# Study of Metaheuristic, Hyper-Heuristic and Machine Learning Approaches to solve the Traveling Salesman Problem

Studienarbeit von Steffen Müller
Tag der Einreichung:

1. Gutachten: Jun.-Prof. Dr. Simon Emde
2. Gutachten: Lukas Polten, M.Sc.

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachgebiet Management Science und Operations Research

Study of Metaheuristic, Hyper-Heuristic and Machine Learning Approaches to solve the Traveling Salesman Problem

Vorgelegte Studienarbeit von Steffen Müller

1. Gutachten: Jun.-Prof. Dr. Simon Emde
2. Gutachten: Lukas Polten, M.Sc.

Tag der Einreichung:

**Erklärung zur Studienarbeit**

Hiermit versichere ich, die vorliegende Studienarbeit ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 03. August 2018

(Steffen Müller)

## Abstract

In this research project, different methods for solving the Traveling Salesman Problem (TSP) are presented. The TSP is the problem of finding a minimal length Hamiltonian cycle of a graph, where a Hamiltonian cycle is a closed tour visiting each node exactly once. It belongs to the $\mathcal{NP}$-complete optimization problems and an optimal solution is hard to obtain. Therefore, (meta-)heuristic approaches are used to approximate the optimal solution. In this work, three metaheuristic approaches are presented and defined: the Ant System, the Ant Colony System and Genetic Algorithms.
These algorithms describe a general solution scheme for combinatorial problems and require less computational effort than exact algorithms. In addition, two machine learning approaches are applied. The first approach uses reinforcement learning, where a self-learning agent optimizes a single problem instance in order to obtain a close-to-optimal solution. The second machine learning approach concerns the Algorithm Selection Problem. Each metaheuristic leads to varying performances on different problems or instances of a problem. A supervised meta-learning algorithm can be trained in order to predict the best metaheuristic for unseen problem instances. Hyper-heuristics expand this method and represent a high-level approach to select the best low-level heuristic at each decision step during construction or improvement of a solution.

## Contents

## List of Figures

## List of Tables

# 1 Introduction

In many fields of operation, it is important to find the shortest path through a collection of points. The most common example is the delivery of packages. The shortest route is chosen to save fuel and labour cost. In Operations Research this type of problem is called Traveling Salesman Problem (TSP). The problem states that a salesman has to visit a set of cities all exactly once, using the route with the overall shortest traveling distance. Solving the TSP seems fairly simple but its complexity increases more than exponentially with the number of points due to its $\mathcal{NP}$-completeness as proven in [1].
In computational complexity theory, a $\mathcal{NP}$-complete (non-deterministic polynomial-time complete) decision problem belongs to the $\mathcal{NP}$ and the $\mathcal{NP}$-hard complexity classes. An arbitrary problem $C$ is $\mathcal{NP}$-hard if every problem $D$ in $\mathcal{NP}$ can be reduced to $C$ in polynomial time. If solving $C$ is possible in a short period of time, we can use this solution to solve $D$ in polynomial time. As a result, finding a polynomial algorithm to solve any $\mathcal{NP}$-hard problem would return polynomial algorithms for all problems in $\mathcal{NP}$.
Although any given solution to a $\mathcal{NP}$-complete problem can be verified quickly (in polynomial time), there is no known efficient way to obtain this solution. The time required to solve the problem using any currently known algorithm increases very quickly as the size of the problem grows [2]. Therefore, new methods have to be investigated to obtain a close-to-optimal solution for large TSP instances with lower computation effort. These approximation algorithms or heuristics reduce the number of possible solutions leading to a lower combinatorial complexity of the given TSP.
Generally, researchers are interested in the TSP, because it belongs to the large class of combinatorial problems. These problems occur in many variations, but can also be modeled as a TSP. Therefore, TSPs are still an important field of research and new optimization algorithms/heuristics are benchmarked against other algorithms using them.

In this research project, different methods for solving the TSP are presented. The next sections of this chapter review the history of the TSP and its relation to combinatorial problems. In the following chapter, the TSP is mathematically defined and three extensions of the standard case are described. Metaheuristic methods for solving the TSP are discussed in Chapter 3. In addition, another method based on a machine learning approach is introduced in Chapter 4. Combining machine learning and heuristic methods leads to a special approach called supervised meta-learning, which is described in Sec. 4.6. After that, the general idea of hyper-heuristics is introduced in Chapter 5. Finally in Chapter 6, a summary is presented and a conclusion is drawn.

## 1.1 History of the Traveling Salesman Problem

The Traveling Salesman Problem (TSP) describes a salesman who travels through cities. The order in which he is visiting the cities is not important, as long as he is able to visit each city exactly once and the last city coincides with the starting city. All cities are connected with each other by some kind of weighted links. The weight can be a distance or a cost of traveling between the connected cities. The goal of the salesman is to minimize the distance or the cost of travel [3].
This type of problem was firstly mentioned in 1832 in a handbook for traveling salesmen with the title *'Der Handlungsreisende – wie er sein soll und was er zu thun hat, um Aufträge zu erhalten und eines glücklichen Erfolgs in seinen Geschäften gewiß zu sein – von einem alten Commis-Voyageur'*. It includes examples for tours through Germany and Switzerland, but without any mathematical treatment of the tour problem. One of the examples is shown in Fig. 1.1.
The mathematics behind this problem was formulated in the 1800's by the Irish mathematician W. R. Hamilton and the British mathematician T. Kirkman for Hamilton's Icosian Game [4]. As illustrated in

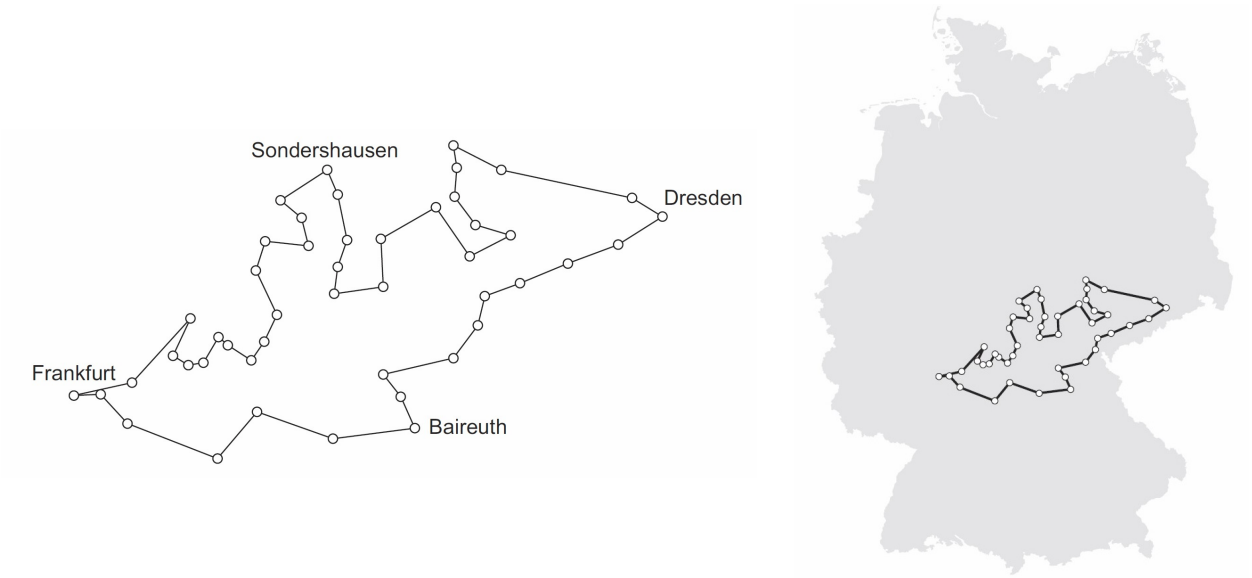
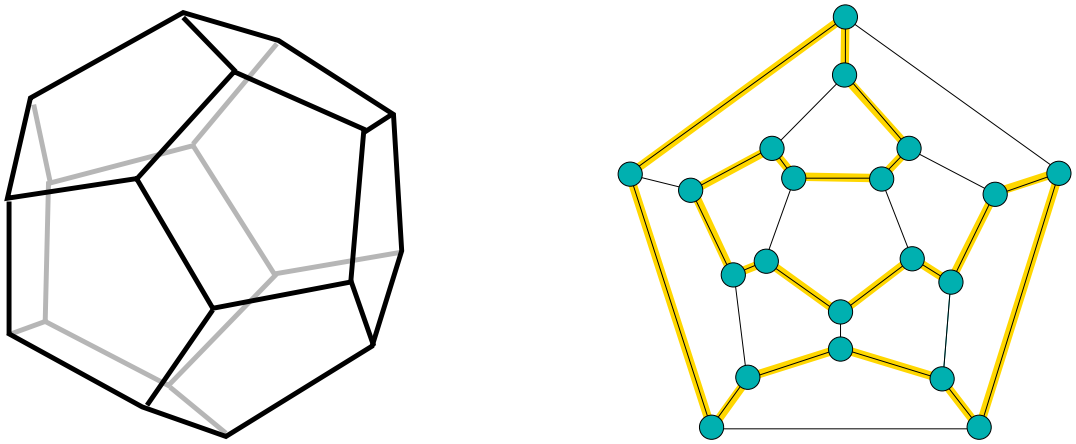



**Figure 1.1:** The 47-city tour from the Commis-Voyageur in Germany [4].

Fig. 1.2, it is a puzzle based on finding a Hamiltonian cycle along the edges of a dodecahedron (any polyhedron with twelve flat faces) such that every node is visited a single time and the ending point is the same as the starting point [5]. The explanation of a Hamiltonian cycle is part of the mathematical definition of the TSP in Sec. 2.1.



**(a)** dodecahedron

**(b)** example Hamiltonian cycle in the 2D-space

**Figure 1.2:** Icosian Game puzzle.

During the 1930's, mathematicians in Vienna and at Harvard University, notably K. Menger, studied the general form of the TSP and defined the problem mathematically. H. Whitney at Princeton University introduced the name *'Traveling Salesman Problem'* soon after [4]. In the following decades, the problem became increasingly popular, because of its relation to many other combinatorial problems, which will be introduced in Sec. 1.2.
As described before, the number of possible solutions grows more than exponentially with an increasing number of nodes. Nevertheless, researchers try to obtain the optimal solution for larger and larger TSP instances. The history of records for the optimal tour is shown in Table 1.1. Exponentially increasing computation power helps finding an optimal tour for a larger set of cities.

| Year | Research Team | Size of the TSP |
|---|---|---|
| 1954 | Dantzig, Fulkerson and Johnson | 49 nodes |
| 1971 | Held and Karp | 64 nodes |
| 1975 | Camerini, Fratta and Maffioli | 67 nodes |
| 1977 | Grötschel | 120 nodes |
| 1980 | Crowder and Padberg | 318 nodes |
| 1987 | Padberg and Rinaldi | 532 nodes |
| 1987 | Grötschel and Holland | 666 nodes |
| 1991 | Padberg and Rinaldi | 2392 nodes |
| 1995 | Applegate, Bixby, Chvátal and Cook | 7397 nodes |
| 1998 | Applegate, Bixby, Chvátal and Cook | 13509 nodes |
| 2001 | Applegate, Bixby, Chvátal and Cook | 15112 nodes |
| 2004 | Applegate, Bixby, Chvátal, Cook and Helsgaun | 24978 nodes |
| 2007 | Cook, Espinoza and Goycoolea | 33810 nodes |
| 2009 | Applegate, Bixby, Chvátal, Cook, Espinoza, Goycoolea and Helsgaun | 85900 nodes |

**Table 1.1:** Milestones in the optimal solution for the TSP [17].

Nowadays, the challenge is to find a solution, which is as close as possible to the calculated lower bound of the TSP. The world tour, a $1,904,711$-city instance of locations throughout the world, has the current best lower bound of length $7,512,218,268$. The current shortest tour was found by K. Helsgaun in March 2018. A comparison to the lower bound reveals that K. Helsgaun's tour has a length which is only 0.0474% greater than the length of the optimal tour [6].

As seen by the description of the TSP, the main goal is to obtain a short or cheap tour by altering the order in which the cities or nodes are visited. Thus, the TSP belongs to the group of combinatorial problems, which will be discussed in the following.

## 1.2 Related Combinatorial Problems

One group of combinatorial problems are routing problems. In literature, this type of problem is often called Package Delivery, School Bus Routing or Vehicle Routing Problem. It can be described as the problem of designing optimal delivery or collection routes from one or multiple depots to a number of geographically scattered cities or customers [7]. For example, there exist several school buses which have to pick up students in each suburb. An illustration for these types of problems is given in Fig. 1.3. In addition to the combinatorial part of this optimization problem, the following additional constraints can be introduced to fit the model to a particular scenario [8]:

- The number of routes can be minimized.
- The total cost can be minimized.
- The total distance traveled by all buses is kept at minimum.
- No bus is overloaded.
- The time required to traverse any route does not exceed a maximum allowed policy.
- The length of the routes should be similar.
- The route should be robust against traffic jam.

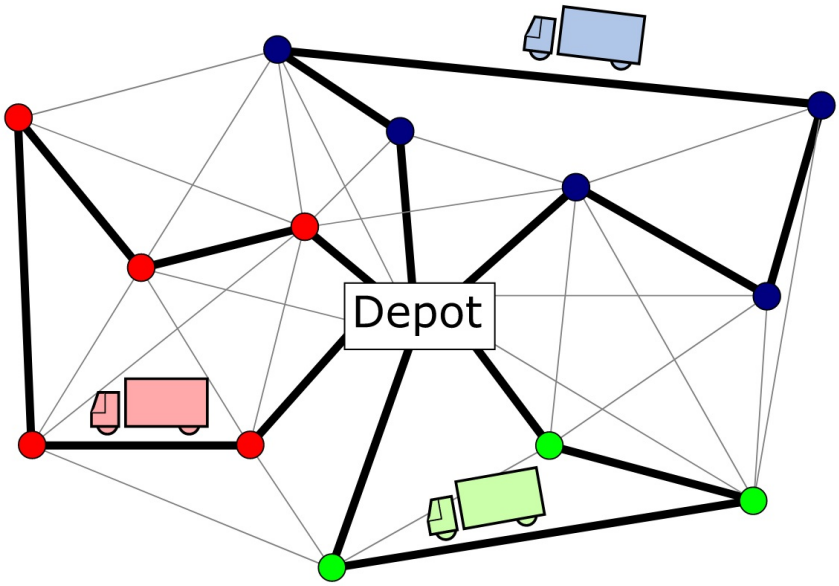



**Figure 1.3:** Example Vehicle Routing Problem.

As described before, combinatorial problems can also be modeled as a TSP. In this case, Vehicle Routing Problems can be modeled as multiple TSPs. This extension of the standard TSP is presented in Sec. 2.2.2.

Another group of combinatorial problems are scheduling problems. These problems include the scheduling of resources such as machinery (job shop scheduling), labor (crew scheduling) and timetabling. For example, airline crew scheduling has received much attention from researchers, because it can be formulated as a set partitioning problem [9]. Scheduling involves the assignment of crew members to jobs in such a way that the correct number of people are scheduled at the right time, performing the right tasks with the right ability.
In job shop scheduling, each job in a given schedule has a varying processing time on each machine. The task is to minimize the total time or set-up costs used to complete all jobs. If a job has to be worked off in a given order on different machines, precedence conditions have to be implemented for each step. These scheduling problems can also be modeled as a TSP. For example, the set-up costs of a machine can be seen as the distances between the nodes (products). In order to reflect the precedence conditions, the standard TSP can be extended to the TSP with Precedence Conditions (TSPPC) as shown in Sec. 2.2.3.

One more common example for a combinatorial problem is the Knapsack Problem. It belongs to the class of Resource Allocation Problems. A knapsack has a prescribed capacity. A set of items is given where each item has a weight and a value. The task is to determine which items should be taken in order to maximize the value, but without exceeding the limited capacity of the knapsack.
Resource Allocation Problems can be modeled as a TSP with Profits [10]. This is a generalization of the standard TSP, where it is not necessary to visit all nodes. Each node has an associated profit and the overall goal is the simultaneous optimization of the collected profit and the travel costs. These two optimization criteria appear either in the objective function or as a constraint. Overall, three different generic problems can be modeled using a TSP with Profits, depending on the way in which the two objectives of the problem are addressed:

1. Both objectives are combined in the objective function. The aim is to find a route that minimizes travel costs minus collected profit. In literature, this problem is called the Profitable Tour Problem (PTP) [11].

2. The travel cost objective is stated as a constraint. The goal is to find a route that maximizes the collected profit such that the travel costs do not exceed a preset value. In literature, this problem is called the Orienteering Problem (OP) [12].

3. The profit objective is stated as a constraint. The goal is to find a route that minimizes travel costs and where the collected profit is not smaller than a preset value. In literature, this problem is called the Prize Collecting TSP (PCTSP) [13].

The Knapsack Problem itself can be modeled as a Prize Collecting TSP. The salesman travels between pairs of cities (items) at a cost depending only on the pair (weight), gets a prize (value) in every city he visits and pays a penalty to every city that he fails to visit. He wants to minimize the costs and penalties while visiting enough cities to collect a prescribed amount of prize money.

In the next Chapter 2, standard and extended TSPs are mathematically defined and methods finding the optimal solution are presented.

# 2 Traveling Salesman Problem (TSP)

## 2.1 Symmetric TSP

In the following, we will adopt the terminology as defined by Míča in [14], La Maire et al. in [15] and Potvin in [16]. The main issue in TSPs is to find a Hamiltonian cycle. That means to visit all nodes or cities of a graph exactly once with the shortest possible route and return to the origin node.
This problem can be described as a graph $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{1, 2, \ldots, N\}$ is the set of nodes and $\mathcal{E} = \{\langle i, j\rangle : i, j \in \mathcal{V}\}$ is the set of edges or paths between the nodes and each edge has its own length. Each city corresponds to a particular node $v \in \mathcal{V}$ and each edge $\langle i, j\rangle$ connects nodes $i$ and $j$. The distance between node $i$ and node $j$, using the corresponding edge, is represented by $d_{ij}$. A TSP is symmetric if $d_{ij} = d_{ji}$ for all edges $\langle i, j\rangle$; otherwise it is asymmetric [17].
The network is a complete graph when each pair of nodes is connected by an edge. When the graph is not complete, fictive edges with infinite length can be added between two unconnected nodes without affecting the optimal solution [14]. In addition to the graph representation, we can formulate an Integer Optimization Problem [15]:

$$\text{minimize} \quad \sum_i \sum_j d_{ij} x_{ij} \tag{2.1a}$$

$$\text{subject to} \quad \sum_j x_{ij} = 1 \qquad \forall i = 1, \ldots, N \tag{2.1b}$$

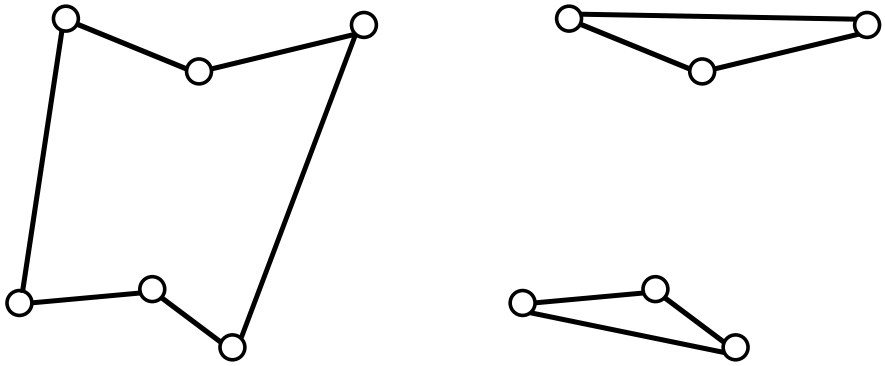
$$\sum_i x_{ij} = 1 \qquad \forall j = 1, \ldots, N \tag{2.1c}$$

$$\sum_{i,j \in \mathcal{S}_\mathcal{V}} x_{ij} \leq |\mathcal{S}_\mathcal{V}| - 1 \qquad (\mathcal{S}_\mathcal{V} \subset \mathcal{V};\ 2 \leq |\mathcal{S}_\mathcal{V}| \leq N - 2) \tag{2.1d}$$

$$x_{ij} = \{0, 1\} \qquad i, j = 1, \ldots, N \quad i \neq j \tag{2.1e}$$

The first Eq. (2.1a) describes the linear objective function, which has to be minimized. The binary variable $x_{ij}$ is a decision variable that has a value of 0 if the edge from city $i$ to city $j$ is not used and 1 if the edge is used. The constraints in Eq. (2.1) are added to ensure that the solution forms a valid tour.
Constraints (2.1b) and (2.1c) reflect that every city is entered and left once. Moreover, no fraction of an edge can be added to the solution. For this reason, $x_{ij}$ has to be an integer value as shown in Eq. (2.1e). Without any other constraints, the solution of the Integer Optimization Problem shown in Eq. (2.1) can contain subtours. This means that every node is visited once, but there is more than one tour.
Our goal is to find the optimal tour including all nodes. Thus, we have to add another constraint shown in Eq. (2.1d). It ensures that no subtours are formed and that the Hamiltonian cycle is only a single tour. In Eq. (2.1d), $\mathcal{S}_\mathcal{V}$ is defined as some subset of $\mathcal{V}$. Then, $|\mathcal{S}_\mathcal{V}|$ is the cardinality of $\mathcal{S}_\mathcal{V}$. The cardinality of a set is defined as a measure of the number of elements in the set. Constraint (2.1d) prohibits subtours with less than $N$ nodes leading to a single large tour. If there exists a subtour on some subset of nodes $\mathcal{S}_\mathcal{V}$, this subtour would contain $|\mathcal{S}_\mathcal{V}|$ edges. Consequently, the left-hand side of Eq. (2.1d) would be equal to $|\mathcal{S}_\mathcal{V}|$, which is greater than $|\mathcal{S}_\mathcal{V}| - 1$, meaning that the constraint is violated for this particular subset.

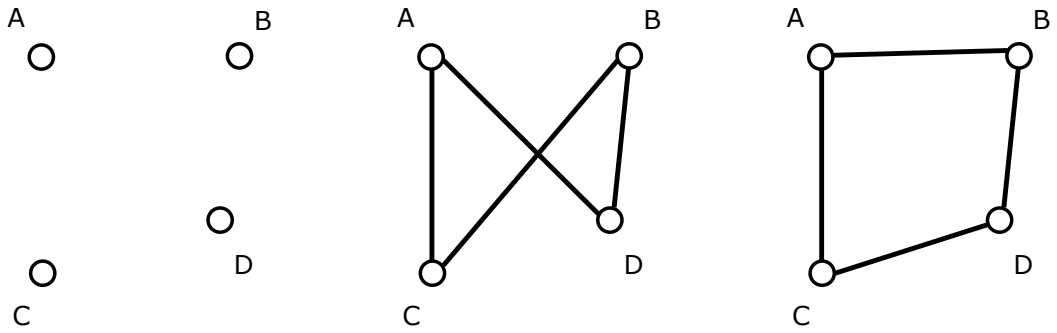
Without the subtour-breaking constraint, the TSP reduces to an Assignment Problem [16]. The solution of a TSP and an Assignment Problem with the same 6-city example is shown in Fig. 2.3. As described before, the solution of the Assignment Problem would violate the subtour constraint.



**(a)** TSP **(b)** Assignment Problem

**Figure 2.1:** Example 6-city problem [16].

The main characteristic of an optimal solution for the euclidean symmetric TSP is shown in Fig. 2.2, where a 4-city example is considered.




