FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Solving The Travelling Thief Problem with an Evolutionary Algorithm

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Christoph Wachter
Matrikelnummer 0525340

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Günther Raidl
Mitwirkung: Univ.-Ass. Dipl.-Ing. Benjamin Biesinger
Univ.-Ass. Dipl.-Ing. Dr.techn. Bin Hu

Wien, 04.08.2015

(Unterschrift Verfasserin) (Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Solving The Travelling Thief Problem with an Evolutionary Algorithm

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Christoph Wachter
Registration Number 0525340

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Günther Raidl
Assistance: Univ.-Ass. Dipl.-Ing. Benjamin Biesinger
Univ.-Ass. Dipl.-Ing. Dr.techn. Bin Hu

Vienna, 04.08.2015

(Signature of Author)

(Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Christoph Wachter
Lange Gasse 26/), 1080 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasserin)

Acknowledgements

I would like to express my sincere thanks to all people who made the accomplishment of this master thesis: my advisers Günther Raidl, Benjamin Biesinger and Bin Hu at the *Algorithms and Complexity Group* department. Their continues support, advises and expertise made to write this master thesis, to implement its presented algorithms and to do the connected research much easier.

Abstract

Optimization problems in real world scenarios often consist of different, interconnected NP-hard optimization problems. In this thesis the Travelling Thief Problem (TTP) is considered. It combines the well known knapsack problem (KP) and the travelling salesman problem (TSP). In the TTP a person called “thief“ tries to collect as many items, on different nodes he is visiting during his tour, respecting the capacity of his knapsack, as possible. These items have associated a weight and a profit. Collecting an item increases the profit but also decreases the thief’s travel speed on his remaining tour. As he has to pay a rent for his knapsack per time unit he tries to minimize his travel time for his tour and to maximize his profit by collecting items. A possible application for the TTP could be a delivery company which has to pick up packages at different locations which are combined in a route. Picking up packages leads to a profit gain but decreases travel speed and the delivery has a maximum cargo capacity.

This thesis will provide a new state of the art hybrid genetic algorithm to solve the TTP heuristically. This algorithm consists of a genetic algorithm and additional local search operators. Most of these operators will focus on either the embedded TSP or the KP subproblem. For each partial candidate solution the other part is derived from this partial solution. Additionally, an approach where each subproblem will be solved on its own and combined in a solution for the TTP will be provided to examine the importance of the interconnection between the two subproblems. All algorithmic variants are compared to each other and to so far existing solution approaches from the literature. Computational results show that our hybrid genetic algorithm is competitive to these approaches and even outperforms them especially on smaller instances.

Kurzfassung

Optimierungsprobleme in wirklichen Einsatzszenarien bestehen oftmals aus verschiedenen, miteinander verknüpften und sich gegenseitig beeinflussenden NP-schweren Optimierungsproblemen. Daher wird in dieser Diplomarbeit das Travelling Thief Problem (TTP) näher betrachtet. Das TTP besteht aus dem bereits sehr gut erforschten Knapsack Problem (KP) und dem Travelling Salesman Problem (TSP). Beim TTP versucht ein Dieb, auf seiner Tour, so viele Gegenstände wie möglich, einzusammeln, wobei er die Maximalkapazität seines Rucksacks nicht überschreiten darf. Alle Gegenstände sind sowohl mit einem Gewicht als auch einem Profit verknüpft. Durch das Mitnehmen eines Gegenstandes erhöht der Dieb seinen Profit, aber er reduziert auch seine Reisegeschwindigkeit auf seiner restlichen Tour. Da er für seinen Rucksack eine Miete pro Zeiteinheit bezahlen muss versucht er seine Reisezeiten zu minimieren und den Profit durch eingesammelte Gegenstände zu maximieren. Ein mögliches Anwendungsszenario wäre zum Beispiel eine Lieferfirma, welche Packete an bestimmten Orten abholt welche in einer Route abgefahren werden. Das Abholen der Packete erhöht den Profit, erhöht aber die Fahrtdauer der Route und zusätzlich hat der Lieferwagen nur ein begrenztes Fassungsvermögen.

Diese Diplomarbeit wird einen neuen, state-of-the-art hybriden genetischen Algorithmus vorstellen um das TTP heuristisch zu lösen. Der Algorithmus besteht aus einem genetischen Algorithmus und zusätzlichen lokalen Suche Operatoren. Die meisten dieser Operatoren versuchen entweder das TSP oder das KP alleine zu lösen. Die Lösung für das andere Subproblem wird dann von der bereits existierenden Lösung des anderen Subproblems abgeleitet. Zusätzlich wird ein Lösungsversuch vorgestellt, bei dem jedes Subproblem einfach für sich gelöst wird und die beiden Lösungen in einer Gesamtlösung für das TTP zusammengefügt. Dies dient dazu um die Wichtigkeit der Betrachtung der Koppelung der Subprobleme zu untersuchen. Alle Varianten des Algorithmus werden miteinander und so weit existierenden Lösungsversuchen aus der Literatur verglichen. Die Ergebnisse zeigen, dass der eingeführte hybride genetische Algorithmus vergleichbare und auf kleinen Instanzen sogar bessere Resultate erzielt.

Contents

CHAPTER 1

Introduction

In the travelling thief problem (TTP) there is a person called “thief“ who wants to make a round trip through a set of different cities. In each of these cities there are different items which have a weight and a profit. These items can be collected in the knapsack which has a maximum capacity. The goal of the thief is to maximize his profits by collecting the most valuable items without exceeding the knapsacks capacity. But by collecting items along the journey the weight of his knapsack increases and therefore decreases his velocity resulting in a longer total travel time. Additionally the thief has to pay a rent for his knapsack which is proportional to the time needed for the tour. The objective of the thief is to find a tour through all cities and a packing plan which maximizes his profit.

The TTP, which has been introduced by Bonyadi et al. [10], is a relatively new optimization problem as it is a combination of two different NP-hard combinatorial optimization problems which combines two well known optimization problems, the knapsack problem (KP) [22] and the travelling salesman problem (TSP) [24]. The novelty of this optimization problem is that it tries to model the characteristics of real-world problems better by interconnecting different problems with each other. This increases the complexity as the solution of one problem is influencing the other and vice versa.

1.1 Real World Scenario

A possible application for the TTP could be a delivery company which has to pick up packages at different locations which are combined in a route. For each collected package a fee is earned. But each delivery van has only a certain weight capacity for cargo. Additionally, route costs are increased according to the weight of a package as the delivery van consumes more fuel and collecting a package decelerates the whole journey as the delivery man has to pick up the package at the clients location. Therefore costs are increased through increase of fuel consumption, work time and other running costs of the delivery van.

1.2 Outline

The aim of this master thesis is to provide a new state-of-the-art algorithm for this optimization problem by using a hybrid genetic algorithm. In this context this means that the algorithm consists of a genetic algorithm and additional local search operators.

We will first have a look at the definition of the TTP and the related subproblems the TSP and the KP in chapter 1. After that we discuss already existing approaches to solve the TSP and the KP in chapter 2. As I want to implement different variants of evolutionary algorithms to solve the TTP we will also have a look at the benefits and disadvantages of genetic algorithms and local search (chapter 3) and suitable heuristics and operators for the algorithms (chapter 4).

The algorithms will be implemented in java and an evaluation and a comparison with existing algorithms for known problem instances of the TTP will be provided as we will see in chapter 5.

1.3 Formal Problem Description

We take over the problem definition for the *Travelling Thief Problem* (TTP) from Bonyadi et al. [10] who introduced this problem. The TTP is a combination of a *Knapsack Problem* (KP) and a *Travelling Salesman Problem* (TSP). Additionally these two subproblems influence each other. For packing different items into our knapsack the velocity of the thief decreases according to the collected weight of the different items. The more items we pack the more the velocity decrease and therefore travel time increases. For each consumed time unit a rent for the knapsack has to be paid and therefore the total profit decreases. Before we are going to exactly define the TTP we take a look at the KP and the TSP solitarily.

Travelling Salesman Problem (TSP)

In the TSP there are n cities and there is a travelling salesman who is creating a tour where all cities have to be visited once. The task is here to create an optimal tour where the travel time is minimized. Now we have a look at the formal description of the different elements of the TSP definition. Here is to mention that velocity and travel time are not part of the original definition. These have already been taken from the TTP definition [10]. Below we summarize all the details which are needed to describe an instance of the TSP:

city set $X = \{1, 2, ..., n\}$

distance matrix $D = d_{ij}; \forall i, j \in X$

distance d_{ij} weight of the edge connecting city $i \in X$ and city $j \in X$

velocity $v_i \in \mathbb{R}$ constant velocity of the salesman after leaving node $x_i \in X$

tour $\overline{x} = (x_1, x_2, ..., x_n); x_i \in X$ with $i = 1, 2, ..., n$ containing all cities in the order in which they shall be visited

travel time the travel time between city x_i and $x_{(i+1)\, mod\, n}; \forall i = 1, 2, ..., n$ is calculated through $t_{x_i, x_{(i+1)\, mod\, n}} = \frac{d_{x_i, x_{(i+1)\, mod\, n}}}{v_c}$

objective function $min f(\overline{x}) = \sum_{i=1}^{n}(t_{x_i, x_{(i+1)\, mod\, n}})$. The result of function f is the total travel time for the tour $\overline{x}$ which shall be minimized.

Knapsack Problem (KP)

In the KP there is a knapsack with a given maximum capacity c. Additionally there are m items $I_1, ..., I_m$ which are all connected with a weight and a profit. These items get packed into the knapsack and reduce the remaining capacity of the knapsack and increase the gained profit. As the capacity of the knapsack is limited the target is to maximize the profit by choosing an optimal combination of the different items which do not exceed the capacity c of the knapsack. Now we have a look at the formal description of the different elements of the KP definition. Again we can see below all needed details to describe the KP: [22]

items $I_1, ..., I_m$

profit $p_i \in \mathbb{R}$ profit of item $I_i; \forall i = 1, ..., m$

weight $w_i \in \mathbb{R}$ weight of item $I_i; \forall i = 1, ..., m$

knapsack capacity c

item set $\overline{z}$ which defines which items shall be packed. $\overline{z} = (z_1, z_2, ..., z_m); z_i \in \{0; 1\}$; i.e., $z_i = 1$ item i is packed and $z_i = 0$ item i is not packed

objective function $max\ g(\overline{z}) = \sum_{i=1}^{m} p_i z_i$

weight constraint $\sum_{i=1}^{m} w_i z_i \leq c$

Interconnection

These two subproblems now get interconnected in a way that the solution of each subproblem is influencing the solution of the other subproblem. We do that as described in TTP Model 1 [10]. The more items the thief packs the more his velocity decreases. Additionally the thief must pay a rent for his knapsack per time unit. So we need following information to describe the interconnection of these problems:

current knapsack weight W_i is the weight of the knapsack at a certain node $i \in X$ within the tour after the items have been packed of this node.

travel speed $v_i \in [v_{min}; v_{max}]$ current travel speed of the thief at a node $i \in X$ which is dependent on the load of the knapsack within the tour.

travel speed calculation $v_i = v_{max} - W_i \frac{v_{max} - v_{min}}{c}$

knapsack rent R per time unit

Travelling Thief Problem (TTP)

Now we have a look at the whole definition of the TTP [10]. There is a thief who is creating a tour where all cities have to be visited once. The thief rents a knapsack with a constant renting rate R per time unit which has a a given maximum capacity c. By packing items along the tour and increasing the weight of the knapsack the velocity of the thief decreases down to a minimum travel speed v_{min} if the knapsack capacity is completely used. Therefore the travel time for the tour increases. As the capacity of the knapsack is limited and a rent has to be paid for the knapsack the target is to maximize the profit by choosing an optimal combination of the different items which do not exceed the capacity c of the knapsack and lead to a travel time as short as possible. So the TTP is defined as we can see below:

tour $\overline{x} = (x_1, x_2, ..., x_n)$ is a tour through all cities $x_i \in X$

packing plan $\overline{z} = (z_1, z_2, ..., z_m); z_i \in \{0; 1\}$

objective function $h = max \sum_{i=1}^{m} p_i z_i - R \sum_{i=1}^{n} t_{x_i, x_{(i+1)\, mod\, n}}$

weight constraint $\sum_{i=1}^{m} w_i z_i \leq c$

CHAPTER 2

Related Work

Now we will have a look at already existing work dealing with the TSP, the KP and the TTP.

2.1 Travelling Salesman Problem (TSP)

The TSP is one of the best known combinatorial optimization problems and dozens of algorithms to solve it have been presented since it was introduced by the RAND Corporation in 1948 [26].

To measure the quality of the provided solutions of the particular algorithm there are several ways. To compare with already optimal solved instances or to compare with the Held-karp (HK) lower bound [20]. The HK lower bound is the solution of the linear programming relaxation of a standard integer programming formulation and can be computed exactly for instances with up to 30000 nodes. For bigger instances the HK lower bound can be approximated by using iterative Lagrangean relaxation techniques, i.e., the HK lower bound is approximated not the the optimal tour length. [20]

To compare TSP solution with already existing solution the TSPLIB [5] has been created. The TSPLIB contains problem instances with a size from 51 to 85900 cities. Most of them have been optimally solved. The TSPLIB contains the optimal tour length or the lower and upper bounds of the optimal tour length for each instance. [5] The mentioned HK lower bound is often capable of approximating the optimal tour lengths with a deviation less than 2% [20].

The best performing heuristic algorithms basically all use a kind of local search [21]. Which algorithm fits best depends upon your instance size and available computing resources. If instance sizes are very big and computing resources very limited a compromise would be using a simple tour construction heuristic, e.g., the nearest neighbor heuristic which has a worst-case runtime of $O(n^2)$ with an excess of 25% of the HK lower bound in most cases [21]. If a high quality solution within a moderate runtime is needed a good choice would be an effective implementation of the 2-Opt/3-Opt local search or Lin Kernighan Algorithm [21]. There are also plenty of different meta heuristics used to solve the TSP described in the literature like simulated annealing, tabu search or evolutionary algorithms [21].

Instance size	10^2	$10^{2.5}$	10^3	$10^{3.5}$	10^4	$10^{4.5}$	10^5	$10^{5.5}$	10^6
2-Opt	4.5	4.8	4.9	4.9	5.0	4.8	4.9	4.8	4.9
3-Opt	2.5	2.5	3.1	3.0	3.0	2.9	3.0	2.9	3.0
LK	1.5	1.7	2.0	1.9	2.0	1.9	2.0	1.9	2.0

Table 2.1: 2-Opt/3-Opt/LK Comparison: Average HK lower bound excess on Random Euclidean Instances

Instance size	10^2	$10^{2.5}$	10^3	$10^{3.5}$	10^4	$10^{4.5}$	10^5	$10^{5.5}$	10^6
2-Opt (Greedy Start)	< 0.01	0.01	0.03	0.09	0.4	1.5	6	23	87
2-Opt	0.03	0.09	0.34	1.17	3.8	14.5	59	240	940
3-Opt (Greedy Start)	0.01	0.03	0.09	0.32	1.2	4.5	16	61	230
3-Opt	0.04	0.11	0.41	1.40	4.7	17.5	69	280	1080
LK	0.06	0.20	0.77	2.46	9.8	39.4	151	646	2650

Table 2.2: 2-Opt/3-Opt/LK Comparison: Running Time in Seconds on a 150 Mhz SGI Challenge

In table 2.1 and 2.2 we can see a comparison of running times and solution quality of the three algorithms recommended by [21] to use to gain a high quality solution of the TSP in a moderate runtime. The run times are based on the results of [21]. What we can see is that on growing instance size the differences between the run times become bigger. Additionally using a greedy heuristic to create a start solution before using the 2-Opt or 3-Opt operator significantly reduce the running times.

