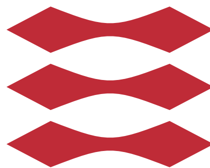
Comparison of Crossover and Diversity-Maintaining Operators in Randomized Search Heuristics

Jens Peter Träff



Kongens Lyngby 2014
DTU Compute-M.Sc.-2014-

Technical University of Denmark
DTU Compute
Matematiktorvet, building 303B,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3351
compute@compute.dtu.dk
www.compute.dtu.dk DTU Compute-M.Sc.-2014-

Summary

In this project a comparison of crossover and diversity-maintaining operators in randomized search heuristics is performed on the Traveling Salesman Problem (TSP).
Through a literature study, an initial overview of the field is provided. Basic original operators and more advanced operators are described. Theoretical results affecting the field are presented.
A framework for solving TSP solutions and testing TSP-solvers is implemented. Three operators are then selected, based on the literature study, implemented and thoroughly tested on problems from the TSP-Lib online library.
An analysis of the results and findings from these tests, combined with the literature study is used to summarize the current state of the art of the field, and present some future design principles of algorithms in the field.
It is found that the traditional pure genetic algorithms are not feasible in order to obtain good performance for solving TSPs. Hybrid algorithms focusing on combining the best properties of crossover with the properties of local search heuristics are the current state-of-the-art of 'genetic algorithms' and has shown the potential to improve solutions obtained by the state-of-the-art algorithms for solving TSPs.
Diversity maintaining mechanisms are found to be important, however the use of a simple injection strategy did not by itself provide an increase in fitness of the solutions obtained.
Four guidelines for future design of hybrid algorithms/crossover-operators in the field, is to have the crossover operator respect the principles of alleles transmission and respectfulnes, to use an efficient local search heuristic, to use some diversity maintaining scheme and to use an operator capable of exploration.

Resumé

I dette projekt vil en sammenligning af crossover og diversitets-vedligeholdende operatorer indenfor randomiserede søge-heuristikker blive udført på Traveling Salesman Problemet (TSP).
Gennem et litteratur studie vil et indledende overblik over feltet blive givet. Grundlæggende originale operatorer og mere advancerede operatorer vil blive beskrevet. Teoretiske resultater der på virker feltet vil ligeledes blive præsenteret.
Et framework til at løse TSP instanser og til at teste algoritmer der løser TSP-instanser implementeres. Derefter udvælges tre operatorer baseret på litteratur studiet, implementeret og grundigt testet på problemer fra TSP-lib biblioteket, der ligger online.
En analyse af de opnåede resultater og andre opdagelser fra de gennemførte tests, kombineret med litteratur studiet bruges til at opsummere det nuværende state-of-the-art indenfor feltet, og til at præsentere nogle fremtidige design principper for algoritmer der anvender crossover på TSP-problemer.
Det viste sig at de traditionelle 'rene' genetiske algoritmer ikke kan betale sig, hvis der skal opnås god performance når TSP-instanser skal løses.
Hybrid algoritmer, der fokuserer på at kombinere de bedste egenskaber fra crossover konceptet og de bedste egenskaber fra lokale søge-heuristikker, er det nuværende state-of-the-art indenfor 'genetiske algoritmer' anvendt på TSP, og har vist potentiale i forhold til at forbedre resultater opnået ved hjælp af generelle state-of-the-art teknikker.
Diversitets vedligeholdende mekanismer bliver fundet til at være vigtige, men det ses også at en simpel injektions strategi, i sig selv, ikke giver en forbedring af hvilken fitness-værdi der opnås.
4 anvisninger for fremtidigt design af hybrid algoritmer/crossover operatorer indenfor feltet er: at få selve crossover operatoren til at overholde principerne for

alleles transmission og respectfulness, at bruge en effektiv lokal søge-heuristik, at anvende en form for diversitets-vedligeholdende koncept og at bruge en operator der er i stand til at sørge for udforskning af løsningsrummet.

Preface

This thesis was prepared at DTU Compute, the Technical University of Denmark in partial fulfillment of the requirements for acquiring the M.Sc. degree in engineering. The project has been done from 9/9 2013 - 24/2 2014 and is worth 35 ECTS points.

This thesis provides an overview of crossover operators and some diversity maintaining mechanisms for the Travelleling Salesman Problem. One particular goal is to provide researchers and others, considering using/studying Genetic Algorithms for the Traveling Salesman Problem, with a thorough background of the field, such that they can choose the approach that best suits their purpose. In this thesis I conduct a literature study of existing crossover operators and of theoretical findings in the field. Selected operators are implemented and thoroughly tested in order to provide an analysis of the field and some suggestions for future design of operators.
The project was done under supervision from Professor Carsten Witt, Technical University of Denmark.

Lyngby, february 2014
Jens Peter Träff

Acknowledgements

I will like to thank my advisor Carsten Witt, who has kindly provided support on how to meet the requirements of DTU, introduced me to the world of evolutionary computing and asked great questions during the process of completing this project.

Contents

Summary i

Resumé iii

Preface v

Acknowledgements vii

1 Introduction 1

2 Known Crossover-operators and Diversity Mechanisms 5

2.0.1 Basic Requirements of Crossover Operators 5

2.1 Original Crossover Operators for TSP 6

2.1.1 Cycle Crossover 7

2.1.2 Partially Matched Crossover 8

2.1.3 Edge Crossover 10

2.1.4 Order Crossover 13

2.2 Theoretical Findings on the Subject of Crossovers on TSP 14

2.3 More complex operators 15

2.3.1 Order and Modified Order Crossover 16

2.3.2 Sequential Constructive Crossover 17

2.3.3 Partition Crossover 19

2.3.4 Generalized Partition Crossover - Hybrid Algorithm . . . 20

2.4 Diversity and the effect on the computations 24

2.5 Other important features of GAs on TSP 25

2.5.1 2-opt 25

2.5.2 Lin Kernighan Heuristic 26

2.5.3 Double Bridge Move 27

2.5.4 Diversity selection 27

2.6 Summary of literature study 28

3 Selection and Implementation of Representative Crossover Operators 31
3.1 Selecting Operators 31
3.2 The Overall Program Structure 33
3.3 Implementing Order Crossover 36
3.4 Implementing Sequential Constructive Crossover 36
3.5 Implementing Generalized Partition Crossover 38
3.5.1 The core GPX operator 38
3.6 Implementing auxillary algorithms: 42
3.6.1 Implementing 2-Opt 43
3.6.2 Lin-Kernighan 43
3.6.3 Implementing Diversity Selection 44
3.6.4 Implementing Double Bridge Move 45
3.7 Implementing Diversity measures 45
3.7.1 Implementing Random Solution 45
3.7.2 Implementing local-search improved solutions 45
3.8 The Graphical User Interface 46
3.8.1 The visual representation 47

4 Experimental results 49
4.1 Selection of Tests 49
4.2 Evaluation of Tests 51
4.3 Test of Order Crossover 51
4.3.1 Parameters for the test 51
4.3.2 The results 51
4.3.3 Test on OX with injection 53
4.4 Test of Sequential Constructive Crossover 55
4.4.1 Parameters for the testing 55
4.4.2 Testresults 55
4.4.3 Injection strategies on SCX 56
4.5 Test of Generalized Partition Crossover 57
4.5.1 The Testversion and parameters 57
4.5.2 The tests 57
4.6 Statistical Tests 58

5 Discussion 61
5.1 Comments on the results 61
5.1.1 The Order operator 61
5.1.2 OX with Injection 62
5.1.3 The SCX operator 63
5.1.4 SCX with injection 65
5.1.5 The GPX operator 66

5.2 Potential of genetic algorithms 70
5.3 Conclusion 72

Bibliography **73**

Chapter 1

Introduction

In this project the Traveling Salesman Person problem (TSP) will be considered. TSP is an old optimization problem dating back to 19th century, and has it roots from a handbook for german traveling salesmen: *the traveling salesman, how he must be and what he should do in order to get comissions and be sure of the happy success in his business, by an old commis-voyageur*[AS]. It was first studied as a mathmatical problems in the 1930'ies.

The problem originally consists of a merchant that wants to visit n cities to sell his goods, however he does not want to visit a city he has already visited once, he wants to start and end in the same city, and he wants to have the tour be as short as possible. Formally this can be described as follows:

The goal is, given a list of n cities and their internal distances, $c_{i,j}$ for all $1 \leq i, j \leq n$, to construct a tour visiting all n cities exactly once, starting and finishing in the same city, and minimizing the sum. Formally that is minimizing the value TotalCost, defined as follows:

$$TotalCost_{tour} = \sum_{i=1}^{n-1} c_{i,i+1} + c_{n,1}$$

This problem can be used to represent various problems in computer science, examples include: vehicle routing, scheduling and dna sequencing and appear as a subcomponent in slightly modified forms in other problems. There are multiple types of the problem. It can be symmetric, where the cost of going from city

i to city j is equal going from j to i, or asymetric where this is not necessarily the case. The distances can be Euclidean, that is the airline distance between the two cities, metric (satisfying the triangle inequality) or arbitrary.
In this project the focus is on euclidean, symmetric problem instances, as they are the most commonly appearing problems and hence the most studied.
The optimization problem has been shown to be computationally hard, belonging to the class of NP-hard problems (Richard M. Karp, 1972). This means that it is believed that no polynomial-time algorithm can solve the problem exactly and thus obtaining an exact solution to a large instance is usually considered infeasible. The combination of a commonly appearing problem, that at the same time is computationally hard has made TSP a subject of considerable research. Today it is often used as a 'benchmark problem', that is a problem where new algorithmic ideas can be tested in order to compare it against other algorithms.
There are a number of different approaches to solve TSPs, including exact solvers, algorithms based on search heuristics and approximation algorithms.
Exact solvers includes branch & bound and cutting planes techniques. They are characterized by using considerable time in order to achieve the optimal solution to a problem instance.
Approximation algorithms tries to approximate the optimal solution, but does so in reasonable time. An example is Christofides algorithm that guarentees a solution at most 50% excess, but finds it very quickly.
Search heuristics are algorithms that often gets very close to the optimal solution, in relatively short time, without having a formally proven guarentee for this. This includes Local Search algorithms and Evolutionary Algorithms. Heuristic search has historically been very effective on TSP problems.
I will in this project look at subset of evolutionary algorithms, genetic algorithms and their performance on TSP.

Evolutionary algorithms (EA) are a class of generic heuristic solvers, that mimics real life evolution theories. Evolutionary algorithms consists of a couple of concepts and operators that that can be adjusted depending on the problem instance:

- It runs for a number of iterations, called 'generations'
- It maintains a number of search points, called 'individuals', these are stored in a 'population'
- It has a fitness evaluation operator, that can assign 'fitness' value to the individuals
- It has a reproduction selection operator, that selects one or more individuals, 'parents' for 'reproduction'

- In the reproduction step an 'offspring' is created from the 'parent(s)'
- The offspring is then with a certain probability 'mutated', that is some part of it is randomly altered
- then 'survival selection' is used to select the individuals for next generations population. Those are chosen from among the current population and the generated offsprings.
- a 'stopping criteria' for when to stop the process. It can be number of generations, when no improvements is made etc.

A general EA will work as follows: first the population is filled with randomly generated individuals. In every generation individuals are selected to reproduce. The offspring(s) are exposed to mutation and the next population is filled by survivor selection. This process continues until the stopping criterion is met.
Genetic Algoritms (GA) are evolutionary algoritms extended with a 'crossover' operator. A crossover takes two or more individuals selected for mating, it then builds one or more offsprings using the parent solutions. They can be recombined in many different ways and there exists a bunch of different crossover operators. Some are generic, where others tries to make as informed choices as possible.
A common reasoning behind crossover operators is the building-block hypothesis [Gol89], [Hol75]. This hypothesis informally states that if one can create multiple solutions that has 'good' subcomponents, that is subcomponents that are have a high fitness value, or are as they should be in the globally optimal solution, crossover should be able to recombine these blocks in order to obtain new and better solutions.

Algorithm 1 Generic GA

```
Create an initial population of search points,p
while some stopping criteria is not met do
   Select two or more search points from p for mating
   Perform crossover to obtain new offsprings
   With a certain probability mutate the offsprings
   Select the appropriate individuals for survival into the next generation pop-
   ulation, p1 based on some parameters specified by the programmer
   Evaluate the stopping criteria
end while
```

Development and selection of appropriate crossover operators for different problems are a large topic within evolutionary computing.
Mutation operators are another frequent topic for research, the purpose of this

operator is normally to ensure that potential 'good' part solutions that are lost, can be recreated. It is also capable of generating new genetic material, that has not been seen before.
Survivor selection and selection for reproduction are other important concepts. When searching with a algorithm using a search heuristic, a key concept for achieving good performance, is balancing the 'exploration' of the search space and the 'exploitation' of good points in the search space. If there is too much focus on exploration, the algorithm will be very slow to converge on a good solution. If there is too much focus on exploitation, it might converge prematurely and be stuck in local minima, thus missing better points in the search space.
For GAs all its various operators can be used to emphasize both exploration and exploitation. In GA's the concept of 'diversity' in a population is used as a measure to how much emphasis is currently on exploitation and exploration. Diversity is a measure for how different the individual search points are from each other. Diversity and the preservation of it is thought to be important on multi-modal fitness landscapes (problems that have a lot of local minima).
GAs have been used on TSPs since the early eightees [GL85]
Another line of research seeks to combine multiple different search heuristics to achieve better performance, those algorithms are called 'Hybrid-algorithms'. An algorithm combining local search with a genetic algorithm is an example of a hybrid-algorithm.

