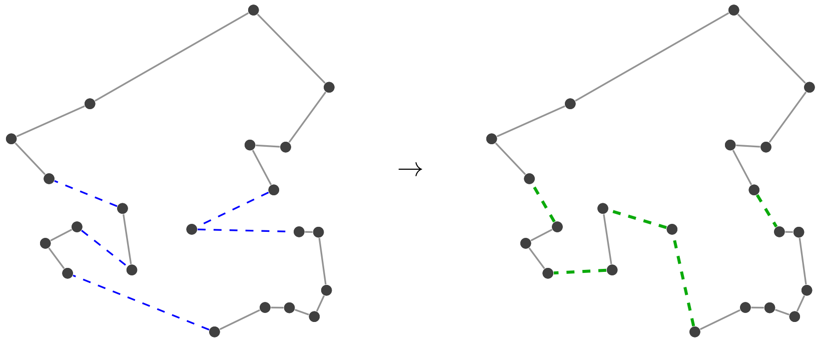


Figure 2.3: A local search move that removes edges X (blue, dashed) and adds edges Y (green, dashed).

With π_t the tour and $E_{\pi,t}$ the directed edges in the tour after t steps, we have

$$E_{\pi,t} = E_{\pi,t-1} \setminus X_t \cup Y_t.$$

Since $|E_{\pi,t-1}| = |E_{\pi,t}| = n$, it must hold that $|X_t| = |Y_t|$. For some cases of local search, the size of the edge swap ($|X_t|$) can be much smaller than n. Local search algorithms can have difficulties when large edge swaps are required to lower the tour length [19].

A potential next solution (the current tour with a potential move applied) is known as a *neighbor solution*. The set of those neighbor solutions is called the *solution neighborhood*. The potential moves define the *search space*: which neighbor solutions there are for the current tour. The size of the solution neighborhood may be independent of n, and evaluated in a running time independent of n.

Local search is a form of *metaheuristic*, a type of algorithm with the goal of efficiently exploring the search space to find a high-quality solution. While there are a wide variety of metaheuristics ranging from simple to complex, for the purposes of this thesis a metaheuristic is the algorithm that guides the solution search process to a good solution, by making calls to subprocedures called *heuristics*. The metaheuristic algorithm defines how the search process is continued based on the output from the heuristic. The heuristic has a specific problem it solves or question it answers, with the output being potentially suboptimal, or even without a known correct answer or objective measure for correctness. For instance, useful heuristics for TSP are

- ”What is the probability this edge is in the optimal tour?”,
- ”Which next node should we add to this partial tour?”, or
- ”Which edges can we remove from the graph without affecting the optimal tour length too much?”

Which move to apply may be based on a different criterion than minimizing the resulting tour length.

If $l(\pi_t) < l(\pi_{t-1}) - \epsilon$ (with $\epsilon > 0$) at all t, the algorithm terminates. Some local search algorithms do not fulfill this property, and some do not terminate. The algorithm may also be terminated after a step by another condition, for instance if $t \geq t_{max}$ for some $t_{max} \geq 0$, or after a certain amount of elapsed wall-clock time.

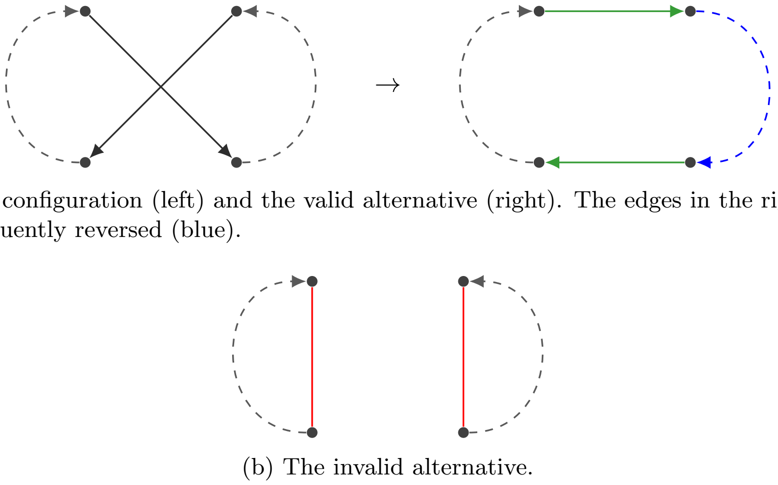
Not every move is valid: there exist X_t, Y_t so that π_t is not connected (often, this means the move breaks the tour into multiple cycles). Making sure a move is valid can cost $O(n)$ time, which may be the bottleneck in computing the next move.

k-opt

Local search with moves that change up to k edges of the solution is referred to as *k-opt*.

The most basic instance of k-opt iterates over every possible subset of k edges in the tour, and replaces these edges with the choice of k edges that minimizes the resulting tour length (potentially the original edges if there is no shorter alternative choice). Either a set number of iterations over all possible subsets is performed, or the procedure is repeated until no more improvement moves can be found.

For symmetric TSP, storing the direction of each edge in the tour, based on the arbitrary direction of the tour as a whole, helps in determining the valid k-opt moves for a given set of k edges. For example, in the case of 2-opt, there are 3 possible undirected configurations of the chosen edges: the current configuration, an alternative valid configuration and an alternative invalid configuration. We can immediately determine the alternative valid configuration if we know the direction in the tour of both existing edges, as shown in Figure 2.4.



(a) The original configuration (left) and the valid alternative (right). The edges in the right-hand outer chain are consequently reversed (blue).

(b) The invalid alternative.

Figure 2.4: A 2-opt move.

Since the number of moves performed is at most the number of moves considered, it is more efficient to update the directions of tour edges after a move than determine them each step. Updating the directions of the tour edges after a move takes at most $O(n)$ time. While alternatives exist, that for instance take amortized $O(\log(n))$ time, they are not necessarily faster in practice than $O(n)$ methods until $n > 10^4$ [20].

Lin-Kernighan heuristic

The *Lin–Kernighan heuristic* (*LKH*) [7], rather than fixing k, constructs sets $X = \{x_1, ...x_k\}$ and $Y = \{y_1, ..., y_k\}$ element by element.

Since we know that if x_1 is removed from the tour, both endpoints of $x_1 = (t_1, t_2)$ need some edge $y \in Y$ incident to them, we can keep adding edges to X and Y in the following fashion:

- Determine $x_1 = (t_1, t_2)$
- Choose some t_3 and take $y_1 = (t_2, t_3)$
- Choose some t_4 and take $x_2 = (t_3, t_4)$
- And so on...
- Take $y_k = (t_{2k}, t_1)$

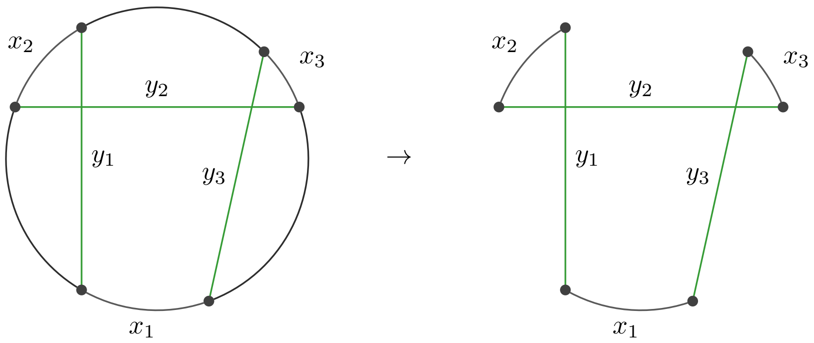


Figure 2.5: An illustration of a 3-opt move constructed by LKH.

This way, LKH constructs a *sequential k-opt move*, a k-opt move that can also be performed by making a sequence of 2-opt moves. The number of changes (k) in each LKH move may differ. An LKH move with $k = 3$ is illustrated in Figure 2.5.

Two important guiding but flexible principles to make sure we end up with a good and valid move are:

- *Feasibility criterion*

  After choosing some $x_i = (t_{2i-1}, t_{2i})$, choosing to finish constructing the move by taking $y_i = (t_{2i}, t_1)$ should yield a valid move

- *Gain criterion*

  The partial sum of the decreases in tour length (*gains*, g) up to each $k' \leq k$ should be positive:

$$\sum_{i=1}^{k'} g(y_i) - g(x_i) > 0$$

LKH offers several major advantages, amongst others:

- It has few local minima because, while each step must provide an improvement on the tour, individual additions to X, Y are not required to provide an improvement; only the partial sum of the decrease in tour length must be positive. This means improvements over the whole that require pairs of x_i, y_i that would increase the tour length can also be found.

- Because it combines many k-opt moves into one tour update, and therefore many edges are removed and added in one tour update, updating the tour edge directions happens less frequently.
- There is much flexibility for heuristic methods, such as determining which next t_i to pick, choosing to backtrack, including non-sequential k-opt moves (which temporarily breaks the feasibility criterion), and more. Many of such methods were proposed in follow-up research.

Some powerful extensions have been proposed and implemented. A common important practical improvement is constructing large X, Y quickly by rather than conceptually using consecutive 2-opt moves, using consecutive k-opt moves for small k such as 5.

The particularly powerful implementations simply named "LKH", "LKH-2" and the more generalized "LKH-3" by K. Helsgaun [21, 22, 23] have, as of January 2021, exactly solved the largest nontrivial instance that has been solved to optimality, an instance with $n = 109399$. Additionally, they have produced optimal solutions for all instances for which an optimal solution is known.

An implementation of the extension Chained Lin-Kernighan [24] is available as part of Concorde [15] under the name *Linkern*. Unless otherwise mentioned, we evaluate LKH by using this implementation, which we will simply refer to as LKH.

2.2 Machine learning

In recent years, *machine learning* (*ML*) has lead to significant breakthroughs in the fields of computer vision, natural language processing and more. Most state-of-the-art results were achieved using *deep learning*: a class of machine learning using multiple layers, that extract information progressively going from low-level features to higher-level features.

The use of the backpropagation algorithm [25] allows for a wide variety of differentiable *ML models*, in deep learning typically called *(neural) networks*. We will first introduce some general concepts and notation, and describe some common components of neural networks such as fully-connected layers. Next, we describe the most relevant classes of network, notably convolutional neural networks and recurrent neural networks.

Finally, we will discuss how these types of networks have been adapted to work with graphs. The resulting models are of interest for applying deep learning to combinatorial optimization problems that involve graphs, such as TSP.

2.2.1 General concepts

A deep learning model M_θ is parametrized by some vector θ. We define a differentiable *loss function* $\mathcal{L}(\hat{y})$ denoting the error of the model output $\hat{y} = M_\theta(x)$ for some input x. We wish to minimize the loss $\mathcal{L}$. This loss can be differentiated with respect to the model parameters θ to get the gradients $\frac{\partial \mathcal{L}}{\partial \theta}$.

We can then iteratively update the parameters θ using gradient descent:

$$\theta := \theta - \alpha \cdot \frac{\partial \mathcal{L}(M_\theta(x))}{\partial \theta}$$

with a chosen learning rate α.

To make use of parallel computation, deep learning operates on *tensors*. This means that, where possible, all operations should be *vectorized*. Tensors are 0-indexed in implementation, but in mathematical notation we will use 1-indexing, as is standard practice in linear algebra.

Some common vectorized operations are:

- Reduction, such as

  sum: $\sum\big((2, 3, 4, -1)\big) = 8$

max: $\max\big((2,3,4,-1)\big) = 4$

argmax: $\operatorname{argmax}\big((2,3,4,-1)\big) = 3$

- *element-wise multiplication*: $\big(2,3,4,-1\big) \circ \big(3,5,0,1\big) = \big(6,15,0,-1\big)$
- *matrix multiplication*: $\begin{pmatrix} 3 & 2 & 4 \end{pmatrix} \times \begin{pmatrix} 0 \\ -1 \\ 2 \end{pmatrix} = \begin{pmatrix} 0 & -3 & 6 \\ 0 & -2 & 4 \\ 0 & -4 & 8 \end{pmatrix}$
  Also written shorthand: $Wx = W \times x$
- *concatenation*: $\big(2,3,4\big) \parallel \big(3,5,0\big) = \big(2,3,4,3,5,0\big)$
- *index along other tensor*: $a = \big(2,3,4,-1\big), i = \big(4,1,2,2,1\big) \Rightarrow a[i] = \big(-1,2,3,3,2\big)$

The model input is typically a tensor x with the size of the first dimension being the number of individual samples. Such input is called a *(mini-)batch* and the number of samples (*(test) cases*) in a mini-batch the *batch size.* Using gradient descent with mini-batches, called *stochastic gradient descent* [26, 27], allows for parallel computation, more stable training and the possibility of normalizing values with respect to the batch. Note that mathematical formulas for model operations usually do not account for this first dimension in the tensor.

Fully-connected layers

The most basic building block of most neural networks is the *fully-connected layer*, also called *linear layer* or *dense layer.* A linear layer, for each element of the output, takes a linear combination of the entire input and applies a nonlinear *activation function* σ to the result. The coefficients of the linear combinations are learnable parameters. The nonlinearity of the activation function allows the network, by changing the coefficients, to compute nontrivial results [28] (since without nonlinearity, sequentially applied linear combinations are always equivalent to just one linear combination). Basic neural networks with only linear layers, called *multilayer perceptrons*, are already able to efficiently learn complex relations.

Formally, a linear layer, for some given input vector x of length m, returns an output vector y of length n, where

$$y = \sigma(Wx + b)$$

with W an $n \times m$ learnable *weight* matrix, b a learnable *bias* vector of length n, and σ a nonlinear activation function. For ease of notation, we can write $\text{lin}_\theta(x) = \sigma(W_\theta x + b_\theta)$, so that lin_θ is parametrized by W_θ, b_θ.

Figure 2.6 illustrates how W, b are applied by the layer.

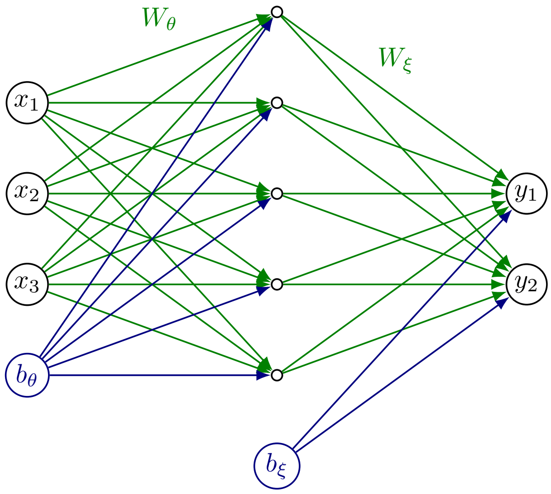


Figure 2.6: Two sequential linear layers $y = \text{lin}_\xi(\text{lin}_\theta(x))$. The activation functions are not visualized.

Common choices of activation function (shown in Figure 2.7) include

- the *logistic function*, also simply called *sigmoid function* in this context:

$$S(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1}$$

- the hyperbolic tangent function $tanh(x)$
- the *Rectified Linear Unit* [29, 30] (*ReLU*):

$$\text{ReLU}(x) = \begin{cases} x & \text{if } x > 0, \\ 0 & \text{otherwise.} \end{cases}$$

- *Leaky ReLU* [31], which allows a small gradient for negative inputs:

$$\text{LeakyReLU}(x) = \begin{cases} x & \text{if } x > 0, \\ 0.01x & \text{otherwise.} \end{cases}$$

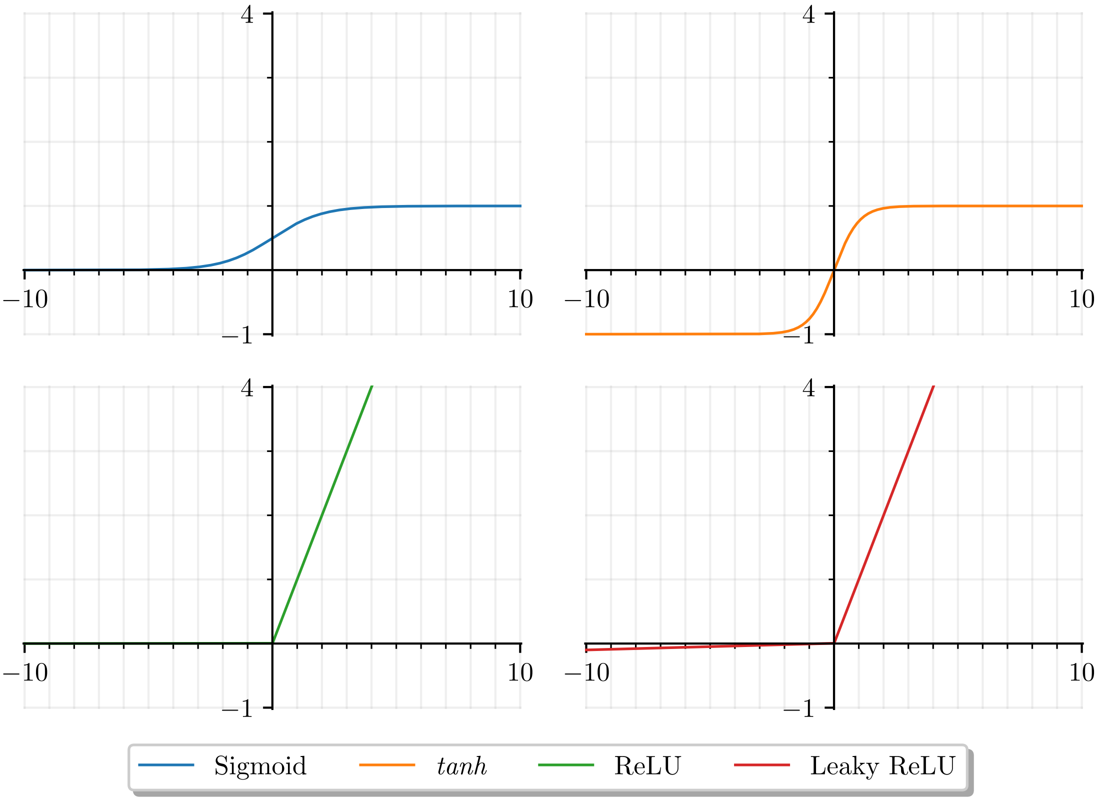


Figure 2.7: Nonlinear functions commonly used as activation functions: the constrained nature of sigmoid and *tanh* versus the linearity for positive inputs of ReLU and Leaky ReLU can be clearly observed.

The sigmoid activation function always returns a value in the range $(0, 1)$, making it useful for classification. Similarly *tanh* always returns a value in the range $(-1, 1)$, and is therefore ideal to return a multiplicative factor before addition/subtraction ($v := v + tanh(a) \cdot b$).

However, both sigmoid and *tanh* suffer from the *vanishing gradient problem*: where backpropagated gradients become so small, that either learning by gradient descent drastically slows down, or the floating-point precision is not enough to store their values. For that reason, the most commonly used activation functions in the intermediate layers of a network are ReLU and Leaky ReLU, which do not suffer from the same problem.

Loss functions

In *supervised learning*, we train the model on pairs of input and desired output x, y by iteratively feeding in the input and comparing the model output to the desired output. We use a loss function $\mathcal{L}(\hat{y}, y)$ that becomes lower if $\hat{y}$ is more similar to y. Common choices of loss function are:

- *Mean squared error* (*MSE*) takes the mean of the squares of the difference for each of the N features in the output:

$$\text{MSE}(\hat{y}, y) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

- *Logistic loss*, also called *Cross-entropy loss* (*CE*) is used if the model output features are probabilities and the desired output features are either 0 or 1:

$$\mathrm{CE}(\hat{y}, y) = -\frac{1}{N} \sum_{i=1}^{N} y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i)$$

Conversely, *unsupervised learning* attempts to learn to perform a task without pairs of inputs and target outputs.

Performance, bias and variance

Simplified for relevance to the context of supervised learning, the *bias* is the error between the computed solution and the target solution on samples in the training dataset, and the *variance* is the error between the computed solution and the target solution on samples in an evaluation dataset.

In other words, *higher bias* means the model *performance*, i.e. the solution quality, on the training dataset decreases: the model is not learning to infer the correct relations when being trained (it is *underfitting*). For problems for which good general model architectures are known, such as for computer vision or natural language processing, bias is often reduced by increasing the model size.

Higher variance means the model performance on the evaluation dataset decreases: the model is not learning to infer the correct relations that apply to the data outside the training set: it is *overfitting* to the training set. Usually, this means the network has learnt to infer relations from noise in the training set, rather than only the relevant features. Under the assumption that the training set has limited noise, variance can be reduced by decreasing the model size, or *regularizing* the network through a wide variety of techniques, such as L2 loss [32], dropout [33] or batch normalization [34]. These techniques often cause bias to increase in varying amounts, thereby leading to a *bias-variance tradeoff*.

While any network architecture can be applied to any data that fits its input-output format, a network architecture is regarded as being suitable for a specific class of data (problem) if it is able to infer the relevant relations, which can be observed as low bias and low variance. For instance, convolutional neural networks have a much better bias-variance tradeoff on image recognition tasks, than do purely multilayer perceptrons.

Combinatorial optimization is slightly different than some other machine learning tasks, in that there are unlimited test cases to use during training. The training set is naturally (but not necessarily) a similar sampling of the same overarching unbounded dataset as the evaluation set. However, there are many ways that variance can be introduced by the network overfitting: specific features may be very rare, and therefore there exist definable sets of instances for which the model performs poorly. When these sets have more occurrences in the intended target distribution than in the distribution used to create the training dataset, this constitutes a quantifiable cause of variance. Similar to machine learning on data sets for which an unlimited number of test cases is not available, identifying the causes of the model giving poor performance on specific test cases can help prevent poor performance on different unknown distributions of the data. For example, for TSP, it is easy to generate an unlimited number of uniformly distributed sets of nodes, but most practical use cases follow very different distributions, such as containing many tight clusters. In this case, making the network performance more stable under greater variation in edge lengths would improve the performance on those types of cases. Therefore, evaluation of machine learning for combinatorial optimization on an evaluation set from the same distribution as the training set will give the appearance of variance being equal to bias, assuming the model has converged on the training set. This is not a result that is particularly meaningful. Instead, if a model is trained on a generated training set, we can get a more meaningful indication of the performance of the model by evaluating it on a standard benchmark dataset containing instances derived from practical sources, such as maps of cities for TSP.

Reinforcement learning

Combinatorial optimization and machine learning are two worlds that are challenging to combine. Evaluating all of a set of discrete choices can lead to state explosion, while not doing so limits the scenarios for which gradients are produced. Many combinatorial optimization problems can be expressed as an environment in which actions are taken. For this type of problem, *reinforcement learning* [35, 36] (*RL*) is commonly used. With reinforcement learning, a model can be trained to take actions, and be trained on sequences of such actions rather than individual actions. This has lead to many state-of-the-art results in problems for which making a purely supervised learning model is difficult. Particularly, games such as Go, which can be represented as a very hard combinatorial optimization problem, have been learned to be played by an ML agent successfully [37, 38].

2.2.2 Convolutional neural networks

Convolutional neural networks [29] (*CNN*s) apply to d-dimensional tensors, typically 2-dimensional images, the following core operations, shown in Figure 2.9:

- *Convolution*

  The convolution operation (denoted by $*$), illustrated in Figure 2.8, takes the sum of an element-wise multiplication of a learnable parametrized *convolution kernel* to each equally sized subtensor of the tensor as whole. It acts as a sliding window over the original tensor, generating a new tensor with the resulting values at each position of the sliding window. After a convolution operation, an activation function is normally applied. Aside from the kernel, convolution typically has an additional bias parameter.
  Convolution is often interpreted as applying a type of generalized mask: detecting certain patterns in the input. Convolution is used to both recognize low-level features from input and higher-level features from lower-level features.

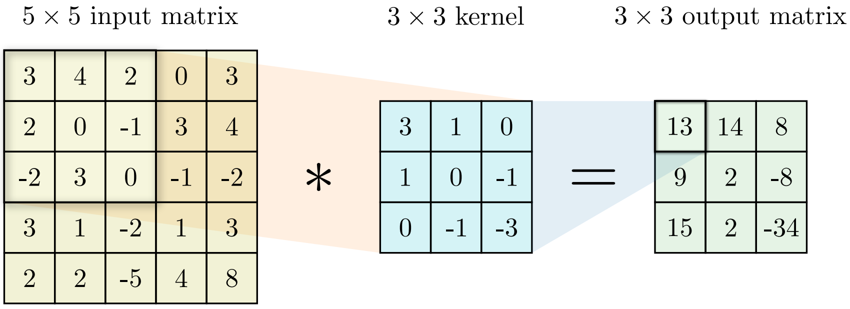


Figure 2.8: Application of the convolution operation (without a bias). When the kernel is applied to the top-left 3 × 3 submatrix of the input, it yields the value 13.

- *Pooling*

  The pooling operation [29] partitions the input into subtensors and aggregates the values of each subtensor together, thereby generating a new tensor with a size that is scaled with respect to the original. Pooling is often interpreted as quick downscaling after feature detection, storing the features detected in a larger area together. For instance, non-parametrized *max-pooling* [39] takes the maximum value of each subtensor, so if the value indicates the likelihood of some condition at that position, max-pooling will keep the highest found likelihood in the subtensor. Pooling is used to quickly downscale (convolutions downscale very little and downscaling with convolutions leads to a very deep network).

- *Fully-connected layers*

Any *d*-dimensional tensor can be *unrolled* into a 1-dimensional tensor (also called *flattening*), after which we may apply fully-connected layers. These fully-connected layers combine the features that were computed over the entire input (e.g. an image). Fully-connected layers are typically used as the last few layers in a CNN, as a step between convolutions and the final output layer (e.g. classification).

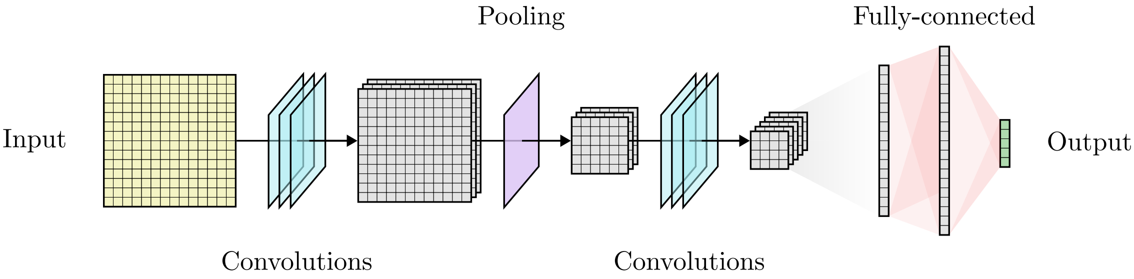


Figure 2.9: The layout of a typical CNN architecture: convolution and pooling layers, followed by a flattening of the features and a number of fully-connected layers.

2.2.3 Recurrent neural networks

Recurrent neural networks [40] (*RNN*s) are networks that can input and/or output sequences of data, by iteratively applying the same parametrized network component over each token in a sequence, and in each iteration returning an output token and a hidden memory state for the next iteration (shown in Figure 2.10). While popular in natural language processing where text is a sequence of tokens, recurrent neural networks can be used in any scenario where there are sequences of variable length, even creating images step-by-step [41]. They are designed to be able to carry the backpropagated gradients across many iterations without vanishing gradients.