**(a)** Input **(b)** Non-optimal tour **(c)** Optimal tour

**Figure 2.2:** Example 4-city TSP [3].

An optimal tour contains no intersections of edges, because of the triangle inequality shown in Eq. (2.2) with distance $D(A,B)$ between two cities $A$ and $B$ [18].

$$D(A,B) + D(C,D) \;\leq\; D(A,D) + D(B,C) \tag{2.2}$$

As described before, the TSP is hard to solve, because of its combinatorial structure. With every additional city, the number of possible solutions grows more than exponentially. Considering a TSP with 4 cities, a salesman located in the starting city has the choice to travel to one of the 3 remaining cities. In the next city, the salesman can choose out of 2 cities and so on. For a TSP with 4 cities, there exist $3! = 3 \cdot 2 \cdot 1 = 6$ possible tours. This is a factorial growth of possible solutions.
In the symmetric case, there is no difference if the salesman chooses tour A→B→C→D→A or A→D→C→B→A, because the tour length is equal. That is why the number of possible tours has to be divided by 2. In general, the number of possible tours for a symmetric TSP with $N$ cities is

$$\frac{(N-1)!}{2}. \tag{2.3}$$

In the next Sec. 2.2, variations of the TSP are presented to reflect different combinatorial problems.

## 2.2 Extended TSP

The symmetric TSP is widely used to represent the class of combinatorial problems. New optimization algorithms are tested on several TSP instances to evaluate the performance and to compare the results to other algorithms. Unfortunately, there exist combinatorial problems, which cannot be transformed into a symmetric TSP. These problems require an extended TSP to represent their characteristics.

### 2.2.1 Asymmetric TSP (ATSP)

As described in Sec. 2.1, a TSP can be asymmetric, meaning that the distance from city $A$ to city $B$ can be different than the distance in the opposite direction from city $B$ to city $A$. For example, there could be a one way street in the euclidean TSP which shortened the distance between two cities, but only when traveling from city $A$ to $B$.
This extension leads to a higher complexity than in the symmetric case. There exist $(N-1)!$ possible tours, which is in sum twice as much as in the symmetric case, because every tour can be passed in two directions. In order to use optimization algorithms or heuristics, which were originally designed for the symmetric case, an asymmetric TSP can be transformed into a symmetric TSP [19].
Let TSP($D$) denote a TSP with distance matrix $D$ and its elements $d_{ij}$, $i,j \in \mathcal{V}$ as defined in Sec. 2.1. Then, we define the distance matrix $\bar{D}$ to be equal to $D$ except $\bar{d}_{ii} = -M$ $(i \in \mathcal{V})$, where $M$ is an arbitrary large number. In addition, we define matrix $U$ with elements $u_{ij} = \infty$ for $i,j \in \mathcal{V}$. Combining these two matrices, we obtain matrix $\tilde{D}$ as follows [19]:

$$\tilde{D} = \begin{bmatrix} U & \bar{D}^T \\ \bar{D} & U \end{bmatrix} \tag{2.4}$$

We will call this TSP($\tilde{D}$). The set of cities for TSP($\tilde{D}$) is $\{1,2,\ldots,N,N+1,\ldots,2N\}$. The optimal solutions for TSP($\tilde{D}$) have finite values and contain $N$ edges of distance $-M$. With $\tilde{D}$ being symmetric, the solutions occur in pairs as shown in Eq. (2.5) to reflect the asymmetric distances between two cities [19]. An optimization algorithm will include edges $d_{i,i+N}$ $i \in \mathcal{V}$ with negative distance $-M$, because our goal is to minimize the sum distance.

$$v_1 \to (v_1+N) \to v_2 \to (v_2+N) \to \cdots \to v_N \to (v_N+N) \to v_1 \qquad v_k \in \mathcal{V} \text{ for } k = 1,2,\ldots,N \tag{2.5}$$

Clearly, there is a one-to-one correspondence between the set of TSP($\tilde{D}$) solutions and solutions of TSP($D$). One must delete the cities greater than $N$ from TSP($\tilde{D}$) and add $N \cdot M$ to its value in order to obtain the corresponding solution for TSP($D$) and its value. A numerical example of this transformation including distance matrices $D$ and $\tilde{D}$ with $M = 10$ is shown in Eq. (2.6).

$$D = \begin{bmatrix} \infty & 1 & 2 \\ 6 & \infty & 3 \\ 5 & 4 & \infty \end{bmatrix} \qquad \tilde{D} = \begin{bmatrix} \infty & \infty & \infty & -10 & 6 & 5 \\ \infty & \infty & \infty & 1 & -10 & 4 \\ \infty & \infty & \infty & 2 & 3 & -10 \\ -10 & 1 & 2 & \infty & \infty & \infty \\ 6 & -10 & 3 & \infty & \infty & \infty \\ 5 & 4 & -10 & \infty & \infty & \infty \end{bmatrix} \tag{2.6}$$

Summing up, this transformation is important and often used, because most optimization algorithms are designed for the symmetric case. That is why the experimental study of heuristics for the ATSP is much less advanced [20].

### 2.2.2 Multiple TSP (mTSP)

The second variation of the original symmetric TSP is called multiple TSP (mTSP). It includes more than one salesman and leads to a more complex problem. Some of the combinatorial problems described in Sec. 1.2 can be modeled as a mTSP. For example, Crew Scheduling Problems and School Bus Routing Problems belong to the group of mTSPs. In general, the constraints for the mTSP optimization task are as follows [21]:

1. All salesmen start in the same city.
2. Each salesman returns to the starting city at the end of a tour.
3. Each and every salesman must travel to a particular set of cities.
4. Each city is visited by exactly one salesman, except the first city.

The idea is to obtain a tour with minimal distance, where each salesman has to travel from the starting location to individual cities and back to the beginning. Thus, the total cost of visiting all cities is lowered. Including more than one salesman leads to an additional task of assigning the cities to each salesman. For this reason, the optimization problem is more complex. In addition to the standard case, there exist several variations of the problem as follows:

- **Multiple depots:** If there exist multiple depots, the salesman at each depot remains the same during and after traveling.
- **Fixed cost:** If the number of salesmen is a limited variable, the usage of each salesman in the solution has an associated fixed cost. In this case, the minimization of this cost can be included in the optimization task.
- **Other restrictions:** For example, a travel distance limit for each salesman can be introduced.

Similar to other TSPs there exist heuristic, meta-heuristic and exact algorithms to solve the mTSP. In order to obtain the Integer Optimization Problem formulation for the mTSP, we have to extend the optimization problem of the standard TSP shown in Eq. (2.1) with two new constraints as follows [8]:

$$\text{minimize} \quad \sum_i \sum_j d_{ij} x_{ij} \tag{2.7a}$$

$$\text{subject to} \quad \sum_{j=2} x_{1j} = m \tag{2.7b}$$

$$\sum_{j=2} x_{j1} = m \tag{2.7c}$$

$$\sum_j x_{ij} = 1 \qquad \forall i = 1, \ldots, N \tag{2.7d}$$

$$\sum_i x_{ij} = 1 \qquad \forall j = 1, \ldots, N \tag{2.7e}$$

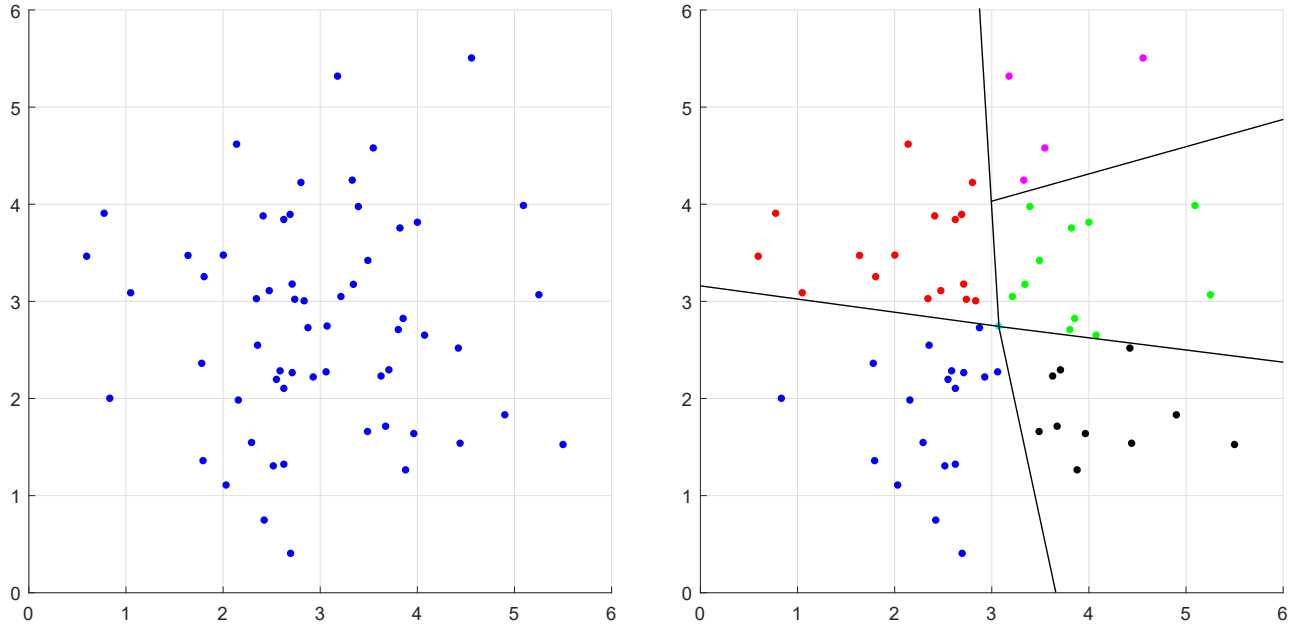
$$\sum_{i,j \in \mathcal{S}_\mathcal{V}} x_{ij} \leq |\mathcal{S}_\mathcal{V}| - 1 \qquad (\mathcal{S}_\mathcal{V} \subset \mathcal{V};\ 2 \leq |\mathcal{S}_\mathcal{V}| \leq N-2) \tag{2.7f}$$

$$x_{ij} = \{0, 1\} \qquad i, j = 1, \ldots, N \quad i \neq j \tag{2.7g}$$

Constraints (2.7d)-(2.7g) are equal to the Integer Optimization Problem constraints for the standard TSP shown in Eq. (2.1). To ensure that exactly $m$ salesmen depart from and return back to the initial starting city 1, constraints (2.7b) and (2.7c) have to be added.

The mTSP can be transformed to a standard symmetric TSP as well. Bellmore et al. showed in [22] that an asymmetric mTSP with $m$ salesmen and $N$ cities can be transformed into an asymmetric TSP with $(N + m - 1)$ cities. Based on this approach, Rao showed in [23] that the symmetric mTSP can be transformed into a standard symmetric TSP with $(N + m - 1)$ cities as well. This is obvious, because the symmetric TSP is a relaxed version of the asymmetric TSP. In both cases, including more than one salesman increases the number of possible tours by $m!$ to $(N + m - 1)!$. In addition, Guoxing showed in [24] that a TSP with $m$ salesmen and $q$ depots can be transformed to a standard TSP.

In the following, an approach to reduce the number of possible tours is described. Decomposing the mTSP to $m$ standard TSPs reduces the number of possible tours to $m \cdot (N-1)!$. In order to split the mTSP in $m$ standard TSPs, Nallusamy et al. and Sze et al. used an unsupervised machine learning algorithm called K-Means Clustering [25, 26]. An overview of the different types of machine learning algorithms is presented in Chapter 4. To sum up, an unsupervised learning algorithm tries to find patterns in given data. The result are clusters of data with similar structure. In the euclidean space, nearby cities would be clustered and therefore labeled differently in comparison to cities which are far away. K-Means Clustering divides a given map with cities into $K$ districts, where $K$ could be the number of salesmen. Fig. **??** shows an example map with 60 cities and the resulting districts after applying the K-Means Clustering algorithm [25].




**(a)** No clustering applied

**(b)** K-Means Clustering applied

**Figure 2.3:** Example 60-city map before and after applying K-Means Clustering.

### 2.2.3 TSP with Precedence Conditions (TSPPC)

Another extension of the standard TSP is called TSP with Precedence Conditions (TSPPC) [28]. It introduces an order in which the nodes or cities should be visited. Industrial problems such as scheduling, routing decision and process sequencing can be modeled as a TSPPC as described in Sec. 1.2. The TSPPC is harder to solve than the standard TSP, because the model formulations are more complex and the algorithms for solving these models are difficult to implement. That is why heuristics are used to obtain a close-to-optimal solution in a short period of time.

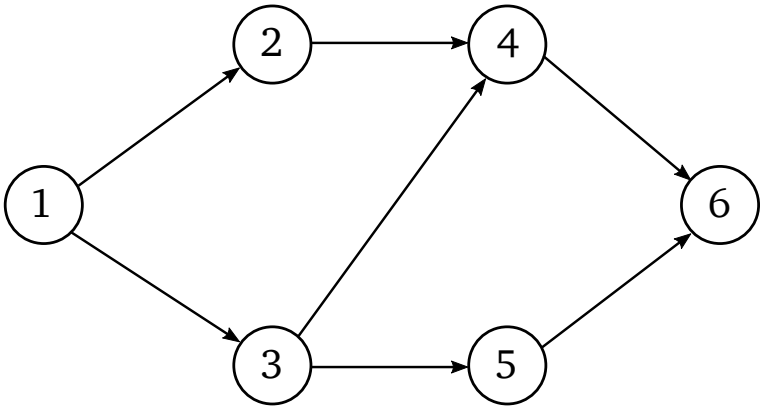



**Figure 2.4:** Example of a directed graph.

The precedence conditions can be modeled using a directed graph in addition to the graph representation of the standard TSP. This standard graph representation $G(\mathcal{V}, \mathcal{E})$ defined in Sec. 2.1 is extended with precedence constraints $\prec$ on $\mathcal{V}$ to obtain the directed graph. The precedence condition $i \prec j$ indicates that node $i$ must be visited before node $j$. An example of a directed graph representing the precedence constraints is shown in Fig. 2.4. Obviously, the directed graph must be acyclic, otherwise no feasible solution exists. In the directed graph scenario, a tour is complete and feasible if the tour includes all cities and does not violate the precedence constraints [27]. In order to find an optimal Hamiltonian cycle, both precedence constraints and minimization of the distance have to be considered.

In order to derive the Integer Optimization Problem for the TSPPC, we have to add all precedence constraints to the Integer Optimization Problem for the standard TSP defined in Eq. (2.1). An example precedence constraint $q \prec r$ can be represented as follows [28]:

1. Set formulation: For all $\mathcal{S} \subseteq \mathcal{V}$, $q \in \mathcal{S}$, $1 \in \mathcal{S}$, $r \notin S$, there exists an edge from $S - \{1\}$ to $\mathcal{V} - \mathcal{S}$ in the tour as shown in Fig. 2.5.

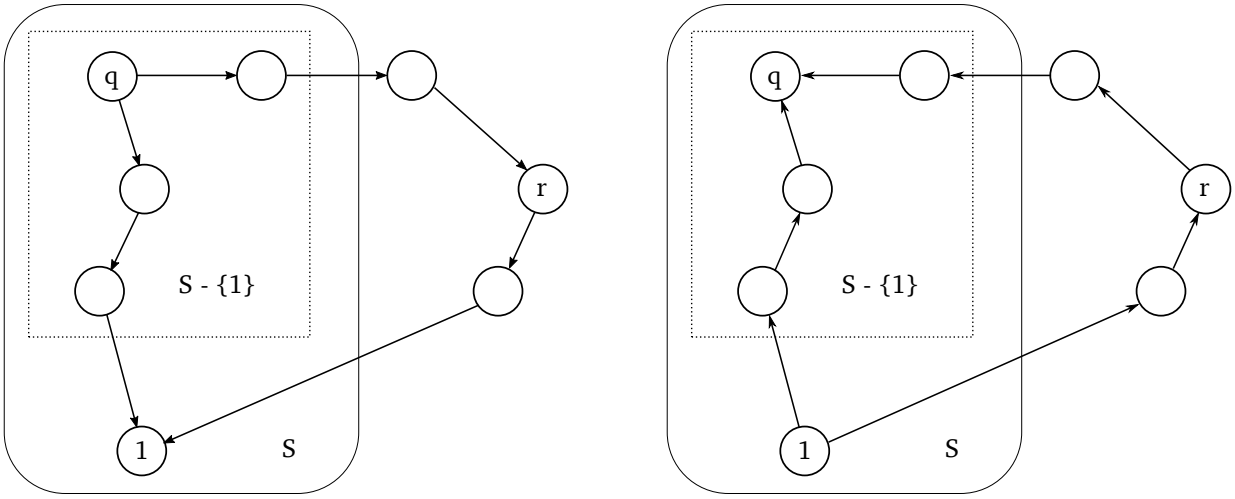
2. Path formulation (this follows from the precedence condition $q \prec r$):
   a) Any path in the feasible tour with starting node $r$ and ending node $q$ must contain node 1.
   b) Any path in the feasible tour with starting node 1 and ending node $r$ must contain node $q$.
   c) Any path in the feasible tour with starting node $q$ and ending node 1 must contain node $r$.

3. Articulation node formulation (a node in an undirected connected graph is an articulation node, if the graph is disconnected when removing this node and all the edges through it):
   a) In the feasible tour, the path with starting node 1 and ending node $q$ and the path with starting node $q$ and ending node $r$ do not intersect each other.

b) In the feasible tour, the path with starting node 1 and ending node $q$ and the path with starting node $r$ and ending node 1 do not intersect each other.

c) In the feasible tour, the path with starting node $q$ and ending node $r$ and the path with starting node $r$ and ending node 1 do not intersect each other.

The derivation for the articulation node formulation and the final integer programming representations for all formulations are shown in [28].




**(a)** feasible tour **(b)** infeasible tour

**Figure 2.5:** Derivation of the set formulation [28].

Before presenting metaheuristics to approximate a solution for the TSP in Chapter 3, approaches to obtain the optimal solution are described in the next Sec. 2.3.

## 2.3 Exact Algorithms

As described in Chapter 1, the TSP belongs to the group of $\mathcal{NP}$-complete problems. Therefore, obtaining the optimal solution becomes more and more difficult for an increasing number of cities. This can be shown by applying a full tree search to a TSP.
The length of all different permutations (all possible tours) can be calculated to evaluate which one is the shortest. This brute force search requires lots of resources and becomes impractical even for a TSP with a small number of cities, because the number of possible solutions $(n-1)!$ increases factorial as described in Sec. 2.1. To overcome this problem, advanced techniques have to be applied. For the definition of all exact algorithms, we will adopt the terminology as defined by Hahsler et al. in [29].

### 2.3.1 Dynamic Programming

In 1962, Held et al. presented the first Dynamic Programming formulation for the TSP in [30]. Using this approach, larger TSPs can be solved exactly.

Given a subset of cities (excluding the first city) $\mathcal{S} \subset \{2, 3, \ldots, N\}$ and $l \in \mathcal{S}$, let $d^*(\mathcal{S}, l)$ denote the length of the shortest path from city 1 to city $l$, visiting all cities in $\mathcal{S}$ in-between. For $\mathcal{S} = \{l\}$, $d^*(\mathcal{S}, l)$ is defined as $d_{1l}$. The shortest path for larger sets with $|\mathcal{S}| > 1$ can be written as:

$$d^*(\mathcal{S}, l) = \min_{m \in \mathcal{S} \setminus \{l\}} \left( d^*(\mathcal{S} \setminus \{l\}, m) + d_{ml} \right) \tag{2.8}$$

In addition, the minimal tour length $d^{**}$ for a complete tour, which includes returning to city 1, is

$$d^{**} = \min_{l \in \{2,3,\ldots,N\}} \left( d^*(\{2, 3, \ldots, N\}, l) + d_{l1} \right). \tag{2.9}$$

Using Eq. (2.8) and Eq. (2.9), the quantities $d^*(\mathcal{S}, l)$ can be computed recursively and the minimal tour length $d^{**}$ can be found. In a second step, the optimal permutation $\pi = \{1, i_2, i_3, \ldots, i_N\}$ of cities 1 through $N$ can be computed in reverse order, starting with $i_N$ and working successively back to $i_2$. This procedure exploits the fact that the permutation $\pi$ can only be optimal, if

$$d^{**} = d^*(\{2, 3, \ldots, N\}, i_N) + d_{i_N 1}, \tag{2.10}$$

and, for $2 \leq p \leq N - 1$,

$$d^*(\{i_2, i_3, \ldots, i_p, i_{p+1}\}, i_{p+1}) = d^*(\{i_2, i_3, \ldots, i_p\}, i_p) + d_{i_p i_{p+1}}. \tag{2.11}$$

A disadvantage of this approach is the space complexity of storing the values for all $d^*(\mathcal{S}, l)$, which is $(N-1)2^{N-2}$. This severely restricts the Dynamic Programming algorithm to smaller TSPs. However, for these TSP instances the approach is fast and efficient [29].

### 2.3.2 Linear Programming

In order to solve larger TSP instances, Linear Programming (LP) can be used. The formulation of the TSP as a Linear Program Optimization Problem is shown in Eq. (2.12) [31]. It is equal to the Integer Optimization Problem for the symmetric TSP shown in Eq (2.1) but formulated in vector notation.

$$\text{minimize} \quad \sum_{i=1}^{E} w_i x_i = \boldsymbol{w}^T \boldsymbol{x} \tag{2.12a}$$

$$\text{subject to} \quad \boldsymbol{x} \in \mathcal{S} \tag{2.12b}$$

The number of edges $e_i$ in a TSP graph $G(\mathcal{V}, \mathcal{E})$ is denoted by $E$ and $w_i \in \boldsymbol{w}$ is the weight of edge $e_i$. Vector $\boldsymbol{x}$ indicates the presence or absence of each edge in the tour. The task is to find the shortest path through all nodes in $G$. This is reflected by the objective function shown in Eq. (2.12a).
The constraints given by Eq. (2.12b) are problematic, because $\boldsymbol{x} \in \mathcal{S}$ contains the set of vectors $\boldsymbol{x}$ of all possible Hamiltonian cycles in $G$. This amounts to a direct search of all $(n-1)!$ possibilities, which is impractical for larger TSPs. However, relaxed versions of the Linear Programming problem including sub-tour elimination constraints can be used to overcome this problem.

The first algorithm solving TSP based on Linear Programming was presented by Dantzig et al. in [31]

and is called cutting-plane. The algorithm is based on an iterative approach where sub-optimal solutions are left over in the next step to reduce the search space and the number of possible solutions. At the beginning of each iteration, the original linear inequality description $\mathcal{S}$ is replaced by the relaxation $\boldsymbol{Ax} \leq \boldsymbol{b}$, where the polyhedron $P$ defined by the relaxation contains $\mathcal{S}$ and is bounded. The optimal solution $\boldsymbol{x}^*$ of the relaxed problem can be obtained using standard Linear Programming solvers.
If $\boldsymbol{x}^*$ belongs to $\mathcal{S}$, the optimal solution of the original problem is obtained. Otherwise, a linear inequality can be found, which is satisfied by all points in $\mathcal{S}$ but violated by $\boldsymbol{x}^*$. Such an inequality is called a cutting plane or cut. A family of cutting planes can be added to the inequality system $\boldsymbol{Ax} \leq \boldsymbol{b}$ to obtain a tighter relaxation for the next iteration. The iterative algorithm is performed until a valid solution in $\mathcal{S}$ is found.

### 2.3.3 Branch and Bound

The well-known Branch and Bound search method has its origins in work on the TSP and has become the most used method for solving $\mathcal{NP}$-hard optimization problems [4]. The Branch and Bound method describes an organized exhaustive search for the best solution in a specified set. It uses a tree representation of the search space with the root node representing the full set.
During the branching step, the algorithm splits the search space into two or more subsets in an attempt to create subproblems that may be easier to solve than the original problem. Before searching a subproblem and possibly splitting it further, a bound is computed on the cost of tours in its subset. The branch is checked against these upper and lower estimated bounds on the optimal solution. It is discarded if it does not lead to a better solution than the best one found so far by the algorithm. Performing this bounding step, the number of possible tours is reduced.
As a consequence, applying the Branch and Bound algorithm can solve large TSPs relatively fast. The latest world records in exact solutions for TSPs shown in Sec. 1.1 are obtained using Linear Programming combined with a modified Branch and Bound algorithm.
Solving even larger TSPs is only possible trying to approximate the optimal solution. The so-called heuristics are described in the next Sec. 2.4.