In this project I will start by conducting a literature study over various crossover operators. The goal is to get an overview of the current state-of-the-art as well as an understanding on what makes a good crossover operator for TSP. I will futher look into some of the auxillary concepts that are important to make a genetic algorithm work.
I will then select 3 different crossover operators and the algorithms they are embedded in, for further study. I will try to apply some diversity-maintaining mechanisms to these algorithms in order to see if it will result in any improvement on the performance.
The 3 algorithms will be implemented, according the the introducing papers if possible, and tested on a number of TSP instances.
Statistical tests are then applied to compare the algorithms fitnesswise, in order to determine if any statistical inference can be made of the performance of the algorithms.
I will then use the results and the insight from the testing, to see if I can identify any common traits, concepts that are a strength in the search, and some general weaknesses/issues.
I will finish by comparing the tested algorithms to each other, and finally to other known solvers on tsp.

Chapter 2

Known Crossover-operators and Diversity Mechanisms

In this chapter I will present the results of a literature study over crossover operators for TSP. During the study many different ideas were encountered, since most of these where quite similar, I have chosen to describe some early original ideas and then what seems to be state of the art. For every operator I describe the idea, present the algorithm, show an example and provide a brief analysis.
I have split the operators into 'basic' and 'more complex' operators. The last category is made up of operators that either combine different ideas, build on the original concepts or are based on concepts that seems to be considered general guidelines in the field.
The literature study was primarily conducted by reading papers describing the operators and corresponding algorithms, survey papers comparing multiple algorithms, papers describing concepts and a slide-format survey of important concepts in crossover for TSP found in [Jak10]. The following papers have constituted to this section: [WSF89][Hol75][GL85][OSH87][Dav85][Ahm10][WHH09][WHH10][RSJJ94][H
[ACR03][RBSMBT][HWH12][FOSW09][CS96][WRE+98][Gol89][SS11][RBP05]

2.0.1 Basic Requirements of Crossover Operators

In traditional genetic algorithms the problem instance is usually encoded as a bitstring. The bitstring representation is very effective in order to represent a TSP problem instance, hence the early attention in the EA community was to find an effective way to represent the problem. The best, and the default, representation for tsp instances was quickly established to be the path-representation. In this, a tour is represented by a list of *city id's*, the *id*'s are listed in the order they are visited in the tour. An edge between the final element and the first element is assumed. Usually 0-based indices are used, and this will be used in this project.
Using this representation, a search with a genetic algorithm basically considers permutations to the initial list of cities.
One problem with this presentation is that it is not feasible to solely use the standard crossover ideas like uniform, 2-point crossover etc. This is due to the restriction that a city must only appear once in the list, as it otherwise would be visited twice.
Using uniform crossover, where for every position in the offspring there is 50% chance for each parent to provide the gene, for example, there is a large chance that the resulting offspring will be illegal, as at least one city would likely appear twice in the offspring.

2.1 Original Crossover Operators for TSP

The first attempts of using Crossover in relation to TSP happened in the early eightees. Originally the focus was on simple crossover schemes that solved the issues from 2.0.1 and thus created legal offsprings, as well as preserving the absolute positions of the cities in either parent.
Whitley et al. [WSF89] [Jak10], suggested that this approach was flawed, and the goal should instead be to preserve the *relative* order of cities, as the most important part was the edges. This reasoning has been largely prevailant since, examples include [WSF89],[Ahm10],[RSJJ94], [WHH09], [WHH10] and its dominance been established through multiple studies. Early implementations attempting to use this concept includes the Edge Recombination operator and the Order operator.
The majority of these early original concepts are, based on their stated results, not feasible compared to state-of-the art techniques, like algorithms based on the Lin-Kernighan search heuristic. For every algorithm, I will explain the workings, write up an abstract algorithmic description, provide an example, followed

by a brief analysis.

2.1.1 Cycle Crossover

Cycle Crossover (CX) is an older method devised in 1987 by olivier et. al. [OSH87]. It is based on the assumption that it is important to preserve the absolute position of cities. Thus a city preferably inherits its position from either parent.
Informally the goal of the algorithm is: 'By looking at cycles, the goal is to get as many cities as possible placed from one parent, while the rest is taken from the second parent'.
In CX a cycle is determined as follows:
Let the element in the first parent at index 0 be c, then find the location in parent 1, of the element at index 0 in parent2. Repeat for the new index.
This is repeated until the considered element from parent 2 is the same as c.
The indices considered in this process constitutes a cycle.

Algorithm 2 CX-crossover

p1 denotes the first parent,
p2 denotes the second parent,
off denotes offspring.
Let i=0;
Let start = p1[i]
//STEP 1 find a 'cycle'
while true **do**
  Let a = p2[i]
  off[i] = p1[i]
  if a == start **then**
    stop cycle 1
  end if
  let i be the location of a in *p1*
end while
STEP2: Fill remaining positions in off with cities from *p2*
STEP3: Repeat the algorithm but swap *p1* and *p2*, to a second offspring.

Example:
Assume we have selected the following two individuals as parents:

p1: | 1 | 2 | 3 | 4 | 5 | 6 | 7 | & *p2:* | 7 | 5 | 1 | 3 | 2 | 6 | 4 |

let off1: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | & *let off2* | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

After step 1 of the algorithm the first offspring looks as follows: | 1 | 0 | 3 | 4 | 0 | 0 | 7 |
After step 2 of the algorithm the first offspring looks as follows: | 1 | 5 | 3 | 4 | 2 | 6 | 7 |

Using the same approach the second offspring is produced.

After step 1: | 7 | 0 | 1 | 3 | 0 | 6 | 0 |
After step 2: | 7 | 2 | 1 | 3 | 5 | 6 | 4 |

Analysis:
This concept emphasizes the cities, where in fact, a good solution is likely to have good connections between cities, that is, the edges between the cities are favorable. In this case we could in a single crossover replace all the old edges with newer ones. It has since been argued that the key is to preserve the relative position of cities and the edges [WSF89].
In fact multiple studies have shown that CX is one of the worst operators considered for TSP, as for example in a thorough study performated at Carnegie Mellon in 1996, [CS96]. In the study, CX gets the following comment:
" ..position based operator working on sequential type problem.."
Thus emphasizing the mismatch between the problem types for which CX is suited for and the actual problemtype of TSP.

2.1.2 Partially Matched Crossover

Partially Matched Crossover (PMX) is one of the first operators suggested, it was presented in 1985 by Goldberg in [GL85]. It aims to preserve the relative ordering of cities.
The main idea is to exchange a varible area between two points (a segment of the tour), and then fill the remaining spots in the offspring, directly with cities from the opposite parent. The remaining cities are placed using a mapping between the exchanged genes, and preserving the order from the oppposite parent.
In 3 I have presented an informal version of the PMX algorithm. *p1* and *p2* are the two individuals chosen for mating.

Example:
Assume we have selected the following two individuals as parents:

p1: | 2 | 5 | 4 | 7 | 8 | 6 | 1 | 3 | & *p2:* | 1 | 2 | 3 | 8 | 4 | 7 | 6 | 5 |

Algorithm 3 PMX-crossover

---

p1 denotes the first parent, *p2* denotes the second parent,
off1 denotes the first offspring. *off2* denotes the second offspring.
//step 1
Choose two points in the string according to the chosen strategy
let *left* = first chosen point
let *right* = second chosen point
//Step 2
for m between left and right **do**
  let *off1*$[m]$ = *p2*$[m]$
  let *off2*$[m]$ = *p1*$[m]$
end for
//STEP 3 fill the remaining spots in the offsprings with cities from the corresponding parent, while legal
for remaining positions, m, in the offsprings **do**
  if setting *off1*$[m]$ = *p1*$[m]$ is legal **then**
    off1$[m]$ = p1$[m]$
  end if
  if setting *off2*$[m]$ = *p2*$[m]$ is legal **then**
    off2$[m]$ = *p2*$[m]$
  end if
end for
//STEP 4 construct a mapping
if any position in the offsprings are unfilled **then**
  construct a mapping between elements in *p1* and *p2* between *left* and *right*
end if
//STEP 5 fill remaining positions.
for every unfilled position in *off1*, i **do**
  find the element *p1*$[i]$
  find the element, a, that *p1*$[i]$ maps to, in the mapping from step 4.
  while a has occurred elsewhere in *off1* **do**
    let a = the element a currently maps to
  end while
  let *off1*$[i]$ = a
end for
repeat STEP 5 for *off2*, reversing the roles of *p1* and *p2*.

---

let off1: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | *& let off2* | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

In step 1, left is chosen to be to 2 and right is is 5.

After step 2 of the algorithm the first offspring looks as follows: | 0 | 0 | 3 | 8 | 4 | 7 | 0 | 0 |
After step 2 of the algorithm the second offspring looks as follows: | 0 | 0 | 4 | 7 | 8 | 6 | 0 | 0 |

After step 3 of the algorithm the first offspring looks as follows: | 2 | 5 | 3 | 8 | 4 | 7 | 1 | 0 |
After step 3 of the algorithm the second offspring looks as follows: | 1 | 2 | 4 | 7 | 8 | 6 | 0 | 5 |

In step 4 this mapping is constructed: $4 \leftrightarrow 3$, $7 \leftrightarrow 8$, $4 \leftrightarrow 8$, $6 \leftrightarrow 7$.

After the final step of the algorithm the first offspring looks as follows: | 2 | 5 | 3 | 8 | 4 | 7 | 1 | 6 |
After the final step of algorithm the second offspring looks as follows: | 1 | 2 | 4 | 7 | 8 | 6 | 3 | 5 |

Analysis:
PMX seeks to maintain the relative ordering, however in some cases it is only the cities between the cut points for which this holds. If we consider the example above, *off2* introduces new edges $6 \leftrightarrow 3$ and $3 \leftrightarrow 5$. *off1* introduces new edges $1 \leftrightarrow 6$ and $6 \leftrightarrow 2$, thus changing the relative ordering of the cities.
It maintains good edges between the cutting points, in most cases. However there is a large chance that new edges are introduced outside of the cutting points.
Studies over various operators have shown that there are more efficient operators than PMX, and more recent operators outperform it significantly [Jak10]. This is likely due that a lot of new edges can get introduced outside of the cutting points. In [Jak10] and insinuated in [RSJJ94] it is stated that good crossover operators preserve the good edges, and does not introduce new edges.

2.1.3 Edge Crossover

Edge Crossover (EX), originally suggested in 1989 by Whitley [WSF89], was the first to explicitly focus on the edges. The operator was designed such, that the number of new edges introduced in every generation, should be as small as possible. This approach was considered a breakthrough for crossover operators on TSP [Jak10] and used as a basis for new operators afterwards.
The algorithm can briefly be described as such:
EX starts by constructing an *Edge-map* for every city. An *Edge-map* for a city, c, is a list of all cities c can reach through one edge in both parents. If c has a

transition to another city j in both parents, this transition is only represented once in c's edgemap (and conversely in j's).
Iteratively the city with the lowest degree (where the size of the *edge-map* is smallest), *c_min*, is chosen. *c_min* is then placed in the offspring and removed from every edgemap. All cities in *c_min*'s *edgemap* is added to a queue of unprocessed nodes.
The next city is now chosen from this queue. If the queue is empty, the next city is chosen at random among cities not yet considered.