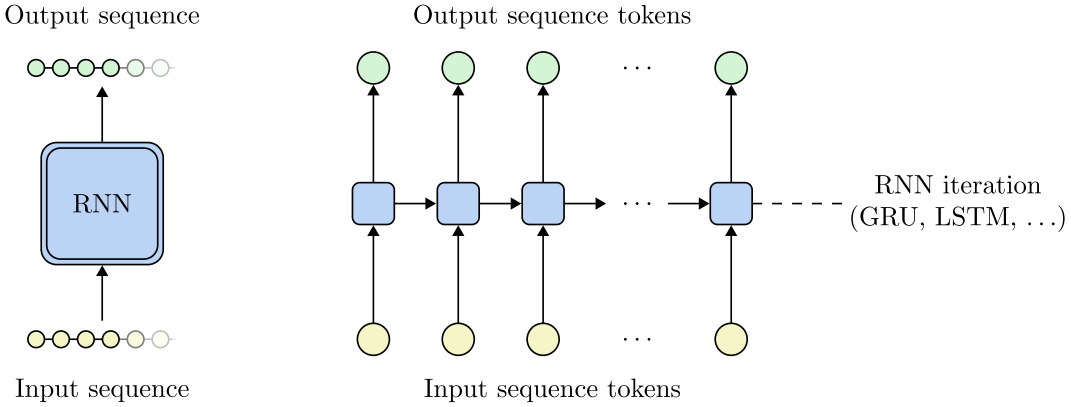


Figure 2.10: A recurrent neural network from two perspectives. Left: from a network architecture perspective, having a sequence as input and output. Right: from a computational perspective, showing the flow of data in the network over time.

The most commonly used types of RNN architecture are *gated recurrent unit* [42] (*GRU*) and *long short-term memory* [43] (*LSTM*). These use gates to control the actions taken in each iteration, such as forgetting the memory cell state, that by nature of their purpose take values in the ranges $[0, 1]$ and $[-1, 1]$. For this reason, the activation functions used within are sigmoid and *tanh*.

To get a non-sequence output, we can simply take the output from the last iteration. To generate a sequence from a non-sequence input, we can simply feed the same input token every iteration.

2.2.4 Graph neural networks

Techniques for applying machine learning directly to graphs have been in development for a number of years. The most common used architecture is the *graph neural network* [44] (*GNN*), and its variants of graph convolutional networks, recurrent graph neural networks and graph attention networks. We will explain each of these, using the graph convolutional network as a basis for recurrent graph neural networks and graph attention networks.

Graph convolutional networks

Graph convolutional networks (*GCN*s) operate on graphs. There are different ways to apply convolution to graphs, two of them being spatial [45] and spectral [46]. When we mention graph convolution, we typically refer to the spatial variant.

Graph convolution can be interpreted as a message passing paradigm [44, 47]: where each node passes a message to each of its neighbors, and each node then aggregates the received messages, combines it with its previous state, and thereby determines its new state. This is illustrated in Figure 2.11. The outgoing messages for a node may be all the same, or different depending on the destination node.

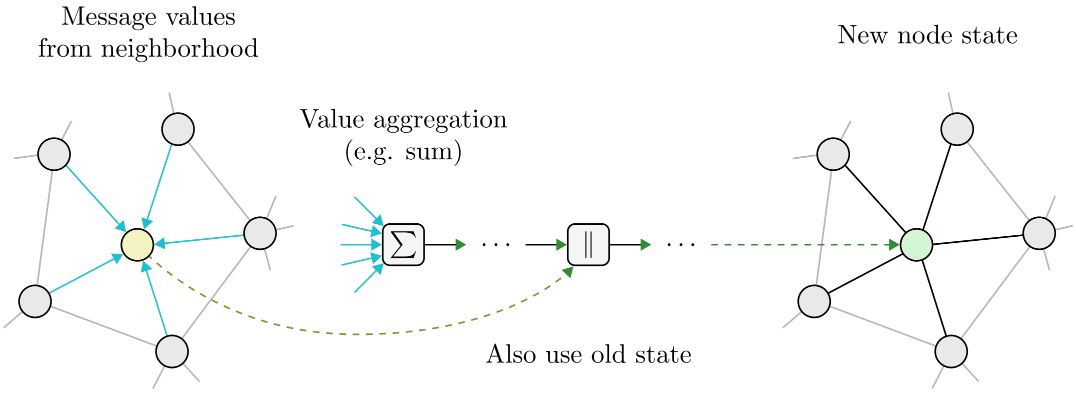


Figure 2.11: A graph convolution: each new node state is determined by aggregating values received from its neighborhood, as well as the previous node state.

Graph convolutions are applied to the node states and message, which usually take the form of hidden vector embeddings. As an architectural component, graph convolution networks often serve one of the two following purposes:

- **Compute an embedding for the entire graph**
  A sequence of graph convolutions are applied, resulting in each node containing an embedding encoding its neighborhood to some distance. Then, all node embeddings are aggregated (with a reduction such as sum or max, or in some architectures concatenation) to one node embedding. This aggregated embedding is then further processed, for example with a series of fully-connected layers. The final output is an embedding of the entire graph, typically of fixed size. This architecture is illustrated in Figure 2.12.

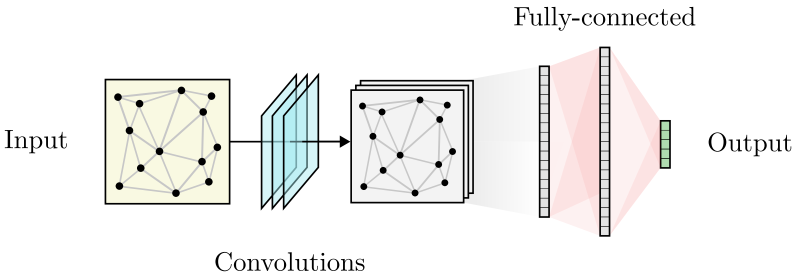


Figure 2.12: The base layout for a GCN that computes a graph embedding.

- **Compute an embedding for each node**
  A sequence of graph convolutions are applied, resulting in each node containing an embedding encoding its neighborhood to some distance. This gives a graph output, and thus a GCN can be used as an architectural component to infer information about each node's surroundings. This architecture is illustrated in Figure 2.13.

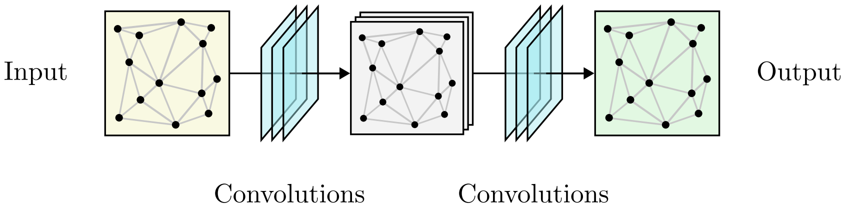


Figure 2.13: The base layout for a GCN that computes node embeddings.

Apart from storing embeddings in nodes, we can also store embeddings in edges. These embeddings can be used to compute the messages used in the message passing paradigm [47]. An edge embedding can also be last-minute computed as a function from its endpoint embeddings when it is needed.

Additionally, graph embeddings can be computed in each step and used as input in the next step (functioning similar to a global node connected to all nodes). This makes it possible to compute both local and global features at the same time [48].

Recurrent graph networks

Gated graph sequence neural networks [49] were proposed as a method for graph-structured inputs as an alternative to problems where otherwise sequence-based models as LSTM could be applied. For simplicity, we shall refer to them as *recurrent graph networks*. These networks, rather than have a number of layers, each with their own parameters (such as weight matrices that are applied to the old state and aggregated messages, or weight matrices that are used to compute the message to be sent), have a parametrized layer that is iteratively applied. This layer may for instance apply an LSTM to the old state and aggregated messages instead.

This means these networks have far fewer parameters when used with many iterations compared to a CNN with many layers. On top of that, they inherit the property of recurrent neural architectures of being able to pass gradients back across many iterations. Because the number of iterations need not be set beforehand, this also lends the network to run iterations until a certain condition is met, which can make the number of iterations applied different for, and depending on, each input. This is done in NeuroSAT [50], an ML-based SAT solver.

Graph attention networks

A *graph attention network* [51] (*GAT*) aggregates the values from a node's neighborhood using *attention* [52]. Attention first computes an attention score vector over the inputs (in this case, the neighborhood node messages) and then uses softmax to turn these attention scores into attention values that sum to 1: these than act as weights for summing the input. There are different methods for determining the attention scores [53], with notable being *dot attention*, which is linear in the size of the embeddings but not parametrized, and *general attention* and *concat attention*, which use a parametrized weight matrix.

While originally introduced for neural machine translation, attention has lead to state-of-the-art results across many disciplines. Some papers have reported better results from graph attention than regular convolution, including the original paper introducing the concept [51].

Attention can also be used to aggregate the node embeddings into a single graph embedding. Joshi et al. [54] compared such aggregation functions and found that using a transformer [55] (an attention-based mechanism) as aggregation does not outperform simple mean and max reduction (evaluated on $n \leq 200$).

An additional use of attention is a *pointer network* [56], which, unlike regular attention which creates an attention distribution that sums to 1 over a fixed number of elements, is also able to create an attention distribution for a variable number of elements. This means the output of a pointer network could denote a choice out of a node's neighborhood, or out of all the edges in a graph.

Chapter 3

Related work

Attempts to solve TSP using deep learning have started only quite recently: nearly all papers on the topic are from the last 4 years. We will first discuss interesting work regarding hybrid metaheuristics for combinatorial optimization in general. Next, we discuss the current state of research in machine learning for TSP, which includes end-to-end constructive approaches and two hybrid metaheuristic approaches.

3.1 Hybrid metaheuristics for combinatorial optimization

Reinforcement learning is a technique commonly used when applying machine learning to combinatorial optimization, because it allows for model formulations that can be trained to take actions in a changing environment, which fits the discrete nature of combinatorial optimization. The resulting algorithm is often an overarching algorithm, such as tree search, that bases its actions, such as expanding a node, on some ML-based heuristic. An example of this is AlphaGo [37].

This technique is also described by Bengio et al. [57] when discussing various generic techniques for machine learning for combinatorial optimization, including branch and bound and tree search where a branching policy is learned. Additionally, the paper specifically describes solving combinatorial optimization using an overarching algorithm making calls to an ML-based subprocedure, making the definition even more general than the concept of a metaheuristic. They note that heuristics can be expected to need to be run often, but this will take up more running time.

With a growing body of research on solving combinatorial optimization problems with machine learning, solutions that are intended to work as an independent algorithm can be adapted to become useful as heuristics in existing algorithms. The NeuroSAT [50] architecture, designed to solve SAT, was adapted to guide existing high-performance SAT solvers, and increased their performance [58].

3.2 Machine learning approaches to TSP

Constructive approaches

Dai et al. [59] propose learning the construction of a tour, similar to greedy search, but replacing the greedy heuristic with an ML-based one. For this, they use graph neural networks. It forms the basis for the idea of a constructive approach: where a tour is constructed element by element.

One can either make a call to an ML model each time a new node is added to the tour, or can make a single call to a model that produces a probability distribution over the edges that can be decoded using greedy search or beam search. The first approach leads to a metaheuristic and may benefit from reinforcement learning. The second approach is not a metaheuristic and the model can be trained end-to-end.

Constructive models include:

- Bello et al., 2017 [60]: RNN to generate tour sequences (pointer network, reinforcement learning)
- Deudon et al., 2018 [61]: attention to encode and decode a set of cities (pointer network, reinforcement learning)
- Kool et al., 2019 [62]: (equivalent to) GAT (reinforcement learning)
- Joshi et al., 2019 [63]: GCN, beam search decoding

Considering these constructive approaches that use a single call to an ML model as end-to-end solutions, Joshi et al., 2020 [54] compare different hyperparameters and architectural choices within these approaches to TSP, notably with GCN. They come to the conclusion that a significant rethinking of the entire solution design pipeline, from network layers and learning paradigms to evaluation protocols, is necessary to solve TSP for larger graph sizes.

Scaling solutions

In 2020, Fu et al. [64] propose a technique to train a GCN architecture on a small TSP instance, and then use the trained model to solve large instances of TSP, by running the model on subgraphs, and merging the edge probability heatmaps for the subgraphs produced by the model.

Guided dynamic programming

In 2021, Kool et al. [65], based on the GCN introduced by Joshi et al. [63], guides a dynamic programming solution to TSP by scoring the partial solutions using the heatmap returned by the GCN model.

Improvement heuristics

Wu et al. [66] introduce the concept of using a reinforcement learning policy as improvement heuristic. They evaluate three pairwise moves on TSP: 2-opt, node swap (swapping 2 nodes in the tour, equivalent to two particular 2-opt moves) and relocate [67] (moving one node to a different position in the tour). They briefly mentioned they found 2-opt to perform the best. The policy network uses attention.

In their 2020 paper called Learning 2-opt Heuristics for the Traveling Salesman Problem via Deep Reinforcement Learning, Da Costa et al. [9] introduce a reinforcement learning algorithm that proposes 2-opt moves by choosing nodes in a way similar to pointer networks. Their approach has various advantages, including

- achieving better performance than constructive approaches in most cases,
- being more easily applicable to different sizes of graphs, even when different from the training data,
- being adaptable to k-opt, as it is based on sequentially selecting nodes for actions.

This thesis builds strongly on the paper by Da Costa et al., which at the time provided state-of-the-art results in improvement heuristics for TSP.

Another type of hybrid heuristic solution is proposed by Zheng et al. [68], who do not use a heuristic to propose improvements directly, but instead use the Lin-Kernighan heuristic to propose improvements, and within LKH, use reinforcement learning to guide it.

Chapter 4

Creating a hybrid metaheuristic with ML

In hybrid metaheuristics using machine learning, we consider the case where a metaheuristic is guided by the information obtained from a ML heuristic. This means we define a higher-level procedure that attempts to find a solution to the problem, that in doing this makes calls to the ML heuristic model.

In this chapter, we explore the step-by-step process of creating a hybrid metaheuristic for TSP. These steps are not prescriptive or generally applicable, but we attempt to mention the notions that guided us. We break down steps we encounter, then provide our response to these, and elaborate on them. Our reasoning is primarily based on our situation, and carries with it constraints and consequences. While we encounter these steps specifically in creating a hybrid local search metaheuristic for TSP, we will attempt to reason from a perspective of creating a hybrid metaheuristic for combinatorial optimization. After considering each step, we determine interesting aspects, and from these observations derive concrete ideas that lead us to construct and evaluate the hybrid metaheuristic approach proposed in this thesis.

First, we provide a motivation for hybrid metaheuristics by listing a number of potential advantages. Next, we define the relation between the metaheuristic and the heuristic: specifically the conditions on the input and output of the heuristic. On the basis of these considerations, we describe and motivate a specific interest in constraining the input and output to induced subgraphs. In the pursuit of the creation of a hybrid metaheuristic, we then consider strategies for contracting or expanding the scope of an existing heuristic within a metaheuristic. This leads us to the general notion of TSP with mandatory edges, which we will formally define in Chapter 5.

4.1 Motivation

There are various reasons for the use of hybrid metaheuristics, specifically for the use of ML-based heuristics within algorithms. Replacing an existing algorithm's already heuristic subprocedure with an ML model can provide a number of advantages:

- **Better performance**: If the current heuristic is not optimal, we may be able to get a model that gives better performance.

- **Shorter running time**: The time complexity of models is usually relatively small, and most ML models have a very predictable computation graph size, which can help in ensuring the running time is predictable and sufficiently bounded. Especially if an exact algorithm or very high-quality solution is known that runs in a high running time or high asymptotic running time, using ML can be a good tool to get reasonable performance based on any pre-chosen running time.

- **Highly adaptable**: The model can be easily retrained and placed in another context entirely, with a new parametrization automatically learned for the characteristics of the data in that context. This means we can specialize it for a new distribution of data, or use it within a modified metaheuristic, even if the overarching problem solved is different.

- **Saving development time**: With a growing body of knowledge about applying machine learning to combinatorial optimization come opportunities for utilizing machine learning in a way that saves on development time, by simply outsourcing some of the algorithmic work to the ML models (usually sacrificing mathematical guarantees and insights).

Additionally, an ML model could just as easily replace a subprocedure for which we know an exact solution rather than a heuristic one (either we have an exact algorithm, or know that such an algorithm exists even if we have not created it yet). This carries the same potential advantages apart from better performance. An ML model can even replace a subprocedure that solves a task for which we do not have any solutions at all, using completely unsupervised techniques.

4.2 Relation between the metaheuristic and heuristic

The first step of creating a metaheuristic is defining the role of the heuristic: the heuristic serves a specific function within the metaheuristic. Based on this function, we can derive the requirements for the input and output. After we have determined the input and output to meet said minimum requirements and desired properties, we can then think about the inner workings of the heuristic, specifically the neural network architecture, without requiring any changes in the rest of the metaheuristic.

The most simple form of heuristic is one that answers only one type of question and has no persistent memory: it is given an input, processes it, and returns an output, and thereby terminates. For instance, a heuristic that chooses two nodes to swap in the tour can be made to have no persistent memory: it can simply be given a matrix of edge lengths as input, and output two integers representing the indices of the chosen nodes.

More complex usage of the heuristic within the metaheuristic is possible, but we wish to think about what matters in defining a basic single-call heuristic without considering all possible more complex cases. Therefore, we abstract away the possibilities of the heuristic having memory and the heuristic answering multiple types of questions:

- A heuristic may have memory and answer multiple queries over time: a heuristic choosing nodes to swap may benefit in terms of performance and running time from having access to the information inferred in the last call, because most of the graph will still be the same. We can abstract this scenario away by interpreting the previous model state as part of the input.

- A heuristic may also answer multiple different questions used (and possibly stored) in different parts of the overarching metaheuristic. For instance, a heuristic that simultaneously chooses nodes to swap, as well as predicts the expected final optimality gap that will be reached: which can be used to terminate the metaheuristic at a specific expected future performance-time tradeoff. We can abstract this scenario away by considering the call to the heuristic independently for each output.

In the following sections, we define what creating a heuristic from an ML model means for the input contract between the metaheuristic and heuristic, and then do the same for the output contract. This helps us to decide if a specific possibility for a heuristic, i.e. a specific role within a metaheuristic, is suitable for creating a ML-based metaheuristic. Simultaneously, we derive specific classes of suitable input and output, that lead us to develop our hybrid metaheuristic.

4.3 Input

Between the metaheuristic and the heuristic lies an input contract: this specifies the input the metaheuristic will provide to the heuristic. We will first introduce requirements and desired properties on the input contract in a progression that follows our order of consideration in creating a hybrid metaheuristic. Then, from this, we derive the concept of heuristics that use as input an induced subgraph containing information about the tour, and motivate its use.

The first condition to consider, is that **the input must be in a format that can be used as input to an ML model.** Machine learning mostly operates on tensors of floating-point numbers. Typical algorithmic data is not stored in such a form, but rather in the form of lists, sets, other more complex data structures chosen for running time guarantees, and references (for example indexing nodes by integers). This means that often we must apply a transformation of the data. If it is exceedingly hard to transform the data for the heuristic into a format that an ML model can use, a different delimitation between metaheuristic and heuristic should be considered. This is relative: a graph may have been judged as difficult to express in tensor form 10 years ago, but with the advent of graph neural networks this has become relatively trivial. Expressing a set of words in tensor form is not difficult using an autoencoder [69] and a sufficient corpus, but can be nearly impossible without a known corpus or list that encompasses the potentially occurring words.

Knowing that the input must be translatable to a format that can be used in an ML model, we decide which input to use. The metaheuristic has some state at the time of the call to the heuristic, and the input will be a subset of this state. **The input must have enough information to derive the desired answer.** If the answer does not need to be optimal, the input must have enough information to be able to derive a qualitatively sufficient answer. This is easily satisfied for existing subprocedures for which we can define a set of information beforehand that is used in this subprocedure and therefore certainly sufficient to answer the question. For example, we know that a distance matrix for a graph is sufficient to judge all potential 2-opt moves, and with enough exploration of the state space, judge all potential sequences of 2-opt moves. Even though we may not immediately have an efficient approach (the first idea would be to loop over all 2-opt moves), we know that we can answer the question of finding the 2-opt move with the greatest decrease in tour length using only this input, and this is sufficient. Alternatively, to find the best k-opt move for k given edges, we merely need the distances between all their endpoints. It may be useful to consider if alternative inputs are known to make it easier to answer the question, but we should assume that once the question can be answered given the chosen input, we can train an ML model for it. The assumption of using machine learning, after all, is that we can automatically obtain a parametrization of a defined neural network to perform a given task. Existing known heuristics that are not optimal, such as the choice of the next node in greedy search, can indicate a lower bound for the potential performance achievable by an ML-based heuristic if provided with the same input (for instance, in the case of greedy search, the distances to the neighbors).

With the input sufficient to solve the problem, we could simply choose to input all parts of the metaheuristic state that we can translate to tensor form. However, to lower time and space constraints and prevent overfitting, we desire, but not require, that **the input must be as small as possible**. We can apply this as an afterthought once we have established an opportunity for a heuristic: if we have a usable definition of the input, we can prune it.

While we can prune the input arbitrarily and simply lower or lose our expectations of the potential performance, it helps to consider what information is important for the given question. This way, we can reduce the input to the heuristic with the least loss to potential performance. In other words, we desire that the input to the model encodes the information significant to the problem. Pruning the input effectively is hard when it is known that the input is sufficient to answer the question (because there is a known algorithm or heuristic solution with some specific performance), but now knowing why (because you do not know how). In the case of AlphaGo [37], which plays the tiled board game of Go, a heuristic chooses a number of tiles of interest, which are then further evaluated by a second heuristic to get a desired output. In that case, it would be very hard to prune the tiles even more before the first heuristic, based on the significance to the second heuristic: we know an effective pruning exists (namely the first heuristic), but we do not understand precisely how it works.

At the same time, we tend to want to apply hybrid metaheuristics to precisely those cases where we have a limited understanding of the underlying significant concepts. Therefore, we propose that when defining a heuristic, we can instead aim to achieve that **the input to the model encodes the significant relationships in the data.** For TSP, where the data consists of a graph and a tour, this would typically include an encoding of edges and their lengths, and some encoding of the tour, such as a sequence of node indices. While we may not know exactly what relationships are important for what problem, we know that the general structure of the input should be so that it retains details about the original structure that can be utilized by the model afterwards. For example, especially in the case of large sparse graphs, it should be at least as easy for the ML model to infer something from the representations of two nodes that share an edge, as to infer something from the representations of two that do not. Concretely, in a GCN, nodes that share an edge will pass information in a convolution step, meaning the model can infer features from their representations, which in the case of GCN are their node embeddings. This would not be possible if the edges were not provided. As a more general example, in image recognition, classifying an image with shuffled pixels is much more challenging. The fact the pixels are correctly ordered means we can use this encoding of their proximity relationship (for example with convolutions).

Encoding the tour is not trivial: encoding the tour as a sequence of the nodes (their indices or some embeddings) to be fed into an RNN is imperfect because it does not explicitly encode the fact that each node will occur exactly once or that the sequence is always of the same length as the number of nodes. Alternatively, marking the next node in the tour for each node is imperfect because while it does encode that each node has one unique successor, it does not encode the fact that each node has a unique predecessor, or the fact the tour is one connected cycle.

Lastly, when we have defined an input definition and have a translation from its form in the metaheuristic to its form in the heuristic, as well as considered whether it contains the right information, we must consider that the ML model that performs the role of the heuristic must be trained. With supervised learning, this means **we should get many input samples for this specific heuristic.** Having or being able to generate a great number of test cases that are valid input for the heuristic, especially if the distribution matches that of typical input instances observed in an evaluation of the metaheuristic, is necessary to train the ML model successfully. For instance, to train a model for a heuristic that chooses a next node for a partial tour, we can generate sample graphs that already contain a partial tour with between 1 and $n-2$ edges.

Heuristics on induced subgraphs

In the case of TSP, limiting the input to a useful subset of the state means we limit the input to some part of the graph and the tour. A particularly well-defined choice of input is to choose a particular set of nodes S and use as input the induced subgraph $G[S]$ defined by these nodes and all edges between them (for reference, denoted as $E[S]$, so that $G[S] = (S, E[S])$). For any heuristic that uses only information within a definable induced subgraph, we can determine some minimal set S for which the induced subgraph $G[S]$ contains the input to the heuristic. For example, a k-opt move considers at most k edges to be removed, so the induced subgraph of the considered edges' endpoints is a minimal induced subgraph that covers the information used in the k-opt heuristic. However, guaranteeing we do not propose k-opt moves that will break apart the tour, we must also know about the tour. The information needed is not necessarily the entire tour, but only which edges in the induced subgraph are part of the tour, and which nodes in the induced subgraph are connected via a chain of tour edges outside of the induced subgraph.

We define an input specification, the *induced subgraph with induced tour*, of a subgraph defined by some nodes S, which contains all edges $E[S]$ between these nodes and marks for edges whether they are part of the tour, and whether their two endpoints are connected via a chain of outer tour edges.

For k-opt moves, we note that which edges within the subgraph are part of the tour is not necessary: the current edges are just one potential output, and can be removed entirely. In such cases where we do not need to know which edges are currently in the tour, but do need to know which edges represent outer edge chains, we call the subgraph an *induced subgraph with induced outer tour*. The *induced outer tour* refers to the outer tour (the tour without those edges in the subgraph) being encoded within the induced subgraph.

The definition of an induced subgraph that covers the necessary information for a heuristic gives rise to a specific class of heuristics, namely those for which there is always an induced subgraph with induced tour of at most k nodes that contains the input normally provided to the heuristic. The input is determined by the metaheuristic, and we therefore can explicitly provide the mechanism that determines this input (which we will have since we are creating the metaheuristic). Therefore we can also certainly define a procedure that explicitly finds an induced subgraph that contains the input normally provided to the heuristic. Because we can explicitly define a way to describe the input to the heuristic as an induced subgraph merely defined by a set of nodes S of size at most k, we call these heuristics *explicitly input-bounded to k*.

We can now see that heuristics that simply use as input an induced subgraph of size $|S| = k$ with induced tour, have as input a superset of the input of all heuristics explicitly input-bounded to k, and therefore generalize over all such heuristics in terms of potential performance on the same problem. For example, some operators such as *relocate* [67], which relocates the node to a different position in the tour, can be performed on a subgraph of 5 nodes: the node to be moved, its predecessor and successor in the tour, and two nodes in the tour between which the node to be moved would be placed. Whether or not applying this relocation will decrease the tour length requires only $O(1)$ to determine, and considering all possible relocations in the graph requires only $O(n^2)$, despite relocate being able to express some moves that cannot be expressed by a sequence of 2-opt moves [70]. Rather than determine such operators concretely, we can observe that relocate is contained within 5-opt. We then construct a metaheuristic that provides an induced subgraph of size 5 with induced outer tour to a heuristic, that comprises an ML model with space and time complexity $O(k^2)$. By applying the heuristic $(\frac{n}{5})^2$ times, we have constructed a metaheuristic with time and running time complexity $O(n^2)$. In this way, we have successfully used the concept of explicitly input-bounded heuristics to outsource the coming up with novel operators to machine learning.

Considering explicitly input-bounded heuristics helps to fulfill the input contract requirements. We can feed the input into an ML model (we know how to encode graphs for graph neural networks) and we know it contains enough information to derive the desired answer. Finding the smallest k to which a heuristic is explicitly input-bounded makes the input as small as possible. This works best for heuristics that likely need most of the data within a subgraph, it works less well for heuristics that use very specific information across a large subgraph (such as that operate on long segments of the tour). The input to the model encodes at least some significant relationships in the data by nature of using an induced subgraph that contains all edges present between any nodes, and the information about the tour that is relevant to that subgraph. And finally, we can obtain many input instances to train the neural network, since we can sample subgraphs from generated larger graphs with tours.

We conclude that heuristics that use as input an induced subgraph of size k with induced tour can generalize in potential performance over heuristics explicitly input-bounded to k, and can be expected to fulfill our desired properties on the input contract. Therefore, we decide to investigate these heuristics further.