2.2 Knapsack Problem (KP)

The KP is like the TSP a well known optimization problem and there are again many different approaches to solve it. In our case we are going to look at the 1-constraint 0-1 KP as we do not have any other constraint than the knapsack capacity. This particular KP is not strongly NP-hard [13] and can be solved in pseudo-polynomial time with dynamic programming [29] in $O(nc)$, where n is the number of items and c is the capacity of the knapsack [28]. It has been shown that beside knapsack capacity and the number of items also a correlation between weights and profits can lead to higher computation times and therefore to a more complex instance. [1,27]

A very good choice for solving the KP exactly is dynamic programming which is even for more complex KP variants and difficult instances one of the best approaches [29].

Additionally there are also effective metaheuristics algorithms which are capable of providing very good to near optimal solutions [13, 13, 15]. This leads to the situation that nearly all standard instances from the literature can be solved [29].

2.3 Travelling Thief Problem (TTP)

As the TTP is a relatively new problem introduced by Bonyadi et al. [10] the list of solution methods for the TTP is relatively short at this moment. But there exist already several solution approaches and a benchmark set [1]. This benchmark set uses instances from the TSPLIB [5] for the tour and combines it with the benchmark of Martello et al. [1] for the KP . In particular they used the code from [6] to create instances for the KP part [1]

The authors of the TTP benchmark set also already presented different approaches to solve this optimization problem. In [1] the authors use a simple (1+1) evolutionary algorithm (EA), a random local search (RLS) and a deterministic constructive heuristic (SH) to solve the TTP problem.

All three approaches use the Chained Lin-Kernighan heuristic to create a fixed tour for the TTP solution. After that only the KP part is considered any more. The SH creates a scoring based on the profits of the items minus the increased renting costs which arise if they get packed. If the same item is at the start of the tour it gets less valuable this way. After that the items get sorted according to their score. Now the SH picks items from the sorted list as long as the capacity of the knapsack is not exceeded. The RLS tries to improve the solution by randomly inverting the packing status of an item. If a better solution has been found the old solution is replaced with that. The EA randomly inverts the packing status of all items with a certain probability. If a better solution has been found the old solution is replaced with that. [1]

CHAPTER 3

Algorithm Overview

First we will have a short look on local search and genetic algorithms. After that I will present a basic overview of the design of the implemented algorithm.

3.1 Local Search

Local search was introduced in the late 1950s and used to solve many different optimization problems. For practically all well studied optimization problems like the TSP or the KP there are attempts to use local search to solve them described in the literature [8].

A local search routine needs a starting solution which gets replaced by better solutions found in the search space. The possible neighbor solutions of which the search space consists are defined by the neighborhood relation. This relation describes how a neighbor can be derived from an existing solution [8]. These solutions get evaluated by an objective function to determine if an improvement has been found. If a better solution has been found the old solution gets replaced by the new one. This iterates until no further improvement can be found or another specified abortion criteria is fulfilled. A disadvantage is that local search algorithms often gets stuck in a local optima [16].

To overcome this problem many metaheuristics which make use of local search use different neighborhood structures, i.e., there are different neighborhood relations. Using different neighborhood structures leads to a larger search space and therefore to a potentially better local optimum. So the effectiveness of local search can be increased to a certain limit [16].

3.2 Genetic Algorithms

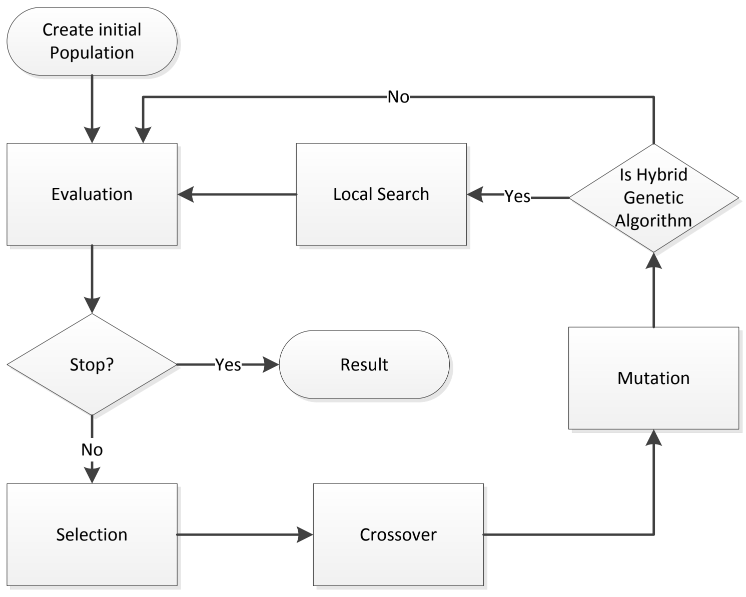

Figure 3.1: Hybrid algorithm architecture

Genetic algorithms were first introduced by Holland [18]. Basically genetic or evolutionary algorithms are „intelligent“ stochastic search algorithms which can be used for all different kinds of optimization problems.

Holland created an algorithm which tries to mimic the principles of biological evolution. So there is a population consisting of individuals which represent the single solutions of the optimization problem. These solutions are encoded into the chromosome which consists of single genes, e.g., the KP chromosome would be the packing plan which is represented by a bitset and each bit which determines if an item gets picked or not would be a single gene. The specification of such a gene, e.g., a gene for the KP is set to 1 is called allele. So a very important part of the design of a genetic algorithm is to define how a chromosome is represented and stored.

In figure 3.1 you see the composition of the genetic and the hybrid genetic algorithm. In this section we will only discuss genetic algorithms. So first we start with the creation of an initial population. After that they get evaluated by using a fitness function. This fitness function calculates an objective value for each individual which makes them comparable. As the fitness function is called very often (each iteration for each individual) it can become very time consuming. So if fitness calculations are very expensive or a precise computing model for objective value calculation does not even exist often also an approximation for the objective value is done.

Based on these calculated objective values some individuals get eliminated, based on their fitness, which is done by the selection operator. A well designed fitness function and selection operator are crucial for the quality of the whole algorithm. The parameters for the selection operator have to be well choosen. A very high selection pressure for example makes the population more uniform and reduces the search space as offsprings of a more uniform population have a lower diversity [31]. Also the population size can have an impact on the solution quality but also on the execution time.

After the selection phase, as we can see in figure 3.1, the crossover is done. This means that the chromosome of two parents is combined to create new offspring. The target is to improve the objective values of the individuals of the population as well performing alleles propagate to the next generation by choosing the most fit individuals which reproduce. Here the challenge lies in the choice how to integrate the newly created offspring into the population: [13]

generational approach replacement of the whole generation

steady-state approach replacement of less fit individuals

Additionally, evolutionary algorithms often come with repair and penalty functions. Depending on the problem it is often the case that crossover operators can create infeasible offsprings, e.g., a choice of items for the KP which exceeds the knapsack capacity. To handle such situations often repair functions are used. A repair function is an algorithm which is capable to derive a feasible offspring from an infeasible offspring. For example taking the KP solution which exceeds the knapsack capacity, a repair algorithm could be a greedy heuristic which unpacks items as long as the capacity is exceeded. Another possibility would be a penalty function. Penalty functions are often used in evolutionary algorithms to penalize infeasible offspring with a very high bias compared to the average objective value of a solution. So the chance is extremely high that these offspring are eliminated in the selection phase.

To further increase diversity and to avoid getting stuck in local optima a mutation operator is used after the crossover phase. This means that some genes get randomly changed to avoid that the individuals in the population become too similar.

To achieve high quality results it is very importation to tune the parameters of the genetic algorithm according to the nature of the problem which include:

- selection method
- population size
- selection pressure
- crossover & mutation rate

3.3 Hybrid Genetic Algorithms

The difference between a genetic algorithm and our hybrid genetic algorithm is that the hybrid genetic algorithm as we can see in figure 3.1 executes a local search on individuals of the population. Hybrid genetic algorithm often perform much better than traditional genetic algorithms as they, use local search for intensification and the traditional genetic algorithm part for diversification [19].

3.4 Algorithm Design

I will implement implement different variants of a hybrid genetic algorithm to solve the TTP. These different variants will either use a tour crossover operator or a packing plan crossover operator to create new offspring. The other part of the solution will then be derived of the already exisiting solution part. As already mentioned in the outline 1.2 an additional approach where each subproblem will be solved on its own and combined in a solution for the TTP instance will be provided to examine the importance of the consideration of interconnection between the two subproblems. In section 4.4 we will discuss all different algorithm variants. The main part of this thesis will be the developement of different operators used by the different algorithm variants to solve the TTP. These operators that are being implemented are based on the literature and have been shown to perform well on the TSP or the KP problem respectively and will also be discussed in chapter 4.

CHAPTER 4

Problem Solution

Now we are going to have a look at the solution representation first. Afterwards we are discussing the problem specific knowledge which could be used to improve our algorithms. Then we discuss the different operators which will be implemented.

4.1 Solution Representation

Before any algorithm or operator can be implemented the representation of the solution has to be determined. The choice of the representation is affected by several factors, e.g., the number of available operators for this representation and how well they perform.

As the TTP is the interconnection of two different subproblems also the solution will consist of two different parts. One part is responsible for storing the tour and the other to store the packing plan of the knapsack.

Tour representation

To store the tour of the TSP part there are many different representation types described in the literature. The most popular would be the path representation. In this representation type the tour is represented as a node list in the order of their appearance in the tour. The advantage of this representation is that most TSP crossover operators are for the path representation and they perform better than the operators for other representation types from the literature. The disadvantage is that standard crossover operators like 1-point,2-point or uniform crossover can not be used on this representation type as the the node list would have to be stored as a bit string like in the binary representation. [12]

I choose the path representation for the tour part as it is the most common and natural way to represent a tour. There are plenty of crossover operators and those operators using the path representation perform quite well compared with crossover operators which use other representation types [12]. Additionally it is also suited for the 2Opt operator which will be presented later in section 4.4.

Packing plan representation (KP)

For the representation of the packing plan there are also different representation types. The most common representation type is the binary representation. There is a binary set which contains a bit for each item which is set to 1 if the item is packed and set to 0 otherwise. It is very common for evolutionary algorithms to use the binary representation. The usage of binary encoding allows to make use of standard crossover operators which we also want to to use. Additionally crossover and mutation operator are very easy to implement for this representation type. A disadvantage is that there is the possibility to create illegal offspring where the knapsack capacity is exceeded as this representation type only stores the information about what is packed but not how much space is left in the knapsack. So a repair algorithm is needed if this representation is used within a genetic algorithm. [11]

For the storage of the packing plan I choose the binary representation. It allows the use of standard crossover operators and crossover and mutation operators are very easy to implement. According to the literature the other representation types are rather used and suited for the the multi dimensional KP (MKP) [11]. As we do not face a MKP there is no benefit of using a more complex representation. Therefore the binary representation is picked and will be combined with a repair and a penalty function.

Solution feasibility

As both sub representations have constraints which can be violated it is important that no infeasible solutions are created.

Tour

A valid tour must contain every node of the graph once, i.e., a Hamiltonian cycle. As we use a path representation and all presented operators do only shift nodes, swap two nodes or create a whole tour it is guaranteed that only new permutations are created which still will be Hamiltonian cycles.

Packing Plan

The usage of crossover operators like 1-point, 2-point or uniform crossover operators leads to the situation that infeasible packing plans could be created. In the literature there are basically two approaches described to deal with infeasible packing plans: [13]

penalty function The fitness function uses a penalty function to penalize infeasible solutions to reduce the chances of survival of infeasible solutions.

repair operator A repair algorithm is used to transform infeasible solutions to feasible solutions again. This approach is often done with greedy heuristics.

I will make use of a repair operator, which is introduced in section 4.4, in my algorithm to deal with infeasible packing plans.

4.2 Problem specific Considerations

Now we discuss some problem specific details which could help us to provide better operators and therefore better solutions for the TTP.

Penalty Function Packing Plan

In figure 4.1 we can see that within a given tour $\overline{x}$ item i gets packed at node x. So we want define how to calculate a corresponding bias for the profit of this item when we are going to determine the packing plan. The idea is to adjust profits according to the renting cost increase of the knapsack through the travel speed increase through the weight gain by packing a certain item. So instead of the total profit we consider the net profit of items.

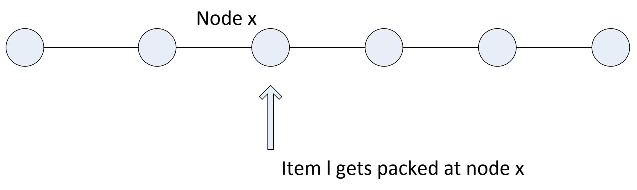

Figure 4.1: Packing item $l \in I$ at node $x \in \overline{x}$

We remember the costs for travelling the distance d_{ij} between two nodes i and j get calculated via:

$$\text{travel costs for distance } d_{ij} = \frac{d_{ij} * R}{v_i}$$

d_{ij} distance between node i and node j ($i, j \in \overline{x}$)
R renting ratio for the knapsack
v_i velocity of thief after packing items at node i

If item l with the weight w_l gets packed the current velocity v_i is decreased by Δv according to following formula:

$$\Delta v = \frac{w_l}{c} * (v_{max} - v_{min})$$

c maximum knapsack capacity
v_{max} maximum travel speed
v_{min} minimum travel speed

This leads to the situation that the thief travels now with a decreased velocity $v_i' = v_i - \Delta v$ for the distance d_{ij} between node i and node j if item l gets packed:

$$\text{travel costs for distance } d_{ij} \text{ if item } l \text{ gets packed} = \frac{d_{ij} * R}{v_i'}$$

So this velocity decrease results in an increase of the travel time and therefore in an increase of the costs for travelling from node i to node j as a rent for the knapsack has to be paid per time unit:

$$\text{travel cost increase} = \frac{d_{ij} * R}{v_i'} - \frac{d_{ij} * r}{v_i}$$

The cost increase has to be calculated this way as we remember from the problem description the travel time t_{ij} between the nodes i and j is calculated through the formula $t_{ij} = \frac{d_{ij}}{v_i}$.

Let's have a look at an example. No item has been picked until now so $v_i = v_{max}$. Now item l is picked at node i:

- distance of the rest tour $d = 10$
- $v_o = v_{max} = 1$
- $v_{min} = 0.1$

- renting rate $R = 1$
- weight of item w_l = 2
- profit of item p_l = 5
- capacity of knapsack c = 10

According to this configuration packing item l results in an cost increase of 2.19 because:

$$\Delta v = \frac{w_l}{c} * (v_{max} - v_{min}) = \frac{2}{10} * (1 - 0.1) = 0.18$$
$$\text{travel costs increase} = \frac{d_{ij} * R}{v_i'} - \frac{d_{ij} * R}{v_i} = \frac{d_{ij} * R}{v_i - \Delta v} - \frac{d_{ij} * R}{v_i} = \frac{10 * 1}{1 - 0.18} - \frac{10 * 1}{1} = 2.19$$

So the penalized profit for item l would be $p_l = 5 - 2.19 = 2.81$.