Algorithm 4 EX-crossover

```
p1 denotes the first parent,
p2 denotes the second parent,
let off denote the offspring.
Construct an edgemap for all cities
Initialise a list, unvisited, consisting of all unvisited cities, intitially this list
contains all cities.
initialise a list of cities, called entry-list
choose a city, i, at random
while unvisited is not empty do
  i is added to the back of off
  i is removed from all edgemaps
  i is removed from the list unvisited
  add the cities in i's edgemap, and not in unvisited to entry-list
  if entry-list is not empty then
    choose the city from entry list with lowest degree
  else if if unvisited is not empty then
    choose a city at random from unvisited
  else
    return the obtained offspring
  end if
end while
```

Example:
Assume we have selected the following two individuals as parents:

p1: | 1 | 2 | 3 | 4 | 5 | 6 | & *p2:* | 2 | 4 | 3 | 1 | 5 | 6 |

let off: | 0 | 0 | 0 | 0 | 0 | 0 |

First the edgemaps are constructed:
1: 2,6,3,5
2: 1,3,6,4

3: 2,4,1
4: 3,5,2
5: 4,6,1
6: 5,1,2

In the initial selection I randomly select city 2:
STEP 3: off: | 0 | 0 | 0 | 0 | 0 | 2 |
STEP 4: 2 is then removed from all edgemaps:
1: 6,3,5
2: 1,3,6,4
3: 4,1
4: 3,5
5: 4,6,1
6: 5,1
STEP 5 and 6: The list unvisited and entrylist:
unvisited: 1,3,4,5,6
entrylist: 1,3,6,4
STEP 7: 4 is chosen as the next city as it has the smallest degree (tied with 3 and 6).
The process is now repeated from STEP 3 until the offspring is complete. One potential offspring completed in this way could be:

2	4	3	1	6	5

Analysis:
EX was one of the first to focus explicitly on the edges. The focus on preserving edges resulted in very few new edges being created in every crossover step.
There are however some drawbacks with this method. One such drawback is that it is not very efficient in 'identifying' and combining good building blocks. Mostly edges are preserved, but ER often has to choose between edges from the two parents.
In every selection step, if there are multiple options, the selection criteria is the size of the *edgemaps*. These *edgemaps* can be of size 1,2,3 or 4, and as seen in my example, the options will often have the same size of *edgemap*, which make the choice stochastic.
In my example, when choosing the next city after considering city 3, the choice was between cities 1,5 and 6. If city 6 had been chosen a new edge would have been introduced ,6 $\leftrightarrow$ 3, but the current selection used an existing edge 3 $\leftrightarrow$ 1. This illustrate a potential problem in this operator, as the former is discouraged behaviour and the latter is encouraged in the theory behind the design of EX.
The core ideas behind EX influenced later design of algorithms in the field.
The concept of *edgemaps* have been used in later algorithms, notably in the Partition crossover and Generalized Partition Crossover presented in [WHH09][WHH10].

These algorithms will be described later in this project.
The focus on preserving good edges, further supported by [RSJJ94], were used in a multitude of later algorithms, as described in [Jak10].
Today EX is by itself no longer competetive with newer algorithms in the field.

2.1.4 Order Crossover

Order Crossover (OX) was one of the most succesful early crossover operators, devised by Davis in 1985, [Dav85]). OX tries to retain the relative order of cities, thus minimizing the number of new edges that needs to be introduced.
As in PMX two cut points are chosen, the cities between those points are placed directly in the respective offsprings. The remaining places from the second cut point and on, are filled, with cities from the opposite parent (disregarding cities already placed in the offspring). Thus it chooses a subsequence from one parent, and tries to keep the relative ordering from the other parent.

Algorithm 5 OX-crossover

p1 denotes the first parent, *p2* denotes the second parent
off1 denotes the first offspring, *off2* denotes the first offspring
choose two cut points, *left* and *right*
Let start = p1[i]
for m between *left* and *right* **do**
  let $off1[m] = p1[m]$
  let $off2[m] = p2[m]$
end for
//STEP 3
let a the first element to the right of *right* in *p2*, that has not yet been placed in *off1*
$off1[right+1]$ = a
for the remaining positions, m **do**
  let a = the next, not yet occured in off1, city to the right of a in *p2*
  off1[m] = a
end for
STEP4: Repeat STEP 3 but swap *p1* and *p2*, to produce *off2*

Example:
Assume we have selected the following two individuals as parents:

p1: | 1 | 3 | 4 | 5 | 8 | 7 | 2 | 6 | & *p2:* | 2 | 4 | 1 | 8 | 7 | 6 | 3 | 5 |

let off1: | 0 | 0 | 0 | 0 | 0 | 0 | 0 | & *let off2* | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

In step 1, left is chosen to be to 2 and right is is 5.

Filling the sequence between cutpoints for the first offspring: | 0 | 0 | 4 | 5 | 8 | 7 | 0 | 0 |
Filling the sequence between cutpoints for the second offspring: | 0 | 0 | 1 | 8 | 7 | 6 | 0 | 0 |

The remaining positions are filled according to step 3
First offspring: | 1 | 6 | 4 | 5 | 8 | 7 | 3 | 2 |
Second offspring: | 4 | 5 | 1 | 8 | 7 | 6 | 2 | 3 |

Analysis:
OX emphasizes maintaining the relative ordering of citites. The chosen subsequence will remain intact, however when filling out the remainder, new edges will most likely be introduced. Consider the follwing:
If a city, c is in the chosen sequence in the first parent, but outside the sequence in the other parent, at least one new edge (likely two) will need to be introduced. This happens as when the opposite parent attempts to fill the offspring outside of the cutpoints, it cannot add c and thus the relative order will be broken at least once.
OX has been shown to be one of the more efficient operators of the early ones. Studies in japan and at Carnegie Mellon [Jak10] [CS96], and later papers on new operators [RBP05], shows that it is superior to other operators presented at the time. It is possible that, this is due to the maintenance of a subsequence, and the (potentially) relatively few new edges introduced outside of the choice points, thus it is closer to the guidelines in [RSJJ94] than many contemporaries. Some work has been done trying to optimize the performance of OX, by finding the best way to choose the cut-points. Several attemps have been made, and results indicate that it indeed improves performance, at least on some instances. [RBP05]
Combining results from recent articles on OX indicates that it is inferior to some recently developed operators [Ahm10],[RBP05].

2.2 Theoretical Findings on the Subject of Crossovers on TSP

Some theoretical work has been done both specifically on this subject, and in the field in general, that applies to this particular subject.
In 1994 Radcliffe and Surry presented a paper in [RSJJ94]. In the paper they

argued that, to achieve the best succes with crossover on this particular problem, the two following principles should be observed:

- The crossover should transmit alleles
- The offsprings should be respectful

Alleles transmission means that all edges in the offspring should come from either of the parents. Thus no new edges are introduced.
Offspring respectfulness means that all edges that are in both parents should also be in the offspring.
Watson argued [WRE+98] that crossover should introduce new edges in order to not be stuck in a local minimum. The problem with this argument is, that this feature can be achieved through other operators used in a genetic algorithm, like the mutation operator. And as such it does not have to be a principle in the design of the crossover operator itself.
A study from Carnegie Mellon in 1996, [CS96] argued amongst others, for the principles og the building block hypothesis, and that by recombining above average sub-subcomponents through crossover crossover could speed up local search heuristic. Suggesting a hybrid approach between genetic algorithms and local search heuristics.
Other suggestion of the Radcliffe article, included combinining crossover operators with local search heuristics, in order to exploit the building blocks hypothesis.
Recently Witt et al. [FOSW09] Argued that diversity is important for EAs working on multimodal fitness landscapes, TSP-problems have such a fitness landscape. Although GAs are different than EAs, these findings suggests that diversity could also be important for genetic algorithms. These thoughts are also expressed by Whitley et al. in [WHH10].

2.3 More complex operators

In the last decade there has been a shift in the design of crossover operators. Earlier the operators were completely stochastic, as usually seen in standard GAs. Now operators, that try to make 'intelligent' choices during crossover, instead of relying only on stochastic choices, are being devised. These operators are usually developed using three different backgrounds.
Many new operators introduce a simple improvement on earlier operators. Improvements could be, adding an informed choice instead of a stochastic choice somewhere in the algorithm, a minor tweak to the processing or the addition of

an extra component. Examples of this includes [RBP05],[SS11].
Some researchers have tried to combine different fields in a single operator. Incorporating greedy or logical reasoning to determine which genes the different parents should contribute to the offspring. Examples include [Ahm10] [RBSMBT].
Finally operators have been devised based on theoretical knowledge and insight of the problems obtained through earlier studies, as briefly described in 2.2. Studies like [RSJJ94] and [CS96] and operators like [WSF89] have suggested a basis for the design of new operators. Operators following these suggestions include Partition Crossover [WHH09] and Generalized Partition Crossover [WHH10].
The standard GAs have been largely unable to compete with local search heuristics such as the Lin Kernighan search. This has prompted some researchers to seek to combine local search heuristics with crossover. They seek to achieve the fast creation of good search points (hopefully containing good building blocks) from local search, and the ability of crossover to combine building blocks. Generalized Partition Crossover is the best example of this, that I could find.

I have selected three operators for closer analysis, which will be presented in this section.

- Extended Order Crossover, that add minor improvements to an existing operator.
- Sequential Constructive Operator, that try to make intelligent choices when constructing the offspring.
- Partition Crossover and Generalized Partition Crossover (which uses partition crossover), they are based upon research and the results of studies like [RSJJ94].

2.3.1 Order and Modified Order Crossover

As OX was one of the initial operators with most succes, numerous attemps to build on it, has been made. Examples of this are the Modified Order Crossover (MOC) [RBP05] and the Modified Order Crossover (MOX) [SS11]. Both of these examples attempts to find a smarter way to choose the subsequence that directly carries over.
The MOC operator tries to limit the size of the subsequence, instead of choosing the length of the string to be random, it is fixed beforehand. By experiments the authors of [RBP05] found that a subsequence length of $l = max2, \alpha$ for

$n/9 \leq \alpha \leq n/7$ where n refers to the number of cities in the problem.
The MOX operator tries to optimize the manner of how the selection points are chosen. The belief is that by adding an element of greedyness, it might be possible to get more 'good' edges in every generation. Specifically it aims to include the minimal cost edge.

Algorithm
The algorithm is exactly as standard OX, but for MOX, when choosing the first point of the subseqence (the first selection point), all edges are scanned and the minimum cost edge is included. Thus the selection point will be before the first city included in the minimum cost edge. The second point is still chosen randomly. For MOC the first point is chosen randomly and the second point is a distance of l to the right.

Analysis:
Both examples constitues minor optimizations. MOX prefers exploitation over exploration, and thus seems to have a better chance of preserving good edges. However since only the best edge is guarenteed to be in the sequence, and the problem is multi-modal, the majority of good edges, could still be outside the sequence.
The computation time is slower, due to the need to find the minimum element which takes at least O(n) extra time. No usable results for MOX were reported.
MOC is primarily used to decrease the computation time of standard OX, without decreasing the quality of the obtained solution. According to [RBP05] the time is lowered, but the results obtained by the MOC is not compared to that of the standard OX. However the results they report, does not seem to indicate that MOC is a major upgrade or downgrade.
In general the extension operators I have found suggests that minor modifications can improve slightly on the original operator, but the capacity of the new algorithm will still be close to the original.

2.3.2 Sequential Constructive Crossover

The SCX operator was suggested by Zakir Ahmed in [Ahm10] in 2010. This operator largely tries to preserve good edges, but adds an element of greediness, in the hope of creating new good edges. It builds the offspring sequentially from the two parents.
It starts from the first element in one of the parents, then in every step it

considers the next unvisited city after this city in both parents. If there is none (it considers the parent tours as paths), then the first unvisited city in the set of cities, sorted by index, is chosen. The two chosen cities are then compared by looking at the edge cost of getting there from the current city. The one with the smallest cost is chosen as the next city in the offspring. This process continues until the offspring is fully constructed.

example

To illustrate this operator, consider this cost matrix:

Costmatrix	1	2	3	4	5
1	0	8	7	4	8
2	8	0	6	5	7
3	7	6	0	9	10
4	4	5	9	0	6
5	8	7	10	6	0

Assume we have selected the following two individuals as parents:

p1: | 2 | 3 | 1 | 5 | 4 | & *p2:* | 1 | 2 | 4 | 5 | 3 |

let offspring: | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

In step 1, 2 is placed and we consider the successor to 2:
There are two options: 3 and 4, since ($2 \leftrightarrow 4$) *is the cheapest edge we choose it*
The offspring is then: | 2 | 4 | 0 | 0 | 0 |

When selecting the next city, 4 has no successor in p1, I then select the minimum index city that has not yet been placed to represent p1. The options are then: 1 and 5, and since ($1 \leftrightarrow 4$) *is the cheapest edge we choose it*
The offspring is then: | 2 | 4 | 1 | 0 | 0 |

This is continued until all places in the offspring is filled.
the final offspring is: | 2 | 4 | 1 | 5 | 3 |

Analysis:

SCX tries to get the immediate best city at every step, while still preserving order if possible. This combination of greediness and preservation appearently provides better results, at least than the operator EX [Ahm10]. It is claimed in the same study that SCX ensures that the offspring inherits good characteristics from the parents. Meaning that the edges, found to be good at the time, will mostly be preserved.