4.4 Output

Conversely, between the heuristic and the metaheuristic lies an output contract: this specifies the output the metaheuristic will receive from the heuristic. We will first introduce requirements and desired properties on the output contract in a progression that follows our order of consideration in creating a hybrid metaheuristic. Then, from this, we derive the concept of local search that uses a heuristic to performs k-opt on induced subgraphs.

Analogous to the input, **the output must be in a format that can be returned by the ML model.** There are only limited existing strategies for ML models constructing data structures, and creating such a procedure is not trivial. If the answer is complex or must fulfill discrete rules, having post-processing happen after the model, so that the problem the model answers becomes only a pre-solution to the actual question at hand, is almost certainly necessary. A tour is a particularly complex data structure: there are many sets of requirements that each are valid and complete descriptions of a tour, but none of these seem to particularly work well with machine learning due to their difficult to encode strict constraints. For instance, requiring that the tour is connected is particularly complex as this creates a very long-distance dependency. A sequence of nodes can be output by an RNN and a pointer network, masking out choices that were already made before. An other solution is the decoding of a tour from a probability distribution over edges, using greedy or beam search. In these cases, it can be hard to predict whether choices down the line will hurt performance, and when such failures happen, there is no clear pathway to solve them. On the other hand, fortunately, for many common questions such as picking one out of a set of provided choices or computing a number, the output being able to be returned by the model is not an issue.

On top of that, for the output to be used to train the model, **there must be a differentiable loss function to evaluate the output.** While at a minimum some loss function must be defined, clearly it should return a qualitative indication of the output that leads the model to converge to give outputs that are desired. The most basic way is using a loss function based on difference, such as MSE or logistic loss, on an optimal output derived from evaluating the input with a separate procedure. Similarity to the optimal output may not always indicate actual quality (for example, for TSP the number of edges chosen that also appear in the optimal tour is not the same as the actual quality of the solution - which is the tour length). This loss function must allow for gradient descent to converge well. For example, if the model should output values between 0 and 1 as probabilities that must sum to 1, softmax or the logistic function must be used, rather than for instance forcing the network to make the probabilities sum to 1 by some loss component that increases when the sum is not 1. Using softmax will lead to smooth convergence, whereas introducing extra loss components will not.

The difficulty of long-distance relationships

We performed preliminary experiments to test the effect of loss functions on end-to-end TSP, which are difficult to define. While these experiments were purely exploratory in nature, and we did not derive sufficiently representative quantitative results to mention here, we will briefly discuss our observations, that lead us to choose to focus on inferring short-distance relationships. We evaluated a graph neural network using DGL [71], with a wide variety of design choices:

- Operations performed in each step were: convolution, attention, LSTM, or multiple in parallel
- The number of layers of convolutions, attention and recurrent blocks in one message-passing iteration were varied
- Edge embeddings were used in the messages passed over them, and after each step derived from their endpoint embeddings using multiple dense layers
- Activation functions ReLU and weighted sigmoid [72] were used
- In some evaluations, a global graph node, with an extra large node embedding, connected to all other nodes was added

The number of design choices was to make sure that we could quickly make some observations that were invariant over these differences, thereby ruling them out as likely causes for the observations. Dense graphs of small sizes, as well as sparse graphs of sizes of around 200 were evaluated.

A probability was computed for each edge separately, as a logit of a specifically indexed value in the edge embedding. We then tested a variety of methods to translate these probabilities to a tour, and loss functions over the logits. To derive the tour, we tried:

- Take the tour as all edges with logit value greater than 0
- Take the tour as the n edges with the highest logit value
- Construct a tour through greedy search, using the probabilities derived from the edge logits

For loss functions:

- A logistic loss on the output edge probability and whether it actually appears in the optimal tour
- A specially designed loss with 3 components:
  - A length loss that minimizes the length of the tour (the sum of each edge's probability multiplied by its length)
  - A degree loss that requires nodes to have degree 2 (if the sum of probabilities of their incident edges is lower than 2, we multiply the difference by the length of the longest edge in the graph to ensure that increasing the probability of any incident edge is always beneficial for this loss component)
  - A connectivity loss that requires nodes to be connected; we
    - stochastically select a few nodes,
    - for each of them compute a single-source shortest path using a fully differentiable adaptation of the Dijkstra algorithm, where instead of minimizing the sum of edge lengths we maximize the product of edge probabilities (which would optimally be 1 for every destination node),
    - then for each of the source nodes sum the resulting value from all destination nodes and use as loss the difference between this sum and n

During these experiments, we failed to create any model that would consistently give output that constitutes a valid tour. The first loss function gave output that did not seem particularly useful, mostly containing short edges in the graph. Trained on the first loss function, the model heavily preferred to output short edges that lead to nodes with degree higher than 2, over outputting long edges that are required in any reasonably qualitative tour. However, we found that, irrespective of the design choices, using the second loss function, the model quickly learns to output a set of short cycles that together contain all nodes in the graph, often including cycles containing up to 8 nodes. However, attempting to get the same models to produce cycles on more nodes, even on graphs with sizes of under 20, seemed to become more difficult at a steep rate. Furthermore, increasing the weight of the loss component decreased the overall quality of the output much more than it lead to finding larger cycles.

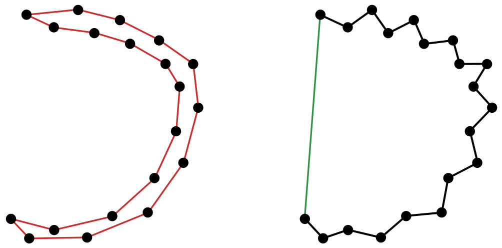


Figure 4.1: A particularly difficult case: the left solution is heavily favored by all of our graph neural network experiments. However, it has an optimality gap of 20% w.r.t. the solution on the right.

Generally, from this, we deduced that graph convolutions are naturally strongly predisposed to short-distance relations, as showcase using a difficult input in Figure 4.1. The number of nodes that information can reach after a few convolutions increases exponentially, but the node embeddings have a static size. If significant short-distance relationships do exist, then their effect will appear an exponential factor greater in the loss function than significant long-distance relationships. Therefore, long-distance relationships would appear to the parameter optimization method, in this case gradient descent, as statistically unlikely. Unless, of course, their significance compared to the short-distance relationships outweighs their smaller number of occurrences in the forward pass of the model.

We therefore assume that existing graph neural networks that are based on the message-passing paradigm, and use the neighborhood relation to determine which nodes to send messages to, have exponentially increasing difficulty in using longer-distance relationships. A TSP tour is an example of a data structure that is typically defined in a way that creates long-distance relationships. Since the number of edges in larger graphs grows quadratically, sparse graphs are often used to combat this growth. The resulting sparsity leads to the increasing distance in number of convolution iterations between nodes that are not directly connected. Considering this, we conclude that if we wish to solve TSP on larger graphs, we must move away from the current model training paradigm of graph convolutional networks, because in its current form, this will lead to an unavoidable set of inputs high in important long-distance relationships for which the model will perform poorly. Worse than just performing poorly for such cases if the model is not trained on them, is that the model can be expected to perform quite poorly on them even when specifically trained on them. This is because it is very hard, through using a specific dataset distribution, to increase the significance of the long-distance relationships enough to combat the exponential decline of their occurrences compared to short-distance relationships. And even then, there will always be a next larger distance relationship for which the model immediately performs poorly.

Consequently, we decide to move our focus towards heuristics that attempt to infer only short-distance relationships in their output. While we believe that this will unavoidably limit the results we can achieve, particularly on a specialized subset of input, it spares us from fruitless optimization and trial-and-error. It also ensures we can judge the effectiveness of the metaheuristic we create within the bounds of what we already expect are upper bounds on performance, given the short-distance relationships we focus on.

Heuristics for induced subgraphs

In the case of TSP, limiting the stated goal of the heuristic to infer only short-distance relationships means that the output of the heuristic involves closely connected nodes and edges. Analogous to how induced subgraphs were a particularly well-defined choice of input, we can take the nodes S on which the heuristic infers information, and define their induced subgraph $G[S]$ with edges $E[S]$. If we can explicitly define a way to describe the output of the heuristic as an induced subgraph merely defined by a set of nodes S of size at most k, we call this heuristic *explicitly output-bounded to* k.

Given our conclusions about the expected higher effectiveness of current approaches on more closely connected subgraphs, we then assume that if we have a heuristic that takes as input an induced subgraph with induced tour $G[S]$, that the heuristic can infer the most useful information on this same induced subgraph. It now naturally follows that we define a heuristic as *explicitly bounded to k* if we can explicitly define a way to find a set of nodes S of size at most k, so that both the input and the relevance of the output of the heuristic are limited to the induced subgraph with induced tour derived from S.

A heuristic that uses as input some induced subgraph with k nodes, and returns an output relevant to the same subgraph, is therefore efficient in terms of training, by nature of the common choice of network architecture being predisposed to short-distance relationships. On top of that, in terms of potential performance, it generalizes over all heuristics explicitly bounded to k. In the case of TSP, this means we can create a hybrid metaheuristic that selects a subgraph induced from a set of nodes S and calls an ML-based heuristic that returns some output pertaining to the same subgraph. Our resulting metaheuristic then already generalizes in terms of potential performance over all such metaheuristics that have explicitly bounded heuristics.

Concretely, we normally observe that different operators that operate on subgraphs containable in small induced subgraphs, such as relocate and small k-opt, have various running times and potential performances. We can simply create a metaheuristic using a heuristic based on a model with specific time and space complexity, that operates on induced subgraphs of at least the same size as the original operator, and then train the model to obtain a hybrid metaheuristic that can approach the original performance to the degree achievable within the chosen complexities.

4.5 Adjusting the scope of the heuristic

When we have created a metaheuristic using an ML model as a heuristic, we may wish to consider modifying the role of the heuristic in the metaheuristic, based on performance of the metaheuristic, perceived opportunities or other reasons. This will carry with it changes in the ML model, since it must now work with potentially different input and produce different output, as well as must perform over a potentially different input distribution and successfully converge during training. Based on the techniques introduced, we derive that we can adjust the scope of heuristics that operate on induced subgraphs to a form in which they solve the problem of subgraph optimization over their induced subgraph, thereby leading to the problem formulation of *TSP with mandatory edges*, where a given set of edges must be included in the tour.

We suggest the following generally applicable modifications to the role of the heuristic within the metaheuristic:

- **Change the scope of the input**: the heuristic may have better performance when it has access to additional information. Also, unnecessary information may be trimmed (which can lead to both better running time and performance). For example, we may choose to increase the size of a provided subgraph by including nodes at a distance of 1 edge from the current subgraph, if we feel the heuristic misses opportunities that can be derived from these extra nodes. As an opposite example, if we wish to select a 2-opt move to perform, we can choose to just use 1/4th of the graph rather than the graph in its entirety: this will save time and often find a good 2-opt move as well.

- **Make the heuristic responsible for an additional task**. Procedures that produce some of the input or use some of the output of the heuristic can be added to the scope of the heuristic, where it then uses those procedures' input or produces their output. We note that particularly for postprocessing performed after the heuristic, assimilating those steps into the model gives the model greater flexibility over the final output. For instance, requiring the model to output a sequence of node indices directly, rather than applying beam search to a probability distribution over the edges, gives the model greater flexibility in determining the final sequence of nodes, but makes it much more difficult to design a successful architecture.

- **Move a task in the heuristic to outside of the heuristic**: this may be necessary when it proves difficult to train the heuristic end-to-end, or useful when an alternative solution to a specific step within the heuristic is known and preferred (for instance, because the current model is slow and we know a non-ML solution to this specific step that is fast, so we decide to move it out of the ML-based heuristic's scope).

- **Merge or split heuristics**: especially with highly overlapping input or highly disjoint input requirements, respectively. When two heuristics share mostly the same input, they may learn many of the same features over this input and therefore it may save on running time and space, and sometimes improve consistency of convergence, to merge the heuristics into one that produces the output for both. When a heuristic produces multiple outputs that seem to depend on highly disjoint sets of input, it may be worth splitting the heuristic into two heuristics to provide greater control over hyperparameters and network architecture, and get better insight into performance and other properties of the heuristics. Note, that even if the input and output of multiple tasks that a heuristic fulfills are completely disjoint, performance on both tasks may benefit from it being the same network due to the same operations learned by the network applied to both the inputs providing useful features for both tasks. For instance, if one model must both detect for a dataset of images whether it contains a parrot and translate another dataset of images to text, their inputs and outputs are disjoint but due to the common nature in input and problem (detecting visual features), they could both benefit from sharing parametrized modules (for example those that perform low-level edge detection). Similarly, if two heuristics both operate on graphs, they may have very different tasks, such as outputting the next node for a partial tour, or judging the probability of an edge being in the optimal solution, but they may both benefit from detecting many of the same features, such as clusters.

- **Collapse an iteration**: for any loop, we can change its order, iterate over only a subset of steps, or just evaluate one step. For the general case it means that we can use a heuristic to determine the steps that will be evaluated. More specifically, in the context of modifying the heuristic, it means that when a heuristic itself is placed in the body of an iteration, we can can write the entire loop as 'for each x, get y, then do z' and the heuristic as 'for x, return y' - and this means that we can reason there is an aggregate result Z over all these steps. We may then think about achieving such an aggregate result using only a heuristic, i.e. replacing the entire loop with a single heuristic 'using all x, do z'. For instance, assume we have a heuristic that selects a number of neighbors N_i for each node i, and it is currently done inside a loop over the nodes, for each node i selecting these neighbors N_i and then taking an action based on $N - i$. We can then instead run a single heuristic with a similar architecture over the entire graph, and get as output these neighbors N_i for each node i, and apply the changes based on those all at once. When this strategy is possible, it can allow for significant parallelization, and constraining the size of the resulting heuristic gives an additional means of control over the tradeoff between performance and running time. This is essentially the basis for the heuristic proposed in Learning 2-opt Heuristics for the Traveling Salesman Problem via Deep Reinforcement Learning [9]. In this paper, instead of traditional 2-opt, which loops over all potential 2-opt moves and for each, judges whether to apply it, a 2-opt move to be applied is repeatedly selected.

TSP with mandatory edges

For an explicitly bounded heuristic, the scope of the relevance of its output is limited to the induced subgraph it receives as input. This output is then used by other steps in the metaheuristic, but we made the assumption that the heuristic should only aim to provide relevant output for the induced subgraph. If we only use the output in the resulting steps to modify details of the induced subgraph, we can make sure we can train the ML model as a heuristic for a problem defined only over the induced subgraph. But, if this is the case, then the best result we could potentially derive directly from this output is an optimal configuration of the tour edges that fall within the induced subgraph: since any other tour edges fall outside of the intended scope of inference.

For any output of the heuristic with a limited scope of relevance, we can describe the *expressiveness of the output*. Based on the output the heuristic produces, the metaheuristic takes steps that potentially lead to a modified configuration of the tour edges within the induced subgraph. The set of potential configurations we can reach is the expressiveness of the output of the heuristic. While a heuristic may have many different types of output, if the output only pertains to a specific induced subgraph, the expressiveness of the output becomes easier to define. After all, the flow of data leading to changes in the tour will be comparatively partitioned along the lines of the used subgraphs.

This allows us to formulate yet another generalization, namely the class of heuristics that are explicitly bounded to some subgraph size k, and return as output a new valid configuration of the tour edges that lie within the induced subgraph. Essentially, these heuristics remove the existing tour edges within the induced subgraph, and only use as input the induced subgraph with induced outer tour. Therefore, their input is a graph with a set of mandatory edges (the edges with endpoints that are connected via an outer edge chain) that must be included in the final tour, and the output is a valid tour over the subgraph consisting of those mandatory edges and other edges within the subgraph. This can be defined as an independent problem, *TSP with mandatory edges*, which adds to TSP the requirement of including a given set of mandatory edges in the output tour. The class of heuristics that solve this problem generalize over all heuristics explicitly bounded to the same subgraph size in terms of the expressiveness of the output.

This notion leads to the proposition of a local search algorithm that iteratively calls an ML-based heuristic to produce a solution to TSP with mandatory edges over a specific induced subgraph, to then apply the resulting tour changes to the tour over the entire graph. In Chapter 5, we will formally define TSP with mandatory edges, and describe the resulting local search algorithm in greater detail. In particular, the heuristic using TSP with mandatory edges is illustrated in Figure 5.1, and the local search metaheuristic in Figure 5.2.

Chapter 5

TSP with mandatory edges

In this chapter, we formally introduce *TSP with mandatory edges*, a problem formulation that adds the requirement of including a given set of mandatory edges to TSP.

While considering the steps to construct a hybrid metaheuristic, we made observations that helped us derive TSP with mandatory edges from its potential use in local search. In this chapter, first we provide a short formalization of the problem and specify an exact solution. Next, we motivate its significance in local search with machine learning. Finally, we define a transformation between the problem and regular TSP, and describe adapted versions of local search and LKH that can be used to solve TSP with mandatory edges.

In the sections that follow, we describe two methods to create a local search algorithm based on machine learning. One method uses an adapted version of beam search to construct a tour from an edge probability distribution computed with machine learning. The other method adapts the reinforcement learning model by De Oliveira da Costa et al. [9] to solve TSP with mandatory edges in a way that benefits from parallel computation.

The next chapter, Chapter 6, will contain the experimental evaluation of primarily the second method, with which we show that by using ML models over TSP with mandatory edges as a subgraph optimization method, we obtain a metaheuristic that significantly expands the range of problem sizes that can be solved with that ML model. Specifically, we are able to perform a stress test on $n = 10^5$, and achieved noteworthy results on TSPLIB instances of sizes up to $n = 10^4$. Notably, the efforts towards parallelization and the introduction of useful hyperparameters to the method that are described in this chapter form an important factor in the results we obtain.

5.1 Definition

We will define TSP with mandatory edges as analogous to TSP, except with the requirement that a given set of mandatory edges must be included in the resulting tour[1].

In TSP with mandatory edges, we are given a graph $G = (V, E)$ and distance matrix c, as in TSP, and additionally a mandatory edge set $E_m \subseteq E$. We wish to find the shortest tour π, as in TSP, with the additional requirement that $E_m \subseteq E_\pi$.

Note that for symmetric TSP we can consider edges to be undirected, so analogously, *symmetric TSP with mandatory edges* does not require mandatory edges to keep the same direction in the tour.

For completeness, we include at least a full exact solution to TSP with mandatory edges. We adapt the Dantzig–Fulkerson–Johnson ILP formulation (for complete TSP with positive edge lengths and no self-loops) [18] by adding a requirement to use all mandatory edges. This requirement can be written as $(i, j) \in E_m \Rightarrow x_{ij} = 1$ and results in the following ILP formulation:

[1] This concept has been described before in the form of *virtual edges* [73] and *fixed edges* [74].

$$
\begin{aligned}
&\min \sum_{i=1}^{n} \sum_{j \neq i, j=1}^{n} c_{ij} x_{ij} : && \\
&x_{ij} \in \{0,1\}; && \\
&\sum_{i=1, i \neq j}^{n} x_{ij} = 1 && j = 1, \dots n; \\
&\sum_{j=1, j \neq i}^{n} x_{ij} = 1 && i = 1, \dots n; \\
&\sum_{i \in Q} \sum_{j \neq i, j \in Q} x_{ij} \leq |Q| - 1 && \forall Q \subsetneq \{1, \dots, n\}, |Q| \geq 2; \\
&x_{ij} = 1 && (i,j) \in E_m
\end{aligned}
$$

5.2 As local search method

5.2.1 Algorithm description

We can use TSP with mandatory edges to construct a local search algorithm on regular TSP.

We iteratively select a subgraph, marking the edges, of which the endpoints are connected by a chain of outer edges, as mandatory edges. Then, we solve TSP with mandatory edges on the subgraph, and perform a local search move on the overarching tour, that replaces the previous subgraph tour edges by those newly found.

We repeat the following local search step, illustrated in Figure 5.1, on a given graph $G = (V, E)$ and tour π (using directed edges in the case of asymmetric TSP):

1. Select a subset of nodes $S \subseteq V$ of size $|S| = k$.
   Get the subgraph $G[S] = (S, E[S])$ defined by these nodes and the edges between them.

2. Define a set of (directed) mandatory edges $E[S]_m$ as all chains of edges that do not appear in $E[S]$.

3. Solve (directed) TSP with mandatory edges over $G[S]$ with mandatory edges $E[S]_m$ to get a tour σ with edges $E[S]_\sigma$.

4. Apply the move X, Y that removes the original tour edges in the subgraph, and adds the non-mandatory tour edges found by TSP with mandatory edges instead, so that $X = E_\pi \cap E[S] \wedge Y = E[S]_\sigma \setminus E[S]_m$.
   This operation takes $O(n)$ time. If any edges in $E[S]_\sigma$ were not in E, we add them to the graph.

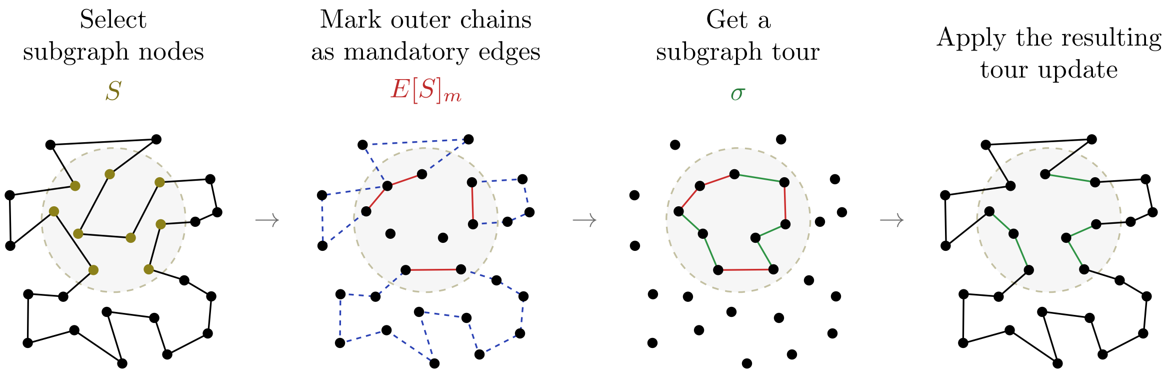

Select subgraph nodes
S
Mark outer chains as mandatory edges
$E[S]_m$
Get a subgraph tour
σ
Apply the resulting tour update
→
→
→

Figure 5.1: One iteration in the local search: we select a subgraph, mark mandatory edges, solve TSP with mandatory edges on the subgraph and apply the changes in the subgraph tour σ to the tour π.

Given a condition for termination of this local search, this leads to the metaheuristic shown in Figure 5.2.

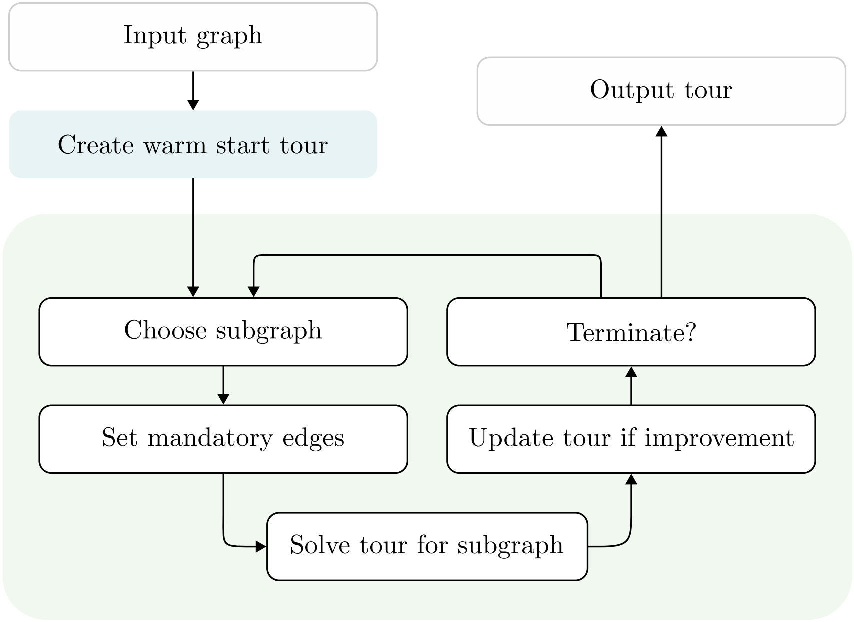


Figure 5.2: An overview of the local search metaheuristic that determines moves by solving TSP with mandatory edges over a subgraph in each step.

5.2.2 Significance in metaheuristics guided by ML

We briefly summarize why we believe local search using TSP with mandatory edges and machine learning can effectively generalize over a whole class of potential metaheuristic designs.

For any local search heuristic, we classify the heuristic as *explicitly bounded* to k if we know a set of nodes $S \subset V$ with induced subgraph $G[S] = (S, E[S])$ before computing X, Y, so that $X \cup Y \subseteq E[S]$. A local search metaheuristic is classified as *explicitly bounded* to some subgraph size $k < n$ if every local search heuristic used within is explicitly bounded to k.

LKH is not explicitly bounded. k-opt is explicitly bounded to subgraph size $2k$ (since each of the k edges, that are considered for removal, have 2 endpoints, so the minimal induced subgraph containing these edges has at most $2k$ nodes).

For any local search heuristic explicitly bounded to some subgraph k, we can turn every outer edge chain in the tour entirely of edges not in $E[S]$ into mandatory edges, and solve TSP with mandatory edges over $G[S]$. The X, Y move found this way leads to a tour of lower or equal length than that of any move found by another type of local search heuristic on this subgraph, since it is optimal with respect to the subgraph (and the outer edge chains are not modified by this heuristic).

Therefore, designing an ML model to solve TSP with mandatory edges has the potential to result in a local search metaheuristic that is at least as expressive w.r.t. moves as any other local search metaheuristic that is explicitly bounded, as mentioned in Section 4.5.

Assume we have a local search metaheuristic that is explicitly bounded to k. Since applying ML to TSP with mandatory edges is at least as expressive, and the ML model contents could be anything, one could assume that an ML-based heuristic exists that can be used in a local search metaheuristic using TSP with mandatory edges, and lead to the same overall performance.

Consequently, we suggest it is particularly interesting to research using machine learning for TSP with mandatory edges as a method of developing hybrid metaheuristics that are based on local search.

5.3 Applying existing solutions for TSP

5.3.1 Local search

We can adapt any local search for TSP, such as k-opt, to work with mandatory edges simply by forbidding any X, Y move that removes any mandatory edges (moves with $X \cap E_m \neq \emptyset$).

5.3.2 TSP with negative edges

We will show that we can transform any instance of TSP with mandatory edges to an instance of regular TSP with the same set of optimal tours. Some edges in the resulting instance will have negative distances. We can then solve TSP with mandatory edges using existing solutions that allow negative distances, such as Chained Lin-Kernighan [24]. Note that when transforming an instance of 2D-TSP with mandatory edges to an instance of TSP, the resulting TSP is not necessarily Euclidean. For a given instance of TSP with mandatory edges with graph $G = (V, E)$, distance matrix c and set of mandatory edges E_m, take the instance of TSP with graph G and distance matrix c', with

$$c'_{ij} = \begin{cases} -N & \text{if } (i,j) \in E_m, \\ c_{ij} & \text{otherwise.} \end{cases}$$

with N some arbitrarily large number that ensures that every optimal solution must include every mandatory edge (if solvable). In particular, this always holds when

$$N \geq n \cdot \left(\max_{i,j \in V} c_{ij} - \min_{i,j \in V} c_{ij} \right) - 1$$

Since no two tours can differ in length by more than $n \cdot (\max_{i,j \in V} c_{ij} - \min_{i,j \in V} c_{ij})$, not including a mandatory edge can never lead to a shorter tour.