In listing 4.1 we can see the pseudo code for the penalty function which we will use in our algorithms:

Listing 4.1: Penalty function pseudo code

```
INPUT :  item set $I$, tour $\overline{x}$
OUTPUT: item set with adapted profits $I'$

$I' = \emptyset$

traverse $\forall l \in I$ {
  $\Delta v = \frac{w_l}{c} * (v_{max} - v_{min})$

  // calculate profit loss $o$ according node position of associated node $i$
      in $\overline{x}$
  $o = \frac{d_{resttour} * R}{v_i - \Delta v} - \frac{d_{resttour} * R}{v_i}$

  $I' = I' \cup l$ with $p_l = p_l - o$
}

return $I'$
```

Non linearity

We have now seen that we are able to calculate the profit decrease if we pack items. But during the tour the knapsack weight will alterate. Therefore we have to check if the velocity loss through packing items is a linear, i.e., do we face the same velocity loss by packing l at a certain node but with different knapsack weights through already packed items. We remember the calculation of the travel costs for a distance d_{ij} between the nodes i and j:

travel costs for distance $d_{ij} = \frac{d_{ij} * R}{v_i}$
$v_i = v_{max} - (v_{max} - v_{min}) * \frac{w}{c}$
w is the weight of the knapsack

This leads to following formula for the calculation of the travel costs for the distance d_{ij} between node i and j:

$$\text{travel costs for distance } d_{ij} = \frac{d_{ij} * R}{v_{max} - (v_{max} - v_{min}) * \frac{w}{c}} = \frac{d_{ij} * R * c}{v_{max} * c - (v_{max} - v_{min}) * w}$$

As we can see in the last transformation of the renting rate formula the variable of the current weight of the knapsack wi is a part of the denominator which means the profit alterations are not linear. This means calculating the profit alteration for a single item depends on the already packed items. So we use this penalty function to approximate the net profit of this item.

Fitness Function

To determine if an individual performs better or worse than the others in the population a fitness function is needed. This fitness function calculates a scalar value to a given solution. We will use our objective function as fitness function. It is to mention here that if an item is collected on the first node of a tour it does not get packed until the end of the tour as the first node is also the last node.

Calculating the objective value this way leads to the situation that a negative profit can occur if the tour rent is higher than the maintained profits. This leads to the question how comparable different objective values are. Therefore the objective values will be normalized to ease the comparison of objective values of different solutions. We will calculate them by setting all objective values into relation with the best objective value of the population.

Solution set: S
objective values: $ov_s; s \in S$
best individual in population: s_b
worst individual in population: s_w
normalized objective value for $s \in S = \frac{ov_s + abs(ov_{s_w})}{ov_{s_b} + abs(ov_{s_w})}$

This leads to the situation where we have to compare negative with positive objective values therefore every objective value is increased with a bias. This bias is the absolute value of the worst performing individual. Therefore it is guaranteed that every normalized objective values is a value within $[0; 1], ov_{s_b} = 1.0$. These normalized objective values where the best individual has a score of 1.0 allow to compare the individuals in relation to the best individual and the whole population which is necessary for the selection operator. Let's summarize this by having a look some examples:

- objective values are $\{6, 1, -3, -4\}$
- a objective value of 6 leads to normalized objective value of $\frac{6+abs(-4)}{6+abs(-4)} = 1.0$
- a objective value of 1 leads to normalized objective value of $\frac{1+abs(-4)}{6+abs(-4)} = 0.5$
- a objective value of -3 leads to normalized objective value of $\frac{-3+abs(-4)}{6+abs(-4)} = 0.1$
- a objective value of -4 leads to normalized objective value of $\frac{-4+abs(-4)}{6+abs(-4)} = 0$

4.3 Greedy strategies

The presented hybrid genetic algorithms will be using different greedy operators. Therefore we will have a look at some greedy strategies from the literature for specific subproblems.

Packing Plan (KP)

Traditional greedy algorithms basically use three different strategies for the 0-1 KP: [32]

Value strategy The items get sorted according their value beginning with the most valuable item. Then the knapsack gets packed with items from this list as long as the capacity of the knapsack is not exceeded.

Capacity strategy Using this strategy the available items get sorted according to their weight beginning with the lightest weight. Then the knapsack gets packed with items from this list as long as the capacity of the knapsack is not exceeded.

Value per Unit strategy Same procedure as above but this time the item list gets sorted according their value to weight ratio beginning with the item with the highest value compared to its weight.

TTP

For the TTP there are not many documented strategies existing in the literature as this problem is a quite new problem. Therefore we will introduce our own greedy strategy. For the TTP we can make following observations:

- If we disregard the fact of different travel costs, through packing items at different positions, we see that we will always receive the same profit whether the item gets packed at the beginning or at the end of the tour. Travel costs will change but the collected profit is the same.

- Packing an item decreases our velocity and therefore rises travel costs.
- There are three possibilities to increase profits. Minimize the tour length, move nodes where very heavy items get collected to the end of the tour or pack a different set of items.

As it is very difficult to concentrate on all the ways to increase profits we neglect to do so and we concentrate only to minimize tour costs or to pack as best as we can. One way to do so could be to create a tour or a packing plan and derive the solution of the other part from that.

Derive packing plan from tour

For a given tour we use the already in section 4.2 introduced penalty function to adapt the profits of all items according to the increased travel costs by collecting them. After that we use these adapted profits to create a packing plan as profitable as possible. A possible greedy strategy which is also used by the greedy pakcing operator, which be introduced later in section 4.4, is now to sort the items descending according to the profit per weight ratio and to pack the items from that sorted list as long as the knapsack does not exceed.

Derive tour from packing plan

We have a packing plan and now we want to derive a tour from that packing plan. One way to reduce travel costs would be to pack the collected items as late as possible as we could shift the travel speed decrease at a later moment and therefore reduce travel costs. We split up all nodes in a node set where we do not collect any items at all and a node set where we collect all the items which get packed. Now we sort the set with nodes where items get collected ascending according to the weight which gets collected at these nodes.

Now first we visit all nodes where we do not collect any items and after that we visit all nodes according to our sorted list beginning with the node where we pack the most lightweight items. The first part of the tour is optimized by minimizing the tour length as we do not collect any items there and we can solely concentrate on minimizing the distances to increase the objective value.

After that we start to traverse the remaining nodes which are sorted ascendingly by the weight of the items which are collected at these nodes. We can see the principle in figure 4.2. We have a tour containing 15 nodes and at 5 nodes we collect items. So we split up the tour into two sections one where we traverse all nodes where no items get collected and a section where we traverse the nodes where items do get collected.

In the first section we minimize the tour length, e.g., with a 2-Opt local search. Therefore we can see in figure 4.2 that the tour length in the first section increases much slower than in the other section although we visit much more nodes as we try to find a minimum tour. Now after node 10 we start collecting items. With every traversed node we collect more and more weight therefore the knapsack capacity starts to exceed faster and faster with every traversed node. The total tour length also increases much faster as we do not try to minimize the tour length now. When we traverse the nodes of the second section we solely try to collect weight as late as possible.

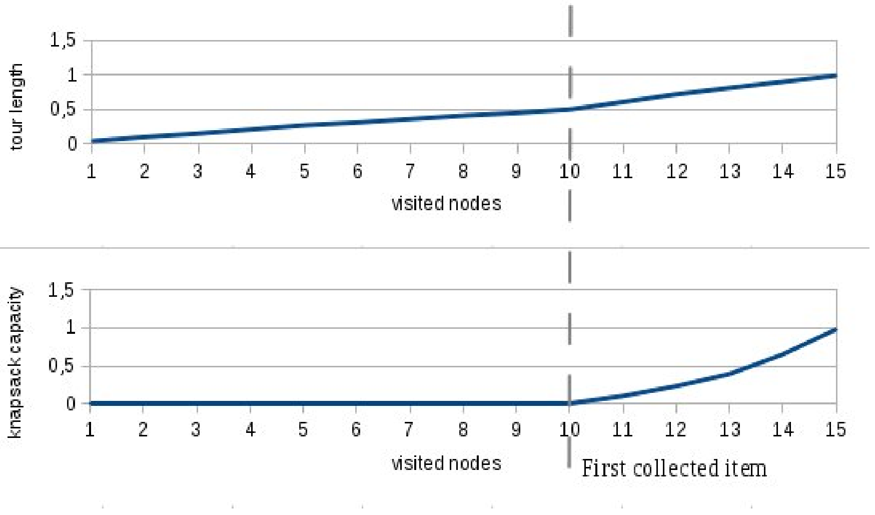

Figure 4.2: Greedy strategy for the TTP

4.4 Operators

Now we are going to discuss all operators which will be used to solve the TTP.

Nearest Neighbor Heuristic

The Nearest Neighbor Heuristic (NN) is a simple tour construction heuristic which starts at a randomly chosen node which is the first node in the resulting tour. Now the distances to all not so far visited nodes are considered and the node with the cheapest distance is picked and gets appended to the tour. Now the same procedure is continued with the new node. The algorithm is finished when a tour containing all nodes has been created. In listing 4.2 you can see the pseudo code for the NN.

Listing 4.2: Nearest Neighbor Heuristic pseudo code

```
INPUT : Distance matrix $D$, Nodeset $X$, Startnode $s$
OUTPUT: resulting Tour $\overline{x}$
```

```
node $x = s$
node $y$

loop unitl $X = \emptyset$ {
  $\overline{x}$.add( $x$ )
  $X$.remove( $x$ )

  pick $\exists y \in X$ where distance  $d_{xy}$ is minimal
  $x = y$
}

return $\overline{x}$
```

The advantage of this operator is that it has a very low execution time with a worst case runtime of $O(n^2)$. Tests [21] have shown that the NN creates tours with an average objective value approximately 25% above the Held-Karp lower bound. The NN is a very good choice for creating start tours which get improved by more advanced algorithms as execution time can be saved as the more advanced algorithm with higher execution times can begin from an better starting point.

2-Opt

The 2-Opt local search is a tour improvement heuristic which expects a valid tour which gets adapted by deleting two edges and recombining the two resulting paths to a tour again by connecting them with 2 other edges. In figure 4.3 you can see the principle of this tour adaption. Additionally it is to say that only non-adjacent edges can be chosen for replacement. [21]

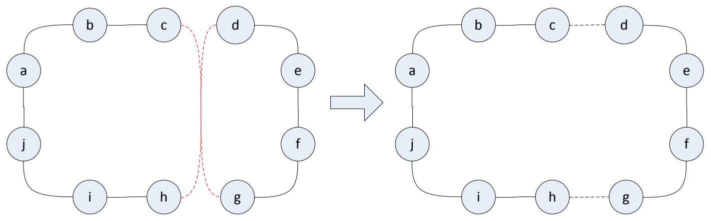

Figure 4.3: A 2-opt edge replacement: start tour left and result right

On implementing the 2-Opt heuristic we have to make the decision if we implement:

2-Opt best fit All possible edge replacement get evaluated and the best edge replacement causing the best tour improvement is done.

2-Opt first fit This variant works like the best fit variant with a nested loop except that after the first possible improvement has been found the edge replacement is done immediately.

The algorithm loops until no further improvement can be found or the time limit has reached. If the 2-Opt heuristic is used on very big instances it can become very time consuming. Therefore a soft deadline has been introduced. If this time limit has been reached no further 2-Opt improvements will be done. In listing 4.3 there is the pseudo code for the 2-Opt best fit operator which has been implemented.

Listing 4.3: 2-Opt best-fit pseudo code

```
INPUT : Distance matrix $D$, Tour $\overline{x}$, timelimit $t$, number of nodes $n$
OUTPUT: Tour $\overline{x}$

loop until no further improvement can be done  or timelimit has reached{

  // best exchanges found
  $g = null$
  $h = null$
  $\Delta = null$

  traverse sequence $i \in (1, 2, ..., n)$ {

    // edge $(j, (j+1) \equiv n)$ must be non adjacent
    traverse sequence: $j = (1, 2, ..., n) \setminus (i, (i+1) \equiv n, i-1)$ {

      //calculate possible tour length reduction
      $\Delta' = (d_{i,(i+1)\equiv n} + d_{j,(j+1)\equiv n}) - (d_{i,j} + d_{(i+1)\equiv n,(j+1)\equiv n})$

      // compare distances and store shortest distance found
      if( $\Delta' > 0$ and $\Delta' > \Delta$ ){
        $g = i$
        $h = j$
        $\Delta = \Delta'$
      }
    }
  }

  if( g, h != null ) {
    replace edges in $\overline{x}$: $(g, (g+1) \equiv n), (h, (h+1) \equiv n)$ with
        $(g, h), ((g+1) \equiv n, (h+1) \equiv n)$
  }
}

return $\overline{x}$
```

The 2-Opt heuristic achieves much better results on tour optimization than the NN alone. As already mentioned in section 2.1 the 2-Opt heuristic has an average Held-Karp lower bound excess of 5% but has much higher execution times than the NN. As we can see in the listing 4.3 one single iteration has a worst case runtime of $O(n^2)$ which is the same as the whole NN. The 2-Opt local search is combined with a nearest neighbor heuristic. This means the nearest neighbor heuristic is run first and the result is improved with the 2-Opt local search. As the start solution of the 2-Opt local search is much better than a random permutation runtimes are reduced, e.g., Freisleben and Merz [14] used a similar approach to create start solutions for a hybrid genetic algorithm which was used to solve the TSP. But they used the Lin-Kernighan algorithm instead of a 2-Opt local search.

As you can see in figure 4.3 the tour $T = (a, b, c, g, f, e, d, h, i, j)$ is adapted through the 2-Opt edge replacement to the new tour $T' = (a, b, c, d, e, f, g, h, i, j)$. So a single 2-Opt edge replacement leads to significant changes of the order of the nodes within the tour. We remember the penalty function in section 4.2 which makes use of the fact that the position of a node within the tour can have great impact on the profit of the items packed at this node as packing an item can lead to a significant velocity decrease. So although a 2-Opt edge replacement can lead to a more cost-effective tour it can worsen the total objective value of the TTP instance as the two subproblems (TSP, KP) are interconnected. So using the traditional 2-Opt heuristic after an initial solution for the tour and the packing plan has been created does not guarantee any increase of the objective value for the whole TTP instance.

Dynamic Programming

As already mentioned in section 2.2 dynamic programming is a very good choice for solving the KP. This is the reason why we also implement a dynamic programming operator to solve the KP part of the TTP. The principle of dynamic programming is very simple. A problem is split up into many different subproblems. This is done because in many cases these subproblems occur more than once in the whole problem and need to be solved more than once. So to save computation time the results of all solved subproblems get stored and reused if needed. [23]

In figure 4.4 and figure 4.5 you can see the principle of dynamic programming on using dynamic programming to calculate a Fibonacci number. A Fibonacci number f_n is calculated through the formula $f_n = f_{n-1} + f_{n-2}; \forall n \geq 2$. So if you want to calculate f_5 you do not calculate each Fibonacci number $f_n; n \leq 4$ again instead you reuse to stored results of these subproblems, e.g., the stored results of f_4 and f_3 are used to calculate f_5. In the case of the knapsack problems the stored subproblems would be if an item should be packed at a given remaining knapsack capacity. A subproblem is here characterized by its remaining knapsack capacity. So if we have two different packing configurations which need the same amount of capacity they are considered as solutions to the equal subproblem.