In the above example, one globally good edge is created, ($1 \leftrightarrow 4$), and two glob-

ally good edges are destroyed, $(3 \leftrightarrow 1)$, $(4 \leftrightarrow 5)$. The new offspring has a higher total fitness than the second parent, and will thus be selected in a deterministic crowding selection, that SCX in [Ahm10] uses. However the offspring and the first parent combined contains all 3 globally good edges, thus the total number of good edges in the population has increased by one. In this case the end-result was positive (an increase in total number of good edges), but the example indicates, that there is a potential of destroying good edges.
In the example presented in the article, 71% of the edges in the offspring comes from either parents [Ahm10]. Thus SCX is not in strict accordance with the guidance of respectfulness and alleless transmission laid down in [RSJJ94], insinuated in [CS96] and introduced in 2.2. This might lead to an expectation that this operator will not work as well as operators fully adhering to the mentioned principles.
Another issue with SCX is that it seems to be trying to perform two roles simultaneously (crossover and mutation). In one crossover step it develops new edges (ca. 28%), these edges are constructed in either greedy or random fashion. There is an issue that the greed could break up good subcomponents, which especially on a multimodal fitness landscape as TSP, could negatively impair the abiliy to use two local minimas to obtain a new and better local minima. However it could also lead to faster convergence times, as new good edges are created. Both concepts were illustrated in the above example.

2.3.3 Partition Crossover

Partition Crossover (PX) is presented in [WHH09] which won the best paper award at the '09 Genetic and Evolutionary Computation Conference, which is the most influential conferences in the evolutionary computing community.
In [WHH09] PX is presented based on earlier knowledge of TSP and GAs. It presents the algorithm but does not include a finished set up for testing. A further improved test version was introduced a year later in [WHH10]. The core concept in PX is a partitioning of the union graph of the two parent tours. A partitioning is in this case a seperation of the graph into multiple disconnected parts called partition components. In PX the goal is to find a single partitioning of cost 2. The cost refers to how many edges that has to be cut in order to construct the partitioning. The goal of a partitioning in PX is thus to split the graph into two partition components by only removing two edges (if there is a common edge, it counts as one).
The Partition operator works by constructing the union of the two parent tours, the *union-graph*. It then searches for a partition of cost 2. If it finds such a partition the graph is split into the two partition components. Inside each partition it chooses to use the edges of only one of the parents. The offspring is

then constructed using the two relevant subtours. If no partitioning is found, it is not feasible to use PX and the parents are returned as offsprings.
The described partitioning problem in PX, is a special instance of the general partitioning problem, which is believed to be NP-hard[WHH09]. In order to construct the partitioning efficiently the authors use the following method to construct the partitioning:

- First the *edgemap* for all cities is constructed. Marking all common edges along the way.

- Then all nodes that have degree 3 or 4 are found (nodes that have 0 or 1 common edge in its edgemap)
- Starting from the first node with degree 3, c, all nodes connected to c through only nodes of degree 3 or 4 are labelled to the first partition *p1*.
- If there are any nodes of degree 3 or 4 not labelled to the first partition, and there are exactly two nodes in this component that has connections to nodes not in the partition. crossover is feasible and the two components are *p1* and the remaining nodes in the other partition.
- otherwise return fail

A visual representation of a union graph, and potential partition is shown in 2.1.
It can be seen that if the common edges are removed from the union-graph, potential partition components will be seperated. This insight is the reason for only considering nodes of degree 3 and 4 in their algorithm.
The resulting offspring is respectful as all edges present in both parents are in the offspring. It has alleles transmission as all edges comes from one of the parents. It thus adheres fully to the principles from [Jak10] and [RSJJ94].

Example and discussion will be done in next subsection under Generalized Partition Crossover.

2.3.4 Generalized Partition Crossover - Hybrid Algorithm

Generalized Partition Crossover (GPX) is presented in [WHH10] and is an enhanced version of the partition crossover (PX) suggested by Whitley et. al in

[WHH09]. This version optimizes a potential weak spot in PX. GPX follows the guidelines established by [Jak10],[RSJJ94] [CS96], offsprings are respectful and GPX is capable of 'tunneling' directly to new local optima. Tunneling means that it is capable of taking two local minima and directly construct a new local minimum.
This operator has been used in conjunction with various local search operators, in hybrid algorithms.
The general idea in GPX, as in PX, is to partition the problem into smaller parts (by parting through common edges between parents) and then try to locally optimize each part as they are independent of other clusters. As opposed to PX, GPX will consider *all* potential partitions (of cost 2), choosing the most promising parent in each partition component.
GPX works (as PX) by taking the union of the two parent tours, then searching for all partitions of cost 2. Inside each of the k partition components, it then makes a greedy choice of either following the edges from one parent or the other. The main difference from PX is, that it uses all possible partitions, to construct as many partition components as possible. If a single partitioning cannot be found, it reports that crossover is unfeasible.
The actual hybrid algorithm combines the crossover operator with Lin-Kernighan search, the scheme presented in [WHH10] is presented in 6, the additional algorithms used in this version will be described later in this chapter.

Algorithm 6 GPX-crossover-hybrid algorithm

```
initialise a population p of size m
apply lin-kernighan search to all m tours and evaluate
while stopping criteria is not met do
    create a temporary population p_temp
    attempt to recombine the best tour in p with the remaining m-1 tours in p

    if crossover is unfeasible with a tour i then
        mutate i with a double bridge move, and place it in p_temp
    end if
    place the best solution among offspring and the previous best tour in p_temp
    from the set of offspring, select tours to fill p_temp using diversity selection

    apply lin-kernighan search to all members of p_temp
    set p to be p_temp
    evaluate stopping criteria
end while
```

Example of the GPX operator itself:
Assume the following two parents have been selected for mating.

p1:	2	1	9	8	4	6	10	3	5	7
p2:	2	8	9	1	4	6	5	3	10	7

The optimal solution is:

opt:	2	1	9	8	4	6	10	3	5	7

The following holds for the edges:
$8 \leftrightarrow 4$ is smaller than $8 \leftrightarrow 3$,
$4 \leftrightarrow 6$ is smaller than $4 \leftrightarrow 3$,
$2 \leftrightarrow 1$ is smaller than $2 \leftrightarrow 8$,
$8 \leftrightarrow 4$ is smaller than $1 \leftrightarrow 4$,
$6 \leftrightarrow 5$ is smaller than $6 \leftrightarrow 10$,
$6 \leftrightarrow 10$ is smaller than $6 \leftrightarrow 3$,
$10 \leftrightarrow 7$ is smaller than $5 \leftrightarrow 7$,
Using GPX, the union graph is first constructed shown in figure 2.1. A potential partition is then found. The edges that needs to be cut in order to partition this graph, are shown by the large dash through them. The pathlength of each parent inside each component is compared in order to pick the best parent inside each component.
In the final result p1 will provide the edges inside the first partition component, and p2 will provide the edges in the second partition component, resulting in the following offspring:

off:	2	1	9	8	4	6	10	3	5	7

Which is actually the optimal solution.

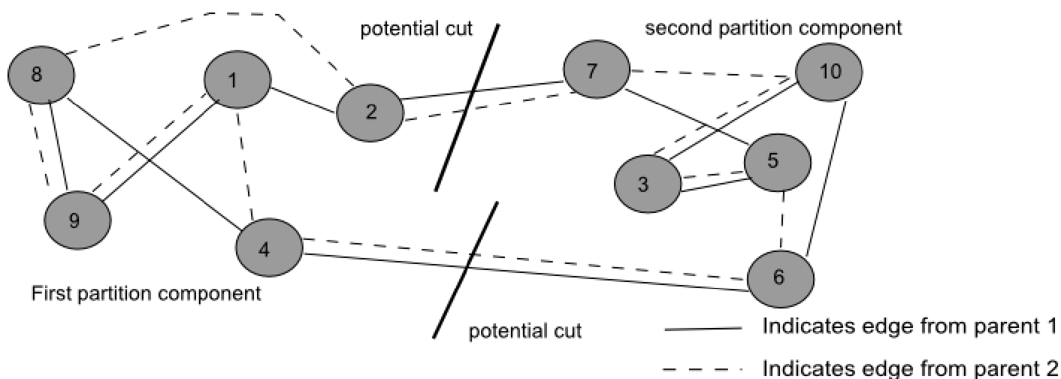


Figure 2.1: a simple example of the GPX operator

Analysis:
GPX is thoroughly analysed by Whitley in the introducing paper [WHH10]. He argues that since all edges come from either of the parents it transmits alleles. Since every common egde will be used by the offspring, the offspring is respectful. He proves that GPX considers up to $2t$ potential offsprings, where t is the population size.
GPX is very good at using local minima to produce new local or global minima. This is due that it makes greedy choices in every subcomponent, but since subcomponents are usually locally optimized in either of the parents, they will turn into excellent building blocks and thus produce new quality solutions.
One drawback is that when recombining the best solution with the remaining solutions, if some of the partition components contain a large number of edges that are uncommon, GPX will have a difficult time of finding the global optimum.
Whitley et al. looked closer at the generated solutions and made some interesting observations:

a: the largest partition component will have the largest amount of uncommon edges.

b: the remaining components have very few uncommon edges.

c: In his experiments, after 5 generations all edges found in the global optimum is present in the population (at least when the size of population is 10)

c suggests that after 5 generations we could stop looking for new edges, and concentrate on recombining the edges already contained. Potentially using some other type of algorithm.
a and b however suggests that to find the global optimum it might be necessary to somehow look individually at the largest partition component. It would suggest that introducing some diversity mechanism occasionally might help to avoid this pitfall. One example could be to pick a random solution, and recombine it with the rest. This might lead to a better edge selection inside the largest partition component.
The hybrid algorithm uses diversity selection as diversity mechanism to retain some individuals that might have the globally optimal edges inside some partition components, even though the total fitness of the individual can be worse than other solutions.

GPX combined with LK-search has produced results comparable and in many cases better than of the , at that time, state of the art algorithms: chained LK-search. Furthermore the three observations by the author suggest that combining this hybrid algorithm with a deterministic local search on the smaller instance of the largest partition component might yield even better results. Other ways of taking advantage of these ideas should be investigated [WHH10]. The author already has started to do this, examplified by the work done in [HWH12]

2.4 Diversity and the effect on the computations

In the field of evolutionary computing, there is always a balancing act of exploitation and exploration. Too much emphasis on exploitation leads to the algorithm being stuck in a local minimum/optimum, while too much experimentation results in slower convergence times, and hence slow running times. One of the primary determinants for performance [FOSW09],[WHH10], especially on multi-modal fitness landscapes is the population diversity, and how it is allowed to affect the computation.
In a study by Witt et al. [FOSW09] it is shown that population diversity is important for EAs on multimodal fitness landscapes. Indicating it could be as well for GAs. In recent studies by Whitley et al.[WHH10], the power of a population was shown. It was demonstrated that in a population of size 10, all edges that also were in the global optimum, were present (after 5 generations of GPX).
These findings suggest that keeping a diverse population will enhance the performance of some GAs in some settings.

The primary advantage of maintaining diversity on problems like TSP, is that more edges can be kept for consideration throughout the computation. This reduces the chance of losing good genetic material, that can then later be used for escaping a local optimum, and potentially reach the global optimum.
The main advantage of small diversity is a quicker convergence time. Other advantages can exists but it depends on the design of the algorithm used and the nature of the problem.

I will in this project, test some simple methods for preserving diversity known from other problems, to see if they can help the algorithm avoid being stuck in local minima. The injection strategy will be primarily be used.
The idea of injection is simply to 'inject' a solution into the population after a certain amount of computations have been made. This solution can be prepared in any way or be completely random. The idea is to hopefully introduce

or re-introduce good genetic material to the population.
In some algorithms roulette wheel selection will be used, others use elitit selection.

2.5 Other important features of GAs on TSP

In order to make these crossover operators work effectively on practical problems, they have to be combined with a good mutation operator and, for the hybrid algorithms, a good local search heuristic. I will in this section describe some mutation and local search heuristics that are used in GAs. Many of these were used in the algorithms described in the past sections.

2.5.1 2-opt

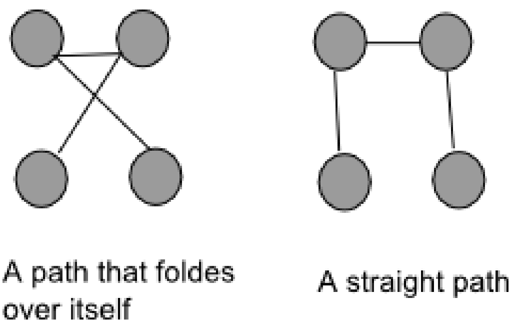


Figure 2.2: a 2-opt improvement on a subpath of a tour

2-opt is a special case of the general k-opt heuristic. It was devised in 1958 by Croes as a method for solving TSPs, and has been used in various forms extensively since.
It was inspired by Euclidean tours, that had a subcomponent (a passage), that 'folded back over itself', see figure 2.2, which intuitively is not optimal for such tours. His idea was then to swap two edges, that is exchange two cities and then reverse the path between them, hoping to unwind any paths like in the figure 2.2. Such a move is called a 2-opt move, [Uni].
In a 2-opt search, all possible exchanges are investigated. In some versions the best move (greed) is selected an carried out, other versions stop as soon as an improvement is found.