Since we can transform any instance of TSP with mandatory edges to an instance of TSP with the same set of optimal tours, we can use LKH with mandatory edges by first applying this transformation. The implementation of Chained Lin-Kernighan we evaluate, Linkern [15], uses signed 32-bit integer arithmetic, and on startup asserts that the values in the input are not too large. We must at least guarantee that the length of any tour fits in a signed 32-bit integer. We achieve this with the following (overly strict) requirement:

$$\forall_{i,j \in V} : \; n \cdot |c'_{ij}| \leq 2^{31} - 1.$$

In practice, we will only evaluate test cases with mandatory edges through LKH with the more strict requirements $n \leq 100$ and $0 \leq c_{ij} < 10^4\sqrt{2}$, which guarantees $\forall_{i,j \in V} : \ n \cdot |c'_{ij}| \leq 100 \cdot \left(100 \cdot 10^4\sqrt{2} + 1\right) \leq 2^{31} - 1$. With these constraints, the startup assertions of Linkern never failed.

5.4 Local search using adapted beam search

In this section, we define a local search method by subgraph optimization that applies beam search, adapted to TSP with mandatory edges, over a probability distribution over the edges. This probability distribution is computed by an ML model. The resulting local search is illustrated in Figure 5.3.

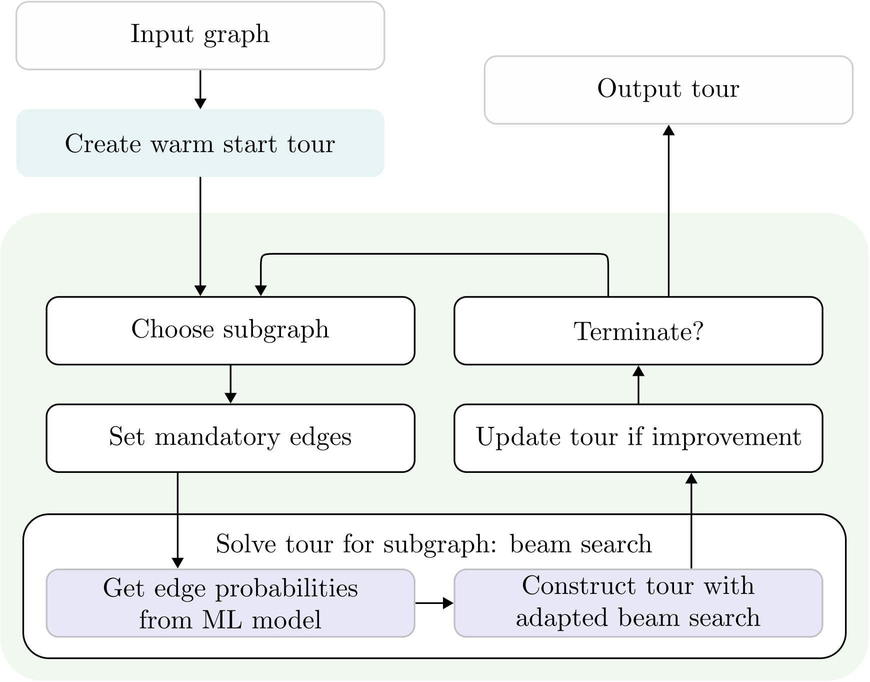

Input graph
Create warm start tour
Output tour
Choose subgraph
Terminate?
Set mandatory edges
Update tour if improvement
Solve tour for subgraph: beam search
Get edge probabilities from ML model
Construct tour with adapted beam search

Figure 5.3: An overview of the modified metaheuristic from Figure 5.2. The tour for a subgraph with mandatory edges is computed by applying adapted beam search over the output of the ML model. The used model need not be aware of the mandatory edges.

5.4.1 Metaheuristic

To optimize a subgraph with mandatory edges, we input the subgraph to an inner ML model that computes a probability distribution. Rather than use a probability distribution directly, we can also use a model that returns a value in $\mathbb{R}$ for each edge, and calculate the probabilities using softmax.

The ML model does not need to be aware of the mandatory edges: the adaptation to beam search that we will introduce guarantees that the mandatory edges will be included in the tour. However, a model that is aware of the mandatory edges can give an output that is more suited to those mandatory edges, knowing that they will be in the tour.

In each local search move, we select a subgraph of size k. We wish to make sure we have a subgraph that is fairly local in terms of distances, and has a number of tour edges included in it. For this purpose, we choose a subgraph by adding nodes together with their successor in the tour, and adding those nodes that are nearest first:

- Choose a random node in the tour $\pi(i)$
- Create a set of subgraph nodes $S = \{\pi(i)\}$
- Repeat the following until $|S = k|$:
    - Add $\pi(i+1)$ to S
    - Optionally (according to hyperparameter $b_{\leftarrow} \in \{0, 1\}$) add $\pi(i-1)$ to S
    - Out of all neighbors of any node in S that are themselves not in S, choose the one with the lowest minimum distance to the nodes in S, and store it as $\pi(i)$

Then, for every chain of edges in the tour that lies entirely outside of the subgraph, we represent it using a mandatory edge between the endpoints that mark the boundary of the subgraph for the tour. That way, the tour in the subgraph is the same as the tour in the graph, albeit with parts of the tour that go outside the subgraph represented as just a single mandatory edge. This means we can use TSP with mandatory edges to optimize this subgraph, and the result can always be turned into a valid tour of the graph as a whole, since we will not have modified the mandatory edges.

After we have selected the subgraph, we use it as input to run an ML model that returns a probability distribution over the edges. We then use an adapted version of beam search to construct a tour according to that probability distribution. If the length of the tour on the subgraph is smaller than it was before, we update the tour on the graph as a whole. If it is not, we do not perform the move and try another move (starting with selecting a new subgraph).

At some point, we must terminate. We choose to terminate after t_X consecutive failures to find a tour improvement. This indicates it becomes harder to find improvements and we are getting closer to the limit of how many improvements we could still find using the currently used ML model. We also add a hyperparameter Δ_X: we will count a move towards the t_X limit if it has either no improvement, or the ratio of the difference in tour length is less than Δ_X:

$$t\text{'th local search move counts as failure if } \frac{l(\pi_{t-1}) - l(\pi_t)}{l(\pi_{t-1})} < \Delta_X$$

This allowed for better control of the expected running time. Note that if a move leads to a shorter tour but with a difference ratio less than Δ_X, we still apply it.

5.4.2 Adapted beam search

We adapt beam search to fulfill two conditions:

- Knowing that we require k steps to form a tour (each tour on a subgraph of size k has length k), evaluating each candidate next node at each step for each beam results in evaluating a candidate next node $k^2 \cdot w$ times (k from tour length, k from number of neighbors to consider, w from number of beams), which in the running time will be further multiplied by overhead from access to data structures. Since the number of local search iterations may be large, and each iteration we need to perform subgraph optimization over all up to $\frac{n}{k}$ subgraphs, we realize we must optimize beam search for our purposes.
- The beam search must return a tour that contains all mandatory edges.

We will now list the relevant modifications to beam search that let us achieve this.

- The initial node must not have 2 incident mandatory edges (or we may later end up closing the tour with less than k nodes).
- If there is a candidate next node to which there is a mandatory edge from the current node, we always choose it (without evaluating other candidate next nodes).

- While evaluating candidate next nodes for all beams, we maintain a sorted data structure that supports queries and updates in logarithmic time [75] of maximum size w, containing the beams for the next step sorted by aggregate value. Each time a beam candidate is considered, we do not add it if it is worse than the worst beam B'. Otherwise, we add the beam candidate and trim the last element of B' if the size exceeds w.
- For each node i, we precompute a sorted list of all nodes j sorted in descending order of probability, and evaluate candidate next nodes j for a beam with last node i in that order. In the case of using softmax on inverse values, this means we sort in ascending order of value.
- When the aggregate value of the beam, which is the product of the probabilities of its chosen edges, or the sum of their logits, resulting from a candidate next node is worse than that of the worst beam in B', we stop evaluating candidate next nodes for the current beam and continue to the next beam.
- For each beam b, we store a persistent balanced binary search tree [76] (a balanced binary search tree where each update does not modify the original structure, but efficiently creates a new balanced binary search tree with the update applied) with k boolean elements indicating for each node i whether i is in the beam already. Because it is persistent, it is fast to construct new instances for each new beam. We only construct new instances at the end of each beam search step, when we know exactly which beams will be kept for the next step.
- For each beam b, we store a persistent stack [77] (a stack where each new element addition does not modify the original structure, but efficiently creates a new stack with the element added) indicating the sequence of nodes in the beam in order. This similarly allows us to construct new beams rapidly.

Limiting bias in aggregate values of beams

Because we must always choose a candidate next node if there is a mandatory edge to there from the current node, the aggregate value of the beam may become biased. If we interpret the values L as probabilities P using softmax, we have a probability of 1 of choosing a mandatory edge. Since we may use probability logits instead of probabilities, and logits for probabilities of 0 and 1 are not finite, one solution to this is inserting a logit for mandatory edges that gives a probability of 0.999% (and for other edges incident to nodes that have incident mandatory edges a probability of 0.001%). However, this introduces the bias that any beam that contains more mandatory edges than another will have a higher aggregate probability, even though eventually all beams of length n will have the same aggregate value contribution from mandatory edges.

Therefore, we modify beam search so that we do not take the aggregate value as the sum of all values L. Instead, we use the mean of all values L for non-mandatory edges (which are those that are actually used in the beam search's decision making).

5.5 Local search using ML 2-opt

The paper Learning 2-opt Heuristics for the Traveling Salesman Problem via Deep Reinforcement Learning by De Oliveira da Costa et al. [9], which we will refer to as *Learning 2-opt* for brevity, has shown good results compared to some other methods, as shown in Table 5.1.

Their paper does not use a constructive approach to decode a tour, but rather guides local search to iteratively improve upon an initial solution. In every iteration, the graph and tour are used by the model to choose two nodes. A 2-opt move defined by those 2 nodes (defined as swapping the nodes' positions in the tour sequence and mirroring the sequence of all nodes between them) is then applied. The model is based on reinforcement learning, which is used to train a model that can make 2-opt moves that are not necessarily an immediate improvement, but allow for improvements down the line. Amongst others, their model uses graph convolution to produce node embeddings, and uses a recurrent neural network to process the embeddings of nodes in the order that they appear in the tour. The local search is ran for a predetermined, but configurable, number of iterations. The authors generally use 2000 steps and primarily evaluate on graph sizes 20, 50 and 100.

Method	$n = 20$		$n = 50$		$n = 100$	
	Opt. gap (%)	Time	Opt. gap (%)	Time	Opt. gap (%)	Time
Concorde [8]	0.00	1m	0.00	2m	0.00	3m
GCN [63]	0.01	12m	**0.01**	18m	1.39	40m
PtrNet [60]			0.95		3.03	
GAT {1280} [62]	0.08	5m	0.52	24m	2.26	1h
GAT-T {5000} [66]	0.00	1h	0.20	1h	1.42	2h
Learning 2-opt	**0.00**	15m	0.12	29m	**0.87**	41m

Table 5.1: A comparison of the performance and time cost of Learning 2-opt and several other methods on 10,000 instances of TSP with n = 20, 50, 100 (results taken from original paper). The optimality gaps and running times of their method are competitive.

Notable is that the approach taken by the paper is competitive without needing a decoding of the tour as a postprocessing step. In fact, this alone would be reason to suspect it may perform better than the alternatives, when adapted to be used with adapted beam search. However, in this case, we can do more than merely call on the unadapted model with pre- and postprocessing: we propose an adaptation of ML 2-opt specifically to be used as a parallelized subgraph optimization method. This can serve as an example of the adaptation of an ML model to be usable on larger graph sizes with the proposed local search metaheuristic.

In this section, we will adapt the model, that we will refer to as *ML 2-opt*, to a method for subgraph optimization in local search. The resulting local search is illustrated in Figure 5.4.

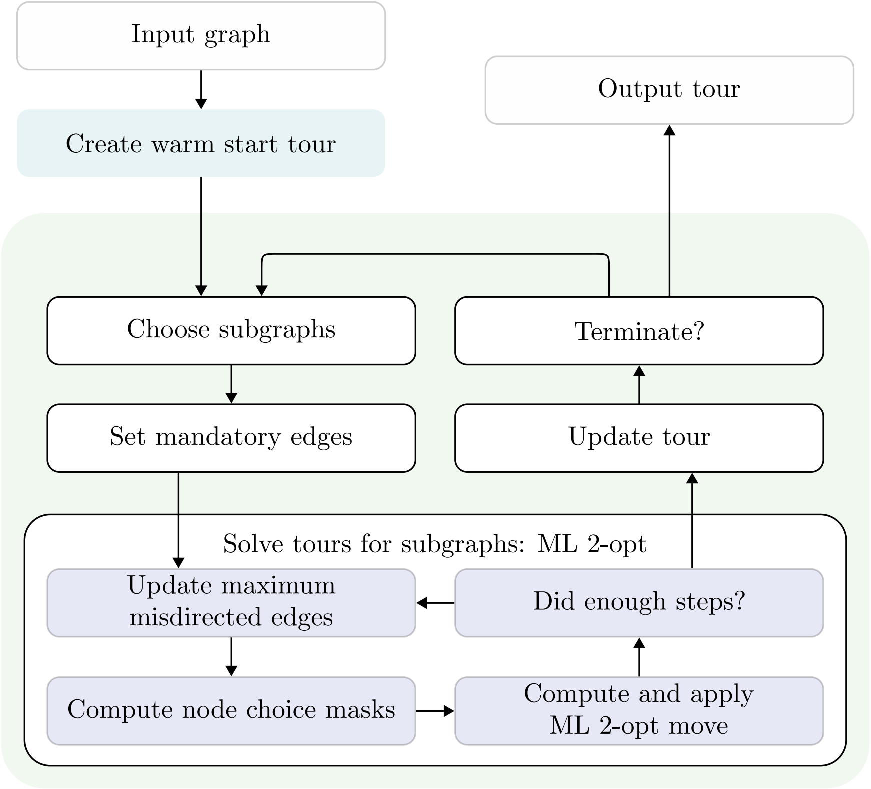

Input graph
Output tour
Create warm start tour
Choose subgraphs
Terminate?
Set mandatory edges
Update tour
Solve tours for subgraphs: ML 2-opt
Update maximum misdirected edges
Did enough steps?
Compute node choice masks
Compute and apply ML 2-opt move

Figure 5.4: An overview of the modified metaheuristic from Figure 5.2. Instead of one subgraph, we choose multiple subgraphs. The tours on these subgraphs are iteratively optimized by ML 2-opt, adapted to conform to directed mandatory edges.

Unlike local search using adapted beam search on subgraphs (as described in Section 5.4.1), we parallelize the optimization of subgraphs. We implemented beam search to run on the CPU, which would only allow for limited parallelization. However, we now get a tour straight out of the ML model, which does most heavy computation on the GPU. It is easy to parallelize in PyTorch by using a larger batch size, and even to parallelize over arbitrary devices using the `DataParallel` module. We found that running the model over a batch size of 1000 takes around 10 times longer than running it over a batch size of just 1, which signifies a great improvement in time cost, and thereby allowing ML 2-opt to provide, apart from potential performance increase, a great improvement in practical running time on typical hardware.

We select multiple disjoint subgraphs, each of size k, set mandatory edges in them to represent outer edge chains, and optimize the tours on them in parallel. Then, we apply the found changes to the graph as whole.

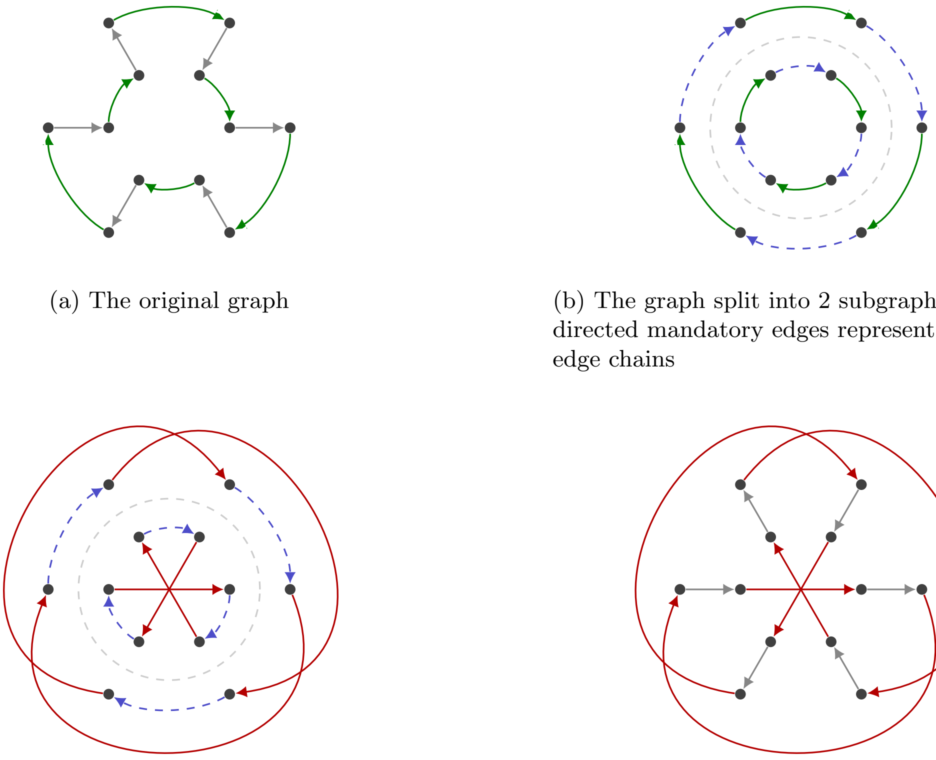
Several changes are made to the subgraph selection described in Section 5.4.1 to ensure they are disjoint yet reasonable:

- The sparsity of available nodes will increase with the selection of more subgraphs. This means the nodes in the next subgraphs will be less close to each other and therefore the subgraphs contain on average more mandatory edges. To avoid having large parts of the subgraph being inoperable for the TSP with mandatory edges solver, we define a maximum ratio of mandatory edges $\rho_m \in [0, 1]$ so that $|E[S]_m| \leq \rho_m \cdot |E[S]|$ (if the subgraph we find has a greater ratio of mandatory edges we count the attempt as a failure and try to select a subgraph again).

- We repeat the subgraph selection until either there are less than k nodes left in the graph that are not in any subgraph, or we fail choosing a subgraph s_X consecutive times.

- The next choice of $\pi(i)$ is only chosen out of nodes that are not already in another subgraph

Optimizing the subgraph tours in parallel poses a problem: the mandatory edges in each subgraph representing the outer edge chains are supposed to ensure that the tour in the whole graph stays whole after it is updated. However, these outer edge chains may no longer be the same after multiple subgraph tour updates are applied to the whole graph. Then, the tour can end up being broken into multiple disjoint cycles. A case of this happening is illustrated in Figure 5.5.



(a) The original graph

(b) The graph split into 2 subgraphs, with directed mandatory edges representing outer edge chains

(c) Each subgraph produces a new tour that conforms to the directed mandatory edges

(d) The new graph after the tour is updated according to the new subgraph tours

Figure 5.5: A move constructed from two subgraph moves that causes the tour to break into multiple disjoint cycles.

We thus face the issue of, in order to optimize the subgraph tours in parallel, finding a way that both:

- can be adhered to by the subgraph optimization method,
- and prevents conflicts between different subgraphs that cause the tour as a whole to become disjoint.

We found that using asymmetric TSP with mandatory edges, even though the overarching problem in the evaluated test cases is symmetric, is a very good solution. This means, for each mandatory edge in the subgraph, not only must it appear in the final tour: additionally, the direction of the tour as a whole must be kept intact. That is, if we list the nodes in the tour in order, each pair of neighboring nodes that represents a mandatory edge must still be in the same order (or, they must all be in the reverse order). We found a subgraph optimization method that is able to use ML 2-opt and yet adheres to the requirement set by asymmetric TSP with mandatory edges. This requirement does not fully fulfill the desired property of preventing all conflicts, but the conflicts that do not happen are so rare that they can be ignored without practical impact. We later evaluate this method over a graph with $n = 10^5$, and found that over 31068 subgraphs for which a better subgraph tour was found, only 11 resulting moves could not be applied due to a conflict.

To adapt ML 2-opt to adhere to asymmetric mandatory edges, in each step in the ML model, we apply a mask to the possible choices of first and second node that ML 2-opt can make. ML 2-opt already does this: it requires that, for the choices $1 \le i, j \le k$ (with k the number of nodes in the subgraph we are optimizing) of first and second node, we have $i < j$. This means that a mask is used for i to ensure $i \neq k$, and a mask is then used for j to ensure $j > i$. We modify this mask to zero out additional options that we must prevent from being taken.

We define *forward mandatory edges* $E_{\rightarrow} \subseteq E[S]$ in the subgraph tour as mandatory edges that, if an arbitrary direction is assigned to the tour, have the same direction as they had in the input to the model. We define *backward mandatory edges* $E_{\leftarrow} \subseteq E[S]$ as the mandatory edges that, with the same tour direction, now have a different direction as they had in the input to the model. We can calculate the number of *misdirected edges* $|E_D| = \min(|E_{\rightarrow}|, |E_{\leftarrow}|)$. For the condition of respecting all mandatory edges to hold, we require that $|E_D| = 0$.

To this end, we define for each 2-opt move that there at most m_D misdirected edges after the move. If the current number of misdirected edges $|E_D| \le m_D + 1$, we can always make a 2-opt move if $\lfloor \rho_m \cdot |E[S]| \rfloor \le \left\lfloor \frac{1}{3}|E[S]| - 1 \right\rfloor$: in this case, there is at least one occurrence of three consecutive non-mandatory edges in the tour. This means we can invert a non-mandatory edge, thereby keeping the number of misdirected edges the same. Additionally, we can invert a mandatory edge with the tour direction that is in the minority, thereby lowering the number of misdirected edges by one. Therefore, we start with $m_D := \left\lfloor \frac{|E[S]|}{2} \right\rfloor$, and then decrease m_D by 1 until eventually $m_D = 0$.

To be precise, we run ML 2-opt with t_L 2-opt steps in total, and for each value of m_D from 0 to $\left\lfloor \frac{|E[S]|}{2} \right\rfloor$ we require t_D steps to have at most that many misdirected edges, starting from the end (so the last t_D steps require 0 misdirected edges).

However, with any number of misdirected edges, we can always choose to invert any specific one edge (changing the number of misdirected edges by a most one). However, if the number of allowed misdirected edges is precisely 0, we can only invert any non-mandatory edge. To prevent these last steps from having very limited improvement, we do something different in the last t_D steps: instead of always requiring 0 misdirected edges, we only require 0 misdirected edges every t_e'th step, with an example choice being each 4th step, giving 3 steps in between where $m_D = 1$. The resulting value of m_D over time is illustrated in Figure 5.6.

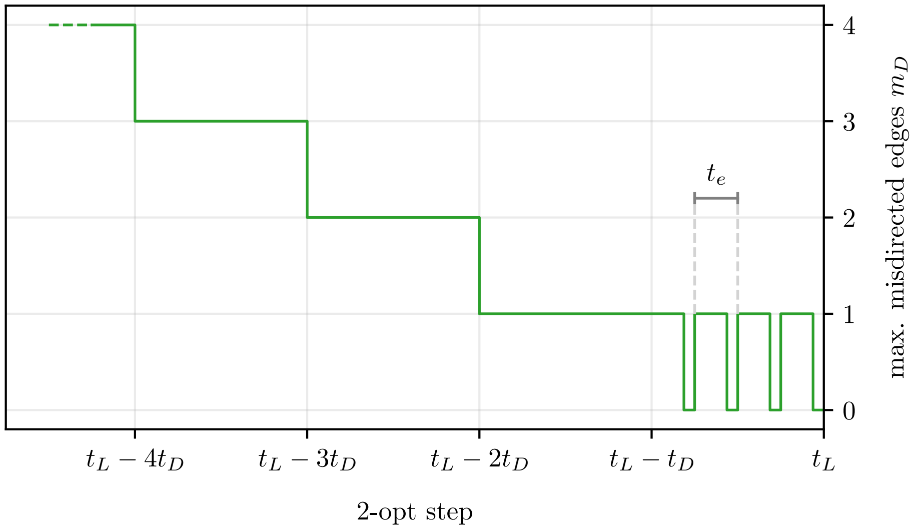


Figure 5.6: The number of misdirected edges m_D towards the end of the ML 2-opt model execution: each t_D 2-opt steps, m_D is decreased by 1, until the last t_D steps where m_D is 0 once every t_e steps, and 1 otherwise.

Having to think about whether choices of these hyperparameters are valid is hard. Therefore, replace t_D by a more easily configurable hyperparameter $\rho_D \in [0,1]$ as the ratio of 2-opt steps that has a maximum number of misdirected edges. Then, we can safely set t_D using

$$t_D = t_e \cdot \left\lfloor \frac{\rho_D \cdot \left\lfloor \frac{t_L}{t_e} \right\rfloor}{\left\lfloor \frac{|E[S]|}{2} \right\rfloor} \right\rfloor$$

and only worry about configuring ρ_D (the formula for t_D is not particularly important, but since we tune the hyperparameter ρ_D later we show its use for completeness).

The final tour outputted is the one with minimal length among all tours resulting from a step that had 0 misdirected edges (whether 0 misdirected edges was required in that step or not).

In the code, we compute the masks for the 2-opt node choices completely using just vectorized operations that are performed on the GPU. This algorithm is not trivial, but too long and specific to detail here.

We can prove by counterexample, shown earlier in Figure 5.5, that this stronger condition of using asymmetric TSP with mandatory edges does not guarantee the resulting tour (after multiple updates from subgraphs) not to break.

A naive, but working, approach to these failures is to simply attempt to apply subgraph tour updates one by one, and when it would cause the tour in its current state to break, ignore it and move on the next subgraph. However, we introduce a better approach, called *tree composition* of moves.

This approach additionally addresses the problem that it costs at least $O(n)$ time to check whether the update would break the tour. This is true regardless of the number of actual edges (of the order $O(k)$, with k the size of the respective subgraph from which the move was derived) being added/removed from the tour. Therefore, we use a tree composition of moves, where we, in a tree-like fashion, apply the updates by combining a set of moves and split it if it fails:

- Given a set of s moves (initially the moves derived from each subgraph) $X_1, \ldots, X_s, Y_1, \ldots, Y_s$ we create one move X, Y equivalent to applying all the individual moves.

- If X, Y would not break the tour into multiple cycles (this check costs $O(n)$ time), apply it, and stop.
- Otherwise, split the moves into two subsets of size $\lfloor \frac{s}{2} \rfloor$ and $s - \lfloor \frac{s}{2} \rfloor$, and apply this procedure to both subsets.

This runs in $O(n \log(n)^d)$ time (with negligible low constant d), and with very few tour-breaking swaps, will tend towards the best-case running time of only $O(n+s)$ rather than $O(n \cdot s)$ for the naive approach. Additionally, to make sure we apply as many changes as possible: if running the procedure does not apply all moves, we can run it again with the leftover moves to see if more moves are applied. To prevent this causing a high running time, with s_0 the number of moves originally found at this step, we simply repeat the procedure a maximum of $\lceil 2 \cdot \log_2(s_0) + 2 \rceil$ times. In practice, this eliminates most of the remaining times a move could not be applied.

In summary, dealing with these tour-breaking swaps caused by parallel processing of subgraphs is highly worth it, because

- The asymmetric mandatory edges are very good at expressing the necessary condition for the subgraphs to conform to: the number of moves that would break the tour are rare. And even then, we can very efficiently attempt to apply them in a way that ultimately skips even fewer.
- We can enforce the condition of asymmetric mandatory edges on ML 2-opt by setting a maximum number of misdirected edges in each step, which can be computed in an efficient and parallelized manner.