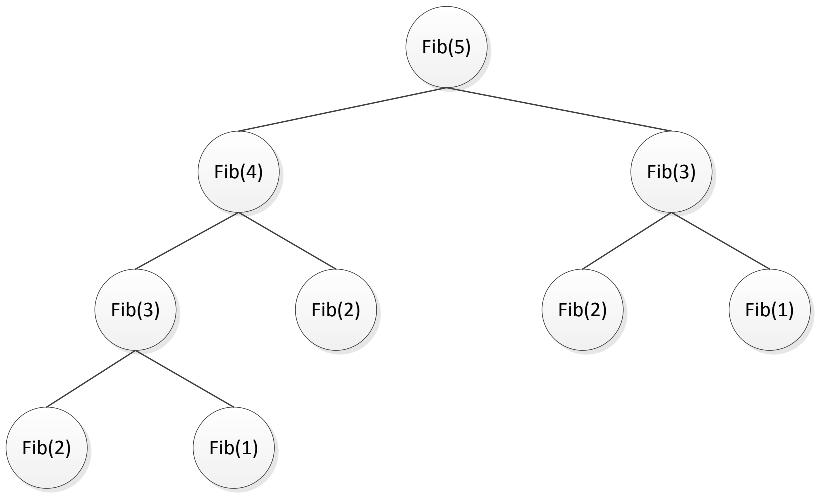

Figure 4.4: fibonacci sequence graph, redrawn based on figure from [2]

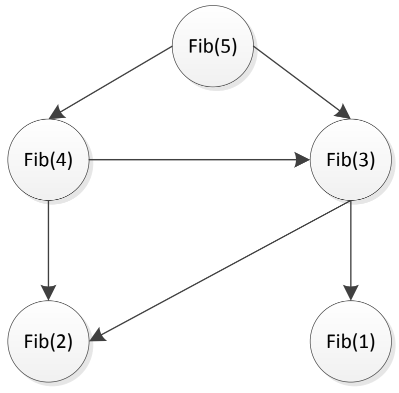

Figure 4.5: fibonacci sequence graph (reduced with dynamic programming), redrawn based on figure from [2]

So we know that we can use dynamic programming to solve the KP part but from section 4.2 we know that we can introduce knowledge about the problem interconnection by using the presented penalty function. You can see the used operator in the listing 4.4. First the the already mentioned penalty function is used to adapt the profits according to the tour.

An additional adaption has been done. As the instances of the TTP benchmark set [1] contain huge KP instances (see section A.2) a scaling mechanism has been introduced. This means that if the knapsack capacity multiplied with the number of items is bigger than 10^6 the weights and the knapsack capacity get divided by an equal factor up to 100 according to the instance size. These factors have to be adapted according to the problem instances, i.e., the knapsack capacity and the item weights. In this case max scaling factor of 100 and the determination of the factor has been chosen in respect to the problem instances of the TTP benchmark set [1]. Scaling reduces the performance of the dynamic programming routine therefore it should be reduced as much as possible and the weights must have a certain size else the weights are too similar and the performance gets badly reduced. We summarize the parameters of the scaling:

scaling factor = $min(100, max(1, \lfloor c * |I|/10^6 \rfloor))$
adapted weight $w' = \frac{w}{scalingFactor}$
adapted knapsack capacity $c' = \frac{c}{scalingFactor}$

After the dynamic programming routine has been finished a repair operator (see section 4.4) is used if an invalid packing plan has been created. This might occur as the weights are represented with integers which may lead to rounding errors.

Besides the scaling and the usage of the penalty function the operator behaves like a standard dynamic programming routine from the literature. This operator is based on [7] but has been adapted to be capable of calculating a packing plan for an TTP instance.

Listing 4.4: DynamicProgramming operator pseudo code

```
INPUT : item set $I$, tour $\overline{x}$, knapsack capacity $c$
OUTPUT: packing plan $\overline{z}$

// penalize item profits  according tour and weight
$I' =$ penaltyFunction$(\overline{x}, I)$

// do scaling of knapsack capacity and item weights $w_i$
$c' = c$
if( $|I| * c > 10^6$ ) {
  scalingFactor $s = min(100, max(1, \lfloor c * |I|/10^6 \rfloor))$
  $c' = \frac{c}{s}$
  $\forall i \in I' : w_i = \frac{w}{s}$
}

// use dynamic programming to create packing plan
$\overline{z} =$ dynamicProgramming$(I', c')$

// repair if necessary
$\overline{z} =$ repairPackingPlan$(I, \overline{x}, \overline{z})$

return $\overline{z}$
```

OX1 Crossover Operator

As already mentioned we have chosen the path representation for storing the tour part of the solution. As for most crossover operators schemata analysis is very difficult there are only a few attempts to measure the performance of the different TSP crossover operators. According to the literature the best operators are the partially-mapped crossover (PMX), the cycle-crossover (CX), order crossover/order based crossover (OX1/OX2), genetic edge recombination (ER), position based crossover (POS). But the order of the performance varies between the different comparison attempts. [25]

So I picked an crossover operator which is according to the literature always under the best performing crossover operators, the order crossover (OX1) operator. We can see that choosing the path representation is a very good choice for the TTP as we can use the 2Opt and OX1 operator using the same datastructures.

The OX1 makes use of the fact that the order of the different nodes of a tour are important and not the actual position. So if we have found a very cost-effective node sequence it does not matter for the TSP if it is situated at the beginning, the middle or the end of the tour. [25]

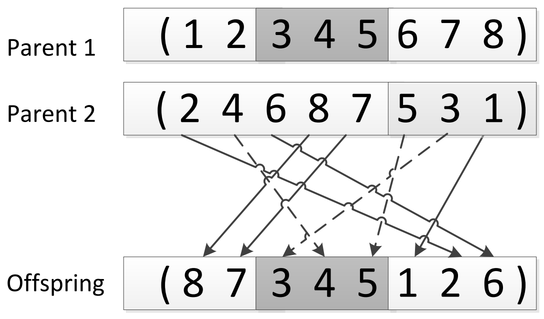

Figure 4.6: order crossover (OX1), redrawn based on figure from [25]

So if a crossover is done a subsequence of nodes is chosen from one parent and copied at the same position in the offspring as in the parent. After that the empty spaces are filled with nodes from the other parent in the same order as they appear there. Nodes which have already been copied to the offspring get skipped. We can see the principle in figure 4.6 and have a more detailed look at the creation of the offspring from figure 4.6.

There are two parents:

parent1	(1 2 3 4 5 6 7 8)
parent2	(2 4 6 8 7 5 3 1)

As we can see from parent1 the sequence $|345|$ is chosen to be copied at the same position in the offspring:

parent1	(1 2 |3 4 5| 6 7 8)
parent2	(2 4 |6 8 7| 5 3 1)
offspring	(* * |3 4 5| * * *)

Now the nodes of parent2 to get ordered starting with the first node after the second cut-point. So we receive the order sequence of the nodes of parent2 in which order they shall be copied into the offspring. We get the node sequence (5 3 1 2 4 6 8 7) from parent2 to fill into off2 where all nodes which are already contained in off2 are skipped. So the nodes (* * 1 * 2 * 6 8 7) get copied into off2 staring after the second cut-point:

parent1	(1 2 |3 4 5| 6 7 8)
parent2	(2 4 |6 8 7| 5 3 1)
offspring	(* * |3 4 5| * * *)
copied nodes from parent2	(8 7 |* * *| 1 2 6)
offspring result	(8 7 |3 4 5| 1 2 6)

In listing 4.5 we can see the pseudo code for the OX1-operator. This operator will produce one offspring for two given parents. If two offspring are needed the operator has to be run a second time and the arguments parent1 and parent2 have to be swapped. Additionally it is to mention that this algorithm will not produce invalid offspring as long as the parent tours are valid tour as the result will always contain all nodes of the node set exactly once and therefore be a valid tour.

Listing 4.5: OX1 operator pseudo code

```
INPUT :  tour parent1 $\overline{x}_1$, tour parent2 $\overline{x}_2$, cut-points $c_1, c_2$
OUTPUT: tour offspring $\overline{x}_o$

copy $\overline{x}_1[c_1, c_2]$ at $\overline{x}_o[c_1, c_2]$
$posOffspring = c_2$

// traverse all nodes starting from $c_2$ until $c_1$, having $c_1 < c_2$ {
traverse sequence $pos = (c_2, c_2 + 1, ...|\overline{x}_2|, 0, 1, ..., c_1)$
  if( not $\overline{x}_2[pos] \in \overline{x}_o$ ){
    $\overline{x}_o[posOffspring]$ $= \overline{x}_2$[pos]

    if( $posOffspring < |\overline{x}_2|$ ) {
            $posOffspring = posOffspring + 1$
          }
          else {
            $posOffspring = 0$
          }

          $posOffspring = posOffspring + 1$
        }
      }

      return $\overline{x}_o$
```

2-Point crossover operator

The 2-Point crossover operator is besides the 1-Point crossover operator, the Uniform crossover operator (UX) or the Half-Uniform crossover operator (HUX) a standard crossover operator for binary representation from the literature. Basically the 2-point crossover operator splits up the chromosomes of the parents into sub chromosomes using two cut-points. We can see the principle in figure 4.7. The parents are sliced into subchromsomes and the different subchromosomes are reunited in the offspring. The cut-points get chosen randomly as well as if the offspring starts with a sub chromosome from parent1 or parent2.

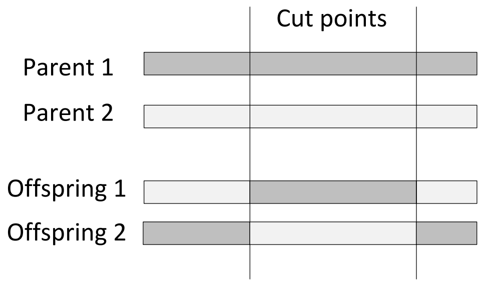

Figure 4.7: 2-Point Crossover (bases on figure from [30])

Additionally it is to mention that the standard crossover operator does not consider any constraints given by the optimization problem. So it is possible that invalid offspring are created. This means that an additional repair operator is needed.

The standard crossover operators which all work on bit strings have two forms of crossover biases and the choice of the used crossover operator can affect the performance of the genetic or hybrid genetic algorithm: [31]

Positional bias This bias describes the frequency of changes of bits in certain areas of the bit sub strings in the chromosome. As the search space is limited by the chromosome of the parents there is no guarantee that an offspring with an arbitrary chromosome can be produced. For example the parents 01110 and 10001 are not capable of producing the offspring 00000 with a 1-Point crossover operator in one generation. But the 2-Point crossover operator or the HUX would be capable of producing such an offspring within one generation. [31]

Distributional bias As the chromosome for standard crossover operators is represented with bit-strings the degree of difference between two individuals can be measured with the hamming distance. This bias measures the number of exchanged bits using a certain crossover operator, i.e., the hamming distance between individuals. [31]

For the packing plan crossover we will use the 2-Point crossover operator because it has a positional bias [31]. We are not only looking for the most profitable packing plan we are looking for a packing plan that also leads to a small knapsack rent. So we are looking for items which are deposited at lucrative node sequences. Such an ideal node sequence would be penned up in a small area and have a high density of profitable nodes which get collected. As we are looking for item groups or node groups we want to have a positional bias and therefore we will use an operator which offers one. That is the reason that we prefer the usage of a 2-Point Crossover operator over, e.g., a HUX which does not have a positional bias [31].

It should also be mentioned that the distributional bias has a very interesting side effect. Crossover operators without positional bias have a higher exchange rate of genes. So using the HUX crossover operator which does not have a positional bias results in a higher alteration rate than operators which have a positional bias like the 1-Point or 2-Point crossover operator. By using operators with varying distributional biases it is possible to adapt the alteration rate of the chromosome between the different generations of offspring. This could be used to have a big alteration rate at the beginning of the evolutionary algorithm and to reduce it when there have been offspring found with better objective values. But we will not use this possibility of controlling the chromosome alteration rate as we always want to have a crossover operator with positional bias. [31]

Mutation Operator

As already mentioned we need a mutation operator to avoid getting stuck in local optima. Therefore we define a mutation operator for the tour part and for the packing plan.

Tour mutation

The implemented operator creates two random numbers ranging from $[0; len(tour))$ and swaps the nodes at these positions as we can see in figure 4.8. Another approach would be to just randomly pick one node and shift it to the end of the tour. As the tour datastructure is an array to offer fast read operations this would leed to many shift operations and a worse performance like we can see in 4.8. Additionally the effect is the same in both cases: 4 edges will be altered as each node is connected with two edges to the other nodes within the tour. Shifting one node at end of the tour also alters 4 edges as the last node is connected with the first node. There is only one exception if the two swaped nodes are adjacent only 2 edges are changed.

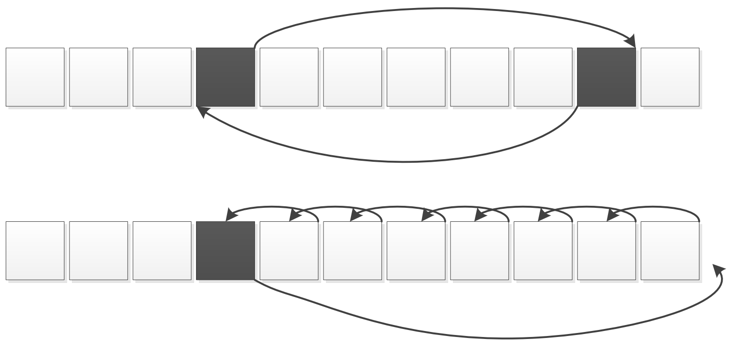

Figure 4.8: Mutation operations for the Tour

The operator needs as input the number of rounds where randomly choosen nodes are swapped.

Listing 4.6: TourMutation operator pseudo code

```
INPUT :  tour $\overline{x}$, mutationRounds $mr$
OUTPUT: mutated tour $\overline{x}$

loop until $mr$ = 0 {
  pick two random numbers $pos_1, pos_2 \in [0; |\overline{x}|]$
  $mr = mr - 1$

  swap $\overline{x}[pos_1], \overline{x}[pos_2]$
}

return $\overline{x}$
```

Packing Plan mutation

As the packing plan is represented with a bit string the mutation operator just randomly flips bits of the packing plan. The mutation operator for the packing plan also needs a mutation rate and a packing plan as input.

Listing 4.7: PackingPlanMutation operator pseudo code

```
INPUT :  packing plan $\overline{z}$, mutationRounds $mr$
OUTPUT: mutated packing plan $\overline{z}$

loop until rounds = 0 {
  pick random item $i \in [0; |\overline{z}|]$
  $\overline{z}[i] = 1 - \overline{z}[i]$

  $rounds = rounds - 1$
}

return $\overline{z}$
```

Repair Operator

As already mentioned we have to deal with infeasible packing plans so we will implement a repair operator which transforms infeasible packing plans into feasible packing plans again. To do so we will make use of the already introduced penalty function. As we remember this function will penalize all item profits according to their caused velocity loss and knapsack rent increase as we only want to pack items which contribute to the total profit.

First we only consider items which increase our total profit and remove the other from the item list. Now this greedy algorithm starts to remove items from the knapsack according to their profit per weight ratio ascending as long as the knapsack capacity is exceeded. After that we sort all unused items whose packing would lead to a profit gain again according to their weight beginning with the item with lowest weight. Then we iterate this sorted list as long as we can and add items without exceeding the knapsack capacity again. In 4.8 we can see the pseudo code of this repair operator which returns a feasible packing plan to a given infeasible packing plan.