Today 2-opt is primarily used as mutation-operater and for generating starting points in a search or population (for GAs).

2.5.2 Lin Kernighan Heuristic

The Lin Kernighan Heuristic was developed by Lin and Kernighan in 1973 for solving TSPs, and has been used as the core of search algorithms that have been highly succesful since, the original algorithm simply called LK-search. For 16 years it was the 'best' algorithm for solving TSPs, and following a series of improvements from 1999-2009 by Keld Helsgaun [Hel00] [Hel09], the slightly modified LK-search has solved a 1 million city instance to under 0.058% in excess of the Held-Karp bound (the accepted method used for estimating optimal solution costs for tsp).
Inspired by the K-opt improvement algorithm, the basic ideas of LK-search can briefly be described as:

- Instead of using a fixed K-value, K can vary doing each iteration
- For K>2, edges to be broken have to be continues such that the endpoint of the first swap is used as a starting point for the next swap.
- In LK search uptil K-1 consecutive worsenings are accepted if the K'th swap finally results in an improvement. This corresponds to going down a hill in the fitness landscape, in order to climb back up on a new and higher hill.
- Only considering the 'interesting' part of the neighborhood.
- Some bookkeeping to ensure an effective search

There are multiple different implementations of the heuristic. For practical purposes every implementation has to have som decision on how to use the ideas listed above, as well as other bookkeeping issues. Some of the algorithms presented in this chapter uses a lin-Kernighan implementation as a subroutine, hence I need to construct a working LK-routine.
I have chosen to follow the approach in [KG11], as it seems to be well-argued for and, compared to the original LK-heuristic, is simpler to implement. This version makes some changes to the heuristic compared to the original LK-search, but every change is carefully considered and the performance of the heuristic should be competetive.

In this version the key ideas are used as follows:
The input tour is examined iteratively from the beginning, whenever an improvement is made, it restarts with the new improved tour.
When trying to improve a path it uses a 'cutoff' depth, α, to determine maximal value of K.
When searching for potential moves, only the interesting part of the neighborhood is investigated. Here the interesting part is defined to be those edges, that by swapping with the considered edge, generates a positive gain. That is, the difference between the original edge and the new edge is positive. I call these swaps *promising*.
For the first edge in the tour, it tries to make a *promising* swap between it and another edge. If closing the new tour is beneficial the changes are done and the search restarted with the new tour as starting point. If not, a new contigous swap is tried. This continues until α is reached, if no improvement has been made by then, the algorithm considers the next promising swap.
If none of the promising swaps works, the next edge in the tour is considered. This continues until all edges have been considered as a starting edge or an improvement has been found.
The running time depends exponentially on the value of alpha, hence small values of alpha should be used.
Today LK-search is used as a subroutine in various different algorithms like hybrid algorithms, Chained-Lin-Kernighan etc.

2.5.3 Double Bridge Move

Double Bridge Move is basically a 4-opt move. It is done by splitting the tour in 4 sections using 3 random points. These sections are then recombined in such a way that 4 swaps are performed.
This move is often seen, in literature, combined with Lin-Kernighan search [WHH10][Hel00][ACR03] and is used to break the search out of local minima.

2.5.4 Diversity selection

Although there might be other uses of the term 'Diversity Selection', it will in this paper refer to the survivor selection strategy used by Darell Whitley in his GPX-hybrid algorithm.
Diversity selection is a method to improve the diversity in a population, designed for use on TSP.

The concept is to count the number of times each edge appear in a population. For all potential candidates all edges are weighted according to the total number of appearances. The edge contributes a value corresponding to the inverse of the number of times the edge has appeared in the population, to a sum d. When all edges have been processed in this way, the d is used as a measure of how much diversity this solution represents.
Once all candidates have had their d-number calculated, the candidates with the highest d-numbers are selected for the next population.
This procedure is in this actual case [WHH10],combined with elitist selection (where the best candidates are found using strictly fitness evaluation).
In the article it is not clear whether the 'population' is the original population, the offspring population, the unfinished next population or the unfinished next population + all potential offsprings. A case can be made for all choices.
I chose in this paper to use the offspring population as the population.
It is interesting to study in this project, whether additional diversity maintaining mechanisms are needed for GPX.

2.6 Summary of literature study

In the early days of crossover operators, the focus was on preserving the cities actual position. The next wave shifted the focus to the edges. With the Radcliffe/Surry paper from 1994, [RSJJ94] and the introduction of the EX in [WSF89] a line with focus on the theoretical foundations for good operators in this field seemed to emerge.
[RSJJ94] introduced two basic principles for designing crossover operators:

- Alleless Transmission
- Respectfulnes

Watson argued in [WRE+98] that crossover needed to introduce new edges, to escape local minima and for diversity. This seems to suggest a tradeoff with the principles of [RSJJ94]. A key point, however is that new edges does not necessarily have to be introduced in the crossover operator, but can be done by using mutation. Diversity can be preserved using various methods such as diversity selection etc. Hence it is sound to focus on the principles of [RSJJ94] in the design of a crossover operator.
Recently the design of new operators seems to fall into three categories, where each follows its own design path. Some are devised as simple improvements to existing algorithms as in [SS11], [RBP05]. In another category different ideas

are combined to create an 'intelligent' choice in the crossover operator as in [Ahm10] and finally there is a category where operators are devised based on theoretical understanding of the problem [WHH10]. Algorithms adhering partly of fully to the principles in 2.2 have recently been devised. These algorithms have delivered promising results [Ahm10], [WHH10].
In order to compete with succesful search heuristics such as Lin-Kernighan search, as suggested in [RSJJ94], attention has been spent on combining crossover with local search operators into hybrid algorithms [WHH10]. The reasoning behind seems to be, to exploit the best of both worlds.
It is hard to numerically compare the performance of various algorithms, as not all of them have been tested in comparison-studies. I hope to be able to provide some head-to-head results in the next chapters.

Chapter 3

Selection and Implementation of Representative Crossover Operators

---

In this chapter I will describe the selection and subsequent implementation of certain crossover operators. I will argue for the choice of operators, describe how central features of the operators are implemented, and elaborate on design choices left open by the underlying papers.

3.1 Selecting Operators

The field of GA operators for TSP is quite large, as evidenced by the number of operators considered in the previous section. In order to be able to treat some operators in depth, I have chosen to focus on a few operators.
I have chosen to select the arguably best solution in the early stages (80-90'ies), and two promising solutions from different design paths. One of which seems to be the current state of the art in GAs for TSP's.

In multiple papers and in surveys [Jak10] [RBP05], Order Crossover (OX) is mentioned as the best operator and used as reference for testing new operators. I have therefore decided to use this as a baseline for performance on the TSP problems. The results obtained by OX will be used as a comparison to see how much ground newer methods adhering partly or fully to the design principles of [RSJJ94] have gained.
The Sequential Constructive Operator (SCX) try to combine the idea of preserving optimal edges and at the same time introduce new good edges, thus adhering to the principles mentioned in [Jak10], from both Watson [WRE+98] and Radcliffe et al. [RSJJ94]. It is done by adding a simple 'intelligent' choice. When building the offspring it starts sequentially from the start, and for all subsequent places, it decides by greed, if it should be filled with a gene from the first or the second parent. Thus this operator represents the second design paths, from last chapter.
The SCX is referenced in some newer papers, but not by any paper, introducing new operators. It seems from my litteratur study that there is a lack of contemporary research for this period and the paper seems to have lacked a thorough peer-review. The last one based on unclear description of some design choices, and places where the formulation is quite different from established standards within the evolutionary computing community.
However the results mentioned in the paper on SCX, [Ahm10] suggests that this operator could be competetive at a larger scale. Hence I have chosen to implement and study it. In 2009 Darell Whitley et al. presented a new operator, the Partition Crossover at the *GECCO* conference [WHH09]. It won the best-paper award at this conference, indicating that it was highly though off in the Evolutionary Computing community.
The operator was based on theoretical principles and insights gained through studies and earlier operators, and hence it represents the third design path, as presented in the last chapter.
This operator was further developed into a working hybrid algorithm presented in 2010 in [WHH10]. It was claimed that it outperformed one of the state-of-the-art algorithms; Chained-lin-kernighan, and thus a major breakthrough for GAs on TSP. This implies that GPX is one of the currently best solutions among GAs. I have therefore selected this operator.

As mentioned in the survey. When using GAs on TSPs respecting solely the principles of [RSJJ94], there is a risk that the algorithm will converge too early to a local optimum and be stuck there. Algorithms like SCX and GPX requires a population where edges, that also exists in the global solutions, are present, in order to be maximally effective.
Preserving a certain amount of diversity in the population during GA search is thus important. In the hybrid GPX algorithm it is embedded through the use of diversity selection. On the other two algorithms, I will in this project try a

simple injection strategy.
I will consider injecting a random solution, and random solutions improved by different local search heuristics. I will do some preliminary experiments to determine which heuristic should be used in the tests.

I will try to implement the algorithms as they are described in the respective papers. If there are some aspects that are unhandled or unclear in the papers, I will implement what seems to be the best solution, but respect the basic idea of the authors.
In the following the implementation of the crossover operators will be described. When embedding the operators into TSP solvers, some auxillary functions are necessary, those will be described as well. Finally the overall structure of the program for the testing will be explained.
I describe the primary issues in the implementation of the operators, as well as how I handled them. More triviel parts will not be described.

3.2 The Overall Program Structure

The framework for the tsp solver is constructed in the *JAVA* programming language, using the model-view-control design paradigm.
The modelling part contains the representation of the problem, methods for constructing a problem representation from a file and the algorithms that are used to solve tsp-problems.
The view part contains the visualisation of the tsp-problem, the solution(s) to it and presents the user with a visual representation of the possible settings.
The control part allows the user-specified input to control the loading of a problem and the calculation of solutions on it.
The visualization and the graphical user interface (GUI) is described later. It is designed to be simple and easy to understand.
The algorithms for solving the tsp-problems are explained in the following sections. I will here describe some important choices for datastructures in the overall framework.
An important aspect is the choice of representation for the problem, and potential solutions. I used the path representation method of representing the solutions, I implemented it using array-list datastructures. This allows me to perform fast look-ups, and provided some flexibility. Although some speed up could have been gained by using specialized datastructures (as will be mentioned later), a different implementation might have been necessary for the different algorithms. The primary task is to compare the algorithms to each other, and since they all use the same datastructure the comparison should not be affected.

Using this argument, I choose flexibility and fast look up.
I choose to represent the cost of traveling between each city, as a precomputed cost-matrix, once the positions of the individuel cities (on euclidean problem instances) are loaded, I calculate the individal edge costs and store them in this matrix. The datastructure for this is a double array of the *double* data type, this provide $O(1)$ lookup time, which is important, as this is an operation that will be performed often.
These are the most important design choices. One issue derived from the use of *doubles*, are that the computation of the fitness of a solution will be a little bit larger than it should be in reallife. The optimal solution to the *berling52* instance is *7544.3* in the evaluations in this program, while it is normally *7542*. I have seen this problem in a number of other studies, and since this is a relative study, and the deviances are very small, it should not affect the evaluation of the various operators.
In figure 3.1 I have shown the classes of the programs, the most important fields and methods, and which segment of the model-view-control they belong to.