Chapter 6

Experimental evaluation

In this chapter, we experimentally evaluate the proposed metaheuristics. First we will provide details of the experimental setup: the software and hardware environment, the used datasets, the preprocessing applied to the input and the list of algorithms we evaluate. After this, an analysis of hyperparameters follows, to determine potential good values to use in general, and to gain insight into their effect: most importantly on the performance-time tradeoff. Finally, we present an evaluation of the metaheuristics on 2D-TSP instances in TSPLIB [10], uniformly generated test cases of various sizes, and a large uniformly generated case with $n = 10^5$.

6.1 Experimental setup

6.1.1 Environment

Software	
Operating system	Windows 10 Home 20H2, build 19042.1083
Python	3.7.6
CUDA	11.1
PyTorch	1.8.1+cu111

Hardware	
GPU	NVIDIA GeForce RTX 2060
CPU	Intel Core i7-9750H @ 2.60GHz
RAM	16.0 GB
SSD	Samsung MZVLB512HBJQ

6.1.2 Datasets

We use random generated sample inputs during development, hyperparameter evaluation and performance evaluation. Finally, we run the algorithm on 2D-TSP instances in the test case library TSPLIB [10].

Randomly generated samples

In each experiment, we generate 2D-TSP samples for a given constant number of nodes n.

All node coordinates are in the range $[0, 1)$. Furthermore, to be able to compare results with the Concorde solver, which limits the maximum coordinate value to slightly above 10^4 to prevent integer overflow, the coordinates have a maximum precision of 10^{-4} (so that for all nodes i with coordinates u_i we have $10^4 u_i \in \mathbb{Z}^2$).

TSPLIB

TSPLIB [10, 78] forms a collection of test cases for several variations of TSP and related combinatorial optimization problems. Specifically, it contains instances for symmetric TSP, of which many are 2D-TSP. All numbers that are used to specify an instance, which may be edge distances or node coordinates, are integral. For all these instances, an optimal solution is known. The 2D-TSP problems are, amongst others, derived from the locations of real world cities. TSPLIB is a commonly used benchmark for evaluating solutions to TSP.

The coordinates of the points are scaled (keeping aspect ratio) so that for any node i with coordinates u_i we have $u_i \in [0, 1]^2$, and the minimum occurring coordinate (either an x or y values) is 0 and the maximum occurring coordinate (either an x or y values) is 1.

6.1.3 Candidate edge graph and initial tour

For each test case, we create an incomplete graph, the *candidate edge graph*, to save on resources. Local search requires that an existing initial tour, the *warm start tour* is provided, so we also generate such a tour for each test case.

Candidate edge graph

From each given test case with n nodes and 2D node coordinates u_i for $1 \leq i \leq n$, the complete graph has $\frac{n(n-1)}{2}$ undirected edges, and each node has $n - 1$ neighbors. Since many operations run over all edges, or over all neighbors of a node, filtering these edges to create a *candidate edge graph* saves on running time.

Most edges in the complete graph can already be assumed not to be in the optimal tour: Joshi et al. [63] found that k-nearest neighbors (KNN) with $k = 20$ gives a candidate edge graph that reasonably contains most edges in the optimal tour, on the 2D-TSP solution space for $n \leq 100$ with all coordinates within the unit square. In a later paper, Joshi et al. [54] found that using KNN with $k = 0.2n$ (a sparsification ratio of 20%) leads to better performance than constant k. The comparison was only evaluated on $n \leq 200$ and causes $|E|$ to be of the order $O(n^2)$ so may be less applicable advice for larger n. Since we wish to potentially run on large $n \geq 10^4$, we opt for using KNN with constant k, so we can find a candidate edge graph with $|E| = O(n)$.

For each given test case we generate some candidate edge graph $G = (V, E)$ defined by the candidate edge set E. We use a procedure based on KNN to construct a candidate edge graph, parametrized by d_{KNN}, k_{KNN} and q_{KNN}, where d_{KNN} is the search distance for neighbors in an initial graph, k_{KNN} is the number of nearest neighbors and q_{KNN} is the number of nearest neighbors per quadrant:

1. Take E_D as the edges in the Delaunay triangulation of the nodes in $O(n \log(n))$ time
2. For each node i, for some parameter d_{KNN}, take the open d_{KNN}-distance neighborhood $N'(i)$, i.e. all other nodes at most d_{KNN} edges in E_D away from i
3. For each node i, compute the set $N(i)$, containing the k_{KNN} nearest neighbors (by Euclidean distance), and the q_{KNN} nearest neighbors per quadrant (so up to $4 \cdot q_{\text{KNN}}$ nodes in total) from $N'(i)$
4. Return the set E of edges either in E_D or in the neighborhood relation defined by N (so that $E = E_D \cup \{(i, j) \mid i \in V, j \in N(i)\}$

An example of a resulting candidate edge graph is shown in Figure 6.1.

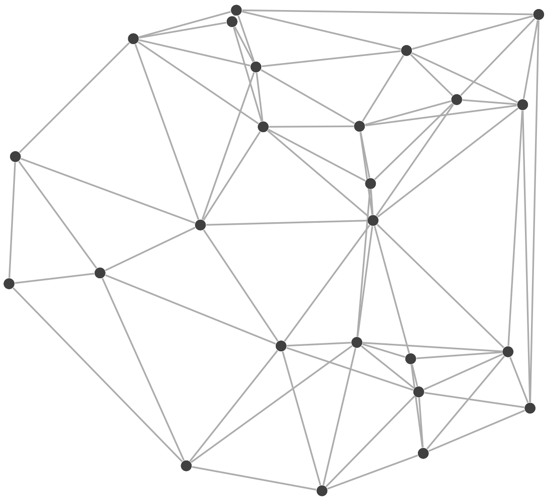


Figure 6.1: A candidate edge graph with $n = 25$, $d_{\mathrm{KNN}} = 3$, $k_{\mathrm{KNN}} = 4$, $q_{\mathrm{KNN}} = 1$.

Initial tour

For the procedure we use to generate the initial tour, we consider the following:

- The procedure must be quick, roughly of the order $O(n \log(n))$, to avoid becoming a bottleneck
- The tour need not be short per se, but shorter is better, to save on the number of needed local search moves (and thus running time) used in each experimental run
- The tour must be improvable upon by local search, i.e. there is a path of local search moves from the tour to a good solution

The Christofides–Serdyukov algorithm [6], an approximation algorithm for TSP with an approximation ratio of 1.5, is used to create a warm start tour in some research, but mostly for small n and complete graphs, because it has running time complexity $O(n^3)$.

Greedy search (where we construct a tour by adding the closest next node) is an improvement over a random tour, but may still lead to a number of very long edges to reach nodes that were forgotten: some edges will suddenly travel far across the graph to reach a node not in the tour yet, as can be seen in Figure!6.2. This creates a problem for the improvability of the tour: for such long edges, it is more unlikely there are nodes that are the neighbor of both endpoints of the edge in the candidate edge graph (because the candidate edge graph is based on distance). This means it is less likely that an explicitly bounded local search will have a suitable replacement edge or edges within their scope that can allow this long edge to be removed.

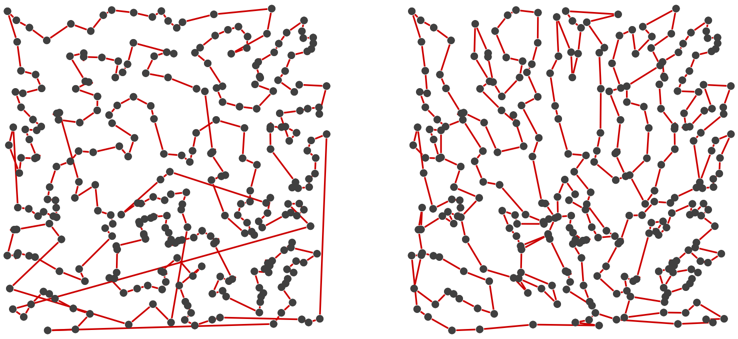


Figure 6.2: A tour constructed by greedy search (left) compared to our version (right).

We will use greedy search with two modifications that aim to result in good improvability of the warm start tour.

The first modification is that, to prevent creating long edges that are caused by exhausting the neighbors around, we force the search to move outward without leaving any unadded nodes behind. This can be observed in Figure 6.2. As a greedy choice, rather than choose the node closest to the last added node, we use the following criterion: given we have already constructed the partial tour τ (a sequence of nodes), choose the next node i^* with coordinates u_{i^*} as the node with the net highest distance to the initial node yet lowest distance to the average node:

$$i^* = \text{argmax}_{i \in V \setminus \tau} \left\| u_i - u_{\tau(1)} \right\|_2 - \left\| u_i - \frac{1}{|\tau|} \sum_{j \in \tau} u_j \right\|_2$$

We apply this to 46 graphs with sizes evenly in the range $[50, 500]$, and the Delaunay triangulation as edges, and observe, as shown in Table 6.1, that the standard deviation in edge lengths is lower than that of a regular greedy search.

	Opt. gap (%)	Tour edge lengths	Time (s)
OPT (Concorde [15])	0.00 ± 0.00	0.04388 ± 0.02893	10.32838 ± 13.85636
Greedy	28.08 ± 5.82	0.05647 ± **0.08063**	0.01569 ± 0.01074
Ours	43.90 ± 9.03	0.06365 ± **0.06725**	0.02130 ± 0.01086

Table 6.1: The tour optimality gaps, tour edge lengths, and construction times of the optimal solution and the two compared warm start tour methods. The small number of very long edges appears in the greedy tour, but not in ours.

The second modification is that for any edge we attempt to ensure k_t mutual neighbors between the endpoints, with those neighbors closer to each endpoint than the length of the edge. This is illustrated in Figure 6.3. In other words: we wish to make sure that for each edge, there exist some nodes that have edges to the endpoints that form a triangle conforming to the triangle inequality, with the original edge being the longest. This ensures that for any comparatively longer edge, there are several pairs of edges in the candidate edge graph that could replace the longer edge, meaning the long edge can be resolved using a local search move that can be found by an explicitly bounded heuristic. This increases the improvability of the tour. If for some edge, there are more than k_{ti} such potential mutual neighbors, we choose the ones with the lowest minimum distance to any of the endpoints. Otherwise, we simply choose all of them.

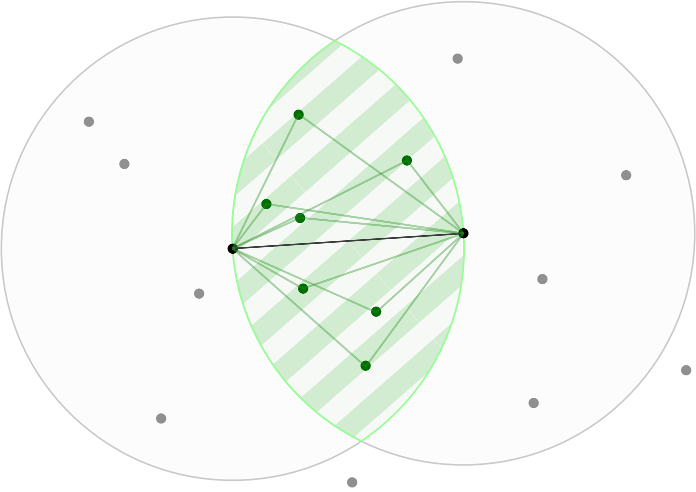


Figure 6.3: The area for a long edge in which we search for mutual neighbors to add edges to, until we have at least k_t such mutual neighbors or run out of nodes to add edges to. The potential edges between the long edge endpoints and the mutual neighbors are shown as dark green edges.

This requirement will be actively renewed after an edge is added to the graph later because the edge appears in a subgraph output tour but not in the whole graph set of edges E.

6.1.4 Evaluated algorithms

To compare our metaheuristic, we also evaluate various other subgraph optimization procedures, namely:

- *2-opt with mandatory edges*
  2-opt constrained only on subgraphs forms an interesting type of local search, and can be compared to 2-opt over the graph as a whole to estimate the quality of the subgraph selection.
- *LKH with mandatory edges*
  Since LKH [24] returns near-optimal solutions, using LKH to optimize subgraphs constitutes an indicative bound on the best performance we can possibly achieve for any given subgraph selection.

Lastly, for some experiments we also run the following algorithms over the graph in its entirety:

- *Concorde*
  Concorde [8, 15] provides an optimal solution (but since, unlike any other algorithm we evaluate, it has exponential time complexity w.r.t. n, and above some n its running time will exceed that of the other algorithms; we found it most reasonable for our purposes to evaluate Concorde only on $n \leq 1000$).
- *LKH*
  LKH [24, 15] provides a near-optimal solution in very minimal time, and can be used a replacement for Concorde in comparisons, when running the same number of Concorde evaluations would be too costly in time.
- *2-opt*
  2-opt is a baseline for local search.
- *OR-Tools*
  OR-Tools [11] is a popular library for combinatorial optimization, by Google. It has the ability to solve TSP.

We evaluate three trained instances of ML 2-opt trained on TSP instances of sizes $m = 20, 50, 100$, specifically the following 3 models [79]:

- $m = 20$: `policy-TSP20-epoch-189`
- $m = 50$: `policy-TSP50-epoch-268`
- $m = 100$: `policy-TSP100-epoch-262`

We do not do hyperparameter optimization specifically on the metaheuristic using adapted beam search. During preliminary experiments, we found we could not match the performance of ML 2-opt. This is not surprising, considering ML 2-opt already performs better on regular TSP than the other models we tried with adapted beam search. Except the choice of beam width w, for which there is already a significant body of research in general, all hyperparameters for adapted beam search are included in the following hyperparameter analysis with ML 2-opt. The following are subgraph optimization procedures we compared against during preliminary experiments that are interesting to note for their purpose in analyzing adapted beam search metaheuristics:

- *Adapted beam search on edge lengths*
  To get an indication of the quality of the edge values produced, we can compare against a beam search that simply uses edge lengths, $L(i, j) = c_{ij}$.
- *Adapted beam search on Concorde tour*
  We can optimize a subgraph with Concorde [8], then run beam search on the edges with probability 1 if they are in the tour returned by Concorde, and 0 otherwise. This mean the result will contain all mandatory edges, and otherwise many edges from the optimal tour. This allows us to see the performance of using adapted beam search on a solution that we know is optimal for regular TSP, but unaware of mandatory edges. Achieving the optimal solution for regular TSP is the goal that an ML model unaware of mandatory edges is trained towards, and therefore this allows us to see specifically the potential loss in performance caused by the model being unaware of mandatory edges.

6.2 Hyperparameter analysis

To determine the effect of the hyperparameters on the performance and running time, we perform various experiments in which we vary these hyperparameters. Since it is very difficult to estimate the significance of some individual hyperparameters intuitively, this mainly serves the purpose of gaining a better underlying understanding of the effects of the respective design choices.

We will also examine which choice of hyperparameters leads to the best performance or running time. This is useful both to speed up the experiments with a roughly known estimate of the performance-time tradeoff, and to get good performance on evaluation data. Note that we do not aim to find an optimal hyperparameter configuration.

List of hyperparameters

Per experiment, we will list the number of test cases, number of input graph nodes, and the relevant hyperparameters:

	Input
T	Number of test cases
n	Number of nodes
	Candidate edge graph and initial tour
d_{KNN}	KNN search distance
k_{KNN}	KNN neighborhood size
q_{KNN}	KNN neighborhood size per quadrant
k_t	Minimum closer mutual neighbors between edge endpoints
	Subgraph
k	Subgraph size
s_X	Maximum subgraph selection attempt consecutive failures
ρ_m	Maximum ratio of mandatory edges
$b_{\leftarrow}$	Whether to also add every node's tour predecessor
	Local search
t_X	Maximum consecutive unsuccessful moves
Δ_X	Minimum improvement ratio
	ML 2-opt
m	The ML 2-opt trained model used (20, 50 or 100)
t_L	Number of ML 2-opt model steps (individual 2-opt steps within the subgraph optimization)
ρ_D	Ratio of ML 2-opt model steps that have an enforced number of misdirected edges m_D
t_e	Interval of ML 2-opt model steps containing 1 step with $m_D = 0$

Presentation of results

Each experiment will be run on T test cases (different for each experiment). For each configuration, for any optimality gap or elapsed time, we have mean μ and standard deviation σ across all test cases. In the respective table, each result will be listed as either μ or $\mu \pm \sigma$. For each time, we will emphasize in bold the best values of the ML-based model. Graphs indicate the mean optimality gap over all test cases, and curves extend until the latest termination across all test cases. Dashed lines extend after the latest test case termination, indicating the mean optimality gap at termination. Boxplots will be centered around the median, as is standard, and outliers are not shown.

6.2.1 Subgraph size k, ML 2-opt model size m

We would like to know what the effect of the subgraph size k is. The subgraph size determines the scope to which the resulting local search metaheuristic will be bounded. For smaller subgraph sizes it will be easier to make high-quality inferences within the scope of the subgraph, but for larger subgraph sizes the expressiveness of the output with regards to moves on the overarching graph is greater, thereby possibly finding moves that are not found by smaller subgraphs.

Additionally, some models may perform better on specific subgraph sizes: especially when an ML model has been trained on a specific size. Therefore for the k evaluated, we try all ML 2-opt model sizes.

The observed optimality gaps over elapsed wall-clock time are shown in Figure 6.4, the optimality gaps at termination are shown in Figure 6.5, and a detailed overview of the results is in Table 6.2.

Hyperparameters

$T = 30$	$d_{\text{KNN}} = 3$	$k = \dots$	$t_X = 3$	$m = \dots$
$n = 500$	$k_{\text{KNN}} = 30$	$s_X = 5$	$\Delta_X = 0.01\%$	$t_L = 175$
	$q_{\text{KNN}} = 4$	$\rho_m = 15\%$		$\rho_D = 99\%$
	$k_t = 4$	$b_{\leftarrow} = 0$		$t_e = 2$

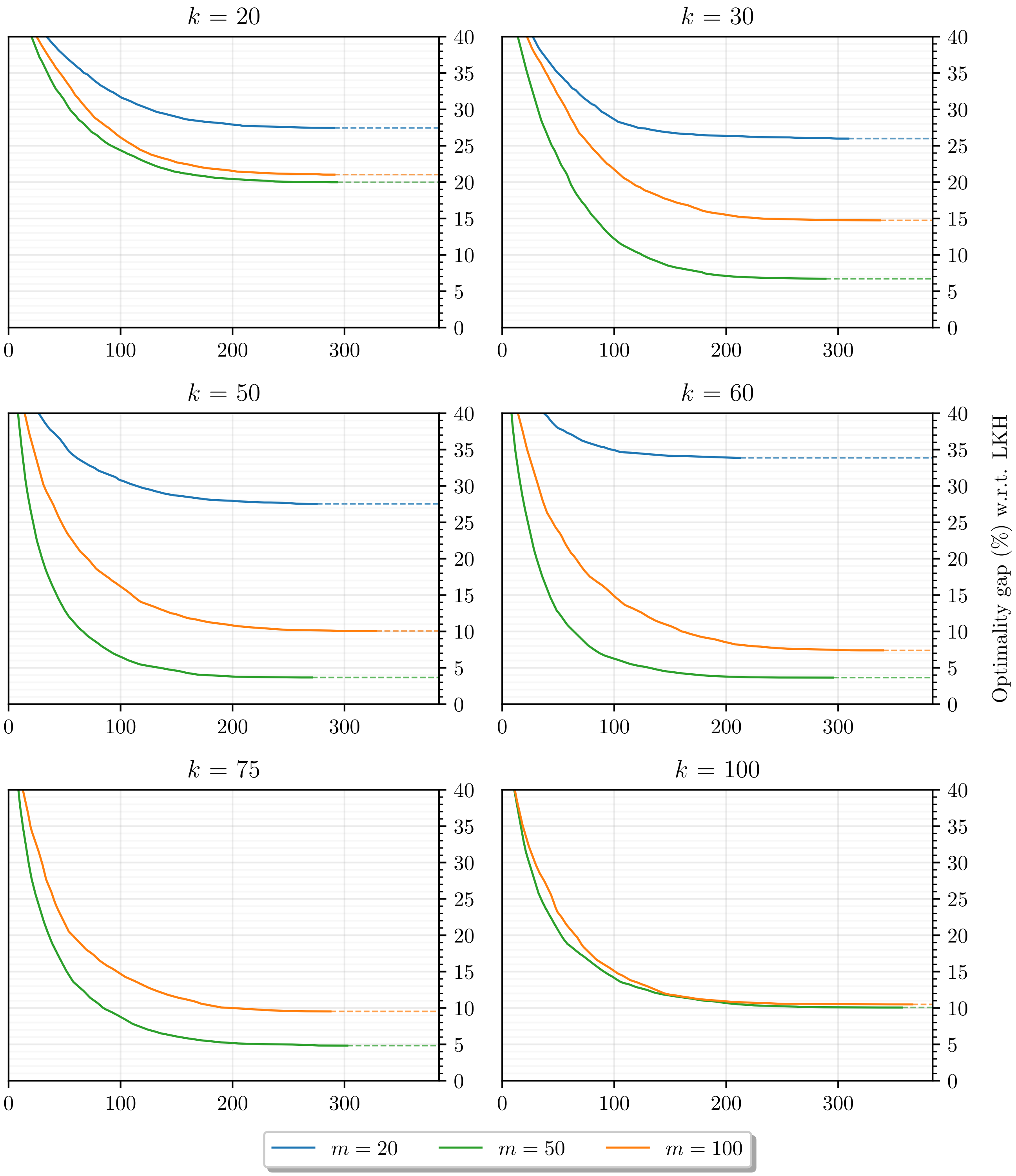


Figure 6.4: The optimality gap over wall-clock time (s) for different values of k and m.

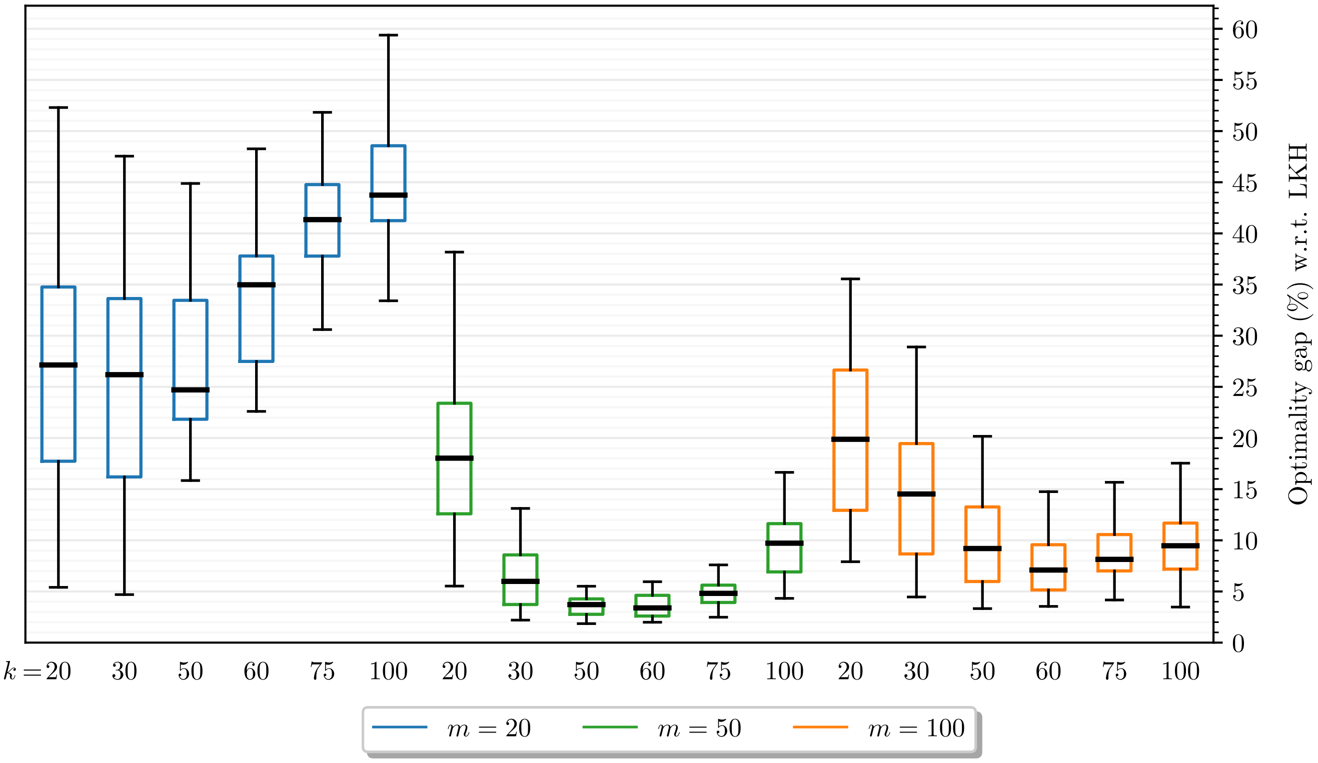


Figure 6.5: The optimality gap at termination for different values of k and m.

			Optimality gap (%) w.r.t. LKH					Total time
			1m	2m	3m	5m	Final	(s)
		LKH	0.00	0.00	0.00	0.00	0.00 ± 0.00	1 ± 0
	k	m						
Subgraph optimization with ML 2-opt	20	20	35.98	30.38	28.22	27.45	27.45 ± 11.72	127 ± 75
		50	29.08	22.93	20.69	19.98	19.98 ± 10.08	144 ± 62
		100	31.89	24.28	21.86	21.03	21.03 ± 10.02	172 ± 64
	30	20	33.36	27.50	26.45	25.99	25.98 ± 11.40	109 ± 63
		50	20.18	10.37	7.51	6.71	6.71 ± 3.49	181 ± 53
		100	29.16	19.47	16.02	14.76	14.74 ± 6.67	193 ± 65
	50	20	33.98	29.72	28.12	27.54	27.54 ± 7.35	121 ± 68
		50	11.01	5.38	**3.97**	3.67	3.67 ± 1.18	178 ± 43
		100	21.84	13.95	11.23	10.07	10.06 ± 4.79	202 ± 65
	60	20	37.19	34.48	34.02	33.87	33.87 ± 6.66	82 ± 50
		50	**10.61**	**5.32**	**3.97**	**3.65**	**3.65** ± 1.16	184 ± 49
		100	21.11	12.91	9.18	7.45	7.39 ± 2.79	229 ± 57
	75	20	42.67	42.02	42.02	42.02	42.02 ± 6.80	46 ± 34
		50	13.29	7.24	5.44	4.83	4.83 ± 1.25	194 ± 48
		100	19.52	13.15	10.37	9.53	9.53 ± 4.21	187 ± 61
	100	20	44.68	44.68	44.68	44.68	44.68 ± 5.76	30 ± 16
		50	18.61	12.87	11.01	10.08	10.07 ± 4.04	183 ± 69
		100	20.97	13.55	11.16	10.54	10.49 ± 6.08	175 ± 67

Table 6.2: The optimality gaps w.r.t. LKH at different wall-clock times for different values of k and m, over 30 test cases.

We observe that the model $m = 50$ appears best or near-best across all subgraph sizes at any time. That model performs best on subgraph size $k = 60$, very closely followed by $k = 50$. Each individual model seems to perform best on approximately the size of graphs it was trained on. Interestingly, the model trained on the largest graphs, is not the best performing, nor is the largest subgraph size, which has the largest output expressiveness with respect to local search moves over the entire graph.

To get a meaningful sense of the performance on certain subgraph sizes k, we can compare the performance against the subgraph optimization procedures

- 2-opt - as a baseline for local search (if the model does not do better than this, the subgraph optimization is qualitatively too bad), and
- LKH - as a near-optimal subgraph optimization method. We use this as an indication of how good the metaheuristic can be if the used ML model is near-perfect.