Listing 4.8: Repair operator pseudo code

```
INPUT : packing plan $\overline{z}$, item set $I$, tour $\overline{x}$
OUTPUT: new packing plan $\overline{z}$

$I_p$ = penaltyFunction($I, \overline{x}$)
remove all items $i \in I_p$ having a negative profit $p_i < 0$
```

```
// remove items until capacity is not exceeded any more
$I' = I_p \cap \overline{z}$
sort $I'$ according profit to weight ratio ascending

iterate $i \in I'$ as long as $\overline{z}$ exceeds capacity {
  $\overline{z} = \overline{z} \setminus i$
}

// try to add items until knapsack exceeds
$I' = I_p \setminus \overline{z}$
sort $I'$ according weight ascending

iterate $i \in I'$ as long as $\overline{z} \cup i$ does not exceed capacity {
  $\overline{z} = \overline{z} \cup i$
}

return $\overline{z}$
```

Greedy packing operator

We have already discussed the greedy strategies which we will use for the TTP. This greedy operator tries to create a packing plan to a given tour. First we use the penalty function to determine the total profit gains of each individual item. After that we use the traditional profit per weight greedy strategy from the KP and iterate all items which lead to a profit gain beginning with the item with the highest profit to weight ratio.

Listing 4.9: Greedy packing operator pseudo code

```
INPUT : item set $I$, tour $\overline{x}$
OUTPUT: new packing plan $\overline{z}$

$I' =$ penaltyFunction($I, \overline{x}$)
sort $I'$ according profit to weight ratio descending

iterate $i \in I'$  {
  if( $\overline{z} \cup i$ does not exceed capacity and profit $p_i > 0$ ) {
    $\overline{z} = \overline{z} \cup i$
  }
}

return $\overline{z}$
```

Greedy tour derive operator

While the greedy packing operator creates a packing plan to a given tour this operator does the opposite: it creates a tour to a given packing plan. Here we are also going to use already discussed greedy strategies from 4.3. First we split up the node set according to the packing plan into two subsets. One containing all nodes where no item is packed and another containing all nodes where items get collected. After that we use a standard procedure from the literature to create a subtour with minimum length containing all nodes where no items get collected. This is done via a combination of the nearest neighbor heuristic and the 2-Opt heuristic. Now the second node subset containing all nodes where items get collected is sorted according to the total weight of all items which get collected at particular nodes ascending. As we remember from section 4.3 if we pack the same items at different positions within the tour we face different travel costs but we gain the same profit. Therefore we try to pack the heaviest items as late as possible. Now all these sorted nodes get added to the already created subtour from before. Then the new tour is returned. We can see the pseudo code in listing 4.10.

Listing 4.10: Greedy tour derive operator pseudo code

```
INPUT : packing plan $\overline{z}$, distance Matrix $D$, city set $X$, timelimit $t$
OUTPUT: new tour $\overline{x}$

// split up nodes (where items get packed and where not)
$X_p$ = all nodes where items get packed
$X_e$ = all nodes where no items get packed

// now create tour with nodes where nothing gets packed
choose random startnode $s \in X_e$
$\overline{x}$ = nearestNeighborHeurisitic($D, X_e, s$)
$\overline{x}$ = twoOpt($D, \overline{x}, t$)

// add remaining nodes according least collected weight
sort $X_p$ according collected weight ascending
$\overline{x} = \overline{x} \cup X_P$

return $\overline{x}$
```

Startnode search

The startnode search operator is an operator which is used to determine the best startnode and startnode successor of a given tour and a given packing plan. Although we might have already created a good solution we possibly can increase profits by alterating the startnode of the tour. We need to determine a startnode and a successor of this startnode as we can travel in two different directions. We can see that figure 4.9:

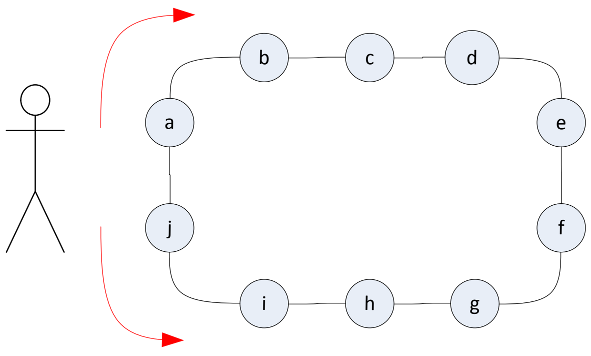

Figure 4.9: The thief can travel in two directions

So we iterate the whole tour in both directions and re-run the evaluation for each startnode and startnode successor pair again and keep the best configuration. We can see the principle in listing 4.11.

Listing 4.11: Startnode search operator pseudo code

```
INPUT : item set $I$, distance Matrix $D$, tour $\overline{x}$, packing plan $\overline{z}$, number of
    nodes $n = |\overline{x}|$
OUTPUT: new solution candidate $(\overline{x}', \overline{z})$

res = $(\overline{x}, \overline{z})$
$\overline{x}' = \overline{x}$
traverse all $x_i \in \overline{x}$ {
  set $x_i$ as startnode in $\overline{x}'$
  set $x_{(i+1) \equiv n}$ as startnode sucessor in $\overline{x}'$

  evaluate($\overline{x}', \overline{z}$)

  if total score of $(\overline{x}', \overline{z})$ > res {
    res = $(\overline{x}', \overline{z})$
  }

  // always consider both directions
  if( $i > 0$ {
    $j = i - 1$
  } else {
    $j = n$
  }
  set $x_j$ as startnode sucessor in $\overline{x}'$

  evalute($\overline{x}', \overline{z}$)
```

```
        if total score of $(\overline{x}',\overline{z})$ > res {
          res = $(\overline{x}',\overline{z})$
        }
      }

      return res
```

Stochastic Universal Sampling

The stochastic universal sampling (SUS) algorithm introduced by James Baker [9] is an alternative to the classical roulette wheel operator known from the literature. A selection operator should keep the population size constant and eliminate individuals which perform bad according to their fitness value. The performance of this operator is measured according following parameters: [9]

bias	the difference between the actual probability that an individual survives and the expected surviving chance according to the fitness value.
spread	the range of the number of possible offspring an individual can have in a generation.
efficiency	execution time spent on the operator

The benefits of the SUS operator are that it has zero bias, a minimum spread and a runtime of $O(n)$. [9]

Like the spinning wheel algorithm each individual gets a normalized objective value on this spinning wheel. This means that each individual i gets following normalized objective value nov_i according to its objective value ov_i assigned: [9]

$$nov_i = \frac{ov_i}{\Sigma_i ov_i}$$

All individuals are now put together on spinning wheel as you can see in figure 4.10. But instead of one we have as many pointers as survivors which are positioned in equal intervals at the spinning wheel. Now all individuals at which a pointer points are sampled and survive. The others get eliminated. To avoid a positional bias the individuals on the way have to be shuffled. Using this operator has the benefit that compared to the spinning wheel that there is no danger that a very strong individual compared to the population is sampled all the time and the variance of the population and therefore the search space is drastically reduced. [9]

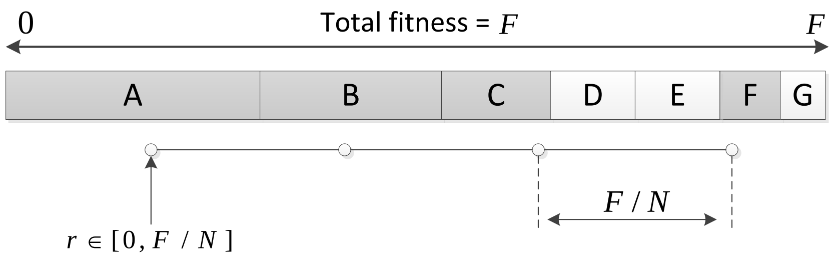

Figure 4.10: SUS operator, redrawn based on figure from [17]

In listing 4.12 we can see the pseudo code of the SUS operator. It is to mention here that a small adaption has been done: the best solution of the current population always survives.

Listing 4.12: SUS operator pseudo code

```
INPUT : solution set $S$, survivorsize $sz$
OUTPUT: new solution set $S'$

// evaluate all solutions
$\forall s \in S$ : evaluate($s$)
calculate normalized objective values $nov_s; \forall s \in S$

// SUS adaption: best solution is always choosen
// therefore best solution gets removed from the wheel
$S' = \{\} \cup$ best $s \in S$
$wheel =$ shuffle $S \setminus S'$

$pointerWidth = \frac{\Sigma nov_s}{sz-1}$
choose $randomStart \in [0; pointerWidth[$
$pointers = \emptyset$

$i = 0$
while( $i < sz - 1$ ) {
  $pointers = pointers \cup (i * pointerWidth + randomStart)$
  $i = i + 1$
}

traverse $p \in pointers$ {
  $S' = S' \cup s; s = wheel[p]$
}

return $S'$
```

4.5 Implemented Evolutionary Algorithms

As we have now defined all used operators we are going to discuss the used hybrid genetic algorithms to solve the TTP. Additionally I will provide an approach where each subproblem will be solved on its own and combined in a solution for the TTP instance and a genetic algorithm to bring out the performance differences between genetic and hybrid genetic algorithms.

Algorithm design decisions

Now we are going to discuss some important implementation details of the hybrid genetic algorithms which all variants have in common.

Selection phase

In the selection phase the population will always be reduced to the size given by the survivor size parameter. Therefore we will use the SUS operator to eliminate the appropriate number of individuals until we have the wished remaining population size. Additional the best result of each generation will always survive.

Offspring generation

The SUS operator reduces the population size to the given fixed parameter size and afterwards the remaining individuals are used by the crossover operators to repopulate.

The introduced crossover operators (OX1, 2-Point crossover) will use the tours or the packing plans of two parents and recombine them in a new packing plan or tour which belongs to the offspring. Afterwards the unsolved subproblem (KP or TSP) of the offspring will be derived from the already solved subproblem, e.g., the OX1 operator creates a tour for the offspring and the offspring packing plan will be derived from this tour using the greedy packing operator.

Solution mutation

Evolutionary algorithms use mutation to avoid to get stuck in local optima. We have already discussed that we use two forms of mutation operators. One which mutates the tour and one that mutates the packing plan. As the two subproblems are interconnected and the now introduced evolutionary algorithms will all derive one subproblem part (TSP or KP) from the other one for each new created offspring we will always mutate both solution parts, i.e., the tour and the packing plan. As the packing plan mutation can lead to infeasible solutions we will fix these with the introduced repair operator.

Abortion criteria

To provide results which are comparable to the TTP benchmark set we use similar abortion criteria. All solution attempts receive the same amount of computation time (10 minutes). [1]

This is a hard deadline this means the algorithm gets exactly 600 000 ms of computation time. After that the thread gets terminated. The best ever achieved result gets immediately stored after it has been found and will not be lost even if the deadline interrupts the algorithm. Additionally the algorithm can stop earlier if there is no improvement in solution quality of the best solution of the population in 100 generations.

Initial Population

For all genetic or hybrid genetic algorithm variants we need to create an initial population. To do so we use a similar procedure as Freisleben and Merz [14]. They start their algorithm by creating an initial population where the tour of each individual gets created by using the nearest neighbor heuristic and the Lin-Kernighan algorithm afterwards to further improve the tours. [14]

We use a similar approach. First a nearest neighbor heuristic creates a set of initial tours. As we have a much more complex problem through the interconnection with the KP we resign of a further tour improvement. After that we create an initial packing plan for each tour by using the greedy packing operator. In listing 4.13 we can see the pseudo code for the creation of the initial population.

Listing 4.13: Initial solution creation

```
INPUT : populationSize
OUTPUT: solution set $S$

solution Set $S = \emptyset$

loop until $|S|$ = populationSize {
  create new $s \in S; s = (\overline{x}_s, \overline{z}_s)$
  pick random $startnode \in X$
  nearestNeighbor($D, X, startnode$)
  greedyPacking($I, \overline{x}_s$)
}

return $S$
```

Algorithm Variants

Now we are going to discuss the different variants of the hybrid genetic algorithm as well as the solution approach where each subprolem gets solved on its own and combined in a solution for the TTP.

Traditional Approach

This approach is a very simple approach where each subproblem (TSP, KP) will be solved on its own and combined in a solution for the TTP. This means we start by picking a random startnode and create an initial tour by using the nearest neighbor heuristic. Afterwards we use up to 90% of the remaining computation time to further reduce the tourlength by using the 2-Opt operator. We use a timelimit as there is the danger that using the 2-Opt best fit heuristic on very large instances without timelimit would exceed the total computation time of 10 minutes. Now as we have a tour we create a packing plan by using the greedy packing operator. This solution will be further improved by the startnode search heuristic where an optimal startnode is searched for this particular solution of the TTP. This is done as the penalty function of the greedy packing operator is only capable of approximating the net profits. In listing 4.14 we can see the pseudo code for this solution approach.

Listing 4.14: Traditional approach

```
INPUT : none
OUTPUT: solution $s$

pick random $startnode \in X$

nearestNeighbor($D, X, startnode$)
twoOpt($D, \overline{x}_s$, remaining executionTime * 0.9)
greedyPacking($I, \overline{x}_s$)
startNodeSearch($I, D, \overline{x}_s, \overline{z}_s$)

return $s$
```

Hybrid Genetic Algorithm

In listing 4.15 we can see the pseudo code of the hybrid genetic algorithm. As already discussed this algorithm will run for 10 minutes or 100 generations without any improvement. First we start by creating an initial population. After that the actual algorithm starts with an selection phase in each round. Now in the next phase an offspring get created. There are three variants how they will be created:

HgaTour in this variant all offspring are created via tour crossover with OX1 crossover operator.

HgaPackingPlan this variants makes solely use of the 2-Point-Crossover operator to create new offspring.

HgaRandom this variant will randomly choose for each new created offspring if it is created via tour or packing plan crossover.

After that the solution of the other subproblem gets derived from the existing solution, i.e., if a tour crossover has been done the packing plan gets derived via a greedy packing or dynamic programming routine and if a packing plan crossover has been done the tour gets derived via the greedy tour derive operator.

Now the tour gets mutated and a 2-Opt local search is done to improve the solution quality of the offspring if it has been created via tour crossover. The 2-Opt improvement is very cost intensive especially if used on big instances so we do not want to waste too much computation time on individuals which get eliminated in further generations. Therefore each member of the population gets 1 second of 2-Opt improvement for each survived generation.

Now the packing plans of all offspring get mutated and repaired if necessary. After that an additional local search is done with the startnode search operator to further improve the quality of the provided solutions.

There is one additional variant *GaTour* which is a pure genetic algorithm without local search. This algorithm is provided to examine if the usage of additional local search operators can improve solution quality. Additionally it is to mention the *HgaTour* variant makes use of the 2-Opt local search to improve the whole population and not only the newly created offspring.