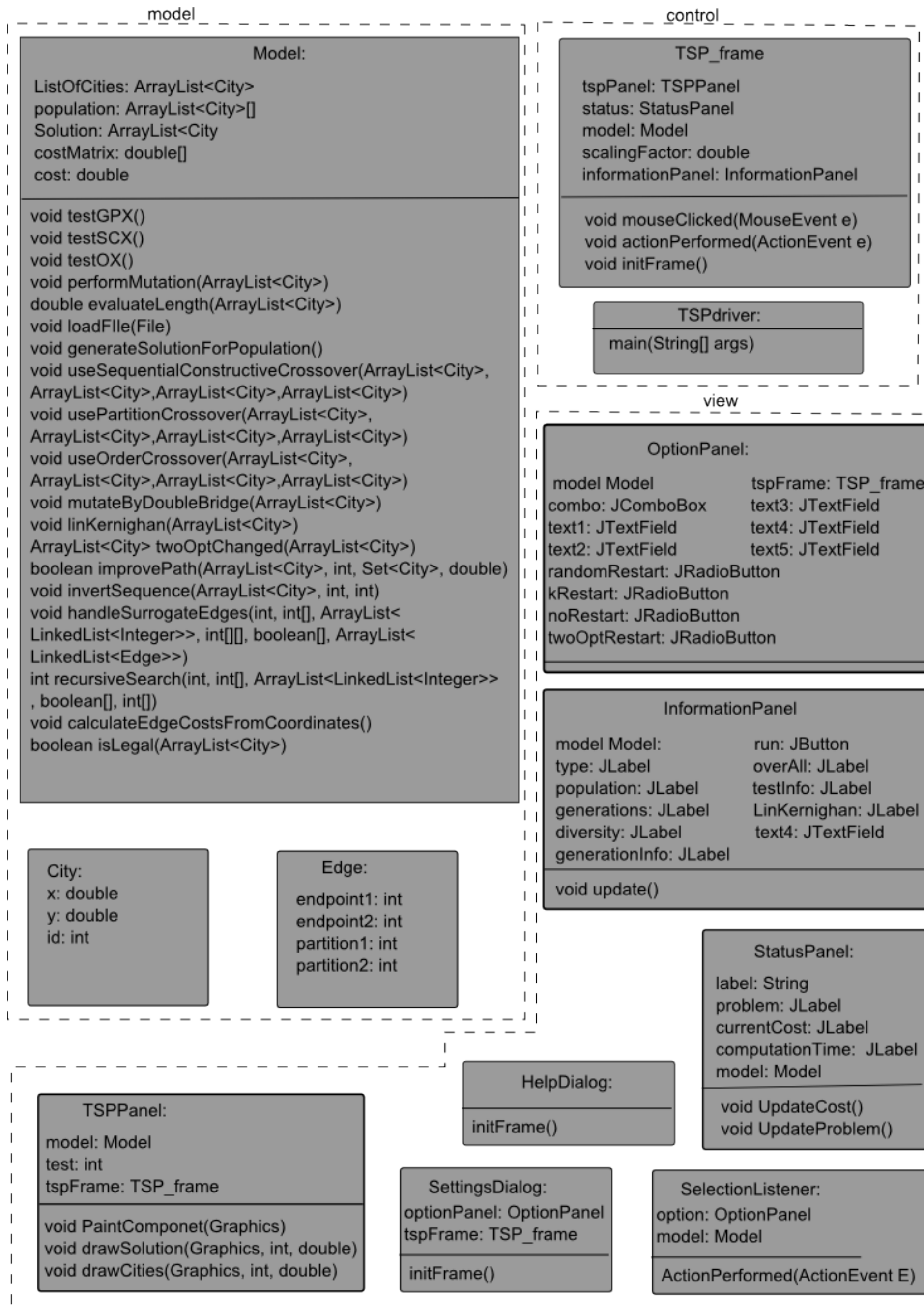


Figure 3.1: An overview of the programming structure

3.3 Implementing Order Crossover

To implement OX there are two problems I need to handle: The first one is to establish the swapping range, this is done by using a pseudo-random generator to generate a number and then a subsequent number between the first number and the size of the parents.
The second issue is to consistently be able to track what positions are already filled in the respective offsprings in constant time. This is done by maintaining two arrays, mapping from *id* to *location in offspring* for the respective offspring. This ensures that, since I have constant access to a given city's id, given its position, I can determine whether an element at location i in the second parent has already appeared somewhere else in the first offspring. Thus deciding if it is legal to use this gene in the offspring (this is a method to avoid repeating genes that have already been transferred from the first parent). Thus I avoid creating duplicates.
This operator produces legal offsprings according to the OX strategy, in O(n) time, where the implicit constant is small.

3.4 Implementing Sequential Constructive Crossover

When implementing this crossover, there are a couple of issues that should be handled. Those are primarily related to selecting the right city for every position:
SCX use the individual edge costs twice for every position in the offspring hence it should be possible to find those quickly. I argued in the overall description of the program that this can be done in constant time with the chosen implementation.
SCX frequently checks whether a given node has already been assigned to the tour. Thus this needs to be checked fast.
I achieve this by using an array per parent. Each array maps from city *id* to *position* in the offspring, if the id has not yet been assigned it is set as -1, thus when I check for legitimate nodes, I can use constant time to look up if a node is 'legal' or not.
When for a given position in the offspring, SCX switches parent nodes compared to the previous, I need to find the position of the considered node in the new parent. If done naïvely, this can take worstcase O(n) time for every node ($O(n^2)$ total running time). To avoid this I create a mapping from *id* to *index* in both parents.
Using these auxillary date structures, the SCX operator is implemented as described by Ahmed in [Ahm10].

3.4.0.1 Embedding the crossover operator into a working algorithm

To use the operator on practical problems, we need to add other features. The author, [Ahm10], explains in his paper which choices he has made, although a couple of those choices were unclear, I have tried to stay true to the intention.

The selection procedure:
The author suggests using a kind of fitness proportional selection called stochastic remainder method. However that does not make much sense for this given set up, I decide on using reciprocal roulette wheel selection (rws) instead, which is a fitness proportional selection strategy. An important aspect of using rws is to determine how much of the fitness is due to the individual genes, when operating with high fitness values, the relative differences becomes quite small. Thus when used on larger instances of tsp (costwise), a good solution will not be particularly preferred to another. Thus there will be close to uniform chance of choosing each indivial for mating.
Another issue, is that I need to do it reciprocal. Thus the best solution is the one, that has lowest fitness cost, and not highest:
To achieve this, I first subtract each fitness value from a 'correction' value slightly smaller than the current lowest fitness value in the population. By doing this subtraction, I ensure that the correlated fitness is below zero (or zero) for all candidate solutions, and that the best candidates correlated fitness now is the highest. By adding the correlated fitness of the worst candidate (now the least-valued candidate) to every candidate plus 2, I ensure all candidates are positive, and that I can calculate the proportion of the total fitness as if it was normal *rws*.
The exact value used in the last paragraph is not important in order to make *rws* reverse, but if the value is chosen appropriately, it can affect how much emphasis is placed on fitter solutions when selecting for mating. Consider this example:

Example *There are two individuals in a population with fitness value 8 and 12 respectively. I want to select one of those for mating.*
If I choose the correction value to be the sum of these two, 20, there would be 60% of selecting the individual with fitness value 8 and 40% for the other, this is close to uniform selection. If I instead choose the correction value to be 6 and follow the suggested algorithm by subtracting the current fitness value and adding the numerical value of the (current) least solution plus two, I get the following new values: 6 (previously 8) and 2 (previously 12). This gives a 75% chance of selecting the 'best' individual and only 25% chance of choosing the other individual.

When the appropriate fitness slots have been found and placed in an array.

I use a random generator to generate random number, and use it to select two solutions for mating.

The mutation operator The author suggests using reciprocal swap (i.e. randomly swapping two cities) which is well known standard mutation operator on TSPs. When the swap is completed the sequence between those two nodes are reversed. This is potentially an expensive operation as I use arraylists as datastructure for storing the solutions, however since this is only done at most once per generation, its impact is small compared to the crossover operator.

Survivor Selection operator The author suggests using Deterministic Crowding (the offspring competes directly with its parents). I have implemented this, using a simple tournament approach where both parents and the offspring compares to each other, scoring a point for each 'victory'. If either of the parents has scored zero points it is replaced by the offspring (which by default will have won at least one comparison).

3.5 Implementing Generalized Partition Crossover

Implementing GPX requires several sub-algorithms. Whitley et al. [WHH10] embedded the GPX operator in an hybrid algorithm, using predetermined mutation, selection and survivor selectors and combining it with the Lin-Kernighan local search heuristic. I will in the following describe how I implemented the core GPX operator. The auxillary algorithms and a working Lin-Kernighan implementation, will be presented later under the section 'auxillary algorithms'.

3.5.1 The core GPX operator

A key part of the GPX is the partitioning of the graph. Naïvely implementing the partitioning part would result in unreasonably large running times as the generel partition problem is believed to be NP-hard [WHH09].
In the precursor article for GPX, where PX is introduced, the author suggests an alternate implementation of the partitioning [WHH09], which can be done in O(n) time. I have implemented this version.
The first step is to construct an edgemap for the two parents. I need to be able to track the degree of each node in the edgemap, and to mark which edges are common (appears in both parents). I use an arraylist of linkedLists to repre-

sent the edgemap, where the first element of the list contains the degree of the node, and the rest the id of the node it links to. A common edge is denoted by flipping the sign, so it becomes negative, special consideration is taken on edges connecting with node 'zero'.
The second step is to construct two lists, one holding the *nodeId*'s, but sorted in ascending order by degree, the second indexing from *ID*'s and into positions in *list1*. This is done by two simple arrays. A pointer is maintained pointing to the first element with degree 3.
I construct an array mapping from *nodeID*'s into partition component numbers, every partition is initialized to -1.
The third step is to search for feasible partitions. I search through all potential nodes that can constitute a partition, that is nodes of degree 3 and 4. The search proceed as follows:

- initially enter the first node with degree 3 or 4 into a FIFO-queue, set the start partition component to 1 and set a pointer, *tracker*, to the final element in the list.
- for as long as there are unvisited nodes with degree 3 or 4 do:
- for as long as there are elements in the queue do:
- select the first element from the queue, c
- assign c to the current partition component
- add all unvisited and uncommon nodes in c's edgemap to the queue
- put c at the position where *tracker* points.
- put the element at the *tracker* position (before the update) to c's previous position
- decrement the tracker
- when there are no more elements in the queue, increment the current partition component, and add the element pointed to by *tracker* to the queue.

The above algorithm is implemented in a straighforward manner by introducing a linkedList, that acts as a queue (by adding last and retrieving first elements in my algorithm) and using previously introduced datastructures.
If there was only one partition component, then it is not possible to specify a meaningful partitioning of the graph, and the algorithm returns without producing a new offspring. I can test this by looking at my current partition component

number.
The fourth step is to determine how many partitions can actually be used (remember: only partition components that can be seperated by cutting just two edges are acceptable). Realising that only nodes of degree 3 are relevant, we consider only these nodes.
First all the nodes are checked to see if they have a common edge, to another partition, if so, they are stored as a cut-edge (meaning they can be used to cut a link between two components).
Then we consider all nodes in the partitionlist, if it is currently unassigned, we investigate it using a custom function, for handling 'surrogate' edges (surrogate edges is the authors word for a series of linked nodes of degree 2, starting and ending in nodes of degree 3):

- given a node of degree 2, investigate both neighbors recursively
- during recursive search, ensure the direction of the search, by noting the node the call came from
- if the current node is unassigned to a partition the search continues
- if the current node is assigned to a partition, partition component number of the node is propagated back, and the id of the endpoint (this current node) is stored.
- when unwinding the recurstion, all nodes set their partition component to the one returned from their recursive call.
- when reaching the original node, check if both endpoints are in the same partition, if yes, set this node's partition to be the same. If not, assign it to the left endpoints partition, and return this common path as a potential cut-edge between the two partitions, and starting at the two endpoint id's.

The implementation is done using a recursive subroutine, *recursiveSearch*, the results are returned in an double array of size number of partitions and size 3. The list of cut-edges from earlier is used to store the new cut-edges. The final step in this phase, is to determine which partition components are feasible. If at least one partition component has exactly two cut-edges the algorithm is feasible. All other partition components, with cut-edge size different than two is grouped together in a single partition component. (Implementation wise, this is achieved by reserving a special partition component number to this partition.)

The final step in the algorithm is using these partitions to build the offsprings. It turned out that there were a few unexpected traps/issues with this part.

First, since GPX base its choice of which parent that contributes the genes in each partition, off greed, I need to know the path length of each parent in all partitions. This is done by going through the parents sequentially. If an edge has an endpoint in two partition components its cost is not stored. Otherwise the cost of the edge is added to the parent's pathcost in the respective partition component.
Now the contributing parent for all partitions can be chosen. In all components the parent yielding the cheapest solution is preferred. However in the construction of the second offspring, for the largest partition component, the parent with the most expensive solution is chosen.

In the the construction itself a key observation is that I am working with graphs and paths, but using arraylists to represent the solutions in a sequential manner. Thus one issue is that the nodes in a given partition component can be listed from left to right in the first parent, but might be listed from right to left in the other parent. I need to handle this when constructing the offspring.
A third issue is that nodes from a parent in a single component, does not have to have a direct connection within the actual component (the path between some nodes could go through another partition componen). Thus I cannot assume that all the nodes in a partition component will appear sequentially, in a parent.
I proceed from the first node, c, in *parent1* and decides which parent should be used for c's partition component. If the other parent should be used, I search for the position of c in *parent2*.
Then I do:

- Check if I should go forward or backward in the parent
- proceed in the chosen direction adding nodes to the offspring along the way
- when crossing to a new partition component, find the parent that should be used
- if a parent switch was made, look up the position of the node in the opposite parent and repeat. If not, continue in the same direction as before.

This continues until all positions in the offspring are filled (and repeated for the second offspring). I use two arrays indexing from *nodeId* to position in the respective parents.
The GPX operator has now completed and returns the two offsprings.