## 2.4 Heuristics

Heuristics are methods finding an approximately optimal solution for a given problem in a shorter period of time than an exact algorithm. The solution can be sub-optimal, but in many cases there is no need for the optimal solution. Instead, it is sufficient to approximate the optimal solution. As presented in Sec. 1.1, the best heuristic for solving large TSP instances has a tour length being only 0.0474% larger than the length of the optimal tour. In order to find an adequate solution in desired time, heuristics have to trade-off between the following criteria [38]:

- **Optimality:** Optimal or desired solution will be found?
- **Completeness:** All solutions will be found, if there exist more than one?
- **Accuracy:** How is the quality of the solution compared to an optimal algorithm?
- **Execution time:** Higher convergence speed compared to other methods?

Generally, heuristics for the TSP can be divided in two groups: tour construction heuristics which create tours from scratch and tour improvement heuristics which use simple local search heuristics to improve existing tours [29]. In the next Sec. 2.4.1, tour construction heuristics are presented.

### 2.4.1 Tour Construction Heuristics

A common tour construction heuristic for TSPs is the Nearest Neighbor algorithm [32]. It follows a simple and greedy procedure. At the beginning, a random city is chosen as the starting city. Then, the nearest not yet visited city is added to the tour until the tour connects all cities.
An extension of this algorithm is to construct different tours by choosing other starting cities. The final solution would be the shortest tour. This extension is called Repetitive Nearest Neighbor.

Another group of tour construction heuristic is called Insertion Algorithms [32]. Each algorithm starts with a tour including a subset of all cities and inserts the remaining cities one after another. In each step, a remaining city $k$ is inserted into the existing tour between two consecutive cities $i$ and $j$ in a way that the insertion cost

$$I = d(i,k) + d(k,j) - d(i,j) \tag{2.13}$$

is minimized. Insertion Algorithms differ in the way a remaining city $k$ is chosen. There exist four different approaches:

- **Nearest insertion:** City $k$ is chosen as the city which is nearest to a city in the tour.
- **Farthest insertion:** City $k$ is chosen as the city which is farthest from any of the cities in the tour.
- **Cheapest insertion:** City $k$ is chosen as the city which minimizes the cost of inserting.
- **Arbitrary insertion:** City $k$ is chosen randomly from all cities not yet in the tour.

Obviously, the tours found by tour construction heuristics are sub-optimal. On the other hand, this type of heuristics can be used as an initial solution for tour improvement heuristics, which are presented in the next Sec. 2.4.2.

### 2.4.2 Tour Improvement Heuristics

Tour improvement heuristics are simple local search heuristics which try to improve an initial tour. These heuristics are characterized by a certain type of basic move to alter the current tour.
A common tour improvement heuristic is the 2-Opt Exchange [33]. It is motivated by the observation that a tour in the euclidean TSP can be easily shortened if it crosses itself. Namely, erase two edges that cross and reconnect the resulting two subtours by edges that do not cross as illustrated in Fig. 2.2. Then, the new tour is shorter than the old one.
In general, a 2-opt move consists of eliminating two edges and reconnecting the two resulting subtours in a different way to obtain a new tour. Improving a tour this way, the two eliminated edges do not necessarily have to cross. Fig. 2.6 shows a 2-opt move involving non crossing edges.

Experimental results show that the 2-opt heuristic using the Nearest Neighbor algorithm as tour construction heuristic leads to relatively good results. This is due to the fact that Nearest Neighborhood tours are locally quite reasonable. The few very sub-optimal edges can be eliminated easily using the 2-opt heuristic [34].
An extension of the 2-opt algorithm is the k-opt algorithm. Typically, a tour $t'$ is a neighbor of another tour $t$, if $t'$ can be obtained from $t$ by deleting $k$ edges and replacing them by a set of different feasible

edges (a k-opt move). In such a structure, a tour can be improved iteratively by always moving from one tour to its best neighbor till no further improvement is possible. The resulting tour represents a local optimum which is called k-optimal [29].

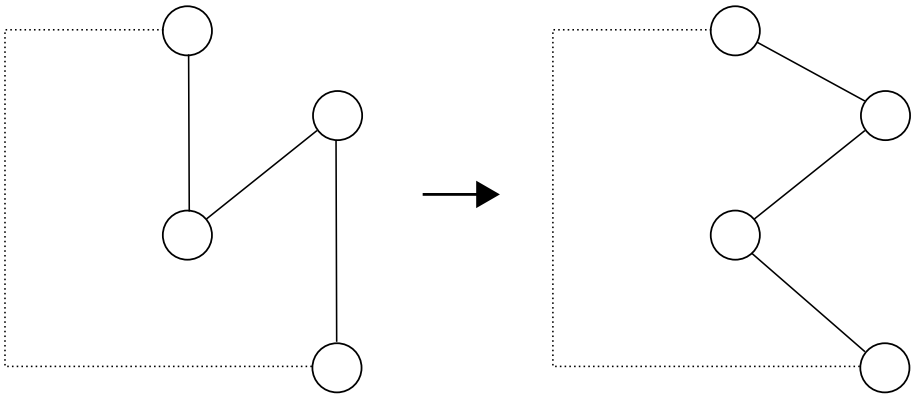


**Figure 2.6:** A 2-opt move [34].

By using full backtracking the optimal solution can always be found. In this case, a trade-off between different criteria has to be made as described in Sec. 2.4. The algorithm has to check nearly all possible solutions which increases the execution time. Therefore, only limited backtracking is reasonable to find better local optima.

Obtaining the optimal solution is hard for heuristics, because these algorithms can get stuck in local optima. In the next Chapter 3, this problem is described and solution approaches in form of metaheuristics are presented.

# 3 Metaheuristics

## 3.1 Characteristics of Metaheuristics

As described in the previous Sec. 2.4, heuristics can be effective to solve TSPs. These approximation algorithms construct an initial solution in a short period of time and improves this solution iteratively. However, a disadvantage is that heuristics have the potential to be far from a global optimum, because heuristics are bounded to a certain local configuration [35]. During the iterative process, each candidate for an improved solution is a neighbor of the previous solution. Therefore, the solution after a new step is also a neighbor of the solution from a previous step. Then it can happen that the heuristic is stuck to a local optimum. Every candidate tour would increase the travel distance meaning that the heuristic would stop after this step. In general, the smaller the interval which is searched for a neighbor solution, the larger the difference between the length of an optimal tour and the tour found by the heuristic.
One way to obtain better performance is to start improvement heuristics many times with different initial tours obtained by construction heuristics, because this increases the chance of finding better local minima [34]. In order to demonstrate this characteristic of heuristics, a simplified one-dimensional search space problem is introduced.

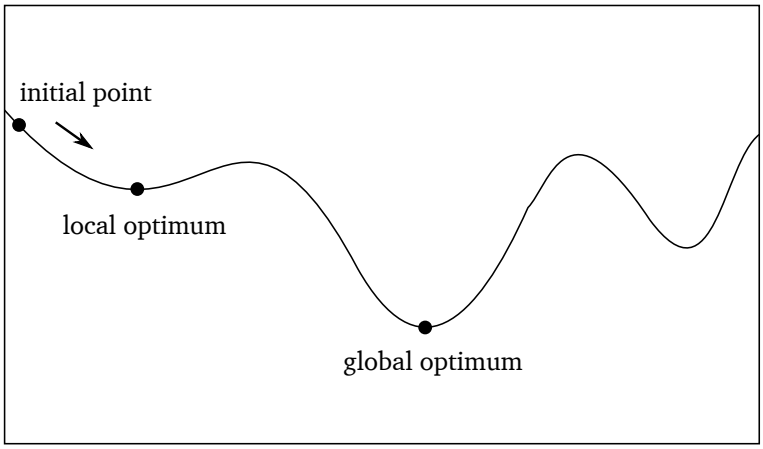



**Figure 3.1:** An example of a local optimum in a simplified, 1-dimensional space [35].

Assume that there is only one global and several local minima in the search space. If we start with the initial point shown in Fig. 3.1 and then look for a better solution by searching the corresponding neighborhood, we can easily be trapped in a local optimum, since the search interval is too small [35]. For this reason, heuristics are also called local search strategies [20].

In order to overcome the problems of heuristics, more general solution schemes can be used. The so-called metaheuristics introduce a class of high-level problem-independent algorithmic frameworks. The term metaheuristics was first used by Glover in 1986 [36]. In comparison to heuristics, which are developed for a certain problem, metaheuristics can be easily applied to new problems as well. This generalized type of heuristics do not put any requirements on the formulation of the optimization problem (such as requiring constraints or objective functions to be expressed as linear functions of the decision variables). However, this flexibility comes at the cost of requiring considerable problem-specific adaptation to achieve good perfomance [37].

There exist several classifications for metaheuristics [38]:

- **Nature inspired versus nonnature inspired:** Many metaheuristics are inspired by natural processes: Evolutionary Algorithms and artificial immune systems from biology; ants, bee colonies, and particle swarm optimization from swarm intelligence.
- **Memory usage versus memoryless methods:** A memoryless metaheuristic does not use dynamically extracted information during the search while other metaheuristics use a memory that contains some information extracted during the search.
- **Deterministic versus stochastic:** Deterministic metaheuristic solve an optimization problem by making deterministic decisions. Stochastic metaheuristics apply some random rules during the search.
- **Iterative versus greedy:** Iterative algorithms start with a complete solution (or population of solutions) and transform it at each iteration using some search operators. Most metaheuristics are iterative algorithms.
- **Population-based search versus single-solution based search:** Single-solution based algorithms (e.g. local search algorithms) manipulate and transform a single solution during the search while in population-based algorithms (e.g. Evolutionary Algorithms) different solutions are combined, either implicitly or explicitly, to create new solutions.

In this work, we divide metaheuristics into the categories single-solution and population-based metaheuristics. These two families have complementary characteristics. Single-solution based metaheuristics are exploitation oriented and have the power to intensify the search in local regions. In contrast, population-based metaheuristics are exploration oriented and allow a better diversification in the whole search space. Exploration can also be seen as a diversification of solutions and exploitation can be seen as an intensification of the best solutions as shown in Fig. 3.2. These two contradictory criteria must be taken into account in designing a metaheuristic.

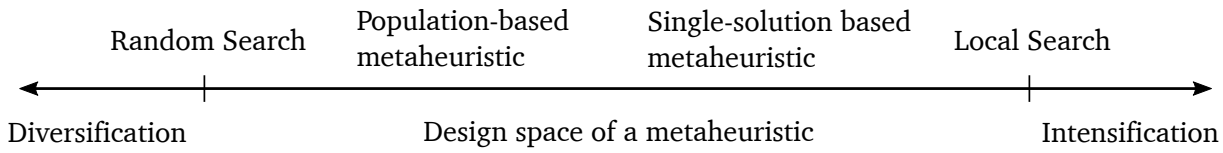



**Figure 3.2:** Two conflicting criteria in designing a metaheuristic [38].

In contrast to standard heuristics, single-solution based and population-based metaheuristics introduce some random factors to overcome the problem of being stuck in local optima. With these random moves, metaheuristics can escape local optima in order to increase the possibility of finding better solutions. An illustration of the so-called perturbation moves is given in Fig. 3.3.

Similar to heuristics presented in Sec. 2.4, single-solution based and population-based metaheuristics can be divided in primarily constructive metaheuristics, where a solution is built from scratch (through the introduction of new elements at each iteration) and improvement metaheuristics, which modify a solution iteratively.

In order to give an impression of single-solution based metaheuristics, the Greedy Randomized Adaptive Search Procedure (GRASP) algorithm is presented in the next Sec. 3.2.

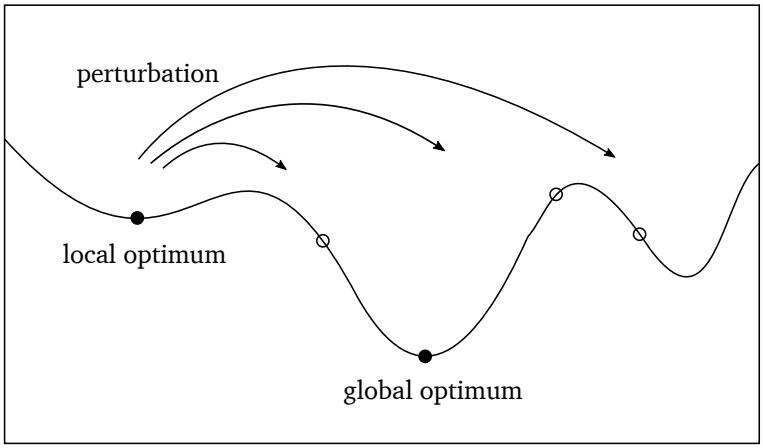



**Figure 3.3:** An example of a perturbation move in a simplified, 1-dimensional space [49].

## 3.2 Greedy Randomized Adaptive Search Procedure (GRASP)

As described in the previous Sec. 3.1, greedy algorithms start with empty solutions and belong to the group of constructive metaheuristics. This group of metaheuristics construct solutions rather than improving complete solutions. This is done by adding one element to a partial solution at each iteration. A greedy algorithm tries to add the best possible element at each iteration. To improve the quality of the final solutions, most constructive metaheuristics include a local search phase after the construction phase [37]. In addition, greedy heuristics are typically deterministic algorithms [38].

Besides applying a large random change (perturbation) to the current solution in order to escape local optima as shown in Fig. 3.3, restarting the search with a new random solution is another way to escape local optima. The Greedy Randomized Adaptive Search Procedure (GRASP) algorithm uses this restarting method. In general, this algorithm is a multi-start procedure consisting of two phases: construction phase and local search phase. Each restart of the procedure applies a randomized greedy construction heuristic to generate an initial solution. For example, the Nearest Neighborhood algorithm described in Sec. 2.4.1 can be used to obtain an initial solution.
At each step of the construction heuristic, elements not yet integrated into the partial solution are evaluated with a greedy function and the best elements are kept in a so-called restricted candidate list. Then, one element is chosen randomly from this list and integrated into the solution. Through randomization, the currently best element is not necessarily chosen, thus leading to a diversity of solutions to explore the whole search space. Then, the solution is improved until a local minimum is found during the local search phase. In each iteration of this phase, the current solution is replaced by a solution from its neighborhood to obtain a local optimum. This is repeated for a given number of restarts and the overall best solution is returned at the end [39]. However, this restarting procedure is one disadvantage of the GRASP algorithm. Each restart is independent from the previous ones and thus prevents the exploitation of previously obtained solutions to guide the search.
In order to solve a TSP, the GRASP algorithm would select a random starting city and apply a construction heuristic described in Sec. 2.4.1 to obtain a valid tour. After that, a local search algorithm (e.g. 2-opt) is applied to find the local optimum in the neighborhood of the initial tour. This means that the algorithm interchanges edges in the tour to minimize the tour length. If the solution cannot be improved any longer, the GRASP algorithm stops and restarts from the beginning.

Applying population-based metaheuristics leads to a more random search and exploration of the search space as illustrated in Fig. 3.2. In this work, we focus on two population-based metaheuristics: Ant (Colony) System and Genetic Algorithms. These algorithms are presented in the next Sec. 3.3-3.5.

## 3.3 Ant System (AS)

In the following, we will adopt the terminology as defined by Dorigo et al. in [40,41] and Cordón in [42]. The first presented population-based metaheuristic is an algorithm which is inspired by swarm intelligence. It is an approach to problem solving that takes inspiration from the social behaviors of insects and other animals. In particular, ants have inspired a number of methods and algorithms among which the most studied and the most successful is the general purpose optimization technique known as Ant System (AS) [43].

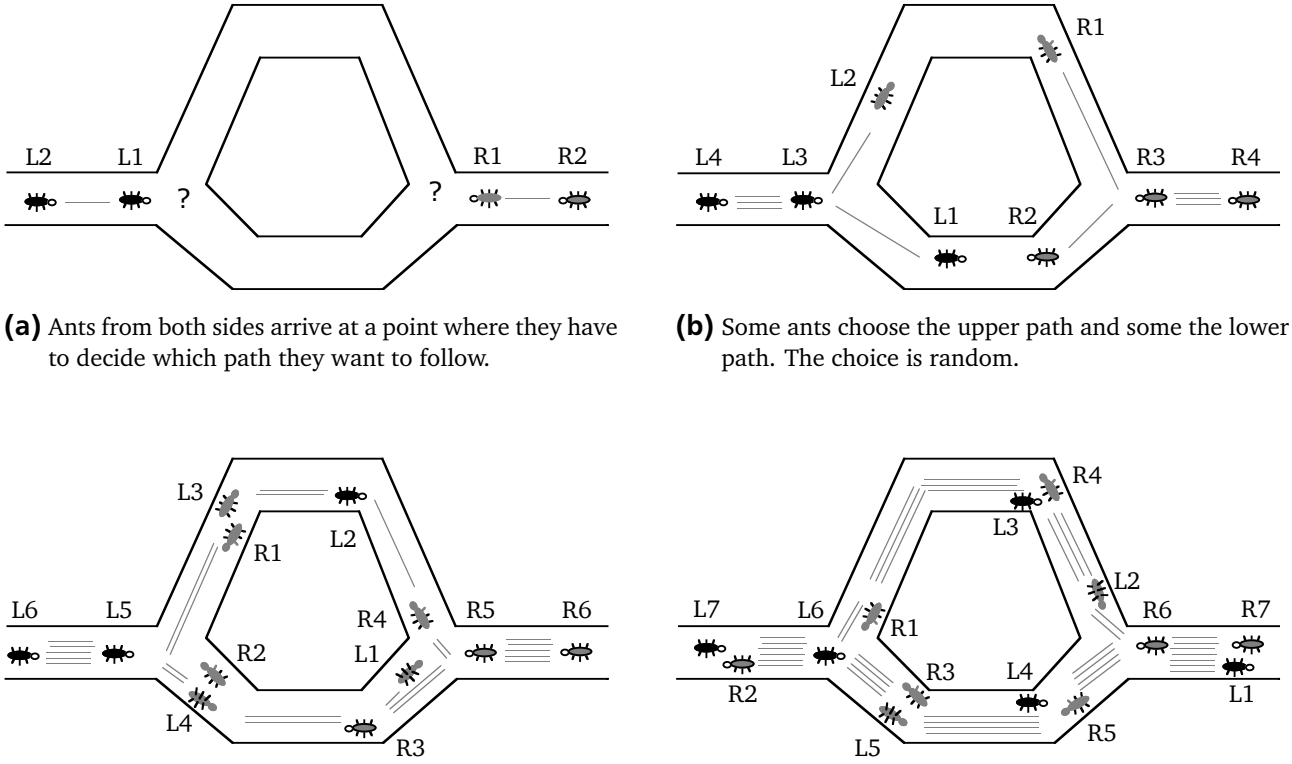
Ants are social insects that live in colonies and, because of their collaborative interaction, they are capable of showing complex behaviors and performing difficult tasks from an ant's local perspective. A very interesting characteristic of several ant species is their ability to find the shortest path between their nest and a food source. This fact is noticeable, because many ant species are almost blind, which avoids the exploitation of visual clues [42].
It was found that the medium used to communicate information among individuals consists of pheromone trails. While walking between their nest and food sources, some ant species deposit a chemical substance in varying quantities on the ground in order to mark the path by a trail of this pheromone. If no pheromone trails are available, ants move essentially at random. Contrary, an ant can detect a previously laid trail and decide with high probability to follow it. In practice, choices between different paths occur when several paths intersect. Then, ants choose the path to follow by a probabilistic decision function biased by the amount of pheromone. The higher the pheromone concentration, the higher the chance that an ant will follow the pheromone trail.
If an ant chooses to follow an existing trail, the pheromone concentration will increase, because the ant will also deposit pheromone on the path. The more ants are following a trail, the more attractive it becomes for being followed. This behavior results in a self-reinforcing process leading to a formation of paths marked by high pheromone concentrations. Thus, the process is characterized by a positive feedback loop, where the probability, with which an ant chooses a path to follow, increases with the number of ants that previously chose the same path [42].

How this mechanism allows ants to find the shortest path between their nest and a food source is illustrated in Fig. 3.4. Regarding Fig. 3.4a, ants arrive at a decision point in which they have to decide whether to turn left or right. Since they have no clue about which is the best choice, they choose randomly as described before. It can be expected that, on average, half of the ants decide to turn left and the other half to turn right. This happens both to ants moving from left to right (name begins with L) and to those moving from right to left (name begins with R).
In Fig. 3.4b and Fig. 3.4c, it is shown what happens next, supposing all ants walk at approximately the same speed. The number of dashed lines is roughly proportional to the pheromone concentration on a path. Since the lower path is shorter than the upper one, more ants will visit it on average. As a consequence, pheromone accumulates faster. After a short transition period, the difference in the amount of pheromone on the two paths is sufficiently large that it will influence the decision of new ants coming into the system. From now on, new ants will prefer, in probability, to choose the shorter path, since they detect a higher amount of pheromone on the lower path at the decision point. This is shown in Fig. 3.4d. The positive feedback loop will lead to an increasing number of ants using the lower path until all ants use this shorter path in the end.

The described behavior of real ants has inspired the Ant System algorithm. It is an algorithm in which a set of artificial ants cooperate to obtain the solution of a problem by exchanging information via a pheromone deposited on graph edges.

In comparison to real ants, artificial ants have the following characteristics:

- Artificial ants have some kind of memory that stores the path followed by the ant.
- They are not completely blind and can make use of heuristic information in the applied stochastic transition policy.
- They live in an environment where time is discrete.




**(a)** Ants from both sides arrive at a point where they have to decide which path they want to follow.

**(b)** Some ants choose the upper path and some the lower path. The choice is random.

**(c)** Since ants move at approximately constant speed, the ants which choose the lower, shorter, path reach the opposite decision point faster than those which choose the upper, longer, path.

**(d)** Pheromone accumulates faster on the shorter path. The number of dashed lines is approximately proportional to the amount of pheromone deposited by ants. As a consequence, new ants prefer the shorter path.

**Figure 3.4:** How real ants find the shortest path [41].

The Ant System was originally designed for the TSP. Each artificial ant in the TSP is a simple agent with the following characteristics:

- It chooses the following city with a probability that is a function of the distance to the city and the amount of pheromone deposited on the connecting edge.
- To force the agent to make legal tours, transitions to already visited cities are not allowed until a tour is completed. This is controlled by a tabu list for each agent.
- When the agent completes a tour, it deposits a substance called trail on each edge visited.

Informally, the Ant System works as follows. Each ant generates a complete tour by choosing the next city according to a probabilistic state transition rule at each iteration. They prefer to move to cities which

are connected by short edges with a high amount of trail. Once all ants have completed their tours, a global pheromone updating rule is applied. A fraction of the trail on all edges evaporates, which causes edges that are not used to become less desirable. Then, each ant deposits an amount of trail on edges which belong to its tour in proportion to the length of the tour. Edges which belong to many short tours receive a higher amount of trail than edges which belong to longer tours. This iterative process restarts including the updated trail concentration on each edge.