Listing 4.15: The Hybrid genetic algorithm pseudo code

```
OUTPUT: new solution $s'$

create initial solution set $|S| = populationSize$

// loop until deadline
while( abortion criteria not fulfilled ) {

  // selection phase
  selection via SUS operator (always pick best $s \in S$, $|S| = survivorSize$ )
  offspring set $O = \emptyset$

  // offspring creation
  while( $|O| + |S| \neq populationSize$ ) {

    create offspring $o$ via chosen crossover type

    if( $o$ created via tour crossover (OX1) ) {
      derive packing plan $o.\overline{z}$ from tour $o.\overline{x}$
      mutate tour $o.\overline{x}$
      do local search via 2Opt on $o.\overline{x}$
    }
    else {
      derive tour $o.\overline{x}$ from packing plan $o.\overline{z}$
      mutate tour $o.\overline{x}$
    }

    $O = O \cup o$
  }

  $\forall o \in O$ : {
    mutate & repair packing plan $o.\overline{z}$
    perform startnode search
  }

  $S = S \cup O$
}

return best $s' \in S$
```

CHAPTER 5

Results

Now we are going to have a look at the instances of the benchmark set first. After that we discuss the operator choice and the configuration of the parameters of the evolutionary algorithms. Then we will examine the results of the algorithm variants.

5.1 Benchmark set instances

As already mentioned in section 2.3 the benchmark set [1] provides problem instances for the TTP based on the TSPLIB [5] combined with a KP problem generator from [6]. In particular each city of an TSPLIB instance is associated with some items which were generated with the KP problem generator. There are following different types of profit and weight distributions for the item:

uncorrelated weight and profit are uniformly distributed within a range of $[1; 10^3]$.

uncorrelated with similar weights weights are uniformly distributed within a range of $[10^3; 10^3 + 10]$ and profits are uniformly distributed within a range of $[1; 10^3]$.

bounded strongly correlated weights w_{ik} are uniformly distributed within a range of $[1; 10^3]$ and profit $p_{ik} = w_{ik} + 100$

Additionally the provided TTP instances differ in the number of items per city (Itemfactor $F_I \in \{1, 3, 5, 10\}$) and the knapsack capacity category $F_c \in 1, 2, 3, 4, 5, 6, 7, 8, 9, 10$. This means that each city contains IF_I items and the knapsack has a capacity of $F_c * c$ where c is given by the instance. Instances with an higher itemfactor also have a higher knapsack capacity. So for each instance the of TSPLIB there are many different KP instances provided which leads to the situation the TTP benchmark set contains 9720 instances in total. [1]

5.2 Hardware Specification & Runtime

All algorithm executions were executed on a Intel Xeon E5540, 2.53 GHz Quad Core with 24 GB RAM. Each algorithm execution for an specific instance received a maximum of 10 minutes cpu time. Each algorithm variant was executed 30 times for each instance.

5.3 Results

Now we are going to examine the performance of our algorithms and operators. It should be mentioned that each TTP instance which is compared is represented with three sub instances with the different weight types. As we remember the TTP benchmark sets distinguishes between bounded strongly correlated, uncorrelated and uncorrelated similar weights [1]. So these comparisons have been done on instances with a certain capacity category and item factor but on all different weight types. Afterwards the results have been averaged. There is one exception tables which declare an explicit datatype show comparisions of a specific datatype.

Operator choice

As we need an initial population and later we want to improve existing tours we will make use of the nearest neighbor heuristic and the 2-Opt operator to create initial tours and to improve them. We also need a packing plan. Therefore we will compare the usage of the greedy packing operator and the dynamic programming operator. Additionally we will examine if we can further improve performance with the usage of the start node search operator.

The problem with dynamic programming is that using dynamic programming on large instances will cause consumption of huge amounts of RAM and computing time. As we remember from section 4.4 we have introduced scaling but the operator can only scale down to maximum of factor 100 as there are no items in the benchmark set with a weight > 1000 and most of these items are within a range of $[100; 1000]$. Therefore we will examine which operators suits better. This evaluation has been done on small instances to save computation time.

We will see now two evaluation tables 5.1 and 5.2. All configurations were executed independently:

Greedy, NN this configuration uses the NearestNeighbor heuristic and the greedy packing operator.

Dynamic Programming this configuration uses the NearestNeighbor heuristic and the dynamic programming operator.

NN, 2-Opt this configuration uses the NearestNeighbor heuristic, the 2-Opt operator and the greedy packing operator.

NN, 2-Opt, SNS this configuration uses the NearestNeighbor heuristic, the 2-Opt operator, the startnode search operator and the greedy packing operator.

On this first run the dynamic programming operator does not use any scaling. Both the greedy and the dynamic programming operator rely on the results of the penalty function how much profit could be gained by packing it. Both operator evaluation use the same tours as input. As we can see in table 5.1 using the dynamic programming operator leads to a slightly increased profit gain. This is not very surprising as the knapsack capacity is very big in relation to the item weights and as the greedy operator packs according to highest profit to weight ratio it performs nearly as good as dynamic programming. Although dynamic programming performs a little better using the greedy operator uses much less computation time and memory and performs only slightly worse. Therefore only the greedy packing operator is used in the hybrid genetic algorithms.

instance	Greedy Packing	Dynamic Programming
eil51-TTP	14150,86	14181,38
berlin52-TTP	15241,13	15275,36
st70-TTP	21651,14	21678,67
eil76-TTP	23038,89	23076,86
eil101-TTP	29645,12	29671,83

Table 5.1: Achieved profit increase according to penalty function; Capacity Category = 1; Item Factor = 3, all data types combined, same tours

Now we have a look at the performance increase through the usage of the startNodeSearch operator. All three presented operator configuration use the greedy packing operator but individual created tours now. We can see in table 5.2 that using the 2-Opt operator has a great impact on the performance. We can further increase the performance by using the startnode search operator for an ideal start point. Here we can see the disadvantages of our penalty function. To increase the objective value even more we would need a more precise penalty function but as we have seen in section 4.2 the runtime would drastically increase therefore a trade off between computation time and solution quality has to be done.

	NN	NN, 2-Opt	NN, 2Opt, SNS
eil51-TTP	-2547,99	271,82	3386,53
berlin52-TTP	-856,37	1140,59	5355,54
st70-TTP	-1500,12	1431,17	7945,49
eil76-TTP	-2653,61	-1124,39	3960,79
eil101-TTP	-9095,01	-2476,78	4048,60

Table 5.2: Comparision of mean objective values; Capacity Category = 1; Item Factor = 3

Parameter adjustment

All variants of the hybrid genetic algorithm and the genetic algorithm have 4 different parameters which determine their performance and also their run times:

population size	the number of individuals within the population
survivor size	the number of surviving individuals of each generation. These individuals are chosen for offspring creation.
mutation rate tour	the mutation rate of the tour in percent. This means how many edges will be different from the not mutated solution.
mutation rate packing plan	the mutation rate of the packing plan in percent. This means how many items from the mutated solution will have a different packing status .

	eil51-TTP	berlin52-TTP	st70-TTP	eil76-TTP	eil101-TTP
TA5	-5080,60	-3024,14	-5018,09	-6919,61	-13545,17
HGARANDOM P10	6545,89	8828,66	11068,50	9183,98	11827,23
HGARANDOM P20	6510,28	8872,01	11047,22	9172,83	11844,62
HGARANDOM P40	6518,78	8829,03	11032,94	9137,20	11755,77
HGARANDOM P80	6461,10	8829,18	11046,92	9101,93	11698,14
HGAPP P10	-1051,43	1795,65	2337,63	140,22	-4502,10
HGAPP P20	-186,85	2039,92	3578,21	1005,67	-3060,33
HGAPP P40	587,30	2404,54	4255,42	1903,22	-1519,38
HGAPP P80	1009,30	2618,94	4669,19	2508,18	-18,61
HGATOUR P10	6679,35	9312,66	11665,71	9565,74	11760,96
HGATOUR P20	6763,37	9366,43	11757,34	9688,72	11990,38
HGATOUR P40	6849,38	9407,65	11860,84	9734,02	12108,06
HGATOUR P80	6859,62	9426,68	11920,99	9862,01	12199,94
GATOUR P10	979,52	3901,30	4511,57	2443,20	88,41
GATOUR P20	1928,24	5007,13	5459,88	3871,16	1667,94
GATOUR P40	2332,90	5495,21	6138,60	4423,47	2463,95
GATOUR P80	2604,49	5836,20	6458,43	4904,23	3174,63

Table 5.3: Comparision of mean objective values for different population sizes; Capacity Category = 1; Item Factor = 3

	eil51-TTP	berlin52-TTP	st70-TTP	eil76-TTP	eil101-TTP
TA5	-5560,60	-2988,52	-5255,20	-6422,41	-13504,75
HGARANDOM S25%	6527,87	8842,57	11080,57	9203,66	11849,71
HGARANDOM S50%	6543,33	8844,80	11051,26	9200,70	11804,64
HGARANDOM S75%	6534,15	8837,39	11036,15	9152,83	11785,84
HGAPP S25%	624,80	2425,08	4255,10	1950,22	-1505,69
HGAPP S50%	529,35	2361,86	4184,47	2018,09	-1501,82
HGAPP S75%	42,34	2378,64	4025,60	1566,29	-2034,41
HGATOUR S25%	6762,52	9348,29	11815,89	9714,29	12048,25
HGATOUR S50%	6778,89	9376,38	11757,15	9651,75	12005,48
HGATOUR S75%	6820,25	9377,33	11686,48	9753,35	11905,09
GATOUR S25%	2166,76	5033,60	5730,27	4123,08	2649,21
GATOUR S50%	2809,15	5819,08	6490,34	4954,47	3489,24
GATOUR S75%	3839,74	7013,73	7391,29	6128,90	4520,67

Table 5.4: Comparison of mean objective values for different survivor rates; Capacity Category = 1; Item Factor = 3

	eil51-TTP	berlin52-TTP	st70-TTP	eil76-TTP	eil101-TTP
TA5	-5587,31	-2229,12	-5152,53	-6043,45	-15117,11
HGARANDOM M2%	6525,09	8820,42	11085,47	9157,12	11806,43
HGARANDOM M5%	6544,23	8822,17	11029,21	9216,22	11781,51
HGARANDOM M10%	6495,57	8807,43	10991,58	9161,92	11714,73
HGAPP M2%	640,84	2427,93	4268,13	1865,80	-1425,93
HGAPP M5%	544,21	2369,52	4132,58	1949,70	-1635,42
HGAPP M10%	497,85	2323,53	4064,33	1781,38	-2408,88
HGATOUR M2%	6745,23	9395,90	11727,71	9680,63	11948,20
HGATOUR M5%	6760,92	9365,74	11749,39	9696,24	11935,64
HGATOUR M10%	6722,59	9336,38	11722,22	9615,98	11833,02
GATOUR M2%	1995,13	5271,71	5981,08	4197,57	2528,84
GATOUR M5%	2126,70	5171,14	5807,69	4101,25	1550,73
GATOUR M10%	1820,53	4919,45	5412,32	3772,18	1152,75

Table 5.5: Comparision of mean objective values for different tour mutation rates; Capacity Category = 1; Item Factor = 3

	eil51-TTP	berlin52-TTP	st70-TTP	eil76-TTP	eil101-TTP
TA5	-5542,74	-2949,33	-5446,50	-5462,61	-12984,34
HGARANDOM M2%	6537,62	8853,73	11041,59	9201,66	11777,82
HGARANDOM M4%	6436,26	8739,97	10920,21	9050,07	11665,81
HGARANDOM M6%	6377,94	8684,78	10836,72	8962,97	11515,19
HGARANDOM M8%	6355,78	8635,41	10801,27	8853,59	11356,89
HGAPP M2%	644,89	2332,38	4193,21	1993,97	-1450,04
HGAPP M4%	554,64	2371,17	4267,39	2066,16	-1612,27
HGAPP M6%	516,19	2330,37	4107,90	1728,80	-1752,11
HGAPP M8%	440,01	2376,59	4116,87	1721,78	-1881,98
HGATOUR M2%	6760,85	9357,38	11753,63	9703,28	11975,65
HGATOUR M4%	6675,47	9318,90	11720,58	9605,24	11858,60
HGATOUR M6%	6635,70	9279,76	11590,04	9512,16	11826,23
HGATOUR M8%	6637,34	9232,98	11563,85	9460,82	11775,30
GATOUR M2%	2172,91	5069,75	5960,81	4158,77	2740,75
GATOUR M4%	1804,59	5058,28	5642,80	3712,54	2036,86
GATOUR M6%	1686,35	4820,17	5239,23	3621,36	2009,44
GATOUR M8%	1682,27	4901,81	5184,82	3617,44	1986,87

Table 5.6: Comparison of mean objective values for different packing plan mutation rates; Capacity Category = 1; Item Factor = 3

Population size

As we can see in table 5.3 the differences between the achieved objective values between the different population sizes are rather small. For the hybrid genetic algorithm it seems that with an population size from 20 to 40 we can achieve good results. I do not set them too big as we are only looking at very small instances and using bigger populations on big instances could lead to the affect that the benefit of the higher diversity is drastically reduced by the lack of execution time as much more local search operations have to be done on different individuals.

For the genetic algorithm approach I choose a even bigger population size of 100 as we can see a significant objective value increase with higher population and this algorithm does not use any local search operators. So we have following population sizes:

HGATOUR	20
HGAPP	40
HGARANDOM	20
GATOUR	100

Survivor Size

Like the examination sizes the hybrid genetic algorithm variants achieve similar objective values on the different survivor rates as we can see in table 5.4. So I choose a survivor rate of 50% to to have a good mix of diversity and computation time savings. For the genetic algorithm we can see having a low survivor rate increase performance so I choose a low survivor rate. So we will use following survivor rates:

HGATOUR	50%
HGAPP	50%
HGARANDOM	50%
GATOUR	25%

Mutation Rate Tour

As we can see in table 5.5 high mutation rates on the tour have a rather negative effect I stick to the lowest mutation rate of 2%. So we will use following mutation rates for the tour:

HGATOUR	2%
HGAPP	2%
HGARANDOM	2%
GATOUR	2%

Mutation Rate Packing Plan

In table 5.6 we can see that the mutation rate of the packing plan has only little effect on the objective values therefore I stick to rather low mutation rate of 2% except for the HGAPP1. For this one I choose one of 4%. Therefore we have following mutation rates for the packing plan:

HGATOUR	2%
HGAPP	4%
HGARANDOM	2%
GATOUR	2%

Final results

Now we have a look at the execution results of bigger instances. TA stands for traditional approach which has been introduced in section 4.5.