3.5.1.1 Running time of GPX

It was claimed by the author that the running time can be done as O(n). I construct a number of datstructures in this algorithm, and none of these takes more than linear time, which I will argue for here.
In the first step I go through all edges in the parents once, spending constant time at each step. This takes O(n) time .
In the second step I construct a linkedlist of nodes of degree 3 and 4, this can be used to sort the nodes by degree, when constructing the two lists. This all takes O(n) time.
In the third step all nodes of degree 3 and 4 are considered. For every node considered, at most four other nodes can be checked. The time spend is constant for all checks and other bookkeeping steps (using the chosen datastructures). Thus the running time is O(n).
In the fourth step, where we handle the surrogate edges, every node previously unassigned is considered at most once. A border node can only be visited once in this step, as it otherwise would have degree 4 or be visited from another node with degree 3 or 4 (which this algorithm does not do). Thus the running time is O(n).
To select usable partitions, the number of cut-edges for all partitions are counted. This can at most be O(n) as there are at most 2n edges in the graph. Then all partitions are updated or checked. This step thus takes at most O(n) time.
Determining the path lengths inside each partition for both parents, requires a single traversal of both parents, constant time is spent for each node, thus this is takes O(n) time.
Finally to construct the offspring I iterate for n steps, one for each position in the offspring. At every step I do various checks and updates, in the worst case I have to switch parents. This requires a lookup in my indexing table which takes O(1) time and then a check to the neighboring nodes to determine direction. The runtime of this step is thus O(n).
Thus the total running time of the GPX operator is O(n), and this implementation complies with the demands and claims of the author in [WHH09] and [WHH10].

3.6 Implementing auxillary algorithms:

A number of auxillary algorithms are necessary to make the TSP-solving algorithms, incorporating the crosssover operators, work. The primary ones are described here.

3.6.1 Implementing 2-Opt

The 2-opt was implemented in the version, where the best move based on greed is chosen.
The 2-opt move is done by choosing two positions in the tour and exchanging them. In order to avoid creating four new edges, the order of the nodes between the two positions are reversed. The positions yielding the best gain is stored during the computation.
The method for reversing is used two times for every possible edge in the tour, this method is hence crucial to fast running times for 2-opt.
The datastructure used right now to represent tours (arraylists) requires linear running time in the distance between the two positions. If we consider all potential swaps involving the first city in the tour, the total running time of the reverse function is then $1 + 2 + 3 + ... + n - 1$ which is $O(\frac{n^2}{2})$. Since this has to be done for all cities, the total running time is $O(n^3)$, which is relatively slow.
A simple method of improving this would be to change the datastructure to something like a Linked list. However lookup operations would then be quite expensive (O(n)). A more thorough possibility is to use 'sway graphs' as used by Helsgaun in this Lin-Kernighan implementation [Hel00], [Uni]. This is, however, not simple to implement.
Since all the chosen algorithms use the feature of reversing a sequence either through 2-opt or LK-search, and that the comparisons in this project, is made relatively between the implemented algorithm, it is acceptable to use the chosen implementation.

3.6.2 Lin-Kernighan

As mentioned in the previous chapter, LK-search is one of the historically most succesful TSP solver. In order to find a good implementation, I originally looked at the implementation presented by Keld Helsgaun in [Hel00], but his implementation consists of more than 4000 code lines and a number of sophisticated datastructures and subroutines.
Since the focus in this work is on crossover and diversity, I have chosen to follow a relatively simple and recursive implementation of the LK heuristic found in [KG11]. Furthermore I use standard datastructures. This should severely impact the running time of the Lin-Kernighan subroutine, especially since it use the same reverse method described in 2-opt, and thus the running time of the hybrid algorithm using GPX.
Since I use relative time, significant differences between the algorithms should hopefully be clear regardless of these choices.

The described algorithm is recursive and consists of an outer method that simply iterates through all potential edges. It then calls the recursive inner method, *improve_path*.
improve-path works as described in [KG11]. I made two primary changes to the algorithm:
I precomputed all costs between cities in advance, to reduce running time. Secondly there was an issue not described in [KG11], that one has to undo potential changes if the final LK-move is rejected. This is done by adding a boolean, specifying if a LK-move was succesful or not, to each call/return for the recursive method.

3.6.3 Implementing Diversity Selection

This method refers to the special method introduced by whitley et al. for their hybrid GPX algorithm. I described it in 2.5.4.
This method requires two iterations of all offsprings, considering all edges in every offspring, thus it is important to minimize the time spend at each iteration. For this purpose I first use a 2-level datastructure. I use an arraylist representing all edges, where every field is a *hashMap*, mapping from recieving edge to number of occurences. Thus if I need to count edge (i, j) where $i < j$, I go to the i'th entry in the arrayList, find the j'th entry in the *hashMap* and update the number of occurences of this (i, j)-edge by one.
To calculate the d-values I construct a treeMapping from d-values, to a linked list of offspring id's. Thus for every value of d, I have the id's of the offsprings that have exactly this value.
Finally to select the k best candidates, where k depends on how many open spots are left, I extract the maximum d-value and corresponding offspring from the mapping holding the d-values. If the list for this d-value is now empty, I delete the entry. This continues until all k spots are filled.
In the first part I used a 2-level structure going from arrayList to a hashtable implementation, to achieve O(1) insertion and look up time.
Another decision was that all edges will be inserted as going from smaller to higher index. Even though this requires some bookkeeping, it reduces the space requirements by 50% and reduces the necessary look up operations by 50%.
These datastructures allows me to find all d-values in $O(n)$ time, using $O(n)$ space. This could be compared to a double-array datastructure which would have required $O(n^2)$ space. I choose to use a tree-Map for the d-values as it basically implements a max-heap, meaning I can get $O(log(n))$ time for inserting and extracting the maximum and $O(1)$ for finding the maximum. Alternatively I could have used a hashtable for obtaining $O(1)$ insertion time, but then I would have to spend more time at extraction and reporting maximum values.

The running time of selecting the k elements is then $O(op \cdot log(op))$ where op equals the size of the offspring population.
The total asymptotical running time is then $O(op \cdot n + op \cdot log(op))$. where op is at most 18, the asymptotical running time is then $O(n)$.

3.6.4 Implementing Double Bridge Move

I implemented this by generating 3 random numbers as splitting points. The 1. and 4. section were then concatenated in an intermediate arraylist, as was the 3. and 2. sections.
Finally the new tour was build by concatenating the elements in the two intermediate lists. The final order combination of sections was 1432.

3.7 Implementing Diversity measures

In the previous chapter, 2.4, I decided on using injection. This can be performed in a variety of ways, I have selected two of those.
The hybrid algorithm using GPX, implements its own version of diversity measure, *diversity selection*, so I expect that the effect of injection will be minimal on this algorithm.

3.7.1 Implementing Random Solution

The first idea is to have a random restart join the existing population. This is done simply by counting the generations and after a certain number, a random element in the population is replaced by a random generated solution.

3.7.2 Implementing local-search improved solutions

The second idea was to generate a random solution, and then improve it by some local-search heuristic before injection. As in the first idea, this is done by, after a certain number of generations, creating a new solution and improving it. This new individual is then placed in the population replacing another individual.
We consider the following local-search methods: LK-search, a series of 2-Opt

moves, until no further improvement can be made, and a fixed number of two-opt moves.
Before recording results, informal experiments were done to settle on when injection should be made, how many times, which local-seach method/which replacement method should be used, which element should be replaced.
I decided to fokus on LK-improved solutions and a series of 2-opt moves until no further improvement can be made. I used these for both the OX and SCX algorithms. The GPX, however, did not benefit at all, so I decided to switch it off for the testings.

3.8 The Graphical User Interface

The testing system is presented with a Graphical user interface. Interacting with this system triggers underlying methods that executes the tests. The end result is displayed visually in the interface. The implemented methods have been presented previously, here I will focus on describing the GUI. A screenshot can be seen in 3.2.

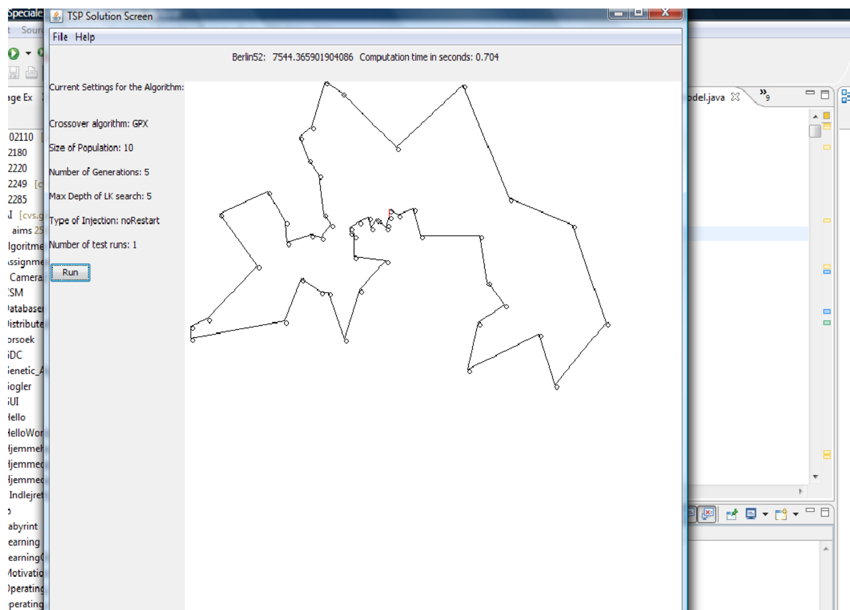


Figure 3.2: An example of the GUI of the framework

3.8.1 The visual representation

The interface consists of:

- A pane for displaying the problem and a solution.
- A menuItem that opens op a screen for selecting problems.
- A menuItem 'settings' that opens a controller that allows the user to choose between various options for the solution algorithm.
- A menuItem that explains the minimum settings for a given problem.
- A side panel for displaying the current settings for the solution algorithm.
- A toppanel containing the solution length, the name of the current problem and the computation time of computing this solution.

3.8.1.1 The solution panel

Visualizes the problem, by representing each city with a small circle at the appropriate coordinate. The solution is shown by drawing a line for every edge present in it.
I use a variable scaling factor to be able to monitor the whole problem in the same frame. When clicking with the left mouse button, a random solution is generated.

3.8.1.2 Problem selection

At present the user has the choice between a number of problems gathered from TSPLib library.

3.8.1.3 Options for the algorithm

The user has a choice of options. He can choose between the Order, Sequential Constructive and General Partition operators. When using GPX it is possible to specify the maximum depth for Lin-Kernighan search. It is possible to specify population size (has to be at least 2), number of generations and how many testruns should be made. Furthermore injection diversity measure can be switched on.

3.8.1.4 Top panel

The toppanel serves as a status panel where the current problem, the cost of the current solution, and the time to calculate the solution is displayed.

Chapter 4

Experimental results

In this chapter, I will execute various tests, using the algorithms described in the previous chapters. I will perform qualitative tests (findings to be discussed in the discussion section), standard numerical tests, and statistical tests.

4.1 Selection of Tests

I have chosen three standard problems from the online library TSP-Lib which can be found at [GR]. TSP-Lib contains a number of real-life inspired problems formulated as TSP instances. Although all instances have been solved, all the papers I read, referenced this library. Based on this, I chose to use problem instances from this library.
The first problem I chose was *Berlin52*. *Berlin52* is 52 locations around Berlin, and is thus indicative of a 'real-life' problem. The problem uses euclidean distances and are symmetric. Results for both OX and SCX (in [Ahm10]) have been reported on this particular problem.
The second problem is *kroA100*. This problem is slighly larger, and should be a challenge for the older operator OX. Results are reported for SCX on this problem in [Ahm10]
The third instance is a problem of medium size by today's standards, but large

for older algorithms. It is called *pr439* and represents the printing of a circuit board. I expect that this will be a problem where OX and SCX might have trouble, but I expect GPX to solve it efficiently, as problems of comparable size have been solved easily [WHH10].

The 3 algorithms with the injection strategies presented in 3.7.1 are run 10 times on all three problem instances, and will be initiated using uniformly random starting points. For GPX I tested before hand if injection is feasible, and found that it was not, hence results will not be presented for this instance.
I will use these test runs to obtain numerical results, a statistical comparison between the algorithms and to observe qualitative behavior of the algorithms.
When reporting the test result I will use a measure called *excess percentage* which is calculated as $\frac{obtainedresult - optimalresult}{optimalresult} \cdot 100\%$. This is the standard numerical measure for quality of solution used when testing TSP-solving algorithms.

Statistical tests can be used to make statistical inference. The idea is to measure small samples of larger populations and use probability theory to gain insight on the larger populations. Since it uses small sample sizes, there is some small chance of error, as for instance, an observation might be very rare, but have occurred in exactly this small sample.

Typically a specific hypothesis is formulated for testing, this hypothesis is then either rejected or accepted with a given level of significance. The level of significance indicates how much chance there is that the conclusion is correct. Usually tests are conducted with a 95% or 99% confidence interval, which means that with 95% or 99% chance respectively, the conclusion of the statistical test are correct.