The mathematical definition of the Ant System, using parts of the definition of the TSP introduced in Sec. 2.1, is as follows. Let $\tau_{ij}(t)$ be the intensity of trail on each edge $\langle i,j\rangle$. At time $t$, each ant chooses the next city where it will be at time $t+1$. An iteration of the Ant System algorithm, where all $m$ ants perform $m$ moves in the interval $(t,t+1)$, is called an iteration. Every $N$ iterations of the algorithm, each ant visited all $N$ cities and has completed a tour. This is called a cycle. At this point, the trail intensity is updated according to the global update rule shown in Eq. (3.1).

$$\tau_{ij}(t+N) \leftarrow (1-\rho)\cdot\tau_{ij}(t) + \sum_{k=1}^{m}\Delta\tau_{ij}^{k} \tag{3.1}$$

The coefficient $\rho$ is chosen is such a way that $(1-\rho)$ represents the evaporation of trail between time $t$ and $t+N$. It has to be set to a value $0 \leq \rho < 1$ to avoid unlimited accumulation of trail. The quantity of trail laid on edge $\langle i,j\rangle$ by ant $k$ is defined as

$$\Delta\tau_{ij}^{k} = \begin{cases} Q/L_k & \text{if ant } k \text{ used edge } \langle i,j\rangle \text{ in its tour,} \\ 0 & \text{otherwise,} \end{cases} \tag{3.2}$$

where $Q$ is a constant and $L_k$ is the length of the tour constructed by ant $k$. During the construction of a solution, ants select the next city to be visited through a stochastic mechanism. When ant $k$ is in city $i$ and has so far constructed the partial solution $s^P$, the probability of going to city $j$ is defined as

$$p_{ij}^{k}(t) = \begin{cases} \dfrac{\tau_{ij}(t)^{\alpha}\cdot\eta_{ij}^{\beta}}{\sum_{\langle i,l\rangle\in\mathcal{N}(s^P)}\tau_{il}(t)^{\alpha}\cdot\eta_{il}^{\beta}} & \text{if } \langle i,l\rangle \in \mathcal{N}(s^P), \\ 0 & \text{otherwise,} \end{cases} \tag{3.3}$$

where $\mathcal{N}(s^P)$ is the set of feasible components; that is, edges $\langle i,l\rangle$ where $l$ is a city not yet visited by ant $k$. Parameters $\alpha$ and $\beta$ with $\alpha+\beta=1$ control the relative importance of trail versus heuristic information $\eta_{ij}$, which is defined as

$$\eta_{ij} = 1/d_{ij}, \tag{3.4}$$

where $d_{ij}$ is defined as the distance between city $i$ and city $j$. To sum up, the transition probability is a trade-off between heuristic information or visibility (which states that close cities should be chosen with high probability) and trail intensity at time $t$ (which states that if there has been a lot of traffic on edge $\langle i,j\rangle$, it is highly desirable).

Using the above definitions, the Ant System algorithm works as follows. At time $t=0$, an initialization phase takes place during which ants are positioned randomly on different cities and initial values for trail intensity $\tau_{ij}(0)$ are deposited on all edges. Then, each ant moves from city $i$ to city $j$ choosing the city to move with probability $p_{ij}^{k}$. The trail $\tau_{ij}$ gives information about how many ants have chosen the same edge $\langle i,j\rangle$ in the past and the visibility $\eta_{ij}$ states that the closer a city, the more desirable it is.

Obviously, setting $\alpha = 0$, the trail level is no longer considered and a stochastic greedy algorithm with multiple starting points is obtained. Moreover, setting $\beta = 0$, all ants tend to use a path which was used from many ants before. This can be a random path, because at the beginning all ants are positioned randomly and move in random directions. Therefore, the exploration of other paths is suppressed.
After $N$ iterations, all ants have completed a tour and the trail intensity is updated according to Eq. (3.1). In addition, the shortest path found by ants is saved. This process is restarted using the updated trail concentration $\tau_{ij}$ and continues restarting until a user defined maximum number of iterations is reached or all ants make the same tour. After the last cycle, the overall best tour found is chosen as the solution.

In order to evaluate if other quantities of trail $\Delta\tau_{ij}^k$ laid on edge $\langle i, j\rangle$ by ant $k$ influence the quality of the solution, two different modifications of the original Ant System algorithm are introduced [40].
The first modification is called AS-density, the second is called AS-quantity. The algorithms differ in the way how the trail is updated. In these two models, each ant lays its trail at each iteration without waiting for the end of the tour. In the AS-density model, an arbitrary quantity of trail $Q$ is left on edge $\langle i, j\rangle$ every time an ant moves from city $i$ to city $j$. In the AS-quantity model, an ant moving from city $i$ to city $j$ leaves a quantity of trail $q/d_{ij}$ on edge $\langle i, j\rangle$. To sum up, the quantity of trail $\Delta\tau_{ij}^k$ left on edges in the AS-density model is

$$\Delta\tau_{ij}^k = \begin{cases} Q & \text{if ant } k \text{ used edge } \langle i, j\rangle \text{ in its tour,} \\ 0 & \text{otherwise,} \end{cases} \tag{3.5}$$

and in the AS-quantity model it is

$$\Delta\tau_{ij}^k = \begin{cases} Q/d_{ij} & \text{if ant } k \text{ used edge } \langle i, j\rangle \text{ in its tour,} \\ 0 & \text{otherwise.} \end{cases} \tag{3.6}$$

Including Eq. (3.5), the quantity of trail $\Delta\tau_{ij}^k$ left on edge $\langle i, j\rangle$ is independent of distance $d_{ij}$ between two cities and the whole tour length $L_k$ of ant $k$. Contrary, shorter edges are made more desirable in the AS-quantity model defined in Eq. (3.6), because only distance $d_{ij}$ between two cities is considered when determining the quantity of trail $\Delta\tau_{ij}^k$ leaving on edge $\langle i, j\rangle$.

Experimental results show that both models lead to worse results than those obtained with the standard Ant System algorithm. The reason is to be found in the kind of feedback information which is used to direct the search process. The standard Ant System algorithm uses global information. The amount of trail left on edges is computed including the quality of the solution. In fact, ants producing shorter paths contribute a higher amount of trail than ants whose tour was poor. On the other side, AS-quantity and AS-density use local information. Their search is not directed by any quality measure of the final result.

Regarding an optimal value for the evaporation coefficient $\rho$, experimental results show that a value of $\rho = 0.5$ is optimal. This can be explained by the fact that the Ant System algorithm uses a greedy heuristic to guide the search in the early phase. Then, exploiting the global information in the values of trail $\tau_{ij}$ on the edge becomes more and more important. For this reason, the Ant System algorithm needs the possibility to forget parts of the experience gained in the past in order to better exploit new incoming global information. The major strength of the Ant System algorithm can be summarized as [40]:

- Within the range of parameter optimality, the algorithm always finds very good solutions.
- The algorithm finds good solutions quickly. Nevertheless, it does not exhibit a stagnation behavior and continues to search for new and possibly better tours.

- It can be applied to similar versions of the same problem. For example, there is a straightforward extension from the TSP to the ATSP. The computational complexity of one cycle of the algorithm remains the same as for the standard TSP, as the only differences occur in the distance and trail matrices, which are no longer symmetric.
- It can be applied to other combinatorial optimization problems such as assignment or scheduling problems requiring only minimal changes.

On the other hand, the main disadvantages of the Ant System algorithm are as follows [40]:

- Solving large TSPs requires many iterations of the algorithm to obtain a good solution.
- In general, the algorithm requires a much longer computation time than any other tested special-purpose heuristic.
- The algorithm can get stuck in local minima, because all ants choose the same sub-optimal tour.

In order to improve the performance of the Ant System algorithm, three main changes can be applied. The enhanced algorithm called Ant Colony System is presented in the next Sec. 3.4.

## 3.4 Ant Colony System (ACS)

The Ant Colony System (ACS) is based on the Ant System presented in the previous Sec. 3.3. There are three main aspects in which the Ant Colony System differs from the Ant System [41]:

1. The state transition rule provides a direct way to balance between exploration of new edges and exploitation of a priori and accumulated knowledge about the problem.
2. The global updating rule is applied only to edges which belong to the best ant tour.
3. While ants construct a solution, a local pheromone updating rule is applied.

Informally, the Ant Colony System algorithm works as follows. There are $m$ ants positioned randomly on $m$ cities chosen according to an initialization rule. Then, each ant builds a tour by repeatedly applying a stochastic greedy rule (the state transition rule). While constructing its tour, an ant also modifies the amount of pheromone on the visited edges by applying the new introduced local updating rule. Once all ants have terminated their tour, the amount of pheromone on edges is modified again by applying the global updating rule. During the construction of feasible tours, ants are guided by heuristic information (they prefer to choose short edges) and by pheromone information (they prefer to choose edges with a high amount of pheromone). The pheromone updating rules are designed in such a way that they tend to give more pheromone to edges which should be visited by ants.

The Ant Colony System algorithm uses a modified state transition rule called pseudo-random-proportional rule. It is based on the state transition rule from the Ant System algorithm defined in Eq. (3.3). Let $k$ be an ant located at city $i$, $q_0 \in [0,1]$ be a parameter and $q$ a random value in $[0,1]$. Then, next city $j$ is chosen randomly according to the following probability distribution [42]:

$$\begin{aligned}
&\text{if } q \leq q_0: \\
&\qquad p_{ij}^k(t) = \begin{cases} 1 & \text{if } j = \underset{\langle i,l\rangle \in \mathcal{N}(s^P)}{\arg\max}\{\tau_{il} \cdot \eta_{il}^\beta\} \\ 0 & \text{otherwise} \end{cases} \\
&\text{else } (q > q_0): \\
&\qquad p_{ij}^k(t) = \begin{cases} \frac{\tau_{ij}(t)^\alpha \cdot \eta_{ij}^\beta}{\sum_{\langle i,l\rangle \in \mathcal{N}(s^P)} \tau_{il}(t)^\alpha \cdot \eta_{il}^\beta} & \text{if } \langle i,l\rangle \in \mathcal{N}(s^P) \\ 0 & \text{otherwise} \end{cases}
\end{aligned} \tag{3.7}$$

Every time an ant in city $i$ has to choose a city $j$ to move, it samples a random number $0 \leq q \leq 1$ and applies the state transition rule. This rule shown in Eq. (3.7) has two goals: when $q \leq q_0$, it exploits the available knowledge while choosing the best option with respect to the heuristic information and the pheromone trail. However, if $q > q_0$, it applies a controlled exploration similar to the transition rule from the Ant System algorithm shown in Eq. (3.3). Summing up, the rule establishes a trade-off between the exploration of new connections and the exploitation of the information available in that moment. Parameter $q_0$ determines the relative importance of exploitation versus exploration.

The second difference to the Ant System algorithm concerns the global updating rule after a cycle is finished and all ants have visited $N$ cities. In the Ant Colony System algorithm, only the globally best ant with the shortest tour from the beginning of the trial is allowed to deposit pheromone. This choice, together with the use of the pseudo-random-proportional rule, is intended to make the search more

directed. This will eliminate one of the main disadvantages of the Ant System algorithm described in Sec. 3.3: the need for many iterations of the algorithm to obtain a good solution.
Applying the enhanced global updating rule, an ant will search in a neighborhood of the best tour for a better solution. For the Ant Colony System algorithm, the global updating rule is defined in Eq. (3.8) [42]. Note that there is no evaporation of trail on edges not being part of the best solution.

$$\tau_{ij}(t+N) \leftarrow \begin{cases} (1-\rho)\cdot\tau_{ij}(t) + \Delta\tau_{ij} & \text{if } \langle i,j\rangle \text{ belongs to the best tour} \\ \tau_{ij}(t) & \text{otherwise} \end{cases} \tag{3.8}$$

In addition to the global updating rule, the Ant Colony System algorithm introduces an online step-by-step trail update during the search. Each time an ant travels an edge $\langle i,j\rangle$, it applies the following rule:

$$\tau_{ij}(t+1) \leftarrow (1-\phi)\cdot\tau_{ij}(t) + \phi\cdot\Delta\tau_0, \tag{3.9}$$

where $\phi \in (0,1]$ is a second pheromone decay parameter and $\tau_0$ is the initial value of the pheromone deposited on all edges. As can be seen, the introduced local updating rule includes pheromone evaporation and deposit similar to the global updating rule. However, because of the fact that the amount of pheromone deposited is very small compared to $\tau_0$, the pheromone trail on the edges traversed by an ant decreases. Hence, this results in an additional exploration technique of the Ant Colony System algorithm. Edges used by an ant are less attractive to following ants. This helps to avoid that every ant follows the same path and the algorithm is stuck in a local minimum.

Experimental results show that the Ant Colony System algorithm outperforms the Ant System algorithm [41]. In addition, applying the Ant Colony System algorithm to symmetric TSP instances, the results are comparable to those obtained by other algorithms. Using the Ant Colony System algorithm to solve large ATSP instances leads to better solutions compared to other algorithms.
In order to improve these very good solutions, a local search heuristic can be applied after an initial solution was found by the Ant Colony System algorithm. It has been shown that it is more effective to apply an improvement heuristic to the last (or best) solution produced by the Ant Colony System, rather than executing a tour improvement heuristic, which starts with a solution generated randomly or by a construction heuristic. For example, the 3-opt tour improvement heuristic can be applied. This is a modified version of the 2-opt heuristic shown in Sec. 2.4.2.

In conclusion, the Ant Colony System algorithm is a powerful metaheuristic and can be applied to many combinatorial optimization problems by defining an appropriate graph representation of the problem. This nature inspired metaheuristic performs very well for a range of problems. In connection with an improvement heuristic, it can outperform other algorithms in special cases. A stochastic component in the Ant Colony System algorithm allows the ants to build a wide variety of different solutions and hence to explore a much larger number of solutions than greedy heuristics. At the same time, the use of heuristic information (e.g. the visibility and the pheromone information) can guide the ants towards the most promising solutions [44].

Similar to the Ant Colony System algorithm and the additional application of a tour improvement heuristic, another class of metaheuristics is able to exploit the search space and explore local solutions, but combined in one algorithm. The so-called Genetic Algorithms are also nature inspired metaheuristics and will be presented in the next Sec. 3.5.

## 3.5 Genetic Algorithms (GA)

Genetic Algorithms are one of the most widely known classes of Evolutionary Algorithms. This class of algorithms is motivated by the way species evolve and adapt to their environment based on the Darwinian principle of natural selection. His theory includes three main observations:

1. All species produce more offspring than needed in order to compensate the mortality rate of the offspring.
2. Individuals of one species can be similar, but never identical.
3. Only genes which lead to an adaption to the environment and the consequent survival will be found in the following generation.

The first observation can be seen as a selection pressure. Despite the surplus of offspring, the size of the population stays constant. This is caused by the fact that there exist limiting factors such as food and other resources. Parts of the population die until being able to reproduce themselves. In conclusion, Darwin states that certain individuals of one species are better adapted to the environment and have a higher chance to survive, even if there is no large difference between the individuals. The environmental conditions lead to a natural selection of the fittest individuals. The so-called *'survival of the fittest'* states that in particular the best adapted individuals will become parents and produce offspring.

| **Metaphor** | **Optimization problem** |
|---|---|
| Evolution | Problem solving |
| Environment | Optimization problem |
| Individual | Solution |
| Chromosome | Encoded solution |
| Gene | Decision variable within a chromosome |
| Allel | Possible values of variables (genes) |
| Locus | Position of a gene within a chromosome |
| Fitness | Objective function |

**Table 3.1:** Evolution process versus solving an optimization problem [38].

The whole observation can be translated to mathematical terms. Evolution can be compared to a search process where the search space is the space of all possible combinations of genes. The goal is to find a combination of genes which helps to survive. The metaphors for mathematical formulations are shown in Table 3.1.

Evolution is essentially based on three principles: mutation, recombination and selection. Their combination is a connection of directed and non-directed search processes. The mutation of genes only produces small changes and thus variants in the gene. As described in Sec. 3.1, the overcoming of local optima is important for metaheuristics. Accidental changes in genes can prevent a population from evolving into a local optimum. Recombination states that two parent individuals combine their most desirable features to create one or two offspring individuals. Selection states that only the fittest individuals will survive and produce offspring [38].
In order to adapt to the environment, there exist two ways. One way is to produce a large number of offspring to obtain a pool of individuals, where only the fittest will survive. The other way is to shorten the reproduction time so that the reproduction is faster than the change in environmental conditions.

Evolutionary Algorithms are based on the notion of competition and the evolution of a population of individuals. In general, these algorithms work as follows. Initially, a population is generated randomly. An objective function associates a fitness value with every individual indicating its suitability to the problem. At each step, individuals are selected to become parents, following the selection paradigm in which individuals with better fitness are selected with a higher probability. Then, the selected individuals are reproduced using variation operators (e.g. crossover and mutation) to generate new offspring. Finally, a replacement scheme is applied to determine which individuals of the population will survive. One iteration of this process is called a generation. An illustration of the described procedure is given in Fig. 3.5.

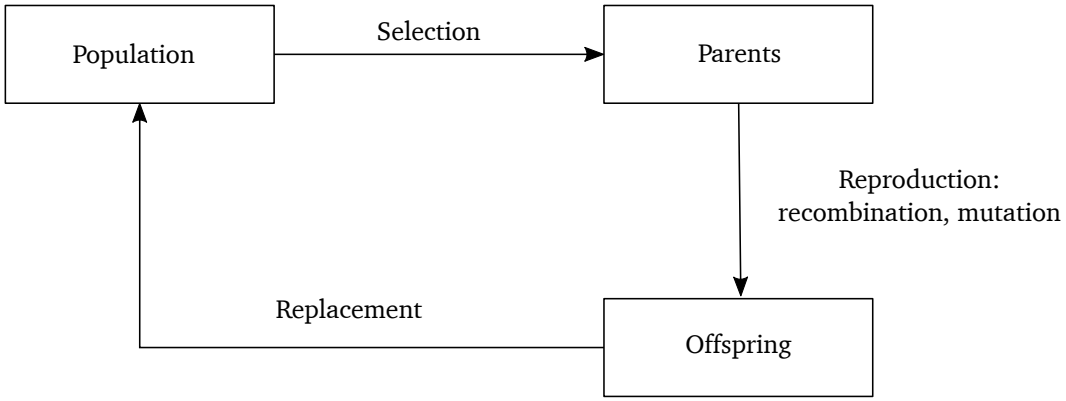



**Figure 3.5:** A generation in Evolutionary Algorithms [38].

Based on this process, Genetic Algorithms can solve different optimization problems such as Resource Allocation Problems presented in Sec. 1.2 or TSP instances. The main design parameters in order to apply Genetic Algorithms are as follows [38]:

1. **Representation:** The solutions have to be encoded in order to apply the algorithm. The encoded solution is referred to as chromosome while the decision variables within a solution (chromosome) are genes. The possible values of variables (genes) are the alleles and the position of an element (gene) within a chromosome is named locus.

2. **Population initialization:** The initial population can be obtained randomly or a solution of a construction heuristic presented in Sec. 2.4.1 can be used.

3. **Objective function:** The objective function (fitness) has to be defined in order to evaluate the quality of a solution.

4. **Selection strategy:** The selection strategy addresses the question which solutions are chosen as parents to produce offspring.

5. **Reproduction strategy:** This strategy consists of choosing suitable mutation and crossover operator(s) to generate new individuals (offspring).

6. **Replacement strategy:** The new offspring compete with old individuals for their place in the next generation.

7. **Stopping criteria:** There are no stopping criteria defined in Genetic Algorithms. Therefore, external stopping criteria have to be included.

In order to show the application of Genetic Algorithms, consider the following simple minimization problem [45]:

$$\begin{aligned} \text{minimize} \quad & F = |(a-b)| + |(b-c)| + |(c-d)| + |(d-a)| && \text{(3.10a)} \\ \text{subject to} \quad & a, b, c, d \leq 63 && \text{(3.10b)} \\ & a, b, c, d \in \mathbb{N} && \text{(3.10c)} \end{aligned}$$

It can be seen that the optimal solution for this problem can be obtained if $a = b = c = d$. Applying Genetic Algorithms requires an initial population. Regarding the example, the value for each variable $a, b, c$ and $d$ is chosen randomly to obtain three initial solutions (chromosomes). The initial population with individuals $I_i$ for $i = 1, \ldots, \lambda$ and population size $\lambda$ is as follows:

$$\begin{aligned} I_1 &: (51, 42, 63, 9) \\ I_2 &: (47, 45, 58, 21) \\ I_3 &: (40, 47, 2, 61) \end{aligned}$$

**Figure 3.6:** Example initial population in decimal representation.

In order to apply a crossover operator, the decimal representation has to be translated into the binary representation, which is often used when applying Genetic Algorithms. The binary representation of the example shown in Fig. 3.6 is as follows:

$$\begin{aligned} I_1 &: (110011, 101010, 111111, 001001) \\ I_2 &: (101111, 101101, 111010, 010101) \\ I_3 &: (100101, 010110, 010001, 111010) \end{aligned}$$

**Figure 3.7:** Example initial population in binary representation.

In addition, the binary representation of the solution has to be converted into a bit string. For this, the binary representations of each variable are connected, leading to the final bit string shown in Fig. 3.8.

$$\begin{aligned} I_1 &: 110011101010111111001001 \\ I_2 &: 101111101101111010010101 \\ I_3 &: 100101010110010001111010 \end{aligned}$$

**Figure 3.8:** Example initial population represented as bit strings.

The selection of possible parents requires a fitness value assigned to each individual. This value is calculated using the result of the objective function in the minimization problem shown in Eq. (3.10). Evaluating the objective function for each chromosome leads to the following values:

$$\begin{aligned} I_1 &: |(51-42)| + |(42-63)| + |(63-9)| + |(9-51)| = 126 \\ I_2 &: |(54-45)| + |(45-58)| + |(58-21)| + |(21-47)| = 78 \\ I_3 &: |(40-47)| + |(47-2)| + |(2-61)| + |(61-40)| = 132 \end{aligned}$$

**Figure 3.9:** Objective function values.

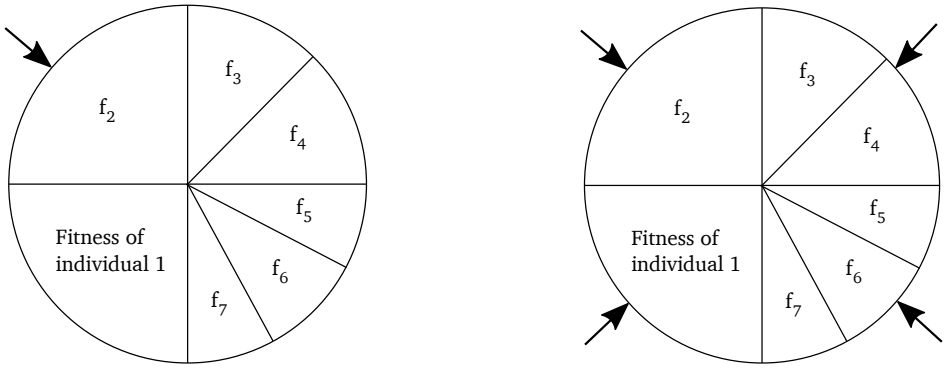
As can be seen in Fig. 3.9, the value of the second chromosome $I_2$ is the lowest. Therefore, the fitness value has to be the highest, because the goal is to minimize the objective function shown in Eq. (3.10a). In order to select two chromosomes as parents, a selection strategy has to be applied. The calculation of the fitness value and three different selection strategies are presented in the next Sec. 3.5.1.

### 3.5.1 Selection Strategies

The most common selection strategy is called Roulette Wheel Selection [38]. It assigns a selection probability to each individual that is proportional to its relative fitness. Let $f_i$ be the fitness of individual $I_i$ and the number of individuals be $\lambda$. Its probability $p_i$ to be selected is defined as follows:

$$p_i = \frac{f_i}{\sum_{j=1}^{\lambda} f_j} \tag{3.15}$$

Suppose a pie graph where each individual is assigned a space on the graph that is proportional to its fitness as illustrated in Fig. 3.10a. An outer roulette wheel with one pointer is placed around the pie. The selection of $\mu$ individuals is performed by $\mu$ independent spins of the roulette wheel. Each spin selects a single individual. Better individuals have more space and as a consequence a higher chance to be chosen. The basic advantage of the Roulette Wheel Selection is that it discards none of the individuals in the population and gives a chance to all of them to be selected [46].
On the other hand, the Roulette Wheel Selection has a bias introduced by outstanding individuals in the beginning of the search. This my cause a premature convergence and a loss of diversity, because these individuals can be chosen more than once. In addition, when all individuals are equally fit, this selection strategy does not introduce a sufficient pressure to select the best individuals [38].