	NN & Greedy	TA	HGA-RANDOM	HGAPP	HGATOUR	GATOUR
pr76	-4928,35	12854,88	15725,37	4248,55	15812,60	8396,44
(pr76)$_{\text{SD}}$	8775,88	4079,77	4175,38	3518,12	4138,68	2227,88
pr124	-10993,30	13904,83	19683,23	6374,58	19455,14	9429,23
(pr124)$_{\text{SD}}$	14867,92	6114,63	2384,00	5267,34	2379,36	4477,00
rat195	1266,71	21114,01	26827,30	17878,58	27550,47	19336,95
(rat195)$_{\text{SD}}$	11015,36	11493,16	7572,35	7160,50	7260,54	7174,40
gil262	-24659,34	13199,44	23452,06	2287,50	28999,08	7025,42
(gil262)$_{\text{SD}}$	22314,51	34770,88	8534,59	10202,98	7000,36	10648,10
rd400	-36810,34	20012,95	19064,39	-8309,14	34008,34	5620,04
(rd400)$_{\text{SD}}$	33634,37	17039,84	17062,76	23850,37	10020,92	18914,49
d657	-79071,80	74239,94	50128,57	24880,95	84676,43	38139,69
(d657)$_{\text{SD}}$	51674,10	18612,89	28487,89	36707,59	16035,47	24765,48
pr1002	-97661,19	39259,34	1169,52	1872,00	80070,34	24025,57
(pr1002)$_{\text{SD}}$	82038,70	36236,48	49072,28	52442,21	28256,33	39508,26
d1291	-100328,00	112329,10	47007,67	52462,94	120777,76	61469,71
(d1291)$_{\text{SD}}$	75907,49	35055,92	35514,74	40670,08	33247,15	38637,06
fl1577	-17883,76	131462,38	55612,61	61906,50	128086,44	70717,72
(fl1577)$_{\text{SD}}$	55936,17	39011,12	34381,29	33455,67	34884,79	35564,53
d2103	-89639,68	199388,74	106792,50	116076,50	249746,33	140637,84
(d2103)$_{\text{SD}}$	137985,29	75447,06	58697,08	64780,27	57439,21	66447,50

Table 5.7: Comparison of mean objective values; Capacity Category = 1; Item Factor = 3

	NN & Greedy	TA	HGARANDOM	HGAPP	HGATOUR	GATOUR
pr76	< 1	< 1	47,76	13,32	62,66	3,46
pr124	< 1	< 1	206,09	28,81	256,71	5,81
rat195	< 1	< 1	429,40	106,08	565,46	8,57
gil262	< 1	< 1	600,00	473,10	600,00	14,34
rd400	< 1	4,96	600,00	600,00	600,00	23,83
d657	< 1	15,81	600,00	600,00	600,00	33,15
pr1002	< 1	16,52	600,00	600,00	600,00	72,99
d1291	< 1	17,52	600,00	600,00	600,00	63,00
fl1577	< 1	18,36	600,00	600,00	600,00	76,54
d2103	< 1	21,21	600,00	600,00	600,00	128,07

Table 5.8: Comparison of mean execution times (s); Capacity Category = 1; Item Factor = 3

In table 5.7 we can see the results of our evaluation of the performance of the different approaches. What we can see here is that the best performer is always one of the evolutionary algorithms. Additionally what we can see that using a packing plan crossover operator and deriving a tour does not really work well. Recombine different tours and recreate a packing plan leads to much better results. The genetic algorithm variant performs much better than HGAPP although it does not use any local search. Also using a hybrid genetic algorithm, i.e., combining a genetic algorithm and local search leads to much better results.

But the evolutionary algorithms need much more runtime as soon as the instances get bigger the execution times exceed as we can see in table 5.8. The genetic algorithm needs the least runtime until convergence of all evolutionary algorithm variants. As soon runtime exceeds the gap between the objective values of the traditional approach and the evolutionary algorithms becomes much smaller.

Summarized we can see that the most promising of the presented approaches seems to be a hybrid genetic algorithm which uses a tour crossover operator. It seems to be necessary to reduce the tour length as much as possible. As we remember table 5.2 reducing the tour length lead to an signifacnt improvement of the achieved objective values. But having only a minimum tour does not guarantee a high objective value. It is very important to look out for a tour with a very profitable position configuration of the collected items. All algorithm variants which were constantly adapting their tours achieved better objective values than algorithm variants which did not constantly adapted their tours as long as they did not run out of run time. Additionally the available runtime has to be adapted in aspect to the instance size.

Comparision Adelaide

Now we are going to compare our results with the results from the TTP benchmark set presented in [1]. We can see the results in table 5.9 and 5.11. In table 5.10 we can see the mean runtimes of the different algorithm variants.

In [1] the authors use a simple (1+1) evolutionary algorithm (EA), a random local search (RLS) and a deterministic constructive heuristic (SH) to solve the TTP problem. All three approaches use the Chained Lin-Kernighan heuristic (CLK) to create a fixed tour for the TTP solution. After that only the KP part is considered any more. To do so in each iteration a new solution for the KP is created and kept if an improvement has been done. The SH creates a scoring based on the profits of the items minus the increased renting costs which arise if they get picked. If the same item is at the start of the tour it gets less valuable this way. After that the items get sorted according their score. Now the SH picks items from the sorted list as long the knapsack does not exceed its capacity. The RLS tries to improve the solution by randomly inverting the packing status of an item. The EA randomly inverts the packing status of all items with a certain probability. The last column PackNone in table 5.11 is a tour created with the CLK heuristic where no items get packed at all.

	NN & Greedy	TA	HGARANDOM	HGAPP	HGATOUR	GATOUR
eil51	-14308,97	6168,24	9429,98	-5628,63	10061,56	1379,10
$(\text{eil51})_{SD}$	6474,52	916,65	441,78	727,71	330,65	1332,77
eil76	-17359,62	8202,34	11638,60	-4418,80	13040,27	2346,30
$(\text{eil76})_{SD}$	7337,98	1799,07	491,65	674,44	507,70	1559,18
kroA100	-16303,96	12627,94	17294,23	4034,37	22443,50	7677,85
$(\text{kroA100})_{SD}$	12718,30	1984,29	532,06	1206,75	443,95	2340,47
u159	-31968,52	18711,91	34242,23	6108,11	38030,46	15072,57
$(\text{u159})_{SD}$	12839,91	15883,37	1097,86	5734,60	1087,38	3566,57
ts225	96,74	49206,49	57060,77	32848,37	52850,39	34017,23
$(\text{ts225})_{SD}$	19773,80	4416,60	961,66	2075,72	1045,77	818,96
a280	-70282,83	30486,18	39408,48	-16821,28	47121,30	1766,61
$(\text{a280})_{SD}$	25135,93	7457,44	2931,65	5327,11	2048,86	7319,98
u574	-151886,70	59721,57	48369,54	-64016,10	91337,31	1101,33
$(\text{u574})_{SD}$	35942,55	20961,42	13650,62	17657,42	6530,86	10888,14
u724	-78449,44	92516,91	38800,86	25998,24	113414,18	41243,25
$(\text{u724})_{SD}$	61037,66	12308,30	21865,99	12193,88	11039,56	13469,82
dsj1000	-559290,25	-387636,02	-356251,64	-350475,43	-113585,36	-269691,60
$(\text{dsj1000})_{SD}$	132891,98	518511,48	27997,57	23252,47	17573,23	26857,47
rl1304	-284573,05	-1160,10	-136828,19	-120010,10	105603,88	-68596,77
$(\text{rl1304})_{SD}$	94426,74	25736,20	20375,56	28741,55	13162,80	30822,23

Table 5.9: Comparison of mean objective values; Capacity Category = 2; Item Factor = 3, datatype = uncorrelated item weights

	NN & Greedy	TA	HGARANDOM	HGAPP	HGATOUR	GATOUR
eil51	< 1	< 1	13,00	2,66	14,00	2,01
eil76	< 1	< 1	33,44	6,55	38,30	3,10
kroA100	< 1	< 1	106,29	13,79	153,56	4,21
u159	< 1	< 1	358,49	23,86	541,25	7,76
ts225	< 1	< 1	600,00	52,45	600,00	8,47
a280	< 1	< 1	600,00	89,42	600,00	18,00
u574	< 1	15,32	600,00	600,00	600,00	35,71
u724	< 1	15,92	600,00	600,00	600,00	33,80
dsj1000	< 1	16,82	600,00	600,00	600,00	74,46
rl1304	< 1	17,85	600,00	600,00	600,00	92,51

Table 5.10: Comparison of mean execution times (s) ; Capacity Category = 2; Item Factor = 3, datatype = uncorrelated item weights

	RLS	SH	EA	PackNone
eil51	8214,06	-3028,93	8217,45	-14614,56
eil76	11568,19	-3653,75	11580,49	-23493,60
kroA100	19310,43	-4834,40	19313,90	-25827,45
u159	40267,22	-5369,61	40263,09	-40422,72
ts225	56997,05	-7492,31	57002,73	-55733,04
a280	63180,82	-20508,54	63182,75	-73686,60
u574	136199,38	-19942,76	136197,28	-145678,78
u724	166509,37	-51624,31	166509,25	-191887,64
dsj1000	111238,25	-189636,78	111233,67	-375087,30
rl1304	311548,14	-96869,65	311552,01	-361981,62

Table 5.11: Comparison of mean objective values from [1]; Capacity Category = 2; Item Factor = 3, datatype = uncorrelated item weights

What we can see is that the HGATOUR algorithm is capable of providing better results than the algorithms from the literature on small instances. But on bigger instances all of our presented algorithm variants provide a worse performance than the best algorithm from [1]. The reason for this is as instances are getting bigger hybrid genetic algorithm lacks on execution time and also the accuracy of the penalty function gets more reduced. Execution times do not get compared as the code of the [1] stores results of the tour generation part on disk and reloads them on repeating runs on the same tour.

To provide a better comparision of the different algorithms I reimplemented the in [1] presented EA and RLS and executed them on the same grid as all presented algorithm variants. In table 5.12 and table 5.13 we can see the results. Algorithm AREA is my implementation of the EA algorithm from [1] and algorithm ARRLS ist my implementation of the RLS algorithm from [1].

	NN & Greedy	TA	AREA	ARRLS	HGARANDOM	HGAPP	HGATOUR	GATOUR
pr76	-4928,35	12854,88	-4572,10	-4020,74	15725,37	4248,55	15812,60	8396,44
$(pr76)_{SD}$	8775,88	4079,77	3609,94	3753,21	4175,38	3518,12	4138,68	2227,88
pr124	-10993,30	13904,83	-13192,33	-13096,42	19683,23	6374,58	19455,14	9429,23
$(pr124)_{SD}$	14867,92	6114,63	8857,92	8934,01	2384,00	5267,34	2379,36	4477,00
rat195	1266,71	21114,01	-18947,32	-19300,25	26827,30	17878,58	27550,47	19336,95
$(rat195)_{SD}$	11015,36	11493,16	11360,52	10889,68	7572,35	7160,50	7260,54	7174,40
gil262	-24659,34	13199,44	-27083,88	-26775,08	23452,06	2287,50	28999,08	7025,42
$(gil262)_{SD}$	22314,51	34770,88	14679,75	15139,89	8534,59	10202,98	7000,36	10648,10
rd400	-36810,34	20012,95	-45160,59	-45168,82	19064,39	-8309,14	34008,34	5620,04
$(rd400)_{SD}$	33634,37	17039,84	26123,77	25881,23	17062,76	23850,37	10020,92	18914,49
d657	-79071,80	74239,94	-60361,95	-61129,55	50128,57	24880,95	84676,43	38139,69
$(d657)_{SD}$	51674,10	18612,89	34856,82	35848,79	28487,89	36707,59	16035,47	24765,48
pr1002	-97661,19	39259,34	-108824,90	-108608,04	1169,52	1872,00	80070,34	24025,57
$(pr1002)_{SD}$	82038,70	36236,48	62035,22	62491,95	49072,28	52442,21	28256,33	39508,26
d1291	-100328,00	112329,10	-142828,29	-148163,53	47007,67	52462,94	120777,76	61469,71
$(d1291)_{SD}$	75907,49	35055,92	80098,82	76709,55	35514,74	40670,08	33247,15	38637,06
fl1577	-17883,76	131462,38	-182602,44	-182624,35	55612,61	61906,50	128086,44	70717,72
$(fl1577)_{SD}$	55936,17	39011,12	91676,87	91155,41	34381,29	33455,67	34884,79	35564,53
d2103	-89639,68	199388,74	-271738,93	-276533,72	106792,50	116076,50	249746,33	140637,84
$(d2103)_{SD}$	137985,29	75447,06	90627,19	88708,29	58697,08	64780,27	57439,21	66447,50

Table 5.12: Comparison of mean objective values; Capacity Category = 1; Item Factor = 3

	NN & Greedy	TA	AREA	ARRLS	HGARANDOM	HGAPP	HGATOUR	GATOUR
pr76	< 1	< 1	< 1	< 1	47,76	13,32	62,66	3,46
pr124	< 1	< 1	< 1	< 1	206,09	28,81	256,71	5,81
rat195	< 1	< 1	< 1	< 1	429,40	106,08	565,46	8,57
gil262	< 1	< 1	< 1	< 1	600,00	473,10	600,00	14,34
rd400	< 1	4,96	1,26	1,28	600,00	600,00	600,00	23,83
d657	< 1	15,81	2,25	2,30	600,00	600,00	600,00	33,15
pr1002	< 1	16,52	2,67	2,66	600,00	600,00	600,00	72,99
d1291	< 1	17,52	2,52	2,53	600,00	600,00	600,00	63,00
fl1577	< 1	18,36	6,29	6,29	600,00	600,00	600,00	76,54
d2103	< 1	21,21	4,52	4,52	600,00	600,00	600,00	128,07

Table 5.13: Comparison of mean execution times (s); Capacity Category = 1; Item Factor = 3

In this comparision we can see that the hybrid genetic algorithm variants are capable of outperforming the algorithms in the literature also on some bigger instances but they need much more execution time as all other compared algorithms. Summarized we can see that the most promising of the presented approaches seems to be a hybrid genetic algorithm which uses a tour crossover operator.

Statistic Evaluations

To prove the assumptions made on the performance of the different algorithms the execution results are tested with an Willcoxon signed rank test with an error level of 5% (created with R 2.14.1). Each algorithm got compared with all other algorithms. In table 5.14 we can see which algorithm was significantly better than other algorithms on which instances. Additionally in table 5.15 we can see on how many instances which algorithm performed better than other algorithms (instances where both algorithms performed equally often better than the other algorithm are not taken into account). Additionally it is to mention that although an algorithm perfomed better than an other algorithm on a specific instance, the differences can be so little that they are not significantly e.g. AREA and RLS. What we can see is that HGARANDOM is the best performing algorithm and is capable of beating existing algorithms from the literature on some instances.