The observed optimality gaps over elapsed wall-clock time are shown in Figure 6.6, and a detailed overview of the results is in Table 6.3.

Hyperparameters

$T = 10$	$d_{\text{KNN}} = 3$	$k = \dots$	$t_X = 3$	$m = 50$
$n = 500$	$k_{\text{KNN}} = 30$	$s_X = 10$	$\Delta_X = 0.01\%$	$t_L = 1000$
	$q_{\text{KNN}} = 4$	$\rho_m = 25\%$		$\rho_D = 60\%$
	$k_t = 4$	$b_{\leftarrow} = 0$		$t_e = 8$

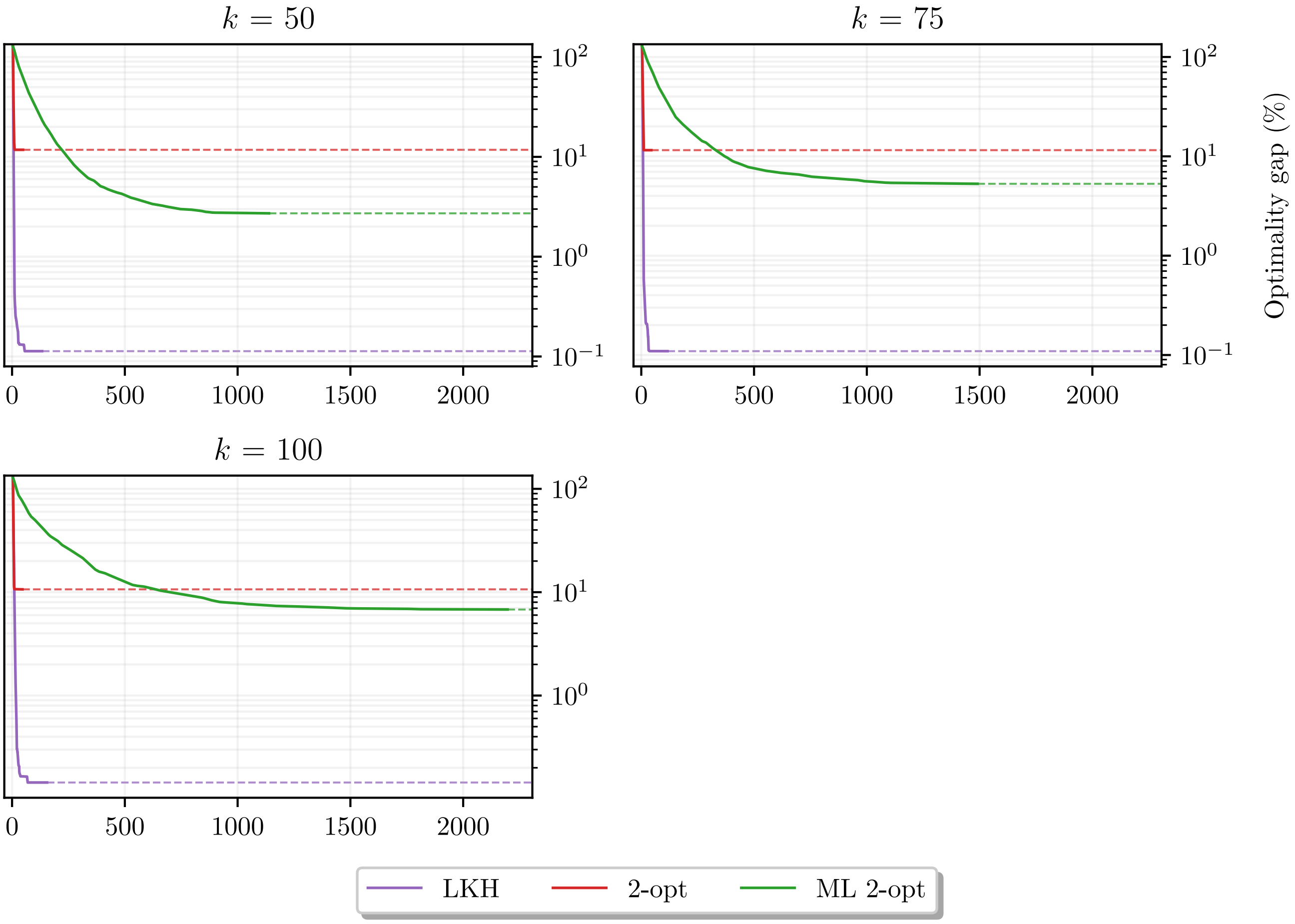


Figure 6.6: The optimality gap over wall-clock time (s) for different values of k and subgraph optimization methods.

			Optimality gap (%)					Total time
			1m	3m	5m	10m	Final	(s)
		Concorde	0.00	0.00	0.00	0.00	0.00 ± 0.00	41 ± 46
		LKH	0.08	0.08	0.08	0.08	0.08 ± 0.09	0 ± 0
		2-opt	11.27	11.27	11.27	11.27	11.27 ± 1.22	9 ± 1
Subgraph optimization	$k = 50$	LKH	0.11	0.11	0.11	0.11	0.11 ± 0.09	100 ± 21
		2-opt	11.77	11.77	11.77	11.77	11.77 ± 1.88	38 ± 6
		ML 2-opt	**51.56**	**15.79**	**7.29**	**3.51**	**2.72** ± 0.79	841 ± 206
	$k = 75$	LKH	0.11	0.11	0.11	0.11	0.11 ± 0.11	100 ± 12
		2-opt	11.54	11.54	11.54	11.54	11.54 ± 1.49	40 ± 4
		ML 2-opt	62.28	21.49	13.00	6.91	5.29 ± 0.93	875 ± 310
	$k = 100$	LKH	0.17	0.14	0.14	0.14	0.14 ± 0.13	111 ± 21
		2-opt	10.65	10.65	10.65	10.65	10.65 ± 0.99	40 ± 4
		ML 2-opt	66.66	33.77	22.56	11.15	6.80 ± 1.87	1141 ± 466

Table 6.3: The optimality gaps at different wall-clock times for different values of k, over 10 test cases.

The earlier observation that the model performs worse on $k = 100$ than on $k = 50$ becomes extra clear here, with subgraph optimization using 2-opt as a baseline of which the performance clearly increases with greater k. From the performance of subgraph optimization using LKH, we can see that there is a lot of potential for local search using TSP with mandatory edges to get near-perfect performance, with optimality gaps nearing 0.1%. Current performance of ML 2-opt is still quite far from there, indicating that there is much room left for improvement.

The slightly slightly better performance for smaller k of subgraph optimization using LKH does not make sense, but we expect it to be caused by the random variation in performance caused by the subgraph selection with respect to the entire graph, and thereby how many mandatory edges it contains. Due to time cost, the experiment was only run over 10 test cases, but we expect that using many more test cases will correct this and give slightly better performance on larger k.

6.2.2 Local search maximum unsuccessful moves t_X

The maximum number of consecutive unsuccessful moves in the local search t_X provides a tradeoff between performance and running time. Since the time it takes to improve the tour length keeps increasing when the tour length decreases, it helps to set a certain cutoff. In this case, the number of consecutive unsuccessful moves (moves with a tour length not sufficiently smaller than the previous) gives an indication of the expected difficulty of finding improvements from thereon out.

Hyperparameters

$T = 200$	$d_{\text{KNN}} = 3$	$k = 50$	$t_X = \dots$	$m = 50$
$n = 500$	$k_{\text{KNN}} = 30$	$s_X = 5$	$\Delta_X = 0.01\%$	$t_L = 175$
	$q_{\text{KNN}} = 4$	$\rho_m = 15\%$		$\rho_D = 99\%$
	$k_t = 4$	$b_{\leftarrow} = 0$		$t_e = 2$

The observed optimality gaps over elapsed wall-clock time are shown in Figure 6.7, the optimality gaps at termination are shown in Figure 6.7, and a detailed overview of the results is in Table 6.4.

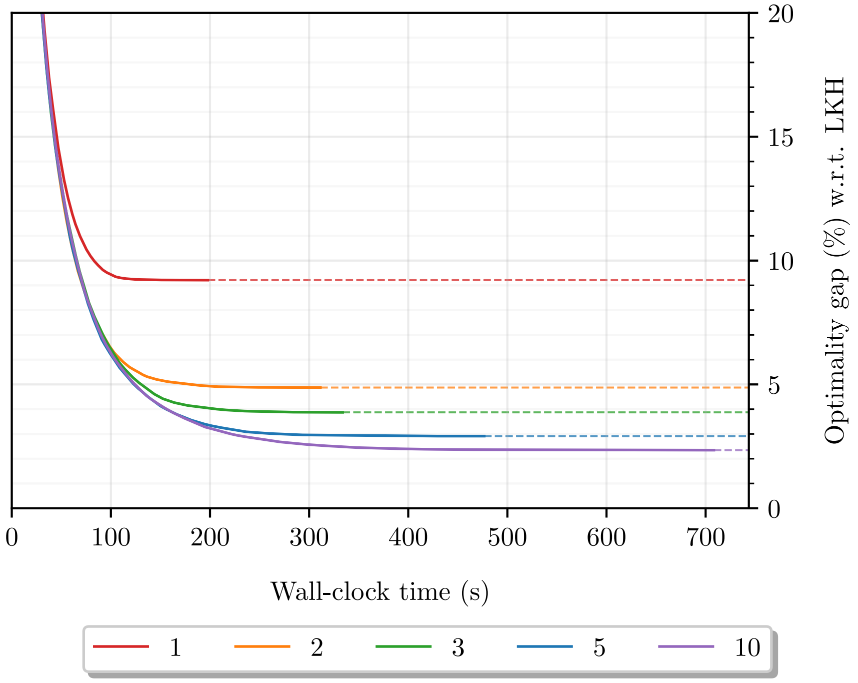


Figure 6.7: The optimality gap over wall-clock time for different values of t_X.

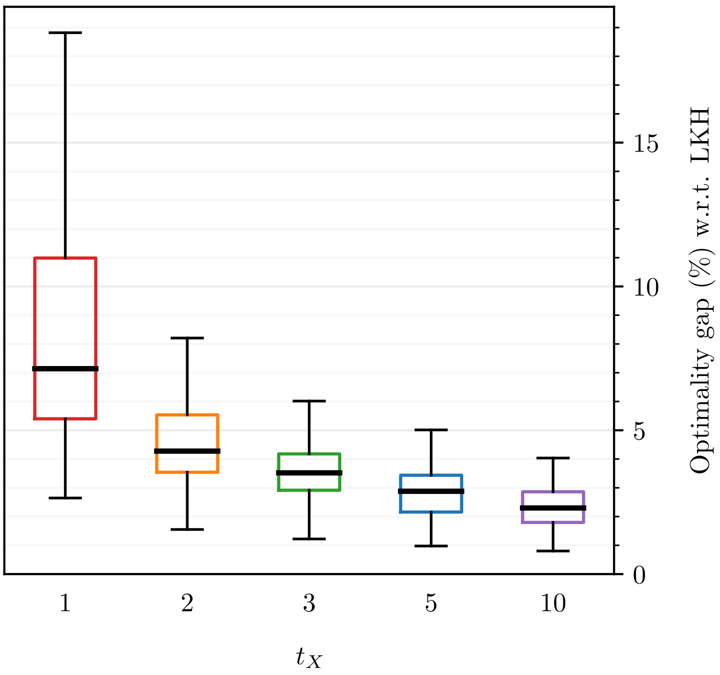


Figure 6.8: The optimality gap at termination for different values of t_X.

		Optimality gap (%) w.r.t. LKH					Total time
		1m	2m	3m	5m	Final	(s)
LKH		0.00	0.00	0.00	0.00	0.00 ± 0.00	1 ± 0
	t_X						
ML 2-opt	1	12.03	9.25	9.21	9.21	9.21 ± 6.71	80 ± 27
	2	**10.68**	5.67	5.00	4.88	4.87 ± 2.17	130 ± 38
	3	11.00	5.36	4.13	3.87	3.87 ± 1.73	165 ± 49
	5	10.71	**5.14**	3.58	2.96	2.91 ± 1.02	226 ± 62
	10	10.93	5.18	**3.53**	**2.57**	**2.35** ± 0.71	336 ± 99

Table 6.4: The optimality gaps w.r.t. LKH at different wall-clock times for different values of t_X, over 200 test cases.

We see that the performance increases with higher t_X, as does the running time, as expected. The curves have the same shape, and it appear the performance for higher t_X also increases consistently over every moment in time.

6.2.3 Local search minimum improvement Δ_X

Similar to t_X, changing Δ_X provides a tradeoff between performance and running time. During preliminary evaluation, we found that for larger n, sometimes the improvements on the tour being found become smaller for long times with hardly any unsuccessful moves. In that scenario, tweaking this hyperparameter may be a better way to achieve a certain performance-time tradeoff.

Hyperparameters

$T = 200$	$d_{\text{KNN}} = 3$	$k = 50$	$t_X = 3$	$m = 50$
$n = 500$	$k_{\text{KNN}} = 30$	$s_X = 5$	$\Delta_X = \ldots$	$t_L = 175$
	$q_{\text{KNN}} = 4$	$\rho_m = 15\%$		$\rho_D = 99\%$
	$k_t = 4$	$b_{\leftarrow} = 0$		$t_e = 2$

The observed optimality gaps over elapsed wall-clock time are shown in Figure 6.9, the optimality gaps at termination are shown in Figure 6.9, and a detailed overview of the results is in Table 6.5.

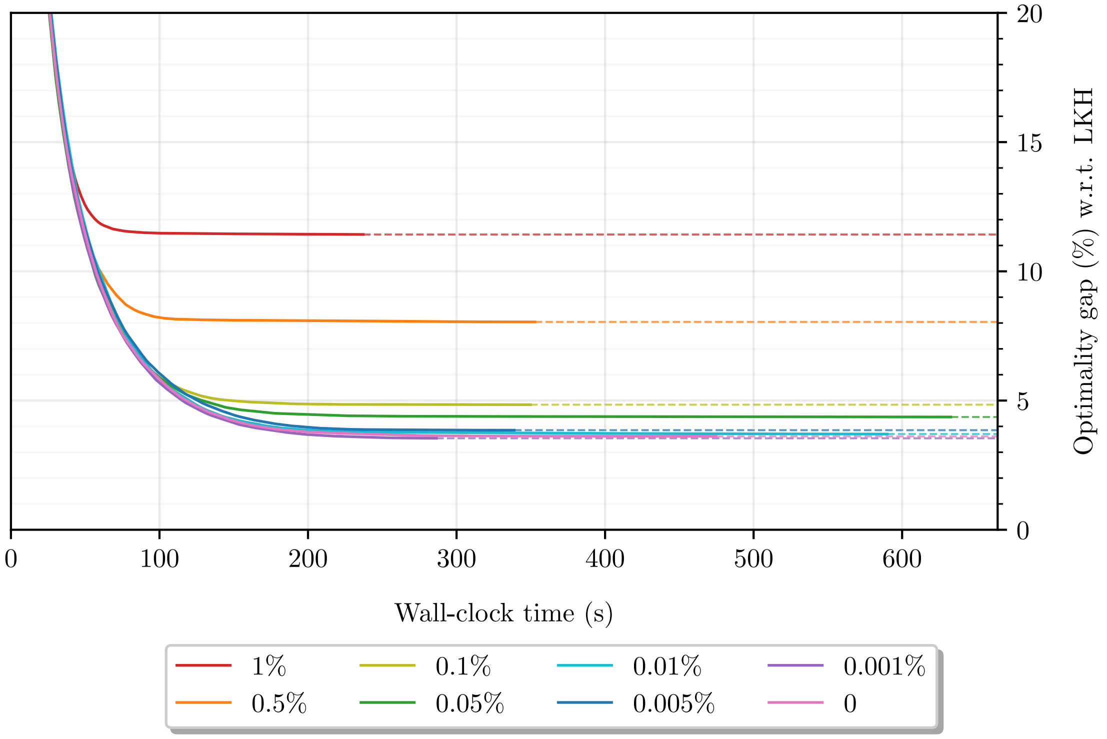


Figure 6.9: The optimality gap over wall-clock time for different values of Δ_X.

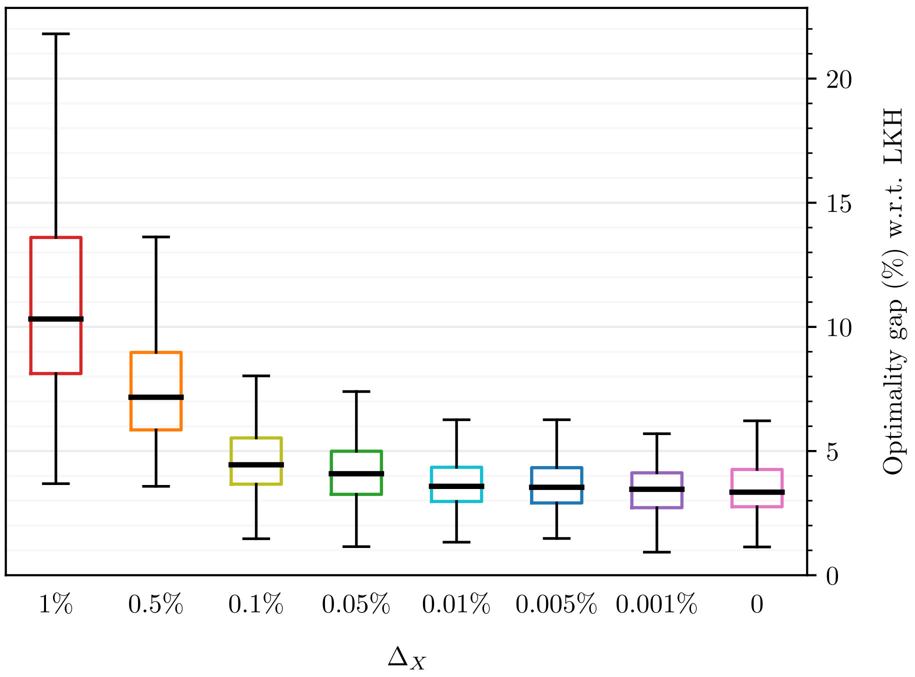


Figure 6.10: The optimality gap at termination for different values of Δ_X.

		Optimality gap (%) w.r.t. LKH					Total time
		1m	2m	3m	5m	Final	(s)
	LKH	0.00	0.00	0.00	0.00	0.00 ± 0.00	1 ± 1
	Δ_X						
ML 2-opt	1%	11.85	11.46	11.44	11.43	11.43 ± 4.82	51 ± 19
	0.5%	10.01	8.14	8.10	8.05	8.04 ± 3.36	71 ± 27
	0.1%	9.57	5.33	4.89	4.84	4.84 ± 1.92	115 ± 37
	0.05%	**9.34**	5.18	4.50	4.39	4.36 ± 1.66	135 ± 52
	0.01%	9.91	5.00	3.95	3.75	3.70 ± 1.15	165 ± 54
	0.005%	9.74	5.17	4.08	3.85	3.85 ± 1.63	166 ± 46
	0.001%	9.39	**4.84**	**3.81**	**3.54**	**3.54** ± 1.20	172 ± 48
	0	9.55	4.93	3.89	3.63	3.61 ± 1.43	171 ± 54

Table 6.5: The optimality gaps w.r.t. LKH at different wall-clock times for different values of Δ_X, over 200 test cases.

We see that the performance increases with lower Δ_X, as does the running time, as expected. The curves have the same shape, and it appear the performance for lower Δ_X also increases consistently over every moment in time. After $\Delta_X = 0.01\%$, the performance increase from lowering Δ_X becomes quite small.

6.2.4 Maximum mandatory edge ratio ρ_m

A high degree of mandatory edges in a subgraph will decrease the opportunities for improvement, and make it harder for a model not trained on mandatory edges to use the opportunities left. However, a low maximum ratio of mandatory edges ρ_m makes it harder to select valid subgraphs from the overarching graph, especially in the beginning of the local search when two nodes that are neighbors in the tour may not be so close in the graph as a whole. Therefore, we are interested to see what the effect of the value ρ_m is, since this is hard to intuitively predict.

Hyperparameters

$T = 30$	$d_{\text{KNN}} = 3$	$k = 40, 75$	$t_X = 2$	$m = 50$
$n = 500$	$k_{\text{KNN}} = 30$	$s_X = 5$	$\Delta_X = 0.05\%$	$t_L = 250$
	$q_{\text{KNN}} = 4$	$\rho_m = \dots$		$\rho_D = 99\%$
	$k_t = 4$	$b_{\leftarrow} = 0$		$t_e = 2$

The observed optimality gaps over elapsed wall-clock time are shown in Figure 6.11, the optimality gaps at termination are shown in Figure 6.11, and a detailed overview of the results is in Table 6.6.

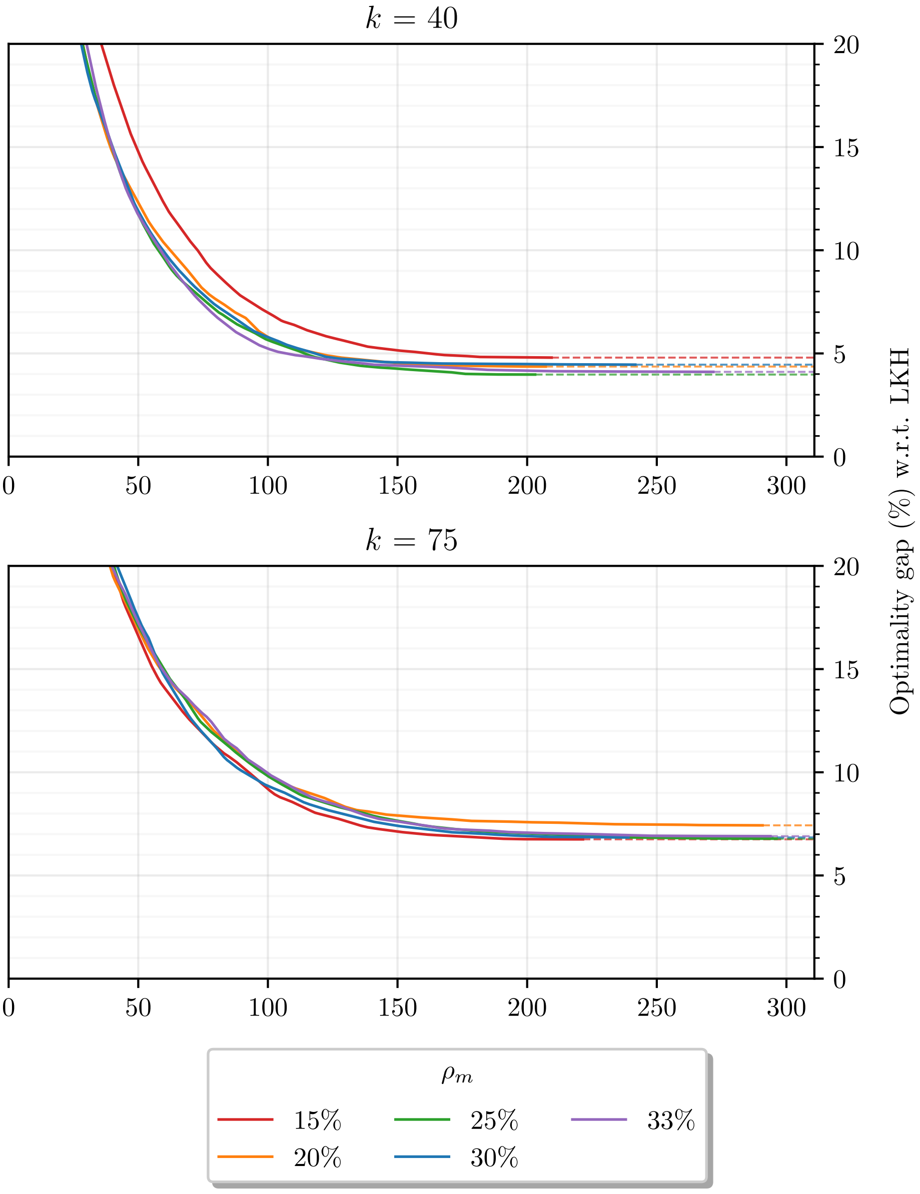


Figure 6.11: The optimality gap over wall-clock time (s) for different values of ρ_m.

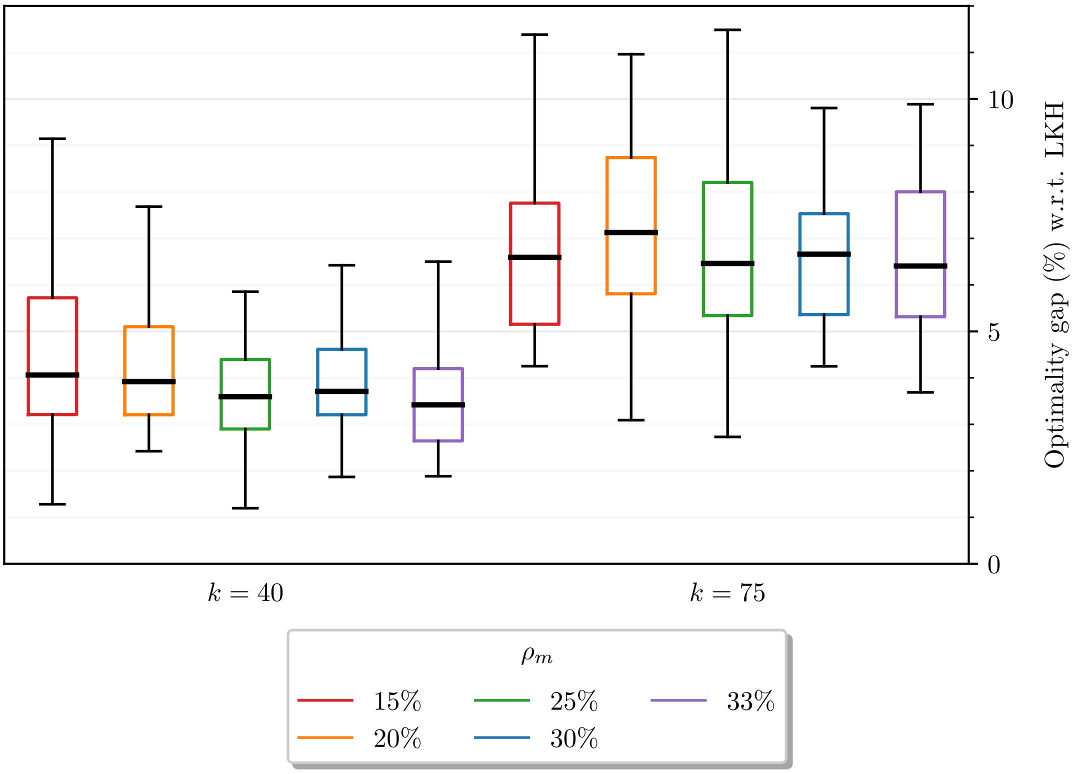


Figure 6.12: The optimality gap at termination for different values of ρ_m.