**(a)** Roulette Wheel Selection **(b)** Stochastic Universal Sampling

**Figure 3.10:** Roulette Wheel Selection. In the standard case, each spin selects a single individual from the population. Using Stochastic Universal Sampling, a spin will select as many individuals as predefined (e.g. four in this example) [38].

In order to reduce the bias of the Roulette Wheel Selection, another selection strategy called Stochastic Universal Sampling can be used. An outer roulette wheel with $\mu$ equally spaced pointers is placed around the pie. Spinning the roulette wheel once will simultaneously select all the $\mu$ individuals for reproduction. An illustration of this selection strategy is given in Fig. 3.10b.
Unfortunately, it is difficult to use Roulette Wheel Selection or Stochastic Universal Sampling on minimization problems whereby the fitness function for minimization has to be be converted to a maximization function as in the case of the TSP. Although this may solve the selection problem, it introduces

confusion to the problem. For example, the best chromosome in the TSP will continually be assigned a fitness value that is the maximum of all other fitness functions. Thus, the shortest tour is searched, but the fitness value is maximized. As a consequence, several other selection techniques with a probability not proportional to the fitness values of each individual have been developed to encounter this selection problem [46].

One of these selection strategies is called Tournament Selection. It consists of selecting $k$ individuals randomly, where $k$ is the size of the tournament group. Then, a tournament is applied to the $k$ members of the group to select the best one as shown in Fig. 3.11. In order to select $\mu$ individuals from the population, the same number of tournaments have to be carried out. A disadvantage of this selection is the expected loss of diversity for larger values of the tournament group, because better individuals will dominate other individuals [47].

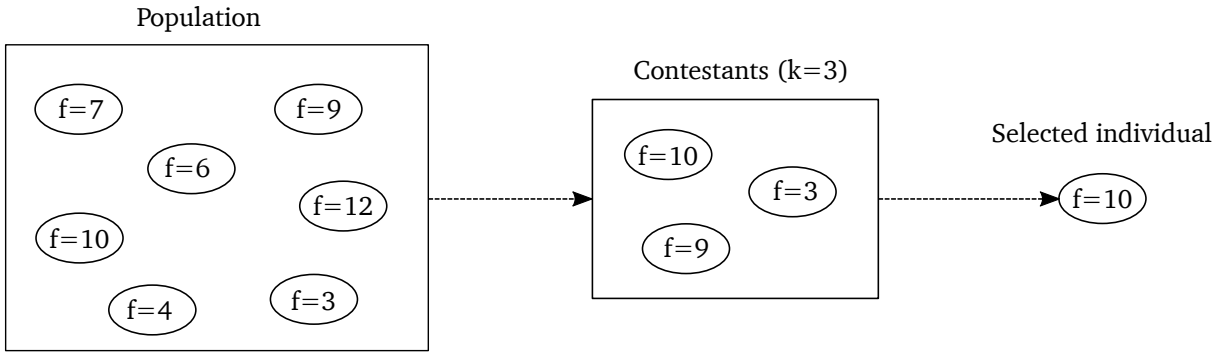



**Figure 3.11:** Tournament Selection [46].

Applying a selection strategy leads to $\mu$ individuals which are chosen to be copied in the so-called mating pool. In case of the Roulette Wheel Selection and the Tournament Selection, there is a chance that an individual is chosen multiple times. That is why some of the selected $\mu$ individuals may be equal. In the next Sec. 3.5.2, the process of creating offspring is described.

### 3.5.2 Reproduction Strategies

After performing the selection of individuals to be parents, a variation of operators such as mutation and crossover are applied in the reproduction phase [38].
Mutation operators are unary operators acting on a single individual. These operators represent small changes in the genes of selected individuals in order to create, by chance, an unexpected good solution [48]. In the case of binary strings, mutation simply means complementing the chosen bit(s). The probability to mutate each element (gene) of a chromosome is denoted as $p_m$. In general, small values are recommended for this probability ($p_m \in [0.001, 0.01]$) [38]. Some strategies set the mutation probability to $1/d$, where $d$ is the number of genes. In this case, in average only one variable is mutated. The following characteristics are important for choosing a suitable mutation strategy [38]:

- **Ergodicity:** The mutation operator should allow every solution in the search space to be reached.
- **Validity:** The mutation operator should create feasible solutions. This is not always possible for constrained optimization problems. For example, the TSP is a special case as described in Sec. 3.5.4.
- **Locality:** The mutation should only create a minimal change. The mutated solution should be a neighborhood solution of the original solution.

The main operation in Genetic Algorithms is crossover. Two parent solutions are chosen from the mating pool to produce offspring while combining their genes. No selection strategy applied and so the probability to be chosen is equal for each individual in the mating pool. The two individuals selected from the mating pool are deleted from that pool after applying the crossover operator. In the following, the simple one-point crossover is described. In Sec. 3.5.6, special crossover operators for the TSP are presented.
One-point crossover is aimed at exchanging bit strings between two parent chromosomes [16]. A random position between 1 and $L-1$ is chosen along the two parent chromosomes, where $L$ is the length of the chromosomes. Then, the chromosomes are cut at the selected position and their parts are exchanged to create two offspring.
In order to copy very good solutions from the old to the new population, a probability is associated to the application of the crossover operation. If the operator is not applied, the selected parents are copied to the new population without any modification. The crossover rate can be related to the "aggressiveness" of the search. High crossover rates create more new offspring at the risk of losing many good chromosomes in the current population. Conversely, low crossover rates tend to preserve the good chromosomes from one population to the next including a weaker exploration of the search space [16].

Regarding the numerical example from Sec. 3.5, we assume that individuals $I_1$ and $I_2$ are chosen from the old population and copied to the mating pool using Tournament Selection. Then, the one-point crossover operation is performed at a random location between two genes to create two offspring $O_1$ and $O_2$ as follows:

$$
\begin{aligned}
I_1 &: \mathbf{11001110} \mid \mathbf{1010111111001001} \\
I_2 &: 10111110 \mid 1101111010010101 \\
\\
O_1 &: \mathbf{11001110}1101111010010101 \\
O_1 &: 10111110\mathbf{1010111111001001}
\end{aligned}
$$

**Figure 3.12:** Example of an one-point crossover operation.

Assuming that no mutation operation is applied, the obtained values evaluating the objective function shown in Eq. (3.10a) are as follows:

$$
\begin{aligned}
O_1 &: |(51-45)| + |(45-58)| + |(58-21)| + |(21-51)| = 86 \\
O_2 &: |(47-42)| + |(42-63)| + |(63-\ \ 9)| + |(\ \ 9-47)| = 118
\end{aligned}
$$

**Figure 3.13:** Objective function values after one-point crossover operation.

In order to select the best individuals to be in the new population, there exist two main strategies which are described in the following Sec. 3.5.3.

### 3.5.3 Replacement Strategies

After creating $\mu$ offspring, the population has a size of $\lambda + \mu$ individuals. Though, by definition, the size of the population has to be constant for each generation. For this reason, $\lambda$ individuals have to be selected to form the new population.
The first strategy is to create more offspring $\mu$ than individuals $\lambda$ being in the old population [49]. Then, the best $\lambda$ individuals are chosen from the $\mu$ offspring to form the new population. This strategy is called comma-strategy $(\lambda, \mu)$. From an optimization point of view, this strategy is sub-optimal, because

all parent solutions are left over. It may happen that a very good solution in the old population is not part of the new population. As described in the previous Sec. 3.5.2, there is a chance that a very good solution is copied to the new population without modification. Nevertheless, there exist another way to avoid this problem.
Applying the second selection strategy, the new population is formed out of the old population and the offspring. This strategy is called plus-strategy ($\lambda+\mu$). The best $\lambda$ solutions from $\lambda$ solutions belonging to the old population and $\mu$ offspring solutions are chosen to form the new population. Very good solutions are preserved and transferred to the next generation. However, this strategy can lead to a convergence to local optima and a loss of diversity [38].

After a new population is created, Genetic Algorithms restart the procedure. This iterative process never stops, because the population size stays constant. For this reason, no solution is chosen to be the final result. However, termination criteria can be included to stop the algorithm and obtain the best solution found so far. The different steps of Genetic Algorithms are summarized in Fig. 3.14.

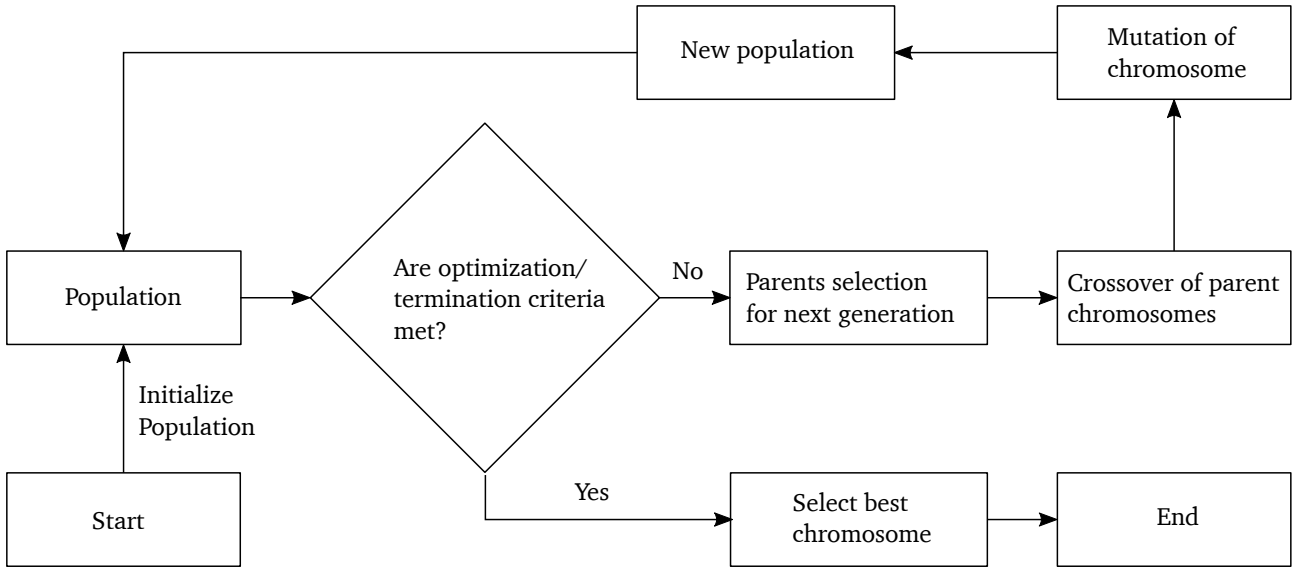



**Figure 3.14:** Genetic Algorithms procedure [46].

Applying Genetic Algorithms to the TSP requires different representations and operators. These are presented in the following Sec. 3.5.4-3.5.6.

### 3.5.4 Application to the TSP

Using Genetic Algorithms with binary representation to solve the TSP causes several problems. Consider the following example. There exist $N = 8$ cities and each city is represented by a 3-digit bit string. Two random initial solutions are as follows:

$$I_1 : (000, 001, 010, 011, 100, 101, 110, 111)$$
$$I_2 : (100, 110, 111, 011, 000, 101, 001, 010)$$

**Figure 3.15:** Random initial solutions for the TSP in binary representation.

As defined in Sec. 2.1, the TSP is the problem of finding the shortest path through all cities visiting each city exactly once. Applying a standard crossover operator, for example the one-point crossover introduced in Sec. 3.5.2, can lead to invalid solutions, because a city may appears twice in the resulting tour. Applying one-point crossover to the parent solutions shown in Fig. 3.15 leads to the following two offspring $O_1$ and $O_2$:

$$
\begin{aligned}
I_1 &: \mathbf{00000101001} \mid \mathbf{1100101110111} \\
I_2 &: 10011011101 \mid 1000101001010 \\
\\
O_1 &: \mathbf{00000101001}1000101001010 \\
O_1 &: 10011011101\mathbf{1100101110111} \\
\\
O_1 &: (000, 001, 010, 011, 000, 101, 001, 010) \\
O_2 &: (100, 110, 111, 011, 100, 101, 110, 111)
\end{aligned}
$$

**Figure 3.16:** Crossover operation using binary representation.

As can be seen in Fig. 3.16, several cities exist twice. Therefore, the created solutions are invalid for the TSP. In addition, the bit strings assigned to each city have a large influence on the search procedure. This is caused by the fact that Genetic Algorithms search in the neighborhood of the parent solution for new solutions. Regarding the TSP, the enumeration of cities using a binary representation is random and therefore, changing the order of cities or flipping single bits applying a mutation operation may not lead to neighboring solutions. Moreover, flipping the first bit can have a larger influence on the solution than flipping the last bit, because of the structure of binary numbers, where the first bit represents a higher decimal value.
In order to overcome this problem, other representations and crossover operators have to be chosen to solve the TSP using Genetic Algorithms [50]. In the next Sec. 3.5.5, two representations for cities in the TSP are presented.

### 3.5.5 Representations for the TSP

In 1985, Grefenstette et al. developed one of the first representations for Genetic Algorithms including the special requirements of the TSP [51]. It is called ordinal representation and is a coding scheme for the standard one-point crossover. With this coding scheme, one-point crossover always generates feasible offspring solutions. This representation is mostly of historical interest, because the sequencing information in the two parent chromosomes is not well transferred to the offspring. Therefore, the resulting search is close to a random search.
The ordinal representation encoding is based on a reference tour. Assume that this reference tour is $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8$ for $N = 8$ cities. The tour is encoded in an iterative step by step procedure. The position of the current city in the reference tour is stored at the corresponding position in the resulting chromosome. Then, the reference tour is updated by deleting that city. After that, the procedure is repeated with the next city in the tour.
An example of the ordinal representation encoding scheme is shown in Fig. 3.17. It shows the initial tour which has to be encoded, the reference tour and the resulting chromosome. The marked city of the initial tour corresponds to the current city. The position of the current city in the reference tour is used to build the ordinal representation.
The resulting chromosome can be easily decoded to the original tour by the inverse process. As described before, two parent chromosomes encoded in this way always create feasible offspring solutions applying one-point crossover.

| Initial tour | Reference tour | Ordinal representation |
|---|---|---|
| **1** 2 5 6 4 3 8 7 | **1** 2 3 4 5 6 7 8 | 1 |
| 1 **2** 5 6 4 3 8 7 | **2** 3 4 5 6 7 8 | 1 1 |
| 1 2 **5** 6 4 3 8 7 | 3 4 **5** 6 7 8 | 1 1 3 |
| 1 2 5 **6** 4 3 8 7 | 3 4 **6** 7 8 | 1 1 3 3 |
| 1 2 5 6 **4** 3 8 7 | 3 **4** 7 8 | 1 1 3 3 2 |
| 1 2 5 6 4 **3** 8 7 | **3** 7 8 | 1 1 3 3 2 1 |
| 1 2 5 6 4 3 **8** 7 | 7 **8** | 1 1 3 3 2 1 2 |
| 1 2 5 6 4 3 8 **7** | **7** | 1 1 3 3 2 1 2 1 |

**Figure 3.17:** Ordinal representation construction [16].

The second representation is called path representation and is a natural way to encode a tour. For example, the tour $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 8$ can be represented simply as (12345678) using the cities indices directly. However, a single tour can be represented in $2N$ distinct ways, because any city can be placed at position 1 and it would be the same tour. In addition, both orientations of the tour are the same in case of a symmetric TSP. Factor $N$ can be removed by fixing a particular city at position 1 in the chromosome [16].
Using the path representation requires special recombination operations. In the next Sec. 3.5.6, two crossover operators which create feasible solutions for the TSP are described.

### 3.5.6 Special Recombination Operators

The fist presented crossover operator is called partially-mapped crossover [52, 53]. At the beginning, this operator selects two random cut points in both parent chromosomes. Then, the substring between the two cut points in the first parent chromosome replaces the corresponding substring in the second parent chromosome. In order to eliminate duplicates, the inverse replacement is applied outside of the cut points. This means that if a city occurs twice in the bit string of an offspring, it is replaced with the city which was originally on the position between the cut points. Consider the following example, where $O_{\text{temp1}}$ and $O_{\text{temp2}}$ are temporary solutions [16]:

$$
\begin{aligned}
I_1 &: (\mathbf{1}\ \mathbf{2} \mid \underline{5}\ \underline{6}\ \underline{4} \mid \mathbf{3}\ \mathbf{7}\ \mathbf{8}) \\
I_2 &: (1\ 4 \mid \underline{2}\ \underline{3}\ \underline{6} \mid 5\ 7\ 8) \\
O_{\text{temp1}} &: (1\ \underline{4} \mid \mathbf{5}\ \mathbf{6}\ \mathbf{4} \mid \underline{5}\ 7\ 8) \\
O_{\text{temp2}} &: (1\ \underline{6} \mid \mathbf{5}\ \mathbf{6}\ \mathbf{4} \mid \underline{2}\ 7\ 8) \\
O_1 &: (1\ \underline{3} \mid \mathbf{5}\ \mathbf{6}\ \mathbf{4} \mid 2\ 7\ 8)
\end{aligned}
$$

**Figure 3.18:** Example of an application of the partially-mapped crossover operator.

As can be seen in Fig. 3.18, replacing the substring between the cut points leads to an invalid solution in $O_{\text{temp1}}$, because city 4 on position 2 and city 5 on position 6 are duplicated. These cities are replaced with the corresponding cities from $I_2$ such that city 4 becomes city 6 and city 5 becomes city 2. This is reflected by $O_{\text{temp2}}$. Unfortunately, city 6 is already part of the solution and has to be replaced with city 3 from $I_2$ which is on the same location in the substring. Then, the feasible solution $O_1$ can be obtained.

The second crossover operator is called cycle crossover [53,54]. It focuses on subsets of cities that occupy the same subset of positions in both parents. These cities are copied from the first parent to the offspring at the same position. Then, the remaining positions are filled with the cities of the second parent. In this way, the position of each city comes from one of the two parents. An example of this procedure is shown in Fig. 3.19 [16]:

$$I_1 : (\mathbf{1}\ \mathbf{3}\ \mathbf{5}\ \mathbf{6}\ \mathbf{4}\ \mathbf{2}\ \mathbf{8}\ \mathbf{7})$$
$$I_2 : (1\ 4\ 2\ 3\ 6\ 5\ 7\ 8)$$
$$O_1 : (1\ \mathbf{3}\ 2\ \mathbf{6}\ \mathbf{4}\ 5\ 7\ 8)$$

**Figure 3.19:** Example of an application of the cycle crossover operator.

Regarding the example shown in Fig. 3.19, the cycle crossover works as follows. The subset of cities $\{3,4,6\}$ occupies the subset of positions $\{2,4,5\}$ in both parents. Hence, an offspring is created by filling positions 2, 4 and 5 with the cities found in the first parent. Then, the remaining positions are filled with the cities found in the second parent.

Genetic Algorithms can be used to obtain an approximate solution of a TSP instance when applying the presented recombination operators. As described before, one or more stopping criteria have to be applied to obtain a final result. For example, the algorithm runs until no improved solution can be found. In addition, a limit for the number of generations can be set.

To sum up, the distinctive feature of Genetic Algorithms is the exploitation of a population of solutions and the creation of new solutions through recombination of two parent solutions. Using Genetic Algorithms for problem solving only requires a suitable representation of the solutions and an adequate recombination operator. The final result can be optimized further by applying a local heuristic.

Both population-based metaheuristics introduced in this chapter are nature inspired and use a distributed set of samples from the space to generate the solution. In addition, these algorithms are problem independent and can handle problems with huge search spaces such as the TSP. Besides heuristic and metaheuristic algorithms, machine learning approaches can be used to approximate the solution of combinatorial optimization problems.
In the next Chapter 4, a machine learning approach is introduced. At first, the general concept of machine learning is introduced in Sec. 4.1. After that in Sec. 4.2, the concept of reinforcement learning is presented. The mathematical framework is defined in Sec. 4.3 and an example learner called Q-learning is described in Sec. 4.4. Then, the application of reinforcement learning to the TSP is presented in Sec. 4.5. Finally, the concept of supervised meta-learning is introduced in Sec. 4.6.

# 4 Machine Learning

## 4.1 Types of Machine Learning Algorithms

Machine learning is a discipline of mathematics and computer science that concentrates on methods and algorithms that can accomplish certain tasks by utilizing given data [55]. There exist three main types of machine learning approaches:

1. Supervised learning
2. Unsupervised learning
3. Reinforcement learning

In supervised learning scenarios, datasets contain pairs of inputs with corresponding outputs. The core idea of supervised learning is to learn from experience and find patterns in a training set of input-output pairs. With this knowledge, the algorithm can obtain the correct output or class for new, unseen input datasets. This process is called classification. An illustration of the procedure in supervised learning is given in Fig. 4.1.
In order to compare and evaluate supervised learners, only a portion of the known, labeled training set is used for training the algorithm. The other portion of the data is then used as a test set. The trained algorithm has to obtain the output class of this data. Comparing the output with the real output shows the performance and the accuracy of the classifier.

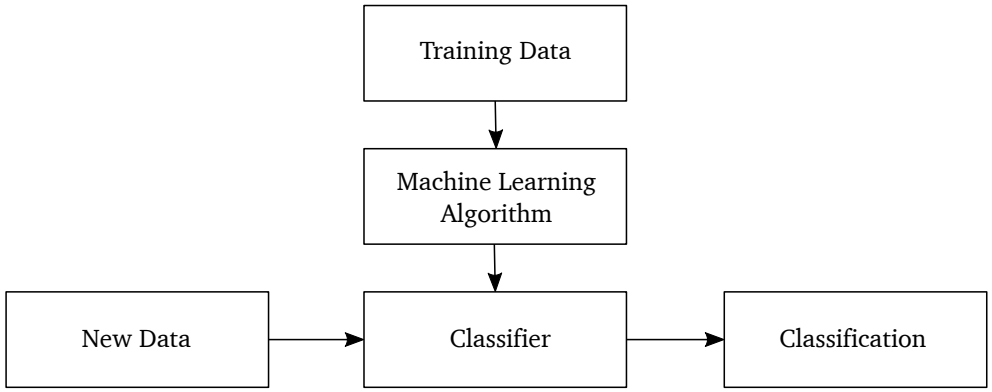



**Figure 4.1:** Supervised machine learning flow chart.

Unsupervised learning describes an approach to discover patterns in the data without having a "correct" classification at hand. The goal is to model the underlying structure or distribution in the data in order to learn more about the data. Unsupervised learning problems can be further grouped into:

- Clustering problems: Find the inherent groupings in the data.
- Association problems: Find rules that describe large portions of the data.

In contrast to supervised learning, in unsupervised learning no pairs of inputs with corresponding outputs exist. An example for an unsupervised learning algorithm is K-Means Clustering. It forms groups of

data with similar characteristics. In the mTSP example shown in Sec. 2.2.2, nearby cities are labeled the same in order to split the TSP in $m$ smaller TSPs, where $m$ is the number of salesmen.

The characteristics of reinforcement learning are described is the next Sec. 4.2. After that, the mathematical framework of a Markov Decision Process is described in Sec. 4.3. Then, a learning schemes is presented in Sec. 4.4. Finally, the application to the TSP is presented in Sec. 4.5.

## 4.2 Reinforcement Learning

Reinforcement learning is learning what to do in order to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them. Often, actions affect not only the immediate reward but also the next situation and all subsequent rewards. These two characteristics, trial-and-error and delayed reward, are the main features of reinforcement learning [56].
In contrast to supervised learning, a reinforcement learner does not learn from labeled training sets. For this reason, reinforcement learning can be applied to most combinatorial optimization problems, because supervised learning is not applicable due to the fact that one does not have access to optimal labels [57]. For example, an agent could learn to play a game by being told whether it wins or loses, but is never given the "correct" action at any given point in time. For this reason, a reinforcement learning approach is used to approximate the best tour for the TSP.
Reinforcement learning is also different from unsupervised learning. Uncovering structures in an agent's experience can certainly be useful in reinforcement learning, but by itself does not address the reinforcement learning problem of maximizing a reward signal.