Algorithm A	Algorithm B	Instances where A is significantly worse than B	Instances where B is significantly worse than A
HGAPP	TA	d1291, d2103, d657, fl1577, gil262, pr1002, pr124, pr76, rat195, rd400	
HGAPP	HGATOUR	d1291, d2103, d657, fl1577, gil262, pr1002, pr124, pr76, rat195, rd400	
HGAPP	AREA		d1291, d2103, d657, fl1577, gil262, pr1002, pr124, pr76, rat195, rd400
HGAPP	GATOUR	d1291, d2103, d657, fl1577, gil262, pr1002, pr124, pr76, rat195, rd400	
HGAPP	RLS		d1291, d2103, d657, fl1577, gil262, pr1002, pr124, pr76, rat195, rd400
HGAPP	NNGREEDY		d1291, d2103, d657, fl1577, gil262, pr1002, pr124, pr76, rat195, rd400
HGAPP	HGARANDOM	d657, gil262, pr124, pr76, rat195, rd400	fl1577
TA	HGATOUR	d1291, d2103, d657, gil262, pr1002, pr124, pr76, rat195, rd400	
TA	AREA		d1291, d2103, d657, fl1577, gil262, pr1002, pr124, pr76, rat195, rd400
TA	GATOUR		d1291, d2103, d657, fl1577, gil262, pr1002, pr124, pr76, rat195, rd400
TA	RLS		d1291, d2103, d657, fl1577, gil262, pr1002, pr124, pr76, rat195, rd400
TA	NNGREEDY		d1291, d2103, d657, fl1577, gil262, pr1002, pr124, pr76, rat195, rd400
TA	HGARANDOM	gil262, pr124, pr76, rat195	d1291, d2103, d657, fl1577, pr1002
HGATOUR	AREA		d1291, d2103, d657, fl1577, gil262, pr1002, pr124, pr76, rat195, rd400
HGATOUR	GATOUR		d1291, d2103, d657, fl1577, gil262, pr1002, pr124, pr76, rat195, rd400
HGATOUR	RLS		d1291, d2103, d657, fl1577, gil262, pr1002, pr124, pr76, rat195, rd400
HGATOUR	NNGREEDY		d1291, d2103, d657, fl1577, gil262, pr1002, pr124, pr76, rat195, rd400
HGATOUR	HGARANDOM	pr124	d1291, d2103, d657, fl1577, gil262, pr1002, rat195, rd400
AREA	GATOUR	d1291, d2103, d657, fl1577, gil262, pr1002, pr124, pr76, rat195, rd400	
AREA	RLS	pr76	d1291
AREA	NNGREEDY	d1291, d2103, fl1577, gil262, pr1002, pr124, rat195, rd400	d657
AREA	HGARANDOM	d1291, d2103, d657, fl1577, gil262, pr1002, pr124, pr76, rat195, rd400	
GATOUR	RLS		d1291, d2103, d657, fl1577, gil262, pr1002, pr124, pr76, rat195, rd400
GATOUR	NNGREEDY		d1291, d2103, d657, fl1577, gil262, pr1002, pr124, pr76, rat195, rd400
GATOUR	HGARANDOM	d657, gil262, pr124, pr76, rat195, rd400	d1291, d2103, fl1577, pr1002
RLS	NNGREEDY	d1291, d2103, fl1577, pr1002, pr124, rat195, rd400	d657
RLS	HGARANDOM	d1291, d2103, d657, fl1577, gil262, pr1002, pr124, pr76, rat195, rd400	
NNGREEDY	HGARANDOM	d1291, d2103, d657, fl1577, gil262, pr1002, pr124, pr76, rat195, rd400	

Table 5.14: Statistic evaluation: 3. column from left: instances where algorithm A performed significantly worse and 4. column from left: instances where algorithm B performed significantly worse

	HGAPP	TA	HGATOUR	AREA	GATOUR	RLS	NNGREEDY	HGARANDOM
HGAPP	-	0	0	10	0	10	10	4
TA	10	-	0	10	10	10	10	6
HGATOUR	10	10	-	10	10	10	10	9
AREA	0	0	0	-	0	6	1	0
GATOUR	10	0	0	10	-	10	10	4
RLS	0	0	0	4	0	-	1	0
NNGREEDY	0	0	0	9	0	9	-	0
HGARANDOM	6	4	1	10	6	10	10	-

Table 5.15: Statistic evaluation of all instances. Algorithms : row: performed on how many instances better than compared algorithm, column: performed on how many instances worse than compared algorithm. Instances where both algorithms performed equally often better than the other algorithm are not taken into account.

CHAPTER 6

Conclusions

Now we are going to recapitulate this master thesis and draw conclusions.

6.1 Summary

The TTP is a relatively new optimization problem which tries to model the requirements of solving real world problems by interconnecting different NP-hard combinatorial optimization problems with each other. The subproblem interconnection further increases complexity as it prevents to efficiently solve the whole problem by simply creating a solution for each subproblem and combining them.

Both subproblems the TSP and the KP are well known optimization problems where lots of research has been done in the last decades and there are solution approaches which provide very good solutions for these problems.

In this thesis we first had a look at existing solution attempts of the TSP, KP and the TTP. After that we gave an overview over genetic algorithms, local search and problem specific characteristics which could help us to find better solutions for the TTP. The main part of this thesis was the discussion and the implementation of the different operators to help the algorithm to find better solution candidates. For the tour creation we used a combination of the nearest neighbor heuristic, 2-Opt local search and greedy operators whereas for the packing plan creation we mainly used a greedy heuristic and a penalty function to respect the problem interconnection.

The algorithms were implemented in Java and for the problem instances the TTP benchmark set [1] was used.

The results showed that from all our different algorithm variants the hybrid genetic algorithm which used a tour crossover operator and derived the packing plan from that tour via a greedy operator and a penalty function performed best. This variant was also the most time consuming variant. As computation time was limited to 10 minutes performance got worse on solving bigger instances. The tour part therefore seems to have a very big influence on the total performance of the whole solution. Additionally we compared this variant with a pure genetic algorithm and could confirm the conclusion from the literature that pure genetic algorithm often have a worse performance than hybrid genetic algorithms. The *HgaPackingPlan* variant which did a crossover on packing plans and derived a tour from that performed much worse compared than the hybrid genetic algorithms using tour crossover. On the comparison with the results of the TTP benchmark set [1] we could achieve better results on small instances but as instances got bigger performance got worse due the lack of computation time and the increasing impreciseness of our penalty function.

6.2 Future Work

Since our hybrid genetic algorithm *HgaTour* was capable to deliver better results than existing algorithms of the literature on some instances our approach seems very promising. To further improve the performance and to be capable to efficiently solve big instances there are different ways which could be examined. Using algorithms which perform better on the tour creation process could lead to better results but will also lead to higher run times. Additional local search operators could be introduced which are used to further improve the packing plan after creation through the greedy operator and the penalty function. Another way to improve performance would be to use a more accurate penalty function.

Bibliography

[1] A Comprehensive Benchmark Set and Heuristics for the Traveling Thief Problem, http://cs.adelaide.edu.au/ optlog/research/ttp.php. Accessed: 2014-10-16.

[2] http://alda.iwr.uni-heidelberg.de/index.php/greedy-algorithmen_und_dynamische_programmierung#dynamische_programmierung. Accessed: 2014-11-13.

[3] http://man7.org/linux/man-pages/man4/random.4.html. Accessed: 2015-02-09.

[4] http://www.math.uwaterloo.ca/tsp/concorde/downloads/downloads.htm. Accessed: 2015-04-25.

[5] Tsplib95, http://www.iwr.uni-heidelberg.de/groups/comopt/software/tsplib95/. Accessed: 2014-10-16.

[6] D. Pisinger. http://www.diku.dk/ pisinger/codes.html. Accessed: 2014-10-16.

[7] Robert Sedgewick and Kevin Wayne . http://introcs.cs.princeton.edu/java/96optimization/knapsack.java.html. Accessed: 2014-11-13.

[8] Emile Aarts and Jan Karel Lenstra. *Local search in combinatorial optimization*. Princeton University Press, 2003.

[9] James Edward Baker. Reducing Bias and Inefficiency in the selection algorithm. *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, pages 14 –21, 1987.

[10] Mohammad Reza Bonyadi, Zbigniew Michalewiez, and Luigi Barone. The travelling thief problem: the first step in the transition from theoretical problems to realistic problems. *Evolutionary Computation (CEC), 2013 IEEE Congress on*, pages 1037–1044, June 2013.

[11] Jürgen Branke, Merve Orbay, and Ima Uyar. The Role of Representations in Dynamic Knapsack Problems. *Applications of Evolutionary Computing*, 3907:764–775, 2006.

[12] Sangit Chatterjee, Cecilia Carrera, and Lucy A. Lynch. Genetic algorithms and traveling salesman problems. *European Journal of Operational Research*, 93:490–510, 1996.

[13] P.C. Chu and J.E. Beasley. A Genetic Algorithm for the Multidimensional Knapsack Problem. *Journal of Heuristics*, 4:63–86, 1998.

[14] Bernd Freisleben and Peter Merz. A Genetic Local Search Algorithm for Solving Symmetric and Asymmetric Traveling Salesman Problems. *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on*, pages 616–621, May 1996.

[15] Arnaud Fréville. The multidimensional 0–1 knapsack problem: An overview. *European Journal of Operational Research*, 155(1):1 – 21, 2004.

[16] Jun Gu and X. Huang. Efficient local search with search space smoothing: a case study of the traveling salesman problem (tsp). *Systems, Man and Cybernetics, IEEE Transactions on*, 24(5):728–735, May 1994.

[17] Simon Hatthon. http://commons.wikimedia.org/wiki/File:Statistically_Uniform.png. Accessed: 2015-02-03.

[18] John H. Holland. *Adaption in Natural and Artificial Systems*. University of Michigan Press, 1975.

[19] Fei Jin, Rui Shan, Yan Zhang, and Huanpeng Wang. Hybrid genetic algorithm based on an effective local search technique. *Automatic Control and Artificial Intelligence (ACAI 2012), International Conference on*, pages 1146–1149, March 2012.

[20] D. S. Johnson, L. A. McGeoch, and E. E. Rothberg. Asymptotic Experimental Analysis for the Held-Karp Traveling Salesman Bound. *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, 7:341–350, 1996.

[21] David S. Johnson and Lyle A. McGeoch. The Traveling Salesman Problem: A Case Study in Local Optimization. 1995.

[22] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack Problems*. Springer, 2004.

[23] Donald E. Kirk. An Introduction to Dynamic Programming. *IEEE Transactions on Education*, E-10, 1967.

[24] Gilbert Laporte. The traveling salesman problem: An overview of exact and approximate algorithms. *European Journal of Operational Research*, 59(2):231 – 247, 1992.

[25] P. Larranaga, C.M.H. Kuijpers, R.H: Murga, I. Inza, and S. Dizdarevic. Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators. *Artificial Intelligence Review*, 13:129–170, 1999.

[26] P. Larranaga, C.M.H Kuijpers, R.H. Murga, I. Inza, and S.Dizdarevic. Genetic Algoirhtms for the Travelling Salesman Problem: A Review of Representations and Operators. *Artificial Intelligence Review*, 13:129–170, 1999.

[27] S. Martello, D. Pisinger, and P. Toth. Dynamic programming and strong bounds for the 0-1 knapsack problem. *Manage. Sci.*, 45:414–424, 1999.

[28] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Implementations*. John Wiley & Sons, 1990.

[29] David Pisinger. Where are the hard knapsack problems? *Computers & Operations Research*, 32(9):2271–2284, 2005.

[30] R0oland. http://en.wikipedia.org/wiki/Crossover. Accessed: 2015-02-03.

[31] Soraya Rana. The distributional Biases of Crossover Operators. *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 549–556, 1999.

[32] JiangFei Zhao and Fei Pang. Genetic algorithm based on Greedy strategy in the 0-1 Knapsack Problem. *Genetic and Evolutionary Computing, 2009. WGEC '09. 3rd International Conference on*, pages 105–107, Oct 2009.

APPENDIX A

Implementation Details

A.1 Deadlines

The deadlines have been chosen in relation to the available hardware and the instance sizes. If you want to reduce system resources or increase instance size you have to check if the chosen deadlines are still appropriate to the changed environment.

A.2 Datastructures

There are two datastructures which should be discussed as their usage on big instances leads to huge RAM consumption.

Dynamic Programming

As the for the dynamic programming operator intermediate results have to be stored, we use an integer array to store them. On a 64 bit machine a long integer needs 64 bit. The dimensions of the array are dependent on the knapsack capacity and the number of items. We will have a look at the ram usage for a few instances of the TTP benchmark set:

TTP instance	|nodes|	|items|	knapsack capacity	ram usage (GB)
eil51_n150_bounded-strongly-corr_01.ttp	51	150	12347	0.014
eil101_n300_uncorr-similar-weights_06.ttp	101	300	164370	0.36
rat195_n1940_uncorr-similar-weights_05.ttp	195	1940	885826	12.800
rd400_n1197_uncorr-similar-weights_05.ttp	400	1197	546570	4.991
pr1002_n10010_uncorr-similar-weights_06.ttp	1002	10010	5484500	409.03

Table A.1: Comparison of ram usage of dynamic programming

As we can see in table A.1 the ram consumption is huge although we only have a look at rather small instances. Therefore we will only make use of the greedy packing operator as dynamic programming operator, although it uses scaling up to factor of 100, consumes too much ram on big instances.

Adjacencymatrix

The TTP benchmark set is making use of the TSP instances from the TSPLIB [5] for the tour part [1]. In the TSPLIB there are coordinates in the form of (x, y) for every city. The distances get then calculated using different distance functions. For the TTP benchmark set the distance function CEIL_2D is used [1]. The distance d_{ij} for two cities i, j are calculated as follows [5]:

$$xd = x[i] - x[j]$$
$$yd = y[i] - y[j]$$
$$d_{ij} = ceil(sqrt(xd^2 + yd^2))$$

($sqrt$ is the square root function)

As the distances between different cities are used by many operators the performance of the TTP can be speed up by using an adjacency matrix which contains all computed distances between all cities. To store the distances a two dimensional double (64 bit) array is used. This can lead also to a very high RAM consumption on big instances:

TSP instance	|nodes|	ram usage (GB)
eil51	51	0.000019
eil101	101	0.000076
rat195	195	0.00028
rd400	400	0.0012
pr1002	1002	0.007
d2103	2103	0.033
d15112	15112	1.7
pla33810	33810	8.51
pla85900	85900	54.97

Table A.2: Comparison of ram usage of the adjacency matrix

As we can see in table A.2 also the adjacency matrix can consume huge amounts of RAM if used on big instances. As we are only looking at instances with up to 2000 nodes it is no problem to use it.

A.3 Random numbers

For the random number generation the standard java runtime generator is used *java.util.Random*. This a pseudo random number generator which is reseeded at program start with the special unix character file */dev/random*.

According to the linux man pages this file gives access to the kernel's random number generator which uses noise from the network card, sound card etc. to create pseudo random numbers. [3]

A.4 Usage

The application is delivered as *jar-file* so it includes all needed dependencies. It needs a *java6* runtime and *ant* installed. To build the program from the java sources just run:

```
ant clean
ant
```

Now it can be executed with the command:

```
java -jar ttp.jar <additional parameters>
```

The arguments (belonging to the parameters) are all mandatory, e.g., parameter e needs an algorithm name as argument.

Parameter	Is optional	argument	description
–help	yes	none	usage output
-e	no	algorithm	use given algorithm
-o	no	output directory	output directory of result
-t	no	deadline	choose deadline in ms
-i	no	instance file	TTP instance file to solve
-a	yes	population size	population size
-b	yes	survivor size	survivor size
-c	yes	mutation rate tour	mutation rate tour
-d	yes	mutation rate packing plan	mutation rate packing plan
-s	yes	start seed	start seed for number generator
-n	yes	none	no usage of the adjacency matrix
-p	yes	property file	property file for algorithm AREA and ARRLS

Table A.3: Parameters

Algorithm code (option -e)	Algorithm Variant
TA1	NN + 2-Opt + greedyPacking + StartNodeSearch
TA5	NN + greedyPacking
TA7	NN + 2-Opt + greedyPacking
HGATOUR1	hybrid genetic algorithm using tour crossover
GATOUR1	genetic algorithm using tour crossover
HGAPP1	hybrid genetic algorithm using packing plan crossover
HGARANDOM1	hybrid genetic algorithm using tour and packing plan crossover
AREA	replica algorithm of EA (TTP Benchmark set)
ARRLS	replica algorithm of RLS (TTP Benchmark set)

Table A.4: Parameters

If algorithm AREA or ARRLS is used you need an implementation of the Chained-Lin-Kernighan heuristic for the TSP (CLK) from [4] which is downloadable as zipped binary. Additonally you need to specify the path of CLK binary and a path of a directory where temporary files can be written in a property file.

property	description	example
tmpFilesDirectory	directory for temp files	tmpFilesDirectory=/home/user1/tmp
pathLinkernProgram	path of CLK binary	pathLinkernProgram=/home/user1/bin/linkern

Table A.5: Properties of property file