Different statistical tests exists with different strengths and weaknesses. In this project I want to determine whether two sets of observations come from the same population, or if they are different. For this purpose I have chosen the Mann-Whitney-U test (also known as Wilcoxon rank-sum).
Mann-Whitney tests are parameter free, statistical tests that is usually used to test the main hypothesis that two sets of observations belong to the same population. The alternative hypothesis is that they are different and that one population has smaller/larger values than another. I will use a 95% confidence interval when performing the tests.
The concept of the Mann-Whitney-U test is to rank the two populations against each other, in order to obtain a u-value, that can be used to either reject or

accept the hypothesis.

4.2 Evaluation of Tests

4.3 Test of Order Crossover

4.3.1 Parameters for the test

It was difficult to find an 'optimal' configuration for OX in papers, so I did some initial experiments to determine what configuration I would use.
OX is used with a population of 100, and run for 50.000 generations. As a sidenote I later found one study [RBSMBT], that actually suggests using a population size of 100 for OX.
I use a mutation operator, that with 1% chance uses 2-Opt, and 5% uses a random swap, and subsequent inversion of the nodes between the two swapped nodes.
After the last generation, 2-Opt is run on the best solution, until the solution could not be further improved by this.

During the process, considerable effort was spend trying to establish the best mutation operator and population. Using the inversion extension combined with occasional 2-Opts decreases the excess percentage by 50% compared to not using 2-Opts and having a standard mutation rate of 1%.

4.3.2 The results

The algorithm was run for 10 testruns:

Testrun	Avg. Excess Percentage	Best	Worst	Wallclock time
1	4.2	0	7.7	22.5 seconds

Table 4.1: Results for OX on Berlin52

For *Berlin52* this resulted in an average excess percentage of 4.2%. The global

optimum was hit 3 times, there was one outlier that was almost 12% in excess. *2-Opt* was used on average 2 times to finish the algorithm.

4.3.2.1 kroA100

The algorithm was run for 10 testruns :

Testrun	Average Excess Percentage	Best	Worst	Wallclock time
1	5.1	3.0	8.5	213.5 seconds

Table 4.2: Results for OX on KroA100

On this instance 50.000 appeared to be to few generations for the algorithm to converge, setting generation numbers to 100.000 and 200.000 produced better results, and it still appeared to not be finished converging. The reported results are for 50.000 generations though.
However when running this many generations, and especially if using 2-Opt, the running time is quite slow. Furthermore the quality of solution is not very good.
The average excess percentage was 5.1. The 2-Opt procedure was used on average 15 times to finish off.
The running time is negatively impacted by the number of twoOpts.

4.3.2.2 pr439

The algorithm was run 10 times, however the running time was so slow, that I decided to switch off the 2-opt mutation, and only use the standard mutation I described in 3.4.0.1

Testrun	Average Excess Percentage	Best	Worst	Wallclock time
1	8.5	4.5	10.5	1110.2 seconds

Table 4.3: Results for OX on pr439

On this larger instance OX was unable to produce solutions of a quality close to that of the performance on the other two instances. An average excess percentage of 8.5% was obtained. The relative worse performance is probably because I removed the occasional 2-opt mutation.

I use the same number of generations on this instance as on *kroA100*, so I expect the difference in running time is how many times 2-opt is used as the finish procedure. It was used on average 400 number of times.

4.3.3 Test on OX with injection

I test the three different injection strategies. Since Lin-Kernighan search is known to be very efficient by itself (usually capable of being within 2-3% of the optimum), I test it seperately as well.The results are presented under each problem:

4.3.3.1 Tests on berlin52 summarized

Type	avg. result excess percentage	best	worst	Wallclock time
LK injection	1.5	0	5.3	11.4 seconds
2-Opt injection	2.2	0	6.2	10.0 seconds
random restart	5.6	2.5	12.6	9.8 seconds
Pure LK	2.6	0	5.6	0.2 seconds

Table 4.4: Results for OX with injection on berlin52

In most of the testruns with lk-injection the final solution was equal to the injected solution, but in four of the testruns, OX bettered the solution found by LK-search. The improvements lowered the excess percentage by 0.8% on average. None of these cases resulted in the optimal solution.
On average the first injection were selected for mating in 500 out of 25000 generations corresponding to 2% of the time. There was a single outlier (excess percentage of 5.3) bumping the average up.
When using 2-Opt injection, OX improves on the injected solution in 60% of the testruns. The average increase is approximately 4%. On average the first injection were selected for mating in 500 out of 25000 generations corresponding to 2% of the time.
Random restart was largely unable to converge on a good solution compared to OX without injection.

4.3.3.2 Tests on kroA100

Type	Avg. Excess Percentage	Best	Worst	Wallclock time
LK-injection	0.6	0.1	0.8	18.4 seconds
2-Opt injection	3.2	0.4	8.2	26.4 seconds
Random restart	5.2	1.7	12.2	21.0 seconds
Pure LK	1.3	0.3	4.0	0.9 seconds

Table 4.5: Results for OX with injection on kroA100

In none of the testruns did OX improve on the injected lk-results. The first injected solution were used in approximately 500/25000 generations corresponding to 2%.
The 2-opt injection was improved in 5 out of 10 cases corresponding to 50%. for an average improvement in excess percentage by 4%
Random restart was again unable to converge within the alloted number of generations.

4.3.3.3 Tests on pr439

Type	Avg. Excess Percentage	Best	Worst	Wallclock time
lk-injection	1.9	0.9	2.7	107.2 seconds
twoOpt injection	8.0	5.8	9.7	2814 seconds
random restart	10.6	10.1	11.5	1241 seconds
Pure LK	2.3	1.0	4.2	13 seconds

Table 4.6: Results for OX with injection on pr439

In three of the testruns did OX improve on the injected lk-results. The improvements lowered the excess percentage by 0.75 % on average. The first injected solution where used in approximately 500/25000 generations corresponding to 2%.
2-Opt injection did not improve notably on the quality of results compared to the original OX operator, but the running time was doubled.
Random Restart yields worse results than standard OX.

4.4 Test of Sequential Constructive Crossover

4.4.1 Parameters for the testing

I tried to follow the exact specifications of the algorithm in [Ahm10], but (perhaps) due to ambiguity in the text for instance regarding selection strategy. I was unable to reproduce the claimed results obtained in [Ahm10], I missed the claimed times by a factor of 10.
I then used SCX in conjunction with some other ideas to see if I could somehow reproduce the achieved results:
I used population size on 200, 10.000 generations, a reciprocal swap mutation but inverting the sequence between the swapped nodes all as claimed in [Ahm10].
I changed the mutation rate to 5%, and used a standard reverse rws selection procedure (as this was a 2+1 algorithm), and hence his suggestions of stochastic remainder selection did not seem to make sense.
I kept the concept of deterministic crowding as survivor selection, this concept was explained earlier in 3.4.0.1.
Finally I added a 'finish' procedure, which I applied to the best solution (by fitness evaluation) after the last generation had been run. I experimented with two local-search finish procedures one was to continously apply 2-opt as long as it yielded a better result and the second was to use LK-search.

4.4.2 Testresults

Instance	finish procedure	Avg. Excess Percentage	Best	Worst	Wallclock time
berlin52	twoOpt	1.4	0.0	4.1	4.1 seconds
berlin52	linKernighan	1.1	0.0	3.1	3.4 seconds
kroA100	twoOpt	2.5	0.4	8.2	7.9 seconds
kroA100	linKernighan	1.9	0.0	6.2	8.5 seconds
pr439	lin Kernighan	2.7	0.8	4.1	77.5 seconds

Table 4.7: Results for SCX

For *Berlin52* an average result on ten runs resulted in excess percentage of 1.1% when using LK search to finish, and 1.4% when using twoOpts. During the ten runs, the global optimum was found at least 5 times. At least one run resulted in an 'outlier' on around 8% excess, that bumps up the total excess percentage. During the runs, some qualitative information was obtained about where it has

trouble which will be discussed in the next chapter.

For *kroA100* the algorithm started to get bogged down. It still produced decent results, although only 1 in ten runs found the optimum. It seems 10.000 generations is to few for it to converge.
The results however are better than those stated in the paper by Zakir Ahmed [Ahm10]. only showing an excess percentage of 2.

For *pr439* It was infeasible to use twoOpt as a finish procedure. With Lin Kernighan finish, it hit 2.7% average excess percentage.
In general it performs well, but it has one outlier that causes the average percentage to be higher. It seems that 10.000 generations are too few for it to converge on an optimum. This number can be varied to obtain better results (using 20.000 generations yielded 2.4% excess percent) at the expense of higher computation costs.

4.4.3 Injection strategies on SCX

I tried to use all three described injection strategies on SCX. Again I inject after 50% and 75% of the total generations are computed. Since the Lin Kernighan finish procedure was found to be better in the previous section, I have chosen to use it as finish procedure in these tests.

Instance	Injection Strategy	Avg. Excess Percentage	Best	Worst	Wallclock time
berlin52	2-Opt	1.6	0.0	3.5	4.2 seconds
berlin52	linKernighan	1.9	0.0	2.7	6.8 seconds
berlin52	randomInjection	1.6	0.0	4.3	3.9 seconds
kroA100	2-Opt	1.6	0.0	3.7	14.1 seconds
kroA100	linKernighan	0.7	0.0	2.6	8.2 seconds
kroA100	randomInjection	0.8	0.0	1.9	6.6 seconds
pr439	linKernighan	1.7	0.7	4.9	52.0 seconds
pr439	2-opt	3.8	2.0	5.8	3012 seconds
pr439	random	2.9	0.7	4.5	49 seconds

Table 4.8: Results for SCX with injection

On *berlin52* the injection strategies were actually a bit worse than standard

SCX. But this is probably just statistical uncertainty.
On *kroA100* and *pr439* the injection strategies yield better results than standard SCX. The twoOpt strategy provides a slight improvement of only 0.3% average excess percentage on the *kroA100 instance* and slight decrease in performance on the *pr439* instance (and an explosion in runnning time).
The LK injection yields an improvement of 1.2% average excess for *kroA100* and 1.0% for *pr439*. Random injection does not yield any improvement on *pr439* and *berlin52*, but does yield improvements similar to injection with LK-search on *kroA100*.

4.5 Test of Generalized Partition Crossover

4.5.1 The Testversion and parameters

The GPX hybrid algorithm is implemented as in the original paper [WHH10]. It was explained in this paper at 6 in the Survey.

4.5.2 The tests

I use 5 generations for *berlin52* and *kroA100*. For *pr439* I use 10 generations. In the original paper between 5 and 50 generations are used, and 5 generations should generate solutions to larger instances of a quality of around 0.6% excess percentage. The maximum depth of the LK search is set to 5. Otherwise I use the settings from [WHH10].

Instance	Avg. Excess Percentage	Best	Worst	Wallclock time
berlin52	0.0	0.0	0.0	0.3 seconds
kroA100	0.03	0.0	0.3	2.7 seconds
pr439	0.7	0.3	1.0	114.0 seconds

Table 4.9: Results for GPX

For *berlin52* the optimal result is found in 1 generation, the running time on the wall clock is 0.3 seconds. Mostly the LK-search produces the right solution in the first generation, so GPX is idle, but in a few cases GPX improves on the LK-produced results via crossover.

For *kroA100* optimal result are mostly found within 3 generations in average 2.7 seconds. Here LK-search is mostly incapable of producing the optimal result alone, but GPX is effectively recombining the various near-optimal solutions to achieve to optimal one.

For *pr439* GPX was not capable of generating the optimal result in 10 generations. The best result was 0.3% percent above the optimal solution, with the average being 0.7% excess. On this problem the strength of GPX is clear, as it repeatedly uses local minima to create better local minima. On a problem of almost similar size, *att532*, the author of [WHH10] reports excess percentages of 0.5% average after 10 generations, which is comparable to the results I achieved here (assuming the hardness of each problem is comparable, and I haven't found anything that indicates otherwise).

4.6 Statistical Tests

In order to be able to determine whether there are significant differences in fitness level between the results produced by the various algorithms, I ran a series of Mann-Whitney-U tests.
The number of test runs, that is 10, for each algorithm provides the population of each algorithm. Since the population is this small, the distribution is tabled, and I can use a simple counting scheme to find the u-values used.
This counting scheme works as follows:

Let the population that seems to be smallest be distribution 1, and the other distribution 2.

Make a complete list of all observations in both distributions, sorted by fitness value.

For all observations in distribution 1, count how many observations from distribution 2, that has a better fitness value than itself.

Add this number to the total u-value

When using the 95% acceptance interval for a population size of 10, the accept interval is for u-values between 23 to 77. If the number is lower, distribution 1 has better values. If the number is larger than 77 distribution has better values.

(with 95% confidence)
In the following table I have listed the results of the statistical tests:
Some notes: I tested SCX linK finish vs. SCX twoOpt finish and they were not statistically different. Hence whenever I write SCX, I mean SCX with a Lin-Kernighan finish.

Instance	Algorithm 1