As described in Sec. 3.3, the Ant System algorithm has the challenge to trade-off between exploration and exploitation. Reinforcement learning has to choose the best proportion of these two features as well. To obtain a lot of reward, a reinforcement learning agent must prefer actions that it has tried in the past and found to be effective in producing reward. On the other hand, it has to try actions that it has not selected before to discover better solutions. The agent has to exploit what it has already experienced in order to obtain a reward, but it has to explore the search space as well in order to make better action selections in the future. The dilemma is that neither exploration nor exploitation can be pursued exclusively without failing at the task. The agent must try a variety of actions and progressively favor those that appear to be best. On a stochastic task, each action must be tried many times to gain a reliable estimate of its expected reward [56].

The main elements of reinforcement learning are as follows [56]:

- **Agent:** It tries to maximize the gained reward in an unknown environment.
- **Environment:** The active decision-making agent interacts with its environment, within which the agent seeks to achieve a goal despite uncertainty about its environment. The agent's actions are permitted to affect the future state of the environment, thereby affecting the options and opportunities available to the agent at later times.
- **Policy:** It defines the learning agent's way of behaving at a given time (strategy). Informally, a policy is a mapping from perceived states of the environment to actions to be taken when in those states. In some cases, the policy may be a simple function or lookup table. The policy is the core of a reinforcement learning agent in the sense that it alone is sufficient to determine the behavior.
- **Reward signal:** It defines the goal in a reinforcement learning problem. On each time step, the environment sends a single number called the reward to the reinforcement learning agent. The

agent's objective is to maximize the total reward it receives over the long run. Thus, the reward signal defines what are good and bad events for the agent. In addition, the reward signal is the primary basis for altering the policy. If an action selected by the policy is followed by low reward, then the policy may be changed to select some other action in that situation in the future.

- **Value function:** Contrary to the reward signal, which indicates what is good in an immediate sense, a value function specifies what is good in the long run. Informally, the value of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state. Whereas rewards determine the immediate desirability of environmental states, values indicate the long-term desirability of states after taking into account the states that are likely to follow and the rewards available in those states.

In general, reinforcement learning is based on the concepts of a Markov Decision Process (MDP), which provides a mathematical framework for modeling decision making in situations where outcomes are partly random and partly under the control of a decision maker. MDPs are an extension of Markov Chains. The difference is the addition of actions (allowing choice) and rewards (giving motivation). In the next Sec. 4.3, the concept of MDPs is described.

## 4.3 Markov Decision Process (MDP)

In the following, we adapt the terminology as defined by Sutton et al. in [56]. The formal problem of finite Markov Decision Processes (MDPs) involves evaluative feedback and choosing different actions in different situations. MDPs are a classical formalization of sequential decision making, where actions influence not just immediate rewards, but also subsequent situations or states. These future rewards can be seen as delayed reward from an action. MDPs involve delayed reward and the need to trade-off immediate and delayed reward. Regarding the TSP, immediate reward is a short path to the next city and delayed reward is the overall tour length. A large immediate reward may lead to a sub-optimal result. For this reason, learning has to be applied to choose the optimal trade-off value.
In MDPs, learning is achieved by interaction with the environment to reach a goal. The learner and decision maker is called agent. It interacts with everything outside itself which is called environment. Both interact continually, the agent selecting actions and the environment responding to these actions and presenting new situations or states to the agent. In addition, the environment gives rise to rewards, which are special numerical values that the agent wants to maximize over time through its choice of actions. This process is illustrated in Fig. 4.2.

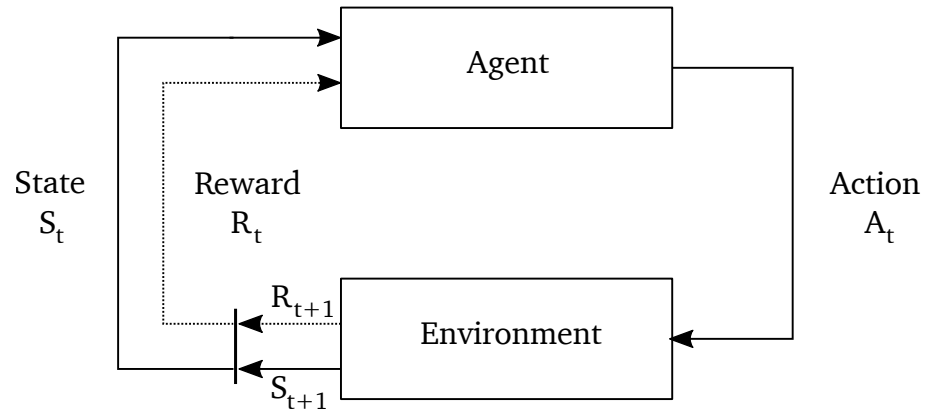



**Figure 4.2:** Markov Decision Process [56].

Agent and environment interact at each of a sequence of discrete time steps $t = 0, 1, 2, 3, \ldots, T$. At each time step $t$, the agent receives some representation of the environment's state $S_t \in \mathcal{S}$. On that basis, the agent selects an action $A_t \in \mathcal{A}(s)$. One time step later, in part as a consequence of action $A_t$, the agent receives a numerical reward $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ and finds itself in a new state $S_{t+1}$. Continuing this process, a sequence or trajectory is started as follows:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots \tag{4.1}$$

In a finite MDP, the sets of states, actions and rewards $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ all have a finite number of elements. In this case, the random variables $R_t$ and $S_t$ have well defined discrete probability distributions dependent only on the preceding state and action. That is, for particular values of these random variables, $s' \in \mathcal{S}$ and $r' \in \mathcal{R}$, there is a probability of these values occurring at time $t$, given particular values of the preceding state and action as shown in Eq. (4.2).

$$p(s', r \,|\, s, a) \doteq \Pr\{S_t = s',\, R_t = r' \mid S_{t-1} = s,\, A_{t-1} = a\} \qquad \forall\, s', s \in \mathcal{S},\, r \in \mathcal{R},\, a \in \mathcal{A}(s) \tag{4.2}$$

Function $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ is an ordinary deterministic function of four arguments. It specifies a probability distribution for each choice of $s$ and $a$. Summing over all conditional probabilities leads to the following Eq. (4.3):

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \,|\, s, a) = 1 \qquad \forall\, s \in \mathcal{S},\, a \in \mathcal{A}(s) \tag{4.3}$$

The MDP framework is abstract and flexible and can be applied to different problems in different ways. For example, time steps need not refer to fixed intervals of real time, but to arbitrary successive stages of decision making and acting. Regarding the TSP, one step would mean traveling from one city to the next city.
In general, anything that cannot be changed arbitrarily by the agent is considered to be outside of it and thus part of the environment. This environment or parts of it do not have to be unknown to the agent. For example, the agent often knows how its rewards are computed as a function of its actions and the states in which the actions are taken. Nevertheless, this computation is not part of the agent, because it defines the task faced by the agent.
To sum up, the MDP framework is an abstraction of the problem of goal-directed learning from interaction. It proposes that whatever objective one is trying to achieve, any problem of learning goal-directed behavior can be reduced to three signals passing back and forth between an agent and its environment: one signal to represent the choices made by the agent (the actions), one signal to represent the basis on which the choices are made (the states) and one signal to define the agent's goal (the rewards).

In order to apply the MDP framework to a problem, goals and rewards have to be defined. In reinforcement learning, the agent's goal is formalized in terms of a special signal, the reward. At each time step, the reward is a simple number $R_t \in \mathbb{R}$ passing from the environment to the agent. Informally, the agent's goal is to maximize the total amount of reward it receives. This means that the agent tries to maximize the cumulative rewards and not the immediate reward. The use of a reward signal to formalize the idea of a goal is one of the most distinctive features of reinforcement learning.
It is important that the reward signal is well defined. In particular, rewards provided to the agent have to be maximized if the goal is achieved. For example, a chess playing agent should be rewarded only for winning and not for achieving subgoals such as taking its opponent's pieces. If achieving these sorts of subgoals were rewarded, the agent might find a way to achieve them without achieving the real goal.

The formal definition of rewards is as follows. In general, we want to maximize the expected return, where return $G_t$ is defined as some specific function of the reward sequence. In the simplest case, the return is the sum of the rewards:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \ldots + R_T, \tag{4.4}$$

where $T$ is a final time step. This approach is useful in applications in which there is a natural notion of a final time step such as playing a game or any sort of repeated interaction. It includes that the agent-environment interaction breaks naturally into subsequences, which are called episodes. Each episode ends in a special state called terminal state, followed by a reset to a starting state or to a sample from a standard distribution of stating states. The next episode begins independently of how the previous one ended. Thus, the episodes can all be considered to end in the same terminal state, only differing in the rewards for different outcomes. Tasks with episodes of this kind are called episodic tasks. On the other hand, there exist continuing tasks which do not break the agent-environment interaction into episodes. Then, the final time step would be $T = \infty$ and the return could be infinite. Regarding the TSP, finding a tour can be seen as one episode. Therefore, the TSP is episodic.

In addition to rewards, the concept of discounting has to be defined. According to this approach, the agent tries to select actions so that the sum of discounted rewards it receives over the future is maximized. In particular, it chooses $A_t$ to maximize the expected discounted return:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{4.5}$$

where $\gamma \in [0, 1]$ is the discount rate. In addition, the returns at successive time steps are related to each other in the following way:

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \ldots \\ &= R_{t+1} + \gamma\big(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \ldots\big) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{4.6}$$

Discount rate $\gamma$ determines the present value of future rewards. A reward received $k$ time steps in the future is worth only $\gamma^{k-1}$ times what it would be worth if it were received immediately. If $\gamma < 1$, the infinite sum in Eq. (4.5) has a finite value as long as the rewards sequence $\{R_k\}$ is bounded. If $\gamma = 0$, the agent only maximizes immediate rewards. In this case, its objective is to learn how to choose $A_t$ to maximize only $R_{t+1}$. As $\gamma$ approaches 1, the agent takes future rewards into account more strongly and becomes more farsighted. However, the discount rate is user defined and the optimal value has to be found for every problem.

In order to choose the right decision, an agent in reinforcement learning tries to maximize the cumulative reward. Unfortunately, the expected reward is unknown at each time step. For this reason, almost all reinforcement learning algorithms introduce value functions, which are functions of states or state-action pairs that estimate the future rewards. The rewards, which the agent can expect to receive in future, depend on what actions it will take. Accordingly, value functions are defined with respect to particular ways of acting, the so-called policies.
Formally, a policy is a mapping from states to probabilities of selecting each possible action. If the agent is following policy $\pi$ at time $t$, then $\pi(a \mid s)$ is the probability that $A_t = a$ if $S_t = s$. Reinforcement learning algorithms specify how the agent's policy is changed as a result of its experience.

The value of state $s$ under policy $\pi$, denoted $v_\pi(s)$, is the expected return when starting in $s$ and following $\pi$ thereafter. For MDPs, this value is defined as:

$$v_\pi(s) \doteq \mathrm{E}_\pi[\, G_t \,|\, S_t = s \,] = \mathrm{E}_\pi\Big[ \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,|\, S_t = s \,\Big] \quad \forall\, s \in \mathcal{S}, \tag{4.7}$$

where $\mathrm{E}_\pi[\cdot]$ denotes the expected value of a random variable given that the agent follows policy $\pi$. Function $v_\pi$ shown in Eq. (4.7) is called the state-value function for policy $\pi$.
Similarly, the value of taking action $a$ in state $s$ under policy $\pi$, denoted $q_\pi(s,a)$, is defined as the expected return starting from $s$, taking action $a$ and thereafter following policy $\pi$:

$$q_\pi(s,a) \doteq \mathrm{E}_\pi[\, S_t = s\,, A_t = a \,] = \mathrm{E}_\pi\Big[ \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,|\, S_t = s\,, A_t = a \,\Big]. \tag{4.8}$$

Function $q_\pi$ shown in Eq. (4.8) is called the action-value function for policy $\pi$. Value functions $v_\pi$ and $q_\pi$ can be estimated from experience. For example, Monte Carlo runs can be performed to obtain the values for each state $S_t$ and action $A_t$.

A main property of value functions used in reinforcement learning is to satisfy recursive relationships. For any policy $\pi$ and any state $s$, the following consistency condition holds between the value of $s$ and the value of its possible successor states:

$$\begin{aligned} v_\pi(s) &\doteq \mathrm{E}_\pi[\, G_t \,|\, S_t = s \,] \\ &= \mathrm{E}_\pi[\, R_{t+1} + \gamma G_{t+1} \,|\, S_t = s \,] \\ &= \sum_a \pi(a\,|\,s) \sum_{s'} \sum_r p(s', r \,|\, s, a) \big[ r + \gamma \mathrm{E}_\pi[\, G_t \,|\, S_t = s' \,] \big] \\ &= \sum_a \pi(a\,|\,s) \sum_{s', r} p(s', r \,|\, s, a) \big[ r + \gamma v_p i(s') \big] \quad \forall\, s \in \mathcal{S}, \end{aligned} \tag{4.9}$$

where it is implicit that actions $a$ are taken from set $\mathcal{A}(s)$, that next states $s'$ are taken from set $\mathcal{S}$ and that rewards $r$ are taken from set $\mathcal{R}$. The first two lines in the above equation come from Eq. (4.6) and lead to the final expression in Eq. (4.9), which can be read as the expected value. It is the sum over all values of the three variables $a$, $s'$ and $r$. For each triple, the probability $\pi(a\,|\,s)\, p(s', r \,|\, s, a)$ is computed weighting the quantity in brackets by the probability and sum over all possibilities to obtain an expected value.

The final Eq. (4.9) is the Bellman equation for $v_\pi$. It expresses a relationship between the value of a state and the values of successor states. Think of looking ahead from a state to its possible successor states as illustrated in Fig. 4.3.
Starting from state $s$, the root node at the top, the agent could take any of some set of actions (three in this example) based on its policy $\pi$. From each of these, the environment could respond with one of several next states $s'$ along with a reward $r$ depending on its dynamics given by the function $p$ shown in Eq. (4.2). The Bellman equation shown in Eq. (4.9) averages over all possibilities, weighting each by its probability of occurring. It states that the value of the current state must be equal to the (discounted) value of the expected next state, plus the rewards expected along the way.
Diagrams like that shown in Fig. 4.3 are called backup diagrams, because the tree structure reveals the relationships that form the basis of the update or backup operations in reinforcement learning. These operations transfer value information back to a state from its successor states.

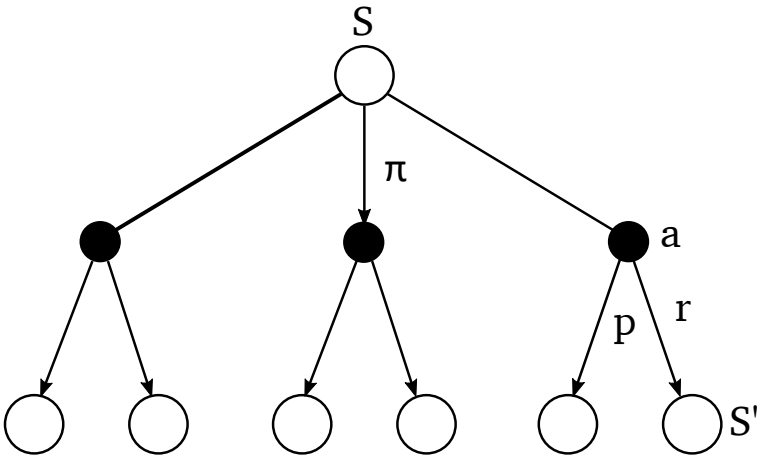



**Figure 4.3:** Example backup diagram for $v_\pi$ [56].

Solving a reinforcement learning task means to find a policy that achieves a lot of reward over the long run. For finite MDPs, the optimal policy can be defined as follows. A policy $\pi$ is defined to be better than or equal to a policy $\pi^{'}$, if its expected return is greater than or equal to that of $\pi^{'}$ for all states. Formerly, $\pi \leq \pi^{'}$ if and only if $v_\pi(s) \leq v_{\pi^{'}}(s)$ for all $s \in \mathcal{S}$. There is always at least one policy that is better than or equal to all other policies. This is an optimal policy. All optimal policies $\pi_*$ are defined as:

$$v_*(s) \doteq \max_\pi v_\pi(s) \qquad \forall\, s \in \mathcal{S} \tag{4.10}$$

Optimal policies also share the same optimal action-value function $q_*$ defined as:

$$q_*(s,a) \doteq \max_\pi q_\pi(s,a) \qquad \forall\, s \in \mathcal{S},\, a \in \mathcal{A}(s) \tag{4.11}$$

An agent learning an optimal policy has done very well, but in practice this rarely happens. It is usually not possible to compute an optimal policy for solving the Bellman optimality formulation shown in Eq. (4.9). For example, computers still cannot calculate the optimal moves, because of the high number of possible state-action pairs and the required computation effort. In addition, the memory available is also an important constraint. In tasks with small, finite state sets, it is possible to form arrays or tables with one entry for each state or state-action pair. Larger tasks require building up approximations of value functions, policies and models. A simple learning method called Q-learning is presented in the next Sec. 4.2.

## 4.4 Q-Learning

All learning methods in reinforcement learning overcome a dilemma. These methods try to learn action values conditional on subsequent optimal behavior, but there is a need to behave non-optimally in order to explore all actions (to find the optimal actions). A straightforward approach is to use two policies, one that is learned about and that becomes the optimal policy, and one that is more exploratory and is used to generate behavior. The policy being learned about is called target policy, and the policy used to generate behavior is called behavior policy. In this case, we say that learning is from data "off" the target policy and the overall process is called off-policy learning.

One of the early breakthroughs in reinforcement learning is called Q-learning and was developed by Watkins et al. [58, 59]. The agent's expected cumulative reward, when it selects action $a$ in state $s$ and acts in the optimal way in the successor states, is defined as $q_*$. Then, the following update rule estimates the action-value function $q_*$ for the nonterminal states $S_t$ [60]:

$$Q(S_t,A_t) \leftarrow (1-\alpha)\underbrace{Q(S_t,A_t)}_{\text{old Q-value}} + \underbrace{\alpha}_{\text{learning rate}} \overbrace{\Big[\underbrace{R_{t+1}}_{\text{feedback}} + \underbrace{\gamma}_{\text{discount factor}} \underbrace{\max_a Q(S_{t+1},a)}_{\text{max future Q-value}}}^{\text{expected discounted feedback}} - \underbrace{Q(S_t,A_t)}_{\text{old Q-value}}\Big], \tag{4.12}$$

where $\alpha$ is a small positive fraction called step-size parameter, which influences the rate of learning. This update rule is an example of a so-called temporal-difference learning method, because its changes are based on differences between estimates at two different times. In this case, the learned action-value function $Q$ directly approximates $q_*$, which is the optimal action-value function defined in the previous Sec. 4.3, and is independent of the policy being followed. The policy still has an effect that it determines which state-action pairs are visited and updated. This learning rule converges to the correct values for $Q$ and $v_*$, assuming that every action is tried in every state infinitely often and that new estimates are blended with previous ones using a slow enough exponentially weighted average [61].

As described in the previous Sec. 4.3 and illustrated in Fig. 4.3, Q-learning learns by backing up experienced rewards through time. An estimated action-value function $Q$ is learned, where $Q(S_t,A_t)$ is the expected return found when executing action $a$ in state $s$ and greedily following the current policy thereafter. The current best policy is generated from $Q$ by simply selecting the action that has the highest value for the current state. For example, the expected reward for each state-action pair is saved in a table, the so-called Q-table. Then, the agent can look up what to do next in order to maximize the total reward. An example of a Q-table with $m$ possible actions for each of the $n$ states is as follows:

$$\text{Q-table} = \begin{bmatrix} Q(1,1) & Q(1,2) & \dots & Q(1,m) \\ Q(2,1) & Q(2,2) & \dots & Q(2,m) \\ \vdots & \vdots & \ddots & \vdots \\ Q(n,1) & Q(n,2) & \dots & Q(n,m) \end{bmatrix} \tag{4.13}$$

To sum up, the pseudo code for applying Q-learning to estimate $\pi \approx \pi_*$ is shown in Fig. 4.4.

Initialize $Q(s,a)$, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$, arbitrarily, and $Q(\text{terminal-state},\cdot) = 0$
Repeat (for each episode):
    Initialize $S$
    Repeat (for each step of episode):
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
        Take action $A$, observe $R$, $S'$
        $Q(S,A) \leftarrow (1-\alpha)Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$
        $S \leftarrow S'$
    until $S$ is terminal
end-Q-Learning

**Figure 4.4:** Q-learning pseudo code [56].

As described before, the main problem in reinforcement learning is to balance the agent's decisions between exploration and exploitation. Exploration takes place when the agent selects an action to learn more about the environment. This is a so-called nongreedy action. On the other hand, exploitation means that the agent exploits the current knowledge of the action values. This is a so-called greedy action. One simple approach that balances these two is the so-called $\epsilon$-greedy policy. The agent selects a random action with chance $\epsilon \in [0,1]$ and the current best action looking up in the Q-table with probability $1-\epsilon$ [62].
An advantage of Q-learning is the exploration insensitiveness. It means that the Q-values will converge to the optimal values, independent of how the agent behaves while the data is being collected (as long as all state-action pairs are tried often enough). As a consequence, the exploration-exploitation issue must be addressed in Q-learning, but the details of the exploration strategy will not affect the convergence of the learning algorithm [63]. Therefore, Q-learning is the most used learning method in reinforcement learning.
In tasks with small, discrete state spaces, $Q$ and $\pi$ can be fully represented in a table. As the state space grows, using a table becomes impractical. That is why reinforcement learning methods often use function approximators such as artificial neural networks to generalize the gained experience. However, by replacing the Q-table with a deep learning network, accuracy is traded for abstraction [64]. In the next Sec. 4.5, it is shown how to apply reinforcement learning to the TSP.

## 4.5 Application to the TSP

Reinforcement learning can be used to approximate the optimal solution for the TSP. In order to apply Q-learning, the variables introduced in the previous Sec. 4.3 have to be declared [65].
When a TSP tour is optimized by the reinforcement algorithm, the state denotes the city number and the action denotes the selection of the next city. The immediate reward $r(i,j)$ between the current city $i$ and the next city $j$ is the reciprocal of the distance $d(i,j)$ between these cities as follows:

$$r(i,j) = \frac{1}{d(i,j)} \tag{4.14}$$

The shorter the distance between two cities $i$ and $j$, the larger the value of the immediate reward. As defined in Sec. 2.1, the TSP is the problem of finding a permutation of cities $\sigma$ (a tour) in a two dimensional space $w = \{v_n\}_{n=1}^{N}$ that includes each city exactly once and has the minimum length. The length of a tour defined by a permutation $\sigma$ is as follows:

$$L(\sigma, w) = \left\| v_{\sigma(N)} - v_{\sigma(1)} \right\|_2 + \sum_{n=1}^{N-1} \left\| v_{\sigma(n)} - v_{\sigma(n+1)} \right\|_2, \tag{4.15}$$

where $\|\cdot\|$ denotes the euclidean norm. Then, the cumulative reward $RQ(i,j)$ can be defined as the reciprocal of the tour length:

$$RQ(i,j) = \frac{1}{L(\sigma, w)} \tag{4.16}$$

The cumulative reward $RQ(i,j)$ is updated according to the following function:

$$RQ(i,j) \leftarrow (1-\alpha) RQ(i,j) + \alpha \left[ r(i,j) + \gamma \max_{z \in L(i)} RQ(j,z) \right], \tag{4.17}$$

where $\alpha \in [0, 1]$ is the learning rate, $\gamma \in [0, 1]$ reflects the discount rate and $L(i)$ denotes the unvisited cities when the current city is $i$. In order to learn an optimal policy, a sufficient number of iterations of the algorithm (episodes) can be used [3]. An episode ends when a tour is completed and the algorithm can be restarted.
Using a Q-table, the algorithm can select the best action according to the highest Q-value. To explore new paths, the $\epsilon$-greedy policy defined in Sec. 4.4 may be used.