				Optimality gap (%) w.r.t. LKH					Total time
				1m	2m	3m	4m	Final	(s)
			LKH	0.00	0.00	0.00	0.00	0.00 ± 0.00	1 ± 0
Subgraph optimization with ML 2-opt	ρ_m	k	m						
	15%	40	50	12.26	5.93	4.84	4.80	4.80 ± 2.21	141 ± 35
		75	50	**14.11**	**7.98**	**6.86**	**6.75**	**6.75** ± 1.73	147 ± 32
	20%	40	50	10.34	4.98	4.40	4.36	4.36 ± 1.68	130 ± 25
		75	50	14.77	8.84	7.64	7.47	7.43 ± 2.43	144 ± 46
	25%	40	50	**9.59**	4.77	**4.00**	**3.98**	**3.98** ± 1.58	135 ± 32
		75	50	14.97	8.60	7.16	6.85	6.78 ± 2.12	156 ± 49
	30%	40	50	9.92	4.91	4.50	4.46	4.46 ± 2.80	131 ± 28
		75	50	14.67	8.28	7.05	6.85	6.85 ± 1.80	147 ± 40
	33%	40	50	9.74	**4.75**	4.24	4.12	4.11 ± 2.31	134 ± 41
		75	50	14.86	8.65	7.20	6.95	6.90 ± 2.33	157 ± 51

Table 6.6: The optimality gaps w.r.t. LKH at different wall-clock times for different values of ρ_m, over 30 test cases. The best optimality gaps per k are in bold.

We observe no significant difference between the different values of ρ_m, for $k = 40$ nor for $k = 75$. We assume this value is much less important than others, and therefore we propose it is best to choose this value safely: not too close to 0% (thereby making it very hard to select subgraphs) and not too close to 33% (thereby making it hard to optimize the subgraph tour), but somewhere in the middle, for general purposes. For a stress test, is may be better to set it high (such as 33%) and, as the subgraph tour optimization may be harder when subgraphs with a high mandatory edge ratio are selected, choose high t_X and low Δ_X to allow these failures to happen and just take up more running time.

6.2.5 ML 2-opt steps t_L

The number of 2-opt steps taken by the 2-opt model, t_L, determines the amount of improvement that is found on each subgraph. Operating for many moves on the same subgraph may decrease the performance-time ratio (it would be better to switch to another subgraph, since we keep repeating the procedure anyway). However, operating too few moves on the same subgraph may lead to the model's inability to deal with misdirected edges gracefully, and also lead to lower performance under the same running time constraints imposed by t_X and Δ_X due to a relatively lower improvement per subgraph optimization step. The original models were trained with $t_L = 2000$.

The observed optimality gaps over elapsed wall-clock time are shown in Figure 6.13, the optimality gaps at termination are shown in Figure 6.13, and a detailed overview of the results is in Table 6.7.

Hyperparameters

$T = 30$	$d_{\text{KNN}} = 3$	$k = 100$	$t_X = 3$	$m = 100$
$n = 500$	$k_{\text{KNN}} = 30$	$s_X = 5$	$\Delta_X = 0.01\%$	$t_L = \dots$
	$q_{\text{KNN}} = 4$	$\rho_m = 15\%$		$\rho_D = 99\%$
	$k_t = 4$	$b_{\leftarrow} = 0$		$t_e = 2$

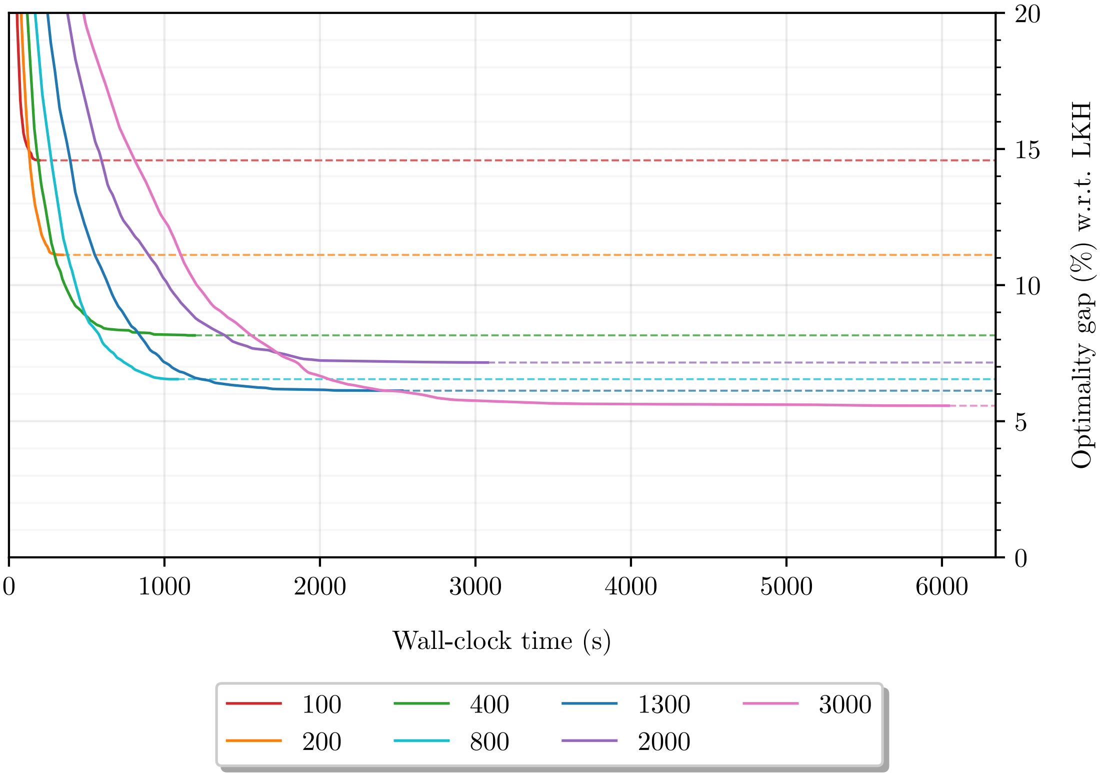


Figure 6.13: The optimality gap over wall-clock time for different values of t_L.

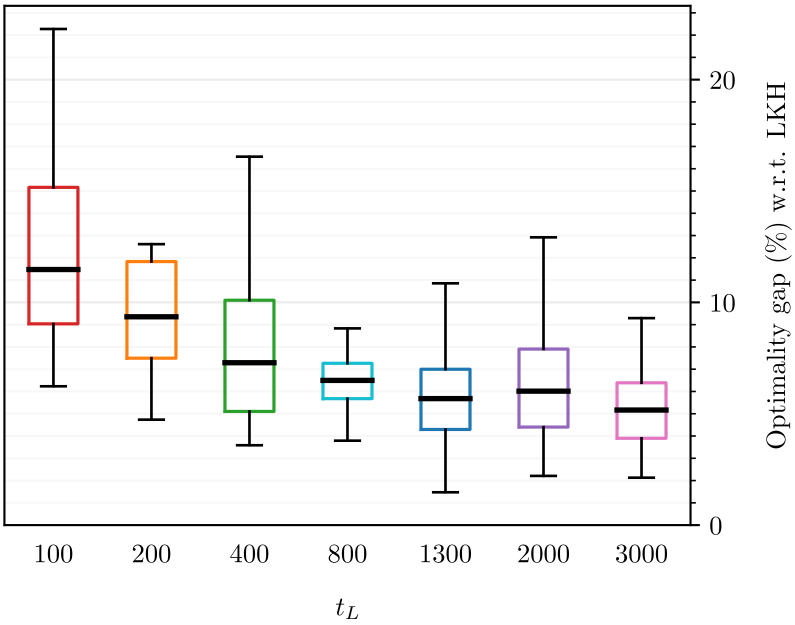


Figure 6.14: The optimality gap at termination for different values of t_L.

		Optimality gap (%) w.r.t. LKH					Total time
		2m	5m	15m	30m	Final	(s)
	LKH	0.00	0.00	0.00	0.00	0.00 $\pm$ 0.00	1 $\pm$ 0
	t_L						
ML 2-opt	100	**15.07**	14.58	14.58	14.58	14.58 $\pm$ 9.00	94 $\pm$ 41
	200	15.53	11.14	11.11	11.11	11.11 $\pm$ 7.08	195 $\pm$ 70
	400	19.86	**10.97**	8.24	8.16	8.16 $\pm$ 3.93	419 $\pm$ 206
	800	24.14	13.54	**6.69**	6.55	6.55 $\pm$ 2.08	734 $\pm$ 206
	1300	28.86	17.65	7.67	**6.17**	6.12 $\pm$ 3.00	1207 $\pm$ 385
	2000	34.24	22.88	11.08	7.42	7.16 $\pm$ 4.54	1521 $\pm$ 503
	3000	34.89	25.64	13.55	7.29	**5.57** $\pm$ 2.63	2280 $\pm$ 1016

Table 6.7: The optimality gaps w.r.t. LKH at different wall-clock times for different values of t_L, over 30 test cases.

As expected, we observe that the performance-time tradeoff increases when using lower t_L: the curve of the performance w.r.t. time is much steeper at the start. However, we also see that as expected, with lower t_L but the same t_X and Δ_X, the final performance becomes significantly worse. An interesting conclusion we can draw is that the model as a whole may run faster if we start with lower t_L, and then gradually increase t_L while it becomes harder to find improvements. Alternatively, we can conclude that it would be better to do away with the hyperparameter t_L and replace it with a stopping condition that stops when a certain improvement is reached, or a certain ratio of improvement over time has been reached.

6.2.6 ML 2-opt interval with no misdirected edges t_e

If the interval of steps t_e, in which one step has a maximum number of misdirected edges $m_D = 0$, is very low, there is not much freedom for the model to make interesting moves before the next. However, if it is too large, there will be less steps with $m_D = 0$ that produce a subgraph tour that is valid to return, thereby possibly lowering the quality of the output of the subgraph optimization.

Hyperparameters

$T = 30$	$d_{\text{KNN}} = 3$	$k = 50$	$t_X = 3$	$m = 50$
$n = 500$	$k_{\text{KNN}} = 30$	$s_X = 5$	$\Delta_X = 0.01\%$	$t_L = 250$
	$q_{\text{KNN}} = 4$	$\rho_m = 25\%$		$\rho_D = 99\%$
	$k_t = 4$	$b_{\leftarrow} = 0$		$t_e = \dots$

The observed optimality gaps over elapsed wall-clock time are shown in Figure 6.15, the optimality gaps at termination are shown in Figure 6.15, and a detailed overview of the results is in Table 6.8.

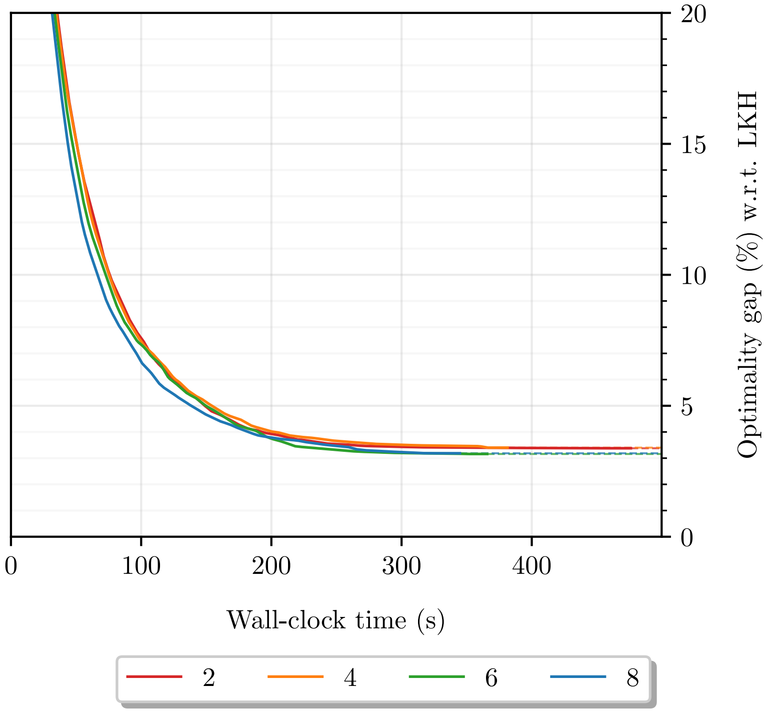


Figure 6.15: The optimality gap over wall-clock time for different values of t_e.

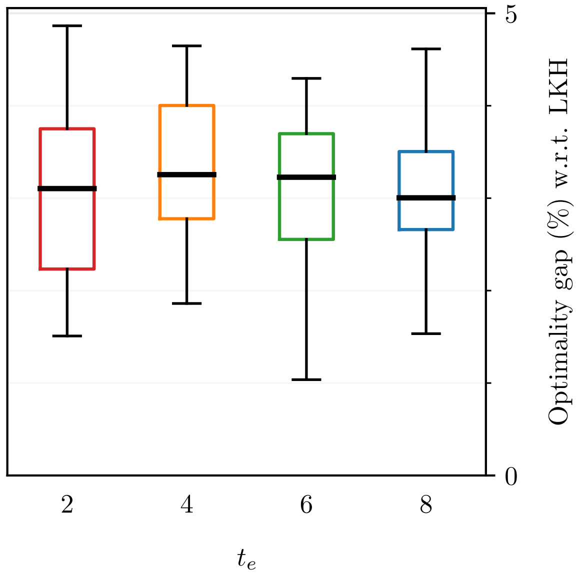


Figure 6.16: The optimality gap at termination for different values of t_e.

		Optimality gap (%) w.r.t. LKH					Total time
		1m	2m	3m	5m	Final	(s)
LKH		0.00	0.00	0.00	0.00	0.00 ± 0.00	1 ± 0
	t_E						
ML 2-opt	2	12.90	6.25	4.17	3.43	3.38 ± 1.82	230 ± 76
	4	12.56	6.38	4.36	3.51	3.40 ± 0.88	232 ± 62
	6	11.89	6.14	4.15	**3.20**	**3.16** ± 0.74	237 ± 61
	8	**10.97**	**5.61**	**4.04**	3.23	3.19 ± 0.85	237 ± 64

Table 6.8: The optimality gaps w.r.t. LKH at different wall-clock times for different values of t_e, over 30 test cases.

It appears that, for the evaluated hyperparameter configuration, either the gained final performance from increasing the number of steps with $m_D = 0$ roughly equals the potential lost performance from the ML model having less choices per 2-opt step, or alternatively, the effect is too little to measure. Since choosing low t_e has a potential downside, but choosing high t_e has a certain downside (although of unknown proportion), we propose to choose $t_e = 2$ for the maximum number of steps with $m_D = 0$.

6.3 Final evaluation

We run two final experiments to evaluate the metaheuristic: a very large instance, and instances in TSPLIB [10]. We do not attempt to achieve state-of-the-art on any stated problem, nor try to achieve the best performance we can. The hyperparameters we have chosen are chosen in order to get a result to provide insight into the potential of the concept only. Additionally, we strongly limit the running time due to hardware and time constraints. Changing these hyperparameters, and running on better hardware and for a longer time, will certainly be able to give much better performance.

Large instance

We evaluate a uniformly generated instance with $n = 10^5$, to get an idea of the potential performance this method can achieve. This is significantly larger than the cases with typical $n \leq 200$ that have been evaluated in most previous research on TSP with machine learning. We do not set termination constraints for this, but terminate it manually.

Hyperparameters

$T = 1$	$d_{\text{KNN}} = 3$	$k = 100$	$t_X = \infty$	$m = 100$
$n = 10^5$	$k_{\text{KNN}} = 100$	$s_X = 3$	$\Delta_X = 0$	$t_L = 400$
	$q_{\text{KNN}} = 25$	$\rho_m = 0.325\%$		$\rho_D = 80\%$
	$k_t = 5$	$b_{\leftarrow} = 1$		$t_e = 4$

TSPLIB 2D-TSP

We evaluate each 2D-TSP instance in TSPLIB with $n < 10^4$. This provides a reference against other research on these graph sizes, since results on TSPLIB for algorithms that run on graph sizes in the range $[10^2, 10^4]$ are available for a large part of the existing research. We will provide a comparison with the popular library OR-Tools [11].

Because some cases in TSPLIB have nodes with relative positions that cause precision problems in the ML 2-opt model, of any pair of nodes that are closer to each other than 10^{-4} we remove the one first occurring in the TSPLIB definition. We also leave out the instances `fl3795` and `vm1748`.

We evaluate ML 2-opt, and adapted beam search with beam width $w = 32$ using the constructive model proposed in Learning TSP Requires Rethinking Generalization [54] (we will refer to this model as *L. TSP* for short), of which we use the pretrained model `tsp_20-50/rl-ar-var-20pnn-gnn-max_20200313T002243` [80]. L. TSP outputs a probability logit value for each directed edge in the complete graph.

Because the metaheuristic with adapted beam search is slow due to its unparallelized nature, we can not run this model with better hyperparameters than we do now, because it would take too long for the larger graph sizes. We have, in preliminary experiments, determined hyperparameter configurations that are better for smaller graphs, and can consistently outperform 2-opt with some margin. However, note that in also in those experiments, apart from being much slower in practice, adapted beam search on L. TSP was consistently outperformed by ML 2-opt, even by ML 2-opt with comparatively uncarefully chosen hyperparameters. Therefore, we expect to see poor performance of this metaheuristic on TSPLIB.

ML 2-opt hyperparameters

$T = 70$	$d_{\text{KNN}} = 3$	$k = 50$	$t_X = \dots$	$m = 50$
$n = \dots$	$k_{\text{KNN}} = 50$	$s_X = 10$	$\Delta_X = \dots$	$t_L = \dots$
	$q_{\text{KNN}} = 10$	$\rho_m = 33\%$		$\rho_D = 99\%$
	$k_t = 4$	$b_{\leftarrow} = 1$		$t_e = 2$

Based on what we have learned from the hyperparameter analysis, we use 4 different hyperparameter configurations for ML 2-opt, from qualitative to fast:

ML 2-opt evaluated configurations

#	t_X	Δ_X	t_L
1	10	0.01%	300
2	5	0.04%	200
3	3	0.1%	150
4	2	1%	100

L. TSP hyperparameters

$T = 70$	$d_{\text{KNN}} = 3$	$k = 40$	$t_X = 50$	$w = 32$
$n = \dots$	$k_{\text{KNN}} = 50$		$\Delta_X = 0$	
	$q_{\text{KNN}} = 10$			
	$k_t = 4$	$b_{\leftarrow} = 1$		

6.4 Final results

Large instance

LKH terminated in 88 seconds. Our procedure for a warm start tour, however, already took 538 seconds. We terminated the run after 40 hours, at which an optimality gap w.r.t. LKH of 3.491% had been reached. An optimality gap of 10% took 4 hours to reach. These running times are too long to be useful for any sort of interpretation, outside of the fact they are infeasible for practical use. However, the large number of local search steps taken (31057 subgraph optimizations, of which each is the composition of 400 2-opt moves), it provides a good opportunity to look at the optimality gap over this large number of steps, which we do in Figure 6.17.

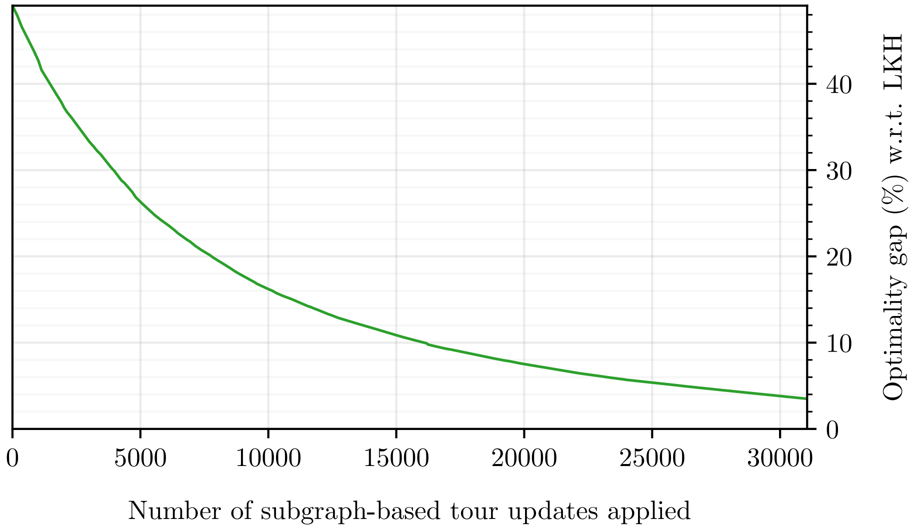


Figure 6.17: The optimality gap of ML 2-opt for $n = 10^5$, over the number of subgraph tours of which the resulting local search move for the entire graph's tour was applied.

We can see that the graph is less sharply curved than performance over elapsed wall-clock time, indicating that the improvement per local search step is more constant. This appears to indicate that each local search move at the start is reasonably saturated: the average improvement found by each of the 400 individual 2-opt moves made by the model is high. This is what we would expect for relatively low t_L.

TSPLIB 2D-TSP

The observed optimality gaps at termination for all instances, over the number of nodes they contain, is shown in Figure 6.18, and a detailed overview of the results is in Table 6.9.

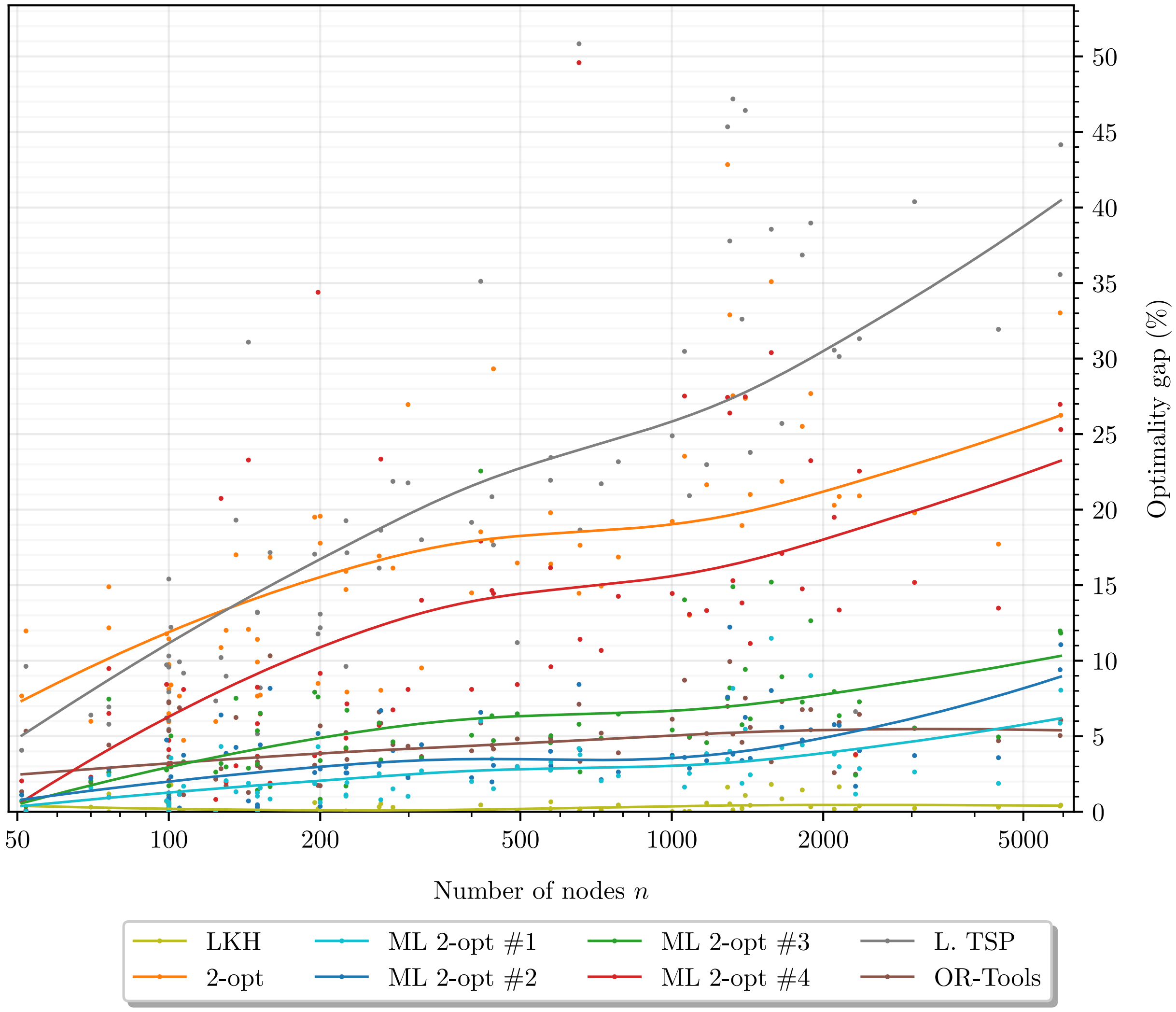


Figure 6.18: The optimality gaps at termination of the evaluated algorithms, over the number of nodes in the instances. Each point marks an individual evaluation, curves are drawn using a Savitzky-Golay filter [81].