To sum up, reinforcement learning algorithms can be applied to solve combinatorial problems such as the TSP. In this case, the salesman learns the best state-action pairs and selects a particular action to maximize the cumulative reward. In contrast to supervised learning, this self-learning method does not need a training set with correct labels. However, supervised learning can be used to select the best meta-heuristic for an optimization problem. The so-called supervised meta-learning approach is described in following Sec. 4.6.

## 4.6 Supervised Meta-Learning

The TSP has been extensively studied in literature and various approximation algorithms such as the ones shown in Chapter 3 are available. However, none of the state-of-the-art metaheuristics for TSP outperforms the others in all problem instances within a given time limit [66]. This so-called *'No Free Lunch Theorem'* is described in the next Chapter 5. Predicting the best performing algorithm for a given problem instance can save computational resources and optimize the results.
In general, this type of concept is the so-called Algorithm Selection Problem formulated by Rice [67]. It includes three main characteristics [68]:

- **Problem space:** The set of all possible instances of the problem. There exist a large number of independent parameters describing the different instances. These parameters are important for algorithm selection and performance measure. However, their influence on algorithm performance is usually unknown.
- **Algorithm space:** The set of all possible algorithms that can be used to solve the problem. The dimension of this set could be unimaginable and the influence of the algorithm parameters is uncertain.
- **Performance measure:** The criteria used to measure the performance of a particular algorithm for a particular problem. It is similar to the fitness value used in Genetic Algorithms presented in Sec. 3.5.

The goal is to relate the performance of algorithms to characteristics or measures of classification datasets. Then, the best algorithm can be predicted for a new problem instance. The mathematical framework can be described as follows. There exist a problem domain $\mathcal{P}$, a set of available algorithms $\mathcal{A}$, a performance space $\mathcal{Y}$ and a feature space $\mathcal{F}$. Performance $y(A, P)$ of algorithm $A \in \mathcal{A}$ on problem $P \in \mathcal{P}$ is given by mapping $y : \mathcal{A} \to \mathcal{Y}$. Features $f(P)$ characterizing the problem $P$ are given by mapping $f : \mathcal{P} \to \mathcal{F}$. The task is to find the mapping $s : \mathcal{F} \to \mathcal{A}$, which for each problem $P \in \mathcal{P}$ selects an algorithm $A_P = s(f(P))$ such that the performance $y(A_P, P)$ is maximized.

In the following, the concept of supervised meta-learning to select the best algorithm solving the TSP is described based on the work of Pihera et al. in [66] and Kanda et al. in [69]. Both apply a classification algorithm to the Algorithm Selection Problem for the TSP. The task is to find the best metaheuristic to solve a particular TSP instance including a runtime limit. Using supervised learning introduced in Sec. 4.1, a suitable algorithm can be chosen to obtain the best possible solution. To make this prediction, a classifier has to be trained using a machine learning algorithm (e.g. a support vector machine). For this reason, it is necessary to find and define relevant features $f(P)$, which characterize the TSP instance well, and to construct a good mapping $s$ in order to apply the algorithm selection. In addition, the set of training instances according to which $s$ is constructed must be representative for the problem. The training instances must not bias the mapping $s$ towards a special class of inputs only.
Training the classifier requires the TSP instances to be characterized by features $f(P)$. These features should represent properties of the instances that affect the relative performance of the metaheuristics [69]. For example, the features shown in Table 4.1 can be used to train a meta-learning model. The values for each feature can be extracted from the graph that represents each TSP instance.

In the next step, a training set of TSP instances is solved using the available metaheuristics. Then, each training sample is labeled with the metaheuristic which leads to the shortest tour in the user defined runtime. This labeled training set can be used to train the classifier of the meta-learning model. An illustration of the general meta-learning construction and classification process is given in Fig. 4.5.

| Feature | Description |
|---|---|
| $V$ | Number of cities |
| $E$ | Number of edges |
| $E_{low}$ | Lowest edge cost |
| $E_{hig}$ | Highest edge cost |
| $E_{avg}$ | Average edge cost |
| $E_{mode}$ | Average of the costs that occur the most frequently in the edges |
| $E_{mediam}$ | Median of the edge costs |
| $E_{lavg}$ | Quantity of edges whose costs are lower than $E_{avg}$ |
| $V_{lower}$ | Sum of the $V$ edges of lowest values |

**Table 4.1:** TSP features used to train a meta-learning model [69].

The construction of the meta-learning model is carried out in two phases. The first is the knowledge acquisition, in which problem instances are obtained from the problem space. Afterwards, problem characteristics are identified and the features are extracted. Then, candidate algorithms are applied to the selected problem instances and their performance is estimated. The resulting values are used to generate target values. In the second phase, the meta-learning model is created. This is achieved by applying machine learning techniques on the data gathered in the first phase. After that, the meta-learning model obtained can be used to predict the relative performance of the candidate algorithms on a new instance and recommend the most promising algorithm.

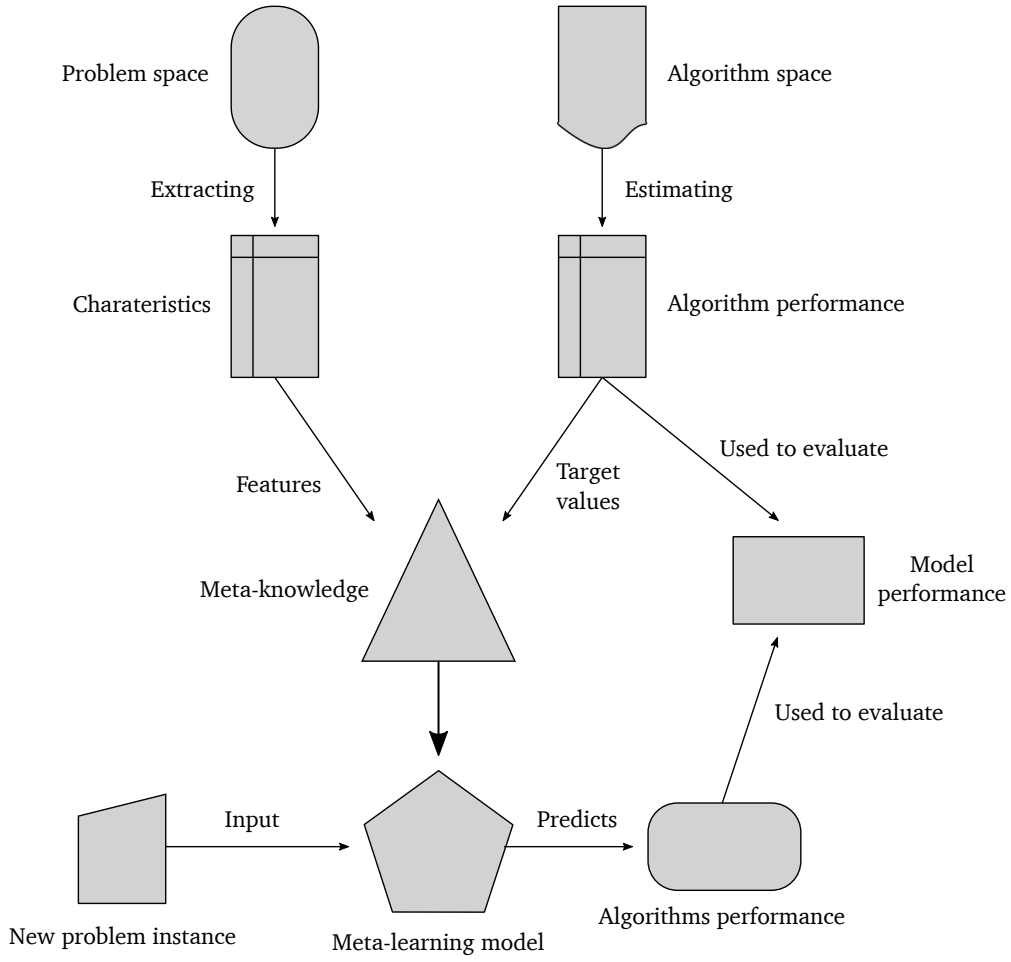



**Figure 4.5:** Construction of the meta-learning model for the Algorithm Selection Problem [69].

As described before, given a trained meta-learning model, the classifier predicts the best algorithm for new, unseen TSP instances. For example, the decision tree learner builds classification models in the form of a tree structure. This learner searches for patterns in the given examples to construct a tree with decision nodes and leaf nodes. A decision node has two or more branches. A leaf node represents a classification or decision with a class label. The topmost decision node in a tree, which corresponds to the best predictor, is called root node. Given a trained decision tree, a new TSP instance can be classified with very low computational effort, because the algorithm only has to compare two values at each decision node. An example for a decision tree obtained from a dataset including 70 cities is shown in Fig. 4.6, where the classifier can choose out of four considered metaheuristics. Regarding this example, only five features are required to classify new TSP instances.

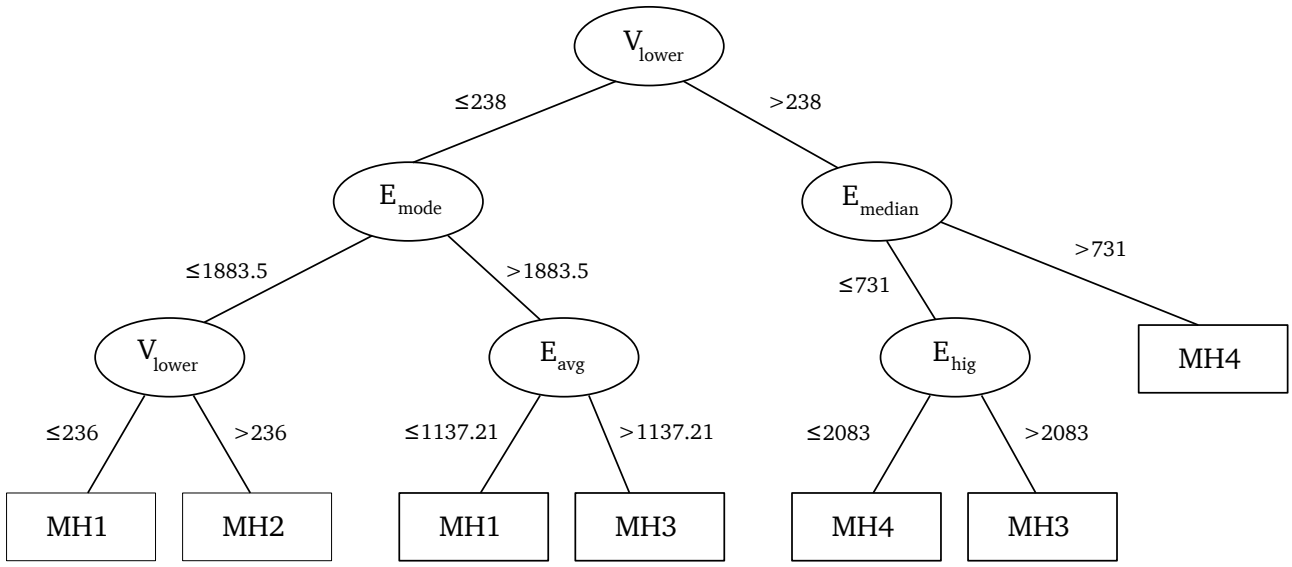



**Figure 4.6:** Decision tree for a dataset including 70 cities [69].

Experimental results show that a high accuracy of nearly 90% can be reached, where accuracy means the percentage of the correctly labeled test samples [69]. Additional features for the TSP instances can further improve the prediction accuracy [66].

To sum up, the proposed meta-learning approach supporting the selection of the most promising metaheuristic can be used successfully to find the shortest tour for unseen TSP instances. This selection process can be further generalized. The so-called hyper-heuristics can be described as heuristics choose heuristics. In the next Chapter 5, the concept of hyper-heuristics is presented.

# 5 Hyper-Heuristics

## 5.1 Characteristics of Hyper-Heuristics

Metaheuristics discussed in Chapter 3 are flexible frameworks that can be used to approximate a solution for any combinatorial optimization problem. However, researchers in metaheuristics spend a large amount of time to properly design and tune their algorithms in trial-and-error fashion [70]. In addition, designing an efficient metaheuristic requires knowledge about the problem domain. There exist difficulties in terms of easily applying metaheuristics to new problems or new instances of similar problems. These difficulties arise mainly from the significant range of parameters such as the population size in Genetic Algorithms presented in Sec. 3.5. Among all parameter settings, only a few will lead to the optimal solution or the best possible solution, but not for all instances and not all the time. Moreover, it is difficult to understand why one metaheuristic does not work effectively when applying it to a new problem. The configuration may has a great influence on the quality of the final solution or the effectiveness of the search method [70].

This characteristic of metaheuristics leads to the so-called *'No Free Lunch Theorem'* introduced by Wolpert et al. in [71]. It states that, when averaged over all problems defined on a given finite search space, all search algorithms have the same average performance. This is an intuitive result since the majority of combinatorial problems have no exploitable structure such as some form of global or local continuity, differentiability or regularity. This type of problem can only be defined by a complete lookup table. The *'No Free Lunch Theorem'* helped to focus attention on the question of what sorts of problems any given algorithm might be particularly useful for [72].

Since different metaheuristics lead to different performances on different problems, an optimization approach would be to design an algorithm containing an infinite switch statement enumerating all finite problems and applying the best known heuristic for each. Then, the best possible solution for a particular problem can be obtained. This approach emerged to the new class of hyper-heuristics. In the first class of hyper-heuristics, or heuristics choose heuristics, the framework is provided with a set of pre-existing, generally widely known heuristics for solving the target problem. This definition was extended to the second class of hyper-heuristics, where the goal is to generate new heuristics from a set of building-blocks or components of known heuristics, which are given to the framework [73]. In the following, the more general second class of hyper-heuristics is described.

Hyper-heuristics can overcome the problems of finding an adequate metaheuristic for a given problem and represent a class of methods that are non-problem specific [74]. The main idea is to develop algorithms that are more generally applicable than many of the current implementations of search methods. This can be done using a set of easy-to-implement low-level heuristics. Then, a hyper-heuristic can be seen as a high-level method which, when given a particular problem instance and a number of low-level heuristics, automatically produces an adequate combination of the provided components to solve the given problem effectively [75]. This includes selecting and applying an appropriate low-level heuristic at each decision point. In contrast to metaheuristics, which are mostly developed for a particular problem and require fine tune of related parameters, hyper-heuristics raise the level of generality and can overcome the *'No Free Lunch Theorem'* to obtain the best possible solution for different problems [76].

In order to describe a hyper-heuristic, three main characteristics can be identified [75]:

1. It is a higher level heuristic that manages a set of low-level heuristics.
2. It searches for a good method to solve the problem rather than for a good solution.
3. It uses only limited problem-specific information.

The key distinction between hyper-heuristics and metaheuristics is the operation space. Hyper-heuristics are search methods that operates on a search space of heuristics (or algorithms or their components), whereas most implementations of metaheuristics search on a space of solutions to a given problem. The goal then is finding or generating high-quality heuristics for a problem, for a certain group of instances of a problem, or even for a particular instance.
This concept is illustrated in Fig. 5.1, where a hyper-heuristic samples a subset of elements in the space of heuristics in order to find a good solver for a problem represented as a search space provided with a fitness function, which reflects the quality of the solution. Each solver sampled by the hyper-heuristic needs to be executed in order to evaluate the quality of such a solver. This involves sampling of a subset of elements of the space of solutions. Since fitness values are associated to these, information flows up the hierarchy to guide the hyper-heuristic. Note that there exist many hyper-heuristics in the space of hyper-heuristics. Each may sample the space of heuristics differently, which in turn may lead to sampling the solution space differently [76].

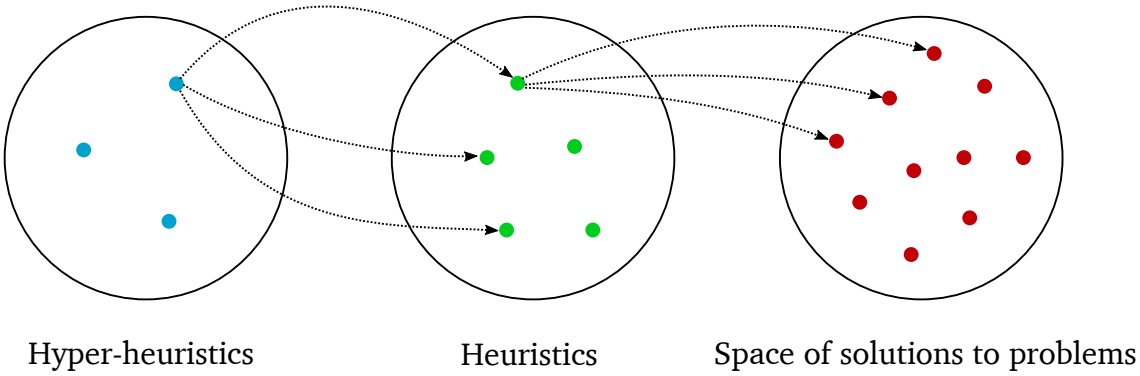



**Figure 5.1:** A hyper-heuristic exploring the space of heuristics for a particular problem [76].

Metaheuristics are often used as the search method in hyper-heuristic approaches. In this case, a metaheuristic is used to search a space of heuristics [77]. For example, Genetic Algorithms presented in Sec. 3.5 can be used. These algorithms generate a population of heuristics or configurations of a metaheuristic and assign a fitness value to each parent solution. Then, crossover and mutation operators are applied to selected parent solutions to create offspring. This process is guided by a performance measure $y(A_P, P)$ that indicates the performance of heuristic $A_P$ in relation to problem $P$ [76].

Each low-level heuristic communicates with the hyper-heuristic using a common problem-independent interface architecture. The hyper-heuristic can choose to call a low-level heuristic in order to see what would happen if the low-level heuristic is used and it can allow the low-level heuristic to change the current solution. The hyper-heuristic may also provide additional information to the low-level heuristic such as the maximum runtime. It is important to note that the hyper-heuristic only knows whether each objective function is to be maximized or minimized and has no direct information about the objective function. The communication between the problem domain and the hyper-heuristic is made through a

barrier, through which domain knowledge is not allowed to cross as illustrated in Fig. 5.2 [78]. The problem-independent interface allows the development for other domains. When implementing a new problem, the user has to supply a set of low-level heuristics and a suitable evaluation function. If the standard interface is used, the hyper-heuristic does not have to be changed in any way. It is able to solve the problem with the given low-level heuristics. This higher level of abstraction is the main goal of hyper-heuristics. There is no need to outperform a custom-made solver or a problem-specific metaheuristic. The applicability over a wide range of problem domains is more crucial. For this reason, hyper-heuristics need to be applied to several problem domains to observe the generalization ability [73, 75].

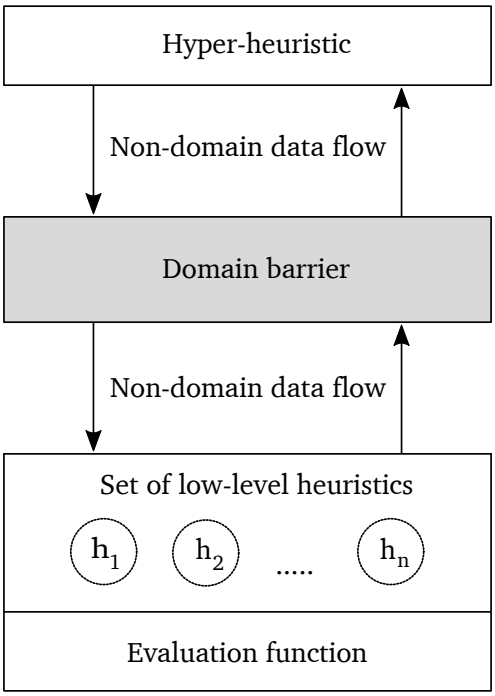



**Figure 5.2:** Hyper-heuristic framework [72].

Hyper-heuristics can be compared to the supervised meta-learning approach presented in Sec. 4.6 [77]. Both try to overcome the *'No Free Lunch Theorem'* finding the best performing algorithm for a given problem. This concept of the Algorithm Selection Problem described in Sec. 4.6 introduces the operation on the algorithm space rather than the solution space of the problem. To guide the learner or the hyper-heuristic, a performance measure (e.g. a fitness function) is evaluated.
The main differences are as follows. Supervised meta-learning is based on a classification learner which learns from experience. The classifier is learned using a training set of problems and the according performance of each algorithm. Then, a classification task can be performed on new problem instances. It is important to evaluate a candidate solution with respect to its generalization ability across many different data instances of the same application domain. Furthermore, the training and testing instances must be drawn independently from the same distribution.
Contrary to this, hyper-heuristics optimize a single problem instance and learning is not required. Moreover, hyper-heuristics can change their preferences during the optimization and are not limited to a fixed setup with a learned classifier as in the supervised meta-learning case, where small changes may require a new learning process. In addition, hyper-heuristics can choose a low-level heuristic at each decision step. The supervised meta-learning approach presented in Sec. 4.6 only selects the best performing algorithm for the whole optimization problem. For example, it could select different heuristics for each construction step of the TSP to take a certain city arrangement into account. However, supervised meta-learning is only one approach in the class of meta-learning methods.

As described before, hyper-heuristics can learn during optimization. This characterisitc can be used to group hyper-heuristics into [73]:

- **Online learning hyper-heuristics:** The learning takes place while the algorithm is solving a problem instance. Therefore, task-dependent local properties can be used by the high-level strategy to determine the appropriate low-level heuristic to be applied. Examples of online learning approaches within hyper-heuristics are: the use of reinforcement learning for heuristic selection and the use of metaheuristics as high-level search strategies across a search space of heuristics.
- **Offline learning hyper-heuristics:** The idea is to gain knowledge in the form of rules or programs and from a set of training instances, which generalize to the process of solving unseen instances. Examples for offline learning approaches within hyper-heuristics are: learning classifier systems and Genetic Algorithms.
- **No learning hyper-heuristics:** No feedback is used from the search process.

Another classification of hyper-heuristics concerns the nature of the heuristic search space. Hyper-heuristics can either select the best performing low-level heuristic at each decision step or generate new heuristics. These new heuristics are suitable for the given problem and perform better than existing heuristics. In order to automatically define new heuristics, genetic programming can be used [79]. In addition, hyper-heuristics can be divided into constructive and perturbative search methods. Perturbative methods work by considering complete candidate solutions and changing them by modifying one or more of their solution components, while constructive methods work by considering partial candidate solutions which are extended iteratively [75]. The classification of hyper-heuristic approaches is summarized in Fig. 5.3. In the next Sec. 5.2, heuristic selection methods are presented.

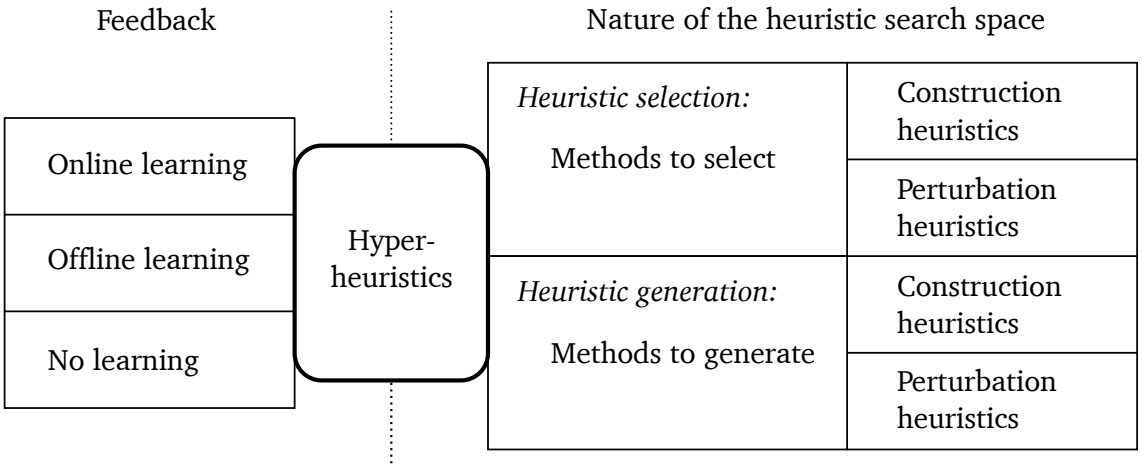



**Figure 5.3:** Classification of hyper-heuristic approaches [73].

## 5.2 Heuristic Selection Methods

Hyper-heuristic selection approaches based on constructive low-level heuristics build a solution incrementally. Similar to construction heuristics presented in Sec. 2.4.1, these algorithms start with an empty solution to gradually build a complete solution. At every decision step, a suitable pre-existing low-level construction heuristic provided to the hyper-heuristic framework is selected to make the best possible

decision for the current problem state. This process continues until the final state, a complete solution, has been reached. Then, the construction process terminates and the solution can be obtained. It follows that the sequence of heuristic choices is finite and determined by the size of the underlying combinatorial problem. Applying online learning during the construction can reveal associations between partial solution stages and adequate heuristics for those stages. In addition, studies in production scheduling show that a combination or sequencing of several constuction heuristics performs better than applying only a single one [73, 75].

The second hyper-heuristic selection approach is based on perturbative low-level heuristics. Similar to improvement heuristics presented in Sec. 2.4.2, these algorithms start with a complete solution and try to improve this solution iteratively. The initial solution can be generated randomly or using simple construction heuristics. The hyper-heuristic framework is provided with a set of neighborhood structures and/or simple local search algorithms. Then, the hyper-heuristic selects these local search algorithms and applies them to the current solution. Similar to Genetic Algorithms presented in Sec. 3.5, this iterative process continues until a stopping condition has been met. For example, the runtime or the improvement quality can be evaluated.
Approaches combining perturbation low-level heuristics in a hyper-heuristic framework use online learning, where knowledge gained during the optimization process can be used directly. For example, reinforcement learning presented in Sec. 4.2 is commonly used to guide the search. In addition, the majority of these hyper-heuristics are single-point algorithms. This means that an initial candidate solution goes through a set of successive stages repeatedly until termination. At first, a heuristic is selected from a set of low-level perturbative heuristics and then applied to a single candidate solution. Then, a decision is made about whether to accept or reject the new solution.
The acceptance strategy is an important component of any local search heuristic. In general, an improved solution is accepted or rejected based on the quality of this solution in comparison to the current solution during a single-point search. There exist two types of acceptance strategies. Deterministic methods make the same decision for acceptance regardless of the decision point during the search using current and new candidate solutions. On the other hand, a non-deterministic method might generate a different decision for the same input. This requires additional parameters such as the time or the current iteration.

In the next Sec. 5.3, the second class of hyper-heuristics is presented: heuristic generation methods.

## 5.3 Heuristic Generation Methods

The main characteristic of hyper-heuristic generation methods concerns the low-level heuristics. Hyper-heuristic generation methods search the space of heuristics constructed from components rather than a space of complete and pre-defined heuristics. New construction or perturbation low-level heuristics can be generated. Genetic programming is an evolutionary computation technique that evolves a population of computer programs and is the most common method used in literature to automatically generate heuristics. Most approaches using genetic programming as a hyper-heuristic use offline learning presented in Sec. 5.2. At the end of a run, a heuristic generator also outputs the new heuristic that produced the solution. This heuristic can be reused on new problem instances and would potentially lead to feasible solutions. However, the performance can be poor if the particular hyper-heuristic method has not been designed with reusability in mind. To be successful when reused, the hyper-heuristic would usually train the generated heuristic offline on a set of representative problem instances.

Hyper-heuristics which automatically generate heuristics can overcome the problem of fine-tuning the parameters of a (meta-)heuristic as described in Sec. 5.1 [70]. The characteristics of problem instances vary and obtaining the best possible result would require to test all input parameters of a heuristic or the

construction of a new heuristic specialized to that instance. In addition, an automated heuristic design process makes it potentially feasible and cost effective to design a heuristic for each problem instance. As the process is automated, it is less demanding on human resources and time. As it is more specialized, a generated heuristic could even lead to better solutions than that obtained by any current human-created heuristic [73, 75].

Regarding the TSP, a linear genetic programming method is presented by Keller et al. in [80]. The introduced hyper-heuristic evolves programs that represent the repeated application of a number of simple local search operations. In general, a method for automatically generating heuristics is to evolve a fundamental part of an existing metaheuristic. This has the potential to produce a much better variation of the original algorithm. For example, an evolved edge selection formula of an Ant Colony System is presented by Runka in [81]. While the evolved formula was tested on only two unseen TSP instances, the results were better than the original human-designed formula [75].

To sum up, hyper-heuristics provide a high-level approach to select or generate the best possible heuristics at each decision step. Applying a hyper-heuristic requires only limited problem-specific information. Therefore, hyper-heuristics can be used to solve different optimization problems without changing the configuration of the hyper-heuristic. Only the low-level heuristics may have to be changed in order to be suitable for the specific problem domain.

In the next Chapter 6, a summary of this research project is presented and a conclusion is drawn.

## 6 Summary and Conclusion

We presented different approaches for solving the Traveling Salesman Problem (TSP). This type of problem can by described as the problem of finding a minimal length Hamiltonian cycle of a graph, where a Hamiltonian cycle is a closed tour visiting each city exactly once. It belongs to the $\mathcal{NP}$-complete optimization problems and its complexity increases factorially with the number of cities. Many combinatorial problems can be modeled as a TSP or an extended TSP. For this reason, researchers benchmark new optimization algorithms or (meta-)heuristics against other methods by using them on instances of a TSP.
The TSP can be formulated as an Integer Optimization Problem as shown in Sec. 2.1. The standard case is the symmetric TSP, where the distances between two cities are equal for both directions. Presented extensions of this standard case are the asymmetric TSP shown in Sec. 2.2.1, the multiple TSP described in Sec. 2.2.2 and the TSP with Precedence Conditions introduced in Sec. 2.2.3. It was shown that each extension can be modeled as a standard symmetric TSP in order to apply optimization approaches developed only for the symmetric case.

Exact algorithms presented in Sec. 2.3 can obtain the optimal solution for a TSP. However, due to the $\mathcal{NP}$-hardness of the problem, this type of algorithm can only be used for small TSP instances. For this reason, the heuristics introduced in Sec. 2.4 were developed. Heuristics are methods finding an approximate solution faster than an exact algorithm. The solution can be sub-optimal, but in many cases there is no need for an optimal solution. Heuristics can be divided in two groups: tour construction and tour improvement heuristics. Construction heuristics start with an empty solution to gradually build a complete solution using a simple and greedy procedure. Contrary, tour improvement heuristics are simple local search heuristics which try to improve an initial tour. Both types of heuristics can be effective to solve TSPs. However, the obtained solution can be far from the global optimum, because heuristics are bounded to a certain local configuration as described in Sec. 3.1.

In order to overcome this problem, metaheuristics can be used as more general solution schemes. In contrast to heuristics, metaheuristics introduce some random factors to escape from local optima as described in Sec. 3.1. There exist two types of metaheuristics: single-solution and population-based metaheuristics. In the first case, only a single solution is concerned during the search. Contrary, in population-based metaheuristics different solutions are combined to create a new solution.
The Greedy Randomized Adaptive Search Procedure (GRAPS) presented in Sec. 3.2 belongs to the class of single-solution based metaheuristics. It is a construction metaheuristic which adds the best possible element to the solution at each iteration. To escape local optima, a large random change (perturbation) is applied. In addition, GRASP is a multiple-start procedure to obtain the best possible solution.
Applying population-based metaheuristics leads to a more random search and exploration of the search space. The first presented population-based metaheuristic is the Ant System (AS) defined in Sec. 3.3. It is inspired by the natural behavior of ants. They can find the shortest path between the nest and a food source using a pheromone. Originally developed for the TSP, the Ant System is an algorithm in which a set of artificial ants cooperate to obtain the solution of a problem by exchanging information via a pheromone deposited on graph edges. These ants prefer to move to cities which are connected by short edges with a high amount of pheromone. Each ant generates a complete tour. At the end of each tour, a global pheromone updating rule is applied. Ants deposit an amount of pheromone on edges which belong to its tour in proportion to the length of the tour. Then, the algorithm is restarted using the updated pheromone amounts.
Improving the Ant System leads to the Ant Colony System (ACS) presented in Sec. 3.4. It introduces three modifications to enhance the performance of the Ant System. Firstly, a state transition rule provides a direct way to balance between exploration and exploitation of new edges. Secondly, the global update

rule is applied only to edges which belong to the best ant tour. Thirdly, a local pheromone updating rule is applied while ants construct a solution.
Another class of population-based metaheuristics are Genetic Algorithms (GA) described in Sec. 3.5. These kind of algorithms belong to the class of Evolutionary Algorithms and are based on the Darwinian principle of natural selection. The so-called *'survival of the fittest'* states that in particular the best adapted individuals will produce offspring. Genetic Algorithms use a population of solutions and select the parent solutions according to selection strategies presented in Sec. 3.5.1. Then, recombination and mutation operators are applied to generate offspring. Similar to the Ant (Colony) System, external stopping criteria have to be included to obtain a final solution. In order to apply Genetic Algorithms to the TSP, special representations and special recombination operators have to be used.

Besides metaheuristics, machine learning can be used to approximate a solution for the TSP. Reinforcement learning presented in Sec. 4.2 is learning what to do in order to maximize a reward. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them. Reinforcement learning introduces an active decision-making agent interacting with its environment. It is based on the Markov Decision Process (MDP) defined in Sec. 4.3, which provides a mathematical framework for modeling decision making. The agent follows a policy, which defines the learning agent's way of behaving at a given time to maximize the cumulative reward. Q-learning presented in Sec. 4.4 is a learning method to obtain an approximation of the optimal policy. Applying Q-learning to the TSP leads to a complete lookup table. Using this table, an agent can select the best action in its current state according to the highest Q-value.
Another machine learning method uses a classification algorithm to choose the best metaheuristic for a given TSP instance. The supervised meta-learning approach presented in Sec. 4.6 tries to overcome the *'No Free Lunch Theorem'*. This problem states that no metaheuristic for TSP outperforms the others in all problem instances within a given time limit. However, it is possible to predict the best performing metaheuristic for each problem instance. The Algorithm Selection Problem includes the search on an algorithm space rather than on the problem space. Using a training set of TSP instances with extracted features, a classification model can be obtained. Then, the performance of each metaheuristic can be predicted and the best performing algorithm can be chosen to solve the TSP.

Similar to supervised-meta learning, hyper-heuristics presented in Chapter 5 try to overcome the *'No Free Lunch Theorem'*. The main idea is to develop algorithms, which are applicable more generally. Hyper-heuristics can be seen as a high-level method which automatically produces an adequate combination of the provided low-level heuristics to solve the given problem effectively. These general solution schemes use only limited problem-specific information and operate on the search space of heuristics. Hyper-heuristics can be divided in heuristic selection and heuristic generation hyper-heuristics. In the first case, the high-level hyper-heuristic selects a suitable low-level heuristic to either build a solution incrementally or to improve an initial solution. Contrary, heuristic generation methods search the space of heuristics constructed from components rather than the space of complete and pre-defined heuristics. Genetic programming can be used to generate new heuristics and to overcome the problem of fine-tuning the parameters of a (meta-)heuristic. As it is more specialized, a generated heuristic could lead to a better solution than the one obtained by any human-created heuristic. For example, an evolved formula for the Ant Colony System leads to shorter tours in two new, unseen TSP instances.

In conclusion, there exist multiple approaches to approximate the optimal solution for the Traveling Salesman Problem and other combinatorial problems. Despite the fact that new approaches such as hyper-hyper-heuristics are developed, simple heuristics and metaheuristics are powerful algorithms to obtain a close-to-optimal solution. In many cases there is no need for an optimal solution. As long as $\mathcal{NP}$-complete problems such as the TSP are hard to solve, heuristic methods provide a good compromise between accuracy and runtime.

## Bibliography


[1] 'The Euclidean Traveling Salesman Problem is NP-complete', C. Papadimitriou, Theoretical Computer Science Vol. 4, 1977

[2] 'NP-Hard Problems', J. Erickson, 2014

[3] 'A Study of Approaches to Solve Traveling Salesman Problem using Machine Learning', M. Aggarwal et al., International Journal of Control Theory and Applications Vol. 9, 2016

[4] 'The Traveling Salesman Problem: A Computational Study', D. Applegate et al., Princeton University Press, 2006

[5] 'The Icosian Game, Revisited', E. Pegg Jr, The Mathematica Journal, 2009

[6] 'TSP World Tour', University of Waterloo, California, http://www.math.uwaterloo.ca/tsp/world/, accessed in June 2018

[7] 'The Vehicle Routing Problem: An Overview of exact and approximate Algorithms', G. Laporte, European Journal of Operational Research Vol. 59, 1991

[8] 'The Multiple Traveling Salesman Problem: an Overview of Formulations and solution Procedures', T. Bektas, Omega International Journal of Management Science, 2005

[9] 'Neural Networks for Combinatorial Optimization: A Review of More Than a Decade of Research', K. A. Smith, Informs Journal on Computing Vol. 11, 1999

[10] 'Traveling Salesman Problems with Profits', D. Feillet et al., Transportation Science Vol. 39, 2005

[11] 'On Prize-collecting Tours and the Asymmetric Travelling Salesman Problem', Dell'Amico et al., International Transactions in Operational Research Vol. 2, 1995

[12] 'An Algorithm for single constraint Maximum Collection Problem', S. Kataoka et al., Journal of the Operations Research Vol. 31, 1988

[13] 'The Prize Collecting Traveling Salesman Problem', G. Balas, Management Science Research Report No. MSRR-539, 1987

[14] 'Comparison of metaheuristic Methods by solving Travelling Salesman Problem', O. Míča, The International Scientific Conference Inproforum, 2015

[15] 'Comparison of Neural Networks for Solving the Travelling Salesman Problem', B. La Maire et al., 11th Symposium on Neural Network Applications in Electrical Engineering, 2012

[16] 'Genetic Algorithms for the Traveling Salesman Problem', J. Potvin, Annals of Operations Research Vol. 63, 1996

[17] 'Selection of Algorithms to Solve Traveling Salesman Problems using Metalearning', J. Kanda et al., International Journal of Hybrid Intelligent Systems Vol. 8, 2011

[18] 'The Convex-Hull-and-Line Traveling Salesman Problem: A Solvable Case', V. Deineko et al., Information Processing Letters Vol. 51, 1994

[19] 'Transforming Asymmetric into Symmetric Traveling Salesman Problems', R. Jonker et al., Operations Research Letters Vol. 2, 1983

[20] 'The Traveling Salesman Problem and its Variations', G. Gutin et al., Springer, 2002

[21] 'A Review on Algorithms Used to Solve Multiple Travelling Salesman Problem', A. Singh, International Research Journal of Engineering and Technology, 2016

[22] 'Transformation of Multi Salesmen Problem to the standard Traveling Salesman Problem', M. Bellmore et al., Journal of Association for Computing Machinery Vol. 21, 1974

[23] 'A note on the Multiple Traveling Salesman Problem', M. R. Rao, Operations Research Vol. 28, 1980

[24] 'Transformation of Multidepot Multisalesmen Problem to the Standard Travelling Salesman Problem', Y. Guoxing, European Journal of Operational Research Vol. 81, 1995

[25] 'Optimization of Non-Linear Multiple Traveling Salesman Problem Using K-Means Clustering, Shrink Wrap Algorithm and Meta-Heuristics', R. Nallusamy et al., International Journal of Nonlinear Science Vol. 9, 2010

[26] 'A Comparison between Heuristic and Meta-Heuristic Methods for Solving the Multiple Traveling Salesman Problem', S. Sze et al., International Journal of Mathematical and Computational Sciences Vol. 1, 2007

[27] 'An efficient Genetic Algorithm for the Traveling Salesman Problem with Precedence Constraints', C. Moon et al., European Journal of Operational Research Vol. 140, 2002

[28] 'The Precedence Constrained Traveling Salesman Problem', M. Kubo, Journal of the Operations Research Society of Japan Vol. 34, 1991

[29] 'TSP - Infrastructure for the Traveling Salesperson Problem', M. Hahsler et al., Journal of Statistical Software Vol. 23, 2007

[30] 'A Dynamic Programming Approach to Sequencing Problems', M. Held et al., Journal of the Society for Industrial and Applied Mathematics Vol. 10, 1962

[31] 'Solution of a Large-Scale Traveling-Salesman Problem', G. Dantzig et al., 1954

[32] 'An Analysis of Several Heuristics for the Traveling Salesman Problem', D. Rosenkrantz, Siam J. Computing Vol. 6, 1977

[33] 'A Method for Solving Traveling-Salesman Problems', G. A. Croes, Operations Research Vol. 6, 1958

[34] 'The Traveling Salesman Computational Solutions for TSP Applications', G. Reinelt, Springer-Verlag, 1994

[35] 'A Tour Construction Heuristic for the Travelling Salesman Problem', C-P. Hwang et al., Journal of the Operational Research Society Vol. 50, 1999

[36] 'Future Paths for Integer Programming and Links to Artificial Intelligence', F. Glover, Computers and Operations Research Vol. 13, 1986

[37] 'Metaheuristics', K. Sörensen et al., in 'Encyclopedia of Operations Research and Management Science', Springer, 1996

[38] 'Metaheuristics - from Design to Implementation', E-G. Talbi, John Wiley & Sons, 2009

[39] 'Survey of Metaheuristic Algorithms for Combinatorial Optimization', M. Baghel, International Journal of Computer Applications Vol. 58, 2012

[40] 'Ant System: Optimization by a Colony of Cooperating Agents', M. Dorigo et al., IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics) Vol. 26, 1996

[41] 'Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem', M. Dorigo et al., IEEE Transactions on Evolutionary Computation Vol. 1, 1997

[42] 'A Review on the Ant Colony Optimization Metaheuristic: Basis Models and New Trends', O. Cordón et al., Mathware & Soft Computing Vol. 9, 2002

[43] 'Ant Colony Optimization - Artificial Ants as a Computational Intelligence Technique', M. Dorigo et al., IEEE Computational Intelligence Magazine, 2006

[44] 'Ant Colony Optimization: Overview and Recent Advances', M. Dorigo et al., IRIDIA – Technical Report Series, 2009

[45] 'Genetische Algorithmen und Evolutionäre Strategien', K. Lienemann et al., Universität Bielefeld, 2003

[46] 'Genetic Algorithm Performance with different Selection Strategies in Solving TSP', N. M. Razali et al., International Conference of Computational Intelligence and Intelligent Systems, 2011

[47] 'A Comparison of Selection Schemes used in Evolutionary Algorithms', T. Blickle et al., Evolutionary Computation Vol. 4, 1996

[48] 'An Adaptive Evolutionary Algorithm for Traveling Salesman Problem with Precedence Constraints', J. Sung et al., The Scientific World Journal, 2014

[49] 'Genetic Algorithms', C. R. Reeves, in 'Handbook on Metaheuristics', Springer, 2010

[50] 'Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators', P. Larrañaga et al., Artificial Intelligence Review Vol 13, 1999

[51] 'Genetic Algorithms for the Traveling Salesman Problem', J. Grefenstette et al., Proceedings of the 1st International Conference on Genetic Algorithms, 1985

[52] 'Alleles, Loci and the Traveling Salesman Problem', D. E. Goldberg et al., Proceedings of the 1st International Conference on Genetic Algorithms, 1985

[53] 'A Comparison of Genetic Sequencing Operators', T. Starkeweather et al., 2002

[54] 'A study of Permutation Crossover Operators on the Traveling Salesman Problem', I. M. Olivier et al., Proceedings of the 2nd International Conference on Genetic Algorithms, 1987

[55] 'A Machine Learning Approach for Coreference Resolution', T. Arnold, Master-Thesis TU Darmstadt Department of Computer Science, 2014

[56] 'Reinforcement Learning: An Introduction', R. S. Sutton et al., The MIT Press, 2nd Edition, 2017

[57] 'Neural Combinatorial Optimization with Reinforcement Learning', I. Bello et al., International Conference on Learning Representations, 2017

[58] 'Learning from Delayed Rewards', C. Watkins, Ph.D. Thesis, 1989

[59] 'Technical Note: Q-learning', C. Watkins et al., Machine Learning Vol. 8, 1992

[60] 'Q-Learning for a Simple Board Game', O. Arviddson, 2010

[61] 'Markov Games as a Framework for multi-agent Reinforcement Learning', M. Littman, Proceedings of the 11th International Conference on International Conference on Machine Learning, 1994

[62] 'Transfer Learning for Reinforcement Learning Domains: A Survey', M. Tylor et al., Journal of Machine Learning Research Vol. 10, 2009

[63] 'Reinforcement Learning: A Survey'. L. P. Kaelbling et al., Journal of Artificial Intelligence Research Vol. 4, 1996

[64] 'Deep Reinforcement Learning compared with Q-table Learning applied to Backgammon', P. Finnman et al., Royal Institute of Technology Stockholm, 2016

[65] 'Study of Genetic Algorithm with Reinforcement Learning to solve the TSP', F. Liu et al., Expert Systems with Applications Vol. 36, 2009

[66] 'Application of Machine Learning to Algorithm Selection for TSP', J. Pihera et al., IEEE 26th International Conference on Tools with Artificial Intelligence, 2014

[67] 'The Algorithm Selection Problem', J. Rice, Advances in Computers Vol. 15, 1976

[68] 'Algorithm Selection: From Meta-Learning to Hyper-Heuristics', L. Cruz-Reyes et al., in 'Intelligent Systems', IntechOpen, 2012

[69] 'Selection of Algorithms to solve Traveling Salesman Problems using Metalearning', J. Kanda et al., International Journal of Hybrid Intelligent Systems Vol. 8, 2011

[70] 'Adaptive and multilevel Metaheuristics', M. Sevaux et al., Springer, 2008

[71] 'No Free Lunch Theorems for Optimization', D. H. Wolpert et al., IEEE Transaction on Evolutionary Computation Vol. 1, 1997

[72] 'Hyper-Heuristics: An Emerging Direction in Modern Search Technology', in 'Handbook on Metaheuristics', Springer, 2003

[73] 'A Classification of Hyper-heuristic Approaches', E. K. Burke et al., in 'Handbook on Metaheuristics', Springer, 2010

[74] 'A Comprehensive Analysis of Hyper-Heuristics', E. Özcan et al., Intelligent Data Analysis Vol. 12, 2008

[75] 'Hyper-Heuristics: A Survey of the State of the Art', E. K. Burke et al., Journal of the Operational Research Society Vol. 64, 2013

[76] 'There Is a Free Lunch for Hyper-Heuristics, Genetic Programming and Computer Scientists', R. Poli et al., 12th European Conference on Genetic Programming, 2009

[77] 'Contrasting Meta-Learning and Hyper-Heuristic Research: The Role of Evolutionary Algorithms', G. Pappa et al., Genetic Programming and Evolvable Machines Vol. 15, 2014

[78] 'A Hyperheuristic Approach to Scheduling a Sales Summit', P. Cowling, International Conference on the Practice and Theory of Automated Timetabling, 2000

[79] 'Exploring Hyper-heuristic Methodologies with Genetic Programming', E. K. Burke et al., in 'Computational Intelligence', Springer, 2009

[80] 'Linear Genetic Programming of Parsimonious Metaheuristic', R. E. Keller, IEEE Congress on Evolutionary Computation, 2007

[81] 'Evolving an Edge Selection Formula for Ant Colony Optimization', A. Runka, in 'Genetic and Evolutionary: Computation Conference', 2009