						ML 2-opt											
		LKH		2-opt		#1		#2		#3		#4		L. TSP		OR-Tools	
Name	n	%	(s)	%	(s)	%	(s)	%	(s)	%	(s)	%	(s)	%	(s)	%	(s)
rl5934	5934	0.4	3	26.2	332	8.0	2718	11.1	650	11.8	471	25.3	158	44.2	699	**6.1**	3357
rl5915	5915	0.4	3	33.0	261	5.9	2509	9.4	701	12.0	361	27.0	155	35.6	619	**5.0**	6038
fnl4461	4461	0.3	3	17.7	136	**1.9**	1917	3.6	449	5.0	284	13.5	93	31.9	247	4.7	2872
pcb3038	3038	0.2	2	19.8	79	**2.6**	1124	3.7	402	5.5	239	15.2	75	40.4	59	5.6	988
pr2392	2360	0.4	2	20.9	73	**2.9**	1529	4.0	485	7.3	215	22.5	66	31.3	233	6.4	718
u2319	2319	0.2	2	3.7	24	**1.2**	784	1.7	178	2.4	69	3.8	25	6.6	30	2.5	349
u2152	2152	1.7	1	20.9	29	**3.0**	1296	5.7	433	6.4	220	13.4	41	30.1	56	5.9	419
d2103	2103	0.2	1	20.3	31	3.8	1085	5.8	300	8.0	177	19.5	50	30.6	140	**2.6**	686
rl1889	1889	0.3	2	27.7	61	9.0	758	**5.4**	407	12.6	153	23.2	67	39.0	179	6.8	457
u1817	1817	1.4	1	25.5	35	**4.4**	838	4.8	350	7.3	147	14.8	49	36.9	96	6.8	397
d1655	1655	0.9	1	21.9	31	**4.3**	892	5.6	397	8.9	160	17.1	40	25.7	227	7.3	230
fl1577	1577	1.8	2	35.1	44	11.5	909	8.0	512	15.2	231	30.4	55	38.6	189	**3.3**	539
u1432	1432	0.4	1	21.0	17	**2.4**	634	3.5	216	6.2	121	11.1	32	23.8	41	5.6	172
fl1400	1400	1.1	6	27.4	47	**5.5**	913	6.2	298	9.4	180	27.5	52	46.4	78	7.5	135
nrw1379	1379	0.2	1	18.9	31	**1.9**	914	3.4	420	5.8	190	13.8	47	32.6	67	4.6	121
rl1323	1323	0.1	1	27.5	36	8.2	1017	**3.8**	366	14.9	146	15.3	62	47.2	82	5.1	119
rl1304	1304	0.5	1	32.9	37	**4.0**	650	12.2	324	8.2	189	26.4	61	37.8	158	9.9	164
d1291	1291	1.6	1	42.8	20	**3.5**	828	7.6	271	7.5	254	27.4	43	45.3	141	7.0	223
pcb1173	1173	0.6	2	21.6	23	3.8	745	**3.4**	220	4.6	105	13.3	33	23.0	160	5.2	134
vm1084	1084	0.0	1	13.0	22	**2.5**	666	2.9	214	5.0	141	13.1	40	20.9	122	4.9	126
u1060	1060	0.0	3	23.5	34	**1.6**	621	3.6	632	14.0	198	27.5	53	30.5	178	8.7	107
pr1002	1002	0.0	1	19.2	29	**3.6**	1197	3.7	538	5.4	256	14.5	66	24.9	177	6.1	117
rat783	783	0.4	0	16.9	21	**2.4**	819	2.6	283	6.5	90	14.3	35	23.2	75	3.9	56
u724	724	0.1	1	14.9	18	**2.0**	645	2.1	309	4.9	127	10.7	35	21.7	94	5.2	39
d657	657	0.2	1	17.6	21	3.8	626	4.1	404	**2.6**	179	11.4	39	18.7	126	3.3	33
p654	654	0.0	2	14.5	21	**4.2**	1268	8.4	303	5.8	235	49.6	22	50.8	25	7.1	13
rat575	575	0.7	0	16.4	15	3.3	667	**3.0**	222	5.0	136	9.6	47	23.5	39	4.7	15
u574	574	0.2	1	19.8	18	**2.7**	545	4.0	239	4.6	141	16.2	38	21.9	69	4.9	23
d493	493	0.1	2	16.5	17	3.0	431	**2.9**	362	6.5	99	8.4	46	11.2	105	4.8	12
pcb442	442	0.0	0	29.3	12	**1.5**	955	3.1	282	4.7	145	14.4	36	17.7	66	4.2	12
pr439	439	0.0	1	17.9	11	3.5	843	**2.0**	289	6.4	146	14.6	43	20.9	73	4.4	11
fl417	417	0.5	2	18.5	17	6.0	1209	6.6	350	22.6	67	17.9	28	35.1	60	**5.9**	10
rd400	400	0.0	1	14.5	12	**2.0**	736	2.3	421	5.1	139	8.1	35	19.2	68	4.0	9
lin318	318	0.0	1	9.5	6	**2.7**	374	4.4	165	3.6	83	14.0	25	18.0	69	3.7	8
pr299	299	0.0	0	27.0	5	**1.0**	324	2.3	104	2.4	82	8.1	20	21.8	55	4.3	6
a280	279	0.3	0	16.1	3	**1.5**	231	4.1	116	4.6	56	6.7	21	21.9	33	4.4	5
pr264	264	0.5	0	8.0	3	**0.8**	443	6.7	91	3.4	98	23.3	7	18.6	33	5.9	4
gil262	262	0.3	0	16.9	5	**2.5**	359	3.1	247	5.9	47	5.7	22	16.1	77	6.6	4
pr226	226	0.0	0	7.9	5	**1.9**	296	2.6	131	6.7	48	7.1	25	17.1	55	3.5	2
ts225	225	0.1	0	14.7	2	**1.0**	206	3.0	82	1.7	43	4.1	17	9.6	45	5.2	2
tsp225	225	0.0	0	15.9	2	1.1	217	2.6	84	4.2	48	4.9	26	19.3	38	**0.0**	2

						ML 2-opt											
		LKH		2-opt		#1		#2		#3		#4		L. TSP		OR-Tools	
Name	n	%	(s)	%	(s)	%	(s)	%	(s)	%	(s)	%	(s)	%	(s)	%	(s)
kroA200	200	0.0	0	17.8	3	0.6	297	**0.4**	131	3.4	45	9.2	15	12.2	56	1.7	1
kroB200	200	0.0	0	19.6	3	**0.0**	275	2.8	117	0.8	58	3.9	29	13.1	54	5.7	2
d198	198	0.2	0	8.5	2	4.3	299	5.2	104	7.6	66	34.4	6	11.8	41	**1.7**	1
rat195	195	0.6	0	19.5	2	**1.9**	403	2.6	108	7.9	41	3.7	23	17.1	51	3.1	2
u159	159	0.0	0	16.8	2	**0.8**	361	8.2	77	1.7	70	1.9	22	17.2	54	10.3	1
pr152	152	0.0	0	7.7	2	**1.6**	287	4.4	80	6.5	45	6.5	17	8.2	41	2.9	1
ch150	150	0.0	0	7.7	2	**0.0**	143	0.3	98	1.4	31	3.6	14	5.2	40	3.3	1
kroA150	150	0.0	0	11.4	2	1.0	148	**0.3**	100	3.2	27	8.2	11	13.2	26	3.7	1
kroB150	150	0.0	0	9.9	2	1.3	279	**0.5**	107	5.4	30	5.8	13	13.2	34	3.0	1
pr144	144	0.0	0	12.1	2	1.9	312	**0.7**	107	2.9	65	23.3	10	31.1	14	1.3	1
pr136	136	0.0	0	17.0	1	**1.3**	218	4.3	106	7.5	20	3.0	18	19.3	15	6.2	1
ch130	130	0.0	0	12.0	2	2.1	159	3.9	55	3.0	36	2.0	16	9.0	30	**1.7**	0
bier127	127	0.0	0	10.9	2	4.3	117	6.4	93	3.2	50	20.7	11	10.2	21	**2.8**	1
pr124	124	0.0	0	6.0	2	**0.0**	208	**0.0**	71	2.6	19	0.8	15	7.3	42	2.2	0
pr107	107	0.0	0	4.7	1	1.7	198	3.7	53	3.3	55	8.1	5	9.2	10	**1.1**	0
lin105	105	0.0	0	7.7	1	1.2	105	**0.3**	71	1.3	46	1.2	15	9.9	25	6.9	1
eil101	101	1.8	0	8.4	1	3.6	313	**2.3**	68	5.0	33	3.2	16	12.2	43	3.0	0
kroA100	100	0.0	0	8.1	1	**0.0**	195	0.3	71	0.1	39	0.8	11	5.5	14	3.2	0
kroB100	100	0.0	0	9.7	1	**0.3**	196	0.9	69	1.7	38	4.7	11	9.6	18	3.6	0
kroC100	100	0.0	0	6.0	1	**0.0**	180	1.3	46	0.3	26	0.5	10	6.0	36	7.2	0
kroD100	100	0.0	0	6.5	1	**0.0**	135	**0.0**	78	0.7	21	3.2	6	15.4	24	5.4	0
kroE100	100	0.0	0	9.8	1	0.5	185	**0.4**	61	1.1	25	4.1	7	7.9	16	2.1	0
rd100	100	0.0	0	11.4	1	0.6	95	0.9	49	**0.5**	18	3.1	14	10.3	15	7.3	0
rat99	99	0.7	0	11.8	1	**0.7**	130	4.8	56	2.8	68	8.4	18	9.7	34	6.2	0
eil76	76	1.2	0	14.9	1	**2.4**	244	2.8	116	7.5	24	6.5	11	6.9	19	4.4	0
pr76	76	0.0	0	12.2	1	**1.0**	99	**1.0**	61	2.6	23	9.5	10	5.8	13	2.6	0
st70	70	0.3	0	6.0	1	**1.6**	191	2.2	53	2.0	35	2.3	8	6.4	21	1.9	0
berlin52	52	0.0	0	12.0	0	**0.0**	84	**0.0**	44	0.4	24	0.1	11	9.6	16	5.3	0
eil51	51	0.7	0	7.7	0	**0.7**	78	1.1	32	1.1	49	2.0	6	4.1	27	1.3	0
Average																	
$1000 \leq n$		0.6	2	23.7	65	**4.3**	1116	5.4	398	8.3	205	18.9	62	32.9	181	5.8	839
$500 \leq n < 1000$		0.3	1	16.7	19	**3.1**	762	4.1	294	4.9	151	18.6	36	26.6	71	4.9	30
$200 \leq n < 500$		0.1	1	16.7	7	**2.0**	480	3.2	198	5.5	80	10.0	27	18.1	59	4.3	6
$100 \leq n < 200$		0.1	0	10.1	1	**1.3**	216	2.2	80	3.2	38	6.8	13	11.8	29	3.9	1
$n < 100$		0.5	0	10.7	1	**1.1**	138	2.0	60	2.7	37	4.8	11	7.1	22	3.6	0
All		0.3	1	16.4	24	**2.5**	595	3.6	222	5.4	109	12.1	33	20.7	86	4.6	268

Table 6.9: The optimality gaps (%) and rounded elapsed wall-clock time (s) at termination of LKH, 2-opt, the 4 ML 2-opt configurations, L. TSP with adapted beam search and OR-Tools, for all evaluated instances in TSPLIB. The lowest non-LKH optimality gap for each instance is in bold (41x ML 2-opt #1, 13x ML 2-opt #2, 4x both ML 2-opt #1 and #2, 2x ML 2-opt #3, and 10x OR-Tools).

It is clear that the ML 2-opt metaheuristic struggles with some of the larger instances, reaching optimality gaps well above 5%. However, it can be clearly seen that 2-opt is improved upon by a wide margin, even so by the fastest configuration in almost all cases. The fastest configuration on the few largest instances is actually both faster in wall-clock time and reaches a lower optimality gap than 2-opt. We can clearly observe the tradeoff between optimality and running time, determined by the hyperparameter configurations of the different ML 2-opt variations. Their optimality gaps general place at various degrees between LKH and 2-opt. At lower n, the ratio in optimality gap between ML 2-opt and 2-opt grows larger. Furthermore notable is that ML 2-opt variations #1 and #2 solve a variety of cases, up to `kroB200` with $n = 200$, to optimality.

Trends in the optimality gap are reasonably consistent across models. Interestingly, the running time trends are also reasonably consistent across models when taking a few instances of comparable size. For local search, this makes sense because those models start from the same initial tour.

Towards larger sizes, the best variations of ML 2-opt appear to underperform compared to OR-Tools, but it is important to note that OR-Tools at this point takes far more time to run, and there is much room for hyperparameter tuning that would pull the optimality gaps produced by ML 2-opt further down.

Generally, we observe highly satisfactory and promising results, with the best ML 2-opt models mostly performing much better than the alternative non-LKH solvers.

Chapter 7

Conclusions and future work

By exploring the topic of hybrid metaheuristics for TSP, we have formulated a list of requirements and considerations, discussed the relevant relationship between the metaheuristic and the ML model and proposed techniques to adjust an existing heuristic. These can be actively used to create a new metaheuristic algorithm. Particularly, we have shown that based on these ideas, we were able to both derive the concept and significance of local search using TSP with mandatory edges, as well as construct such a metaheuristic.

The developed local search metaheuristic using machine-learned 2-opt [9] is applicable to graphs much larger than most previous related work. We have shown that machine learning can be used in local search to solve TSP on graphs with $100,000$ nodes. The obtained optimality gap w.r.t. LKH of 3.5% in this case, as well as the low optimality gaps obtained for TSPLIB, show that this type of method has the potential to achieve high performance. Through this metaheuristic we have provided a concrete way for any existing ML solution to TSP to be adapted to larger graph sizes. Additionally, we have shown that asymmetric mandatory edges and move tree composition allow us to effectively parallelize improvement heuristics on such large graphs. Furthermore, in doing so for ML 2-opt, thereby allowing its reinforcement learning based 2-opt decisions to be applied to graphs of those larger sizes, we have successfully scaled an ML-based improvement heuristic for TSP.

Based on the achieved performance, we conclude that local search using TSP with mandatory edges is indeed a powerful metaheuristic method for TSP. We put forward the argument that the method generalizes over all explicitly bounded local search algorithms. From this we reason that to gain new insights into using machine learning in hybrid metaheuristics, improving the underlying ML model used in this method has greater potential than researching specific such bounded local search algorithms. On the other hand, because of the existence of this generalization relation, we believe that the most meaningful novel insights into local search for combinatorial optimization require explicitly considering local search metaheuristics that are not explicitly bounded.

Using machine learning to create hybrid metaheuristics for combinatorial optimization is a topic with much room for creative, novel ideas. Further investigation into expressing existing algorithms, such as LKH, in steps that machine learning can perform, could provide key insights into ways to develop metaheuristics with ML. Particularly, the ability of a machine learning model to iteratively construct a single local search move, as LKH does, would significantly increase the potential of these hybrid metaheuristics.

In the domain of machine learning, being able to learn both short- and long-distance relationships between elements in large sparse graphs would be helpful to solve NP-hard combinatorial optimization problems, for which improvements in performance increasingly depend on such long-distance relationships. Current models are often explicitly learning short-distance relationships due to the model architecture or the statistical significance of short-distance relationships in the dataset. We strongly expect even algorithms as powerful and efficient as LKH to be overtaken by algorithms that primarily use machine learning. In part, this is due to the general continuous trend of improvement seen in machine learning, with little regard for the status quo in existing solutions. In part, this is because the best existing algorithms will be slowly assimilated by iteratively finding ML-based solutions for their components. However, there are no free lunches [82], and these algorithms may perform well on certain datasets, but always have identifiable, and potentially adversarially generatable [83] inputs on which they fail. Moving to a paradigm where concrete algorithmic steps are produced by ML would ensure that the resulting algorithm has some known bounds and guarantees, but there is no clear pathway from here to such a paradigm.

Finally, specifically as a continuation of the method of local search using TSP with mandatory edges proposed in this thesis, there are many opportunities both for improvements in performance and efficiency. For example, an ML-based heuristic can be used to guide the subgraph selection. Subgraph partition with asymmetric mandatory edges, followed by performing improvement such as single 2-opt moves in parallel through tree composition can be done efficiently and continuously within an ML model running over large sparse graphs, removing the need for a model instance per subgraph.

Bibliography

[1] G. B. Dantzig and J. H. Ramser. 'The Truck Dispatching Problem'. *Management Science* 6.1 (1959), pages 80–91.

[2] E. K. Baker. 'An exact algorithm for the time-constrained traveling salesman problem'. *Operations Research* 31.5 (1983), pages 938–945.

[3] C. C. Murray and A. G. Chu. 'The flying sidekick traveling salesman problem: Optimization of drone-assisted parcel delivery'. *Transportation Research Part C-emerging Technologies* 54 (2015), pages 86–109.

[4] N. A. H. Agatz, P. Bouman and M. Schmidt. 'Optimization Approaches for the Traveling Salesman Problem with Drone'. *Transportation Science* 52 (2018), pages 965–981.

[5] C. H. Papadimitriou. 'The Euclidean travelling salesman problem is NP-complete'. *Theoretical Computer Science* 4.3 (1977), pages 237–244.

[6] N. Christofides. *Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem.* Defense Technical Information Center, 1976.

[7] S. Lin and B. W. Kernighan. 'An Effective Heuristic Algorithm for the Traveling-Salesman Problem'. *Operations Research* 21.2 (1973), pages 498–516.

[8] D. L. Applegate, R. E. Bixby, V. Chvatal and W. J. Cook. *The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics).* Princeton University Press, 2007.

[9] P. R. de O. da Costa, J. Rhuggenaath, Y. Zhang and A. Akcay. 'Learning 2-opt Heuristics for the Traveling Salesman Problem via Deep Reinforcement Learning' (2020). arXiv: 2004.01608.

[10] G. Reinelt. 'TSPLIB—A Traveling Salesman Problem Library'. *ORSA Journal on Computing* 3.4 (1991), pages 376–384.

[11] L. Perron and V. Furnon. 'OR-Tools'. https://developers.google.com/optimization/. Accessed: July 18th 2021.

[12] R. Kumar and H. Li. 'On Asymmetric TSP: Transformation to Symmetric TSP and Performance Bound'. *Journal of Operations Research* (1996).

[13] R. Bellman. 'Dynamic Programming Treatment of the Travelling Salesman Problem'. *J. ACM* 9.1 (1962), pages 61–63.

[14] P. Bouman, N. A. H. Agatz and M. Schmidt. 'Dynamic Programming Approaches for the Traveling Salesman Problem with Drone'. *SSRN Electronic Journal* (2017).

[15] D. L. Applegate, R. E. Bixby, V. Chvatal and W. J. Cook. 'Concorde Downloads'. http://www.math.uwaterloo.ca/tsp/concorde/downloads/downloads.htm. Accessed: July 18th 2021.

[16] R. E. Gomory. 'Outline of an algorithm for integer solutions to linear programs'. *Bulletin of the American Mathematical Society* 64.5 (1958), pages 275–278.

[17] D. L. Applegate, S. Dash and W. J. Cook. 'QSopt Linear Programming Solver'. http://www.math.uwaterloo.ca/~bico/qsopt/index.html. Accessed: July 18th 2021.

[18] G. B. Dantzig, D. R. Fulkerson and S. M. Johnson. *Solution of a Large-Scale Traveling-Salesman Problem.* RAND Corporation, 1954.

[19] C. Papadimitriou and K. Steiglitz. 'Some Examples of Difficult Traveling Salesman Problems'. *Operations Research* 26 (1978), pages 434–443.

[20] M. L. Fredman, D. S. Johnson, L. A. McGeoch and G. Ostheimer. 'Data Structures for Traveling Salesmen'. *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms* (1993), pages 145–154.

[21] K. Helsgaun. 'An effective implementation of the Lin–Kernighan traveling salesman heuristic'. *European Journal of Operational Research* 126.1 (2000), pages 106–130.

[22] K. Helsgaun. 'General k-opt submoves for the Lin–Kernighan TSP heuristic'. *Mathematical Programming Computation* 1 (2009), pages 119–163.

[23] K. Helsgaun. *An Extension of the Lin-Kernighan-Helsgaun TSP Solver for Constrained Traveling Salesman and Vehicle Routing Problems: Technical report.* 2017.

[24] D. L. Applegate, W. J. Cook and A. Rohe. 'Chained Lin-Kernighan for large traveling salesman problems. INFORMS Journal on Computing, 15, 82-92'. *INFORMS Journal on Computing* 15 (2003), pages 82–92.

[25] D. Rumelhart, G. E. Hinton and R. J. Williams. 'Learning representations by back-propagating errors'. *Nature* 323 (1986), pages 533–536.

[26] D. Saad. *On-Line Learning in Neural Networks.* Publications of the Newton Institute. Cambridge University Press, 1999.

[27] H. Robbins and S. Monro. 'A Stochastic Approximation Method'. *The Annals of Mathematical Statistics* 22.3 (1951), pages 400–407.

[28] G. Cybenko. 'Approximation by superpositions of a sigmoidal function'. *Mathematics of Control, Signals and Systems* 2 (1989), pages 303–314.

[29] K. Fukushima and S. Miyake. 'Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Visual Pattern Recognition'. *Competition and Cooperation in Neural Nets* (1982), pages 267–285.

[30] V. Nair and G. E. Hinton. 'Rectified Linear Units Improve Restricted Boltzmann Machines'. ICML'10 (2010), pages 807–814.

[31] A. L. Maas, A. Y. Hannun and A. Y. Ng. 'Rectifier Nonlinearities Improve Neural Network Acoustic Models' (2013).

[32] A. Krogh and J. Hertz. *A Simple Weight Decay Can Improve Generalization.* 1992.

[33] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever and R. Salakhutdinov. 'Dropout: A Simple Way to Prevent Neural Networks from Overfitting'. *Journal of Machine Learning Research* 15 (2014), pages 1929–1958.

[34] I. Sergey and S. Christian. 'Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift' (2015). arXiv: `1502.03167`.

[35] C. Watkins. 'Learning From Delayed Rewards' (Jan. 1989).

[36] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra and M. Riedmiller. 'Playing Atari with Deep Reinforcement Learning' (2013). arXiv: `1312.5602`.

[37] D. Silver et al. 'Mastering the game of Go with deep neural networks and tree search'. *Nature* 529 (Jan. 2016), pages 484–489.

[38] D. Silver et al. 'Mastering the game of Go without human knowledge'. *Nature* 550 (Oct. 2017), pages 354–359.

[39] K. Yamaguchi, K. Sakamoto, T. Akabane and Y. Fujimoto. 'A neural network for speaker-independent isolated word recognition'. *The First International Conference on Spoken Language Processing, ICSLP 1990, Kobe, Japan, November 18-22, 1990* (1990).

[40] D. E. Rumelhart, G. E. Hinton and R. J. Williams. *Learning internal representations by error propagation.* Institute for Cognitive Science, University of California, 1985.

[41] K. Gregor, I. Danihelka, A. Graves, D. Rezende and D. Wierstra. 'DRAW: A Recurrent Neural Network For Image Generation'. *Proceedings of the 32nd International Conference on Machine Learning.* Proceedings of Machine Learning Research 37 (2015), pages 1462–1471.

[42] K. Cho, B. van Merrienboer, Ç. Gülçehre, F. Bougares, H. Schwenk and Y. Bengio. 'Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation' (2014). arXiv: `1406.1078`.

[43] S. Hochreiter and J. Schmidhuber. 'Long Short-term Memory'. *Neural computation* 9 (1997), pages 1735–80.

[44] F. Scarselli, Marco Gori, A. C. Tsoi, M. Hagenbuchner and G. Monfardini. 'The Graph Neural Network Model'. *IEEE Transactions on Neural Networks* 20.1 (2009), pages 61–80.

[45] T. N. Kipf and M. Welling. 'Semi-Supervised Classification with Graph Convolutional Networks' (2016). arXiv: `1609.02907`.

[46] M. Defferrard, X. Bresson and P. Vandergheynst. 'Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering' (2016). arXiv: `1606.09375`.

[47] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals and G. E. Dahl. 'Neural Message Passing for Quantum Chemistry' (2017). arXiv: `1704.01212`.

[48] A. Nammouchi, H. Ghazzai and Y. Massoud. 'A Generative Graph Method to Solve the Travelling Salesman Problem' (2020). arXiv: `2007.04949`.

[49] Y. Li, D. Tarlow, M. Brockschmidt and R. Zemel. 'Gated Graph Sequence Neural Networks' (2017). arXiv: `1511.05493`.

[50] D. Selsam, M. Lamm, B. Bünz, P. Liang, L. de Moura and D. L. Dill. 'Learning a SAT Solver from Single-Bit Supervision' (2018). arXiv: `1802.03685`.

[51] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò and Y. Bengio. 'Graph Attention Networks' (2018). arXiv: `1710.10903`.

[52] D. Bahdanau, K. Cho and Y. Bengio. 'Neural Machine Translation by Jointly Learning to Align and Translate' (2016). arXiv: `1409.0473`.

[53] M.-T. Luong, H. Pham and C. D. Manning. 'Effective Approaches to Attention-based Neural Machine Translation' (2015). arXiv: `1508.04025`.

[54] C. K. Joshi, Q. Cappart, L.-M. Rousseau, T. Laurent and X. Bresson. 'Learning TSP Requires Rethinking Generalization' (2020). arXiv: `2006.07054`.

[55] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser and I. Polosukhin. 'Attention Is All You Need' (2017). arXiv: `1706.03762`.

[56] O. Vinyals, M. Fortunato and N. Jaitly. 'Pointer Networks' (2017). arXiv: `1506.03134`.

[57] Y. Bengio, A. Lodi and A. Prouvost. 'Machine Learning for Combinatorial Optimization: a Methodological Tour d'Horizon' (2020). arXiv: `1811.06128`.

[58] D. Selsam and N. Bjørner. 'NeuroCore: Guiding High-Performance SAT Solvers with Unsat-Core Predictions' (2019). arXiv: `1903.04671`.

[59] H. Dai, E. B. Khalil, Y. Zhang, B. Dilkina and L. Song. 'Learning Combinatorial Optimization Algorithms over Graphs'. *Proceedings of the 31st International Conference on Neural Information Processing Systems.* NIPS'17 (2017), pages 6351–6361.

[60] I. Bello, H. Pham, Q. V. Le, M. Norouzi and S. Bengio. 'Neural Combinatorial Optimization with Reinforcement Learning' (2017). arXiv: `1611.09940`.

[61] M. Deudon, P. Cournut, A. Lacoste, Y. Adulyasak and L.-M. Rousseau. 'Learning Heuristics for the TSP by Policy Gradient'. *Integration of Constraint Programming, Artificial Intelligence, and Operations Research* (2018), pages 170–181.

[62] W. Kool, H. van Hoof and M. Welling. 'Attention, Learn to Solve Routing Problems!' *International Conference on Learning Representations* (2019). arXiv: 1803.08475.

[63] C. K. Joshi, T. Laurent and X. Bresson. 'An Efficient Graph Convolutional Network Technique for the Travelling Salesman Problem' (2019). arXiv: 1906.01227.

[64] Z.-H. Fu, K.-B. Qiu and H. Zha. 'Generalize a Small Pre-trained Model to Arbitrarily Large TSP Instances' (2020). arXiv: 2012.10658.

[65] W. Kool, H. van Hoof, J. A. S. Gromicho and M. Welling. 'Deep Policy Dynamic Programming for Vehicle Routing Problems' (2021). arXiv: 2102.11756.

[66] Y. Wu, W. Song, Z. Cao, J. Zhang and A. Lim. 'Learning Improvement Heuristics for Solving Routing Problems'. *IEEE Transactions on Neural Networks and Learning Systems* (2021), pages 1–13. arXiv: 1912.05784 [cs.AI].

[67] M. Gendreau, A. Hertz and G. Laporte. 'New Insertion and Postoptimization Procedures for the Traveling Salesman Problem'. *Operations Research* 40.6 (1992), pages 1086–1094.

[68] J. Zheng, K. He, J. Zhou, Y. Jin and C.-M. Li. 'Combining Reinforcement Learning with Lin-Kernighan-Helsgaun Algorithm for the Traveling Salesman Problem' (2020). arXiv: 2012.04461.

[69] M. A. Kramer. 'Nonlinear principal component analysis using autoassociative neural networks'. *AIChE Journal* 37.2 (1991), pages 233–243.

[70] L. Sengupta, R. Mariescu-Istodor and P. Fränti. 'Which Local Search Operator Works Best for the Open-Loop TSP?' *Applied Sciences* 9.19 (2019).

[71] M. Wang et al. 'Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks' (2019). arXiv: 1909.01315.

[72] M. Tanaka. 'Weighted Sigmoid Gate Unit for an Activation Function of Deep Neural Network' (2018). arXiv: 1810.01829.

[73] K. Ishikawa, I. Suzuki, M. Yamamoto and M. Furukawa. 'Solving for Large-scale Traveling Salesman Problem with Divide-and-conquer Strategy'. *SCIS & ISIS* 2010 (2010), pages 1168–1173.

[74] G. Reinelt. 'tsp95.dvi'. http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp95.pdf. Accessed: July 18th 2021.

[75] G. Jenks. 'Implementation Details - Sorted Containers 2.4.0 documentation'. http://www.grantjenks.com/docs/sortedcontainers/implementation.html. Accessed: July 18th 2021.

[76] J. R. Driscoll, N. Sarnak, D. D. Sleator and R. E. Tarjan. 'Making data structures persistent'. *Journal of Computer and System Sciences* 38.1 (1989), pages 86–124.

[77] E. Myers. 'An Applicative Random-Access Stack'. *Inf. Process. Lett.* 17 (1983), pages 241–248.

[78] G. Reinelt. 'TSPLIB'. http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/. Accessed: July 18th 2021.

[79] P. R. de O. da Costa, J. Rhuggenaath, Y. Zhang and A. Akcay. 'paulorocosta/learning-2opt-drl: Code Implementation of Learning 2-opt Heuristics for the TSP via Deep Reinforcement Learning'. https://github.com/paulorocosta/learning-2opt-drl. Accessed: July 18th 2021.

[80] C. K. Joshi, Q. Cappart, L.-M. Rousseau, T. Laurent and X. Bresson. 'chaitjo/learning-tsp: Code for the paper 'Learning TSP Requires Rethinking Generalization' (arXiv Pre-print)'. https://github.com/chaitjo/learning-tsp. Accessed: July 18th 2021.

[81] A. Savitzky and M. J. E. Golay. 'Smoothing and Differentiation of Data by Simplified Least Squares Procedures.' *Analytical Chemistry* 36.8 (1964), pages 1627–1639.

[82] D.H. Wolpert and W.G. Macready. ‘No free lunch theorems for optimization’. *IEEE Transactions on Evolutionary Computation* 1.1 (1997), pages 67–82.

[83] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville and Y. Bengio. ‘Generative Adversarial Networks’ (2014). arXiv: `1406.2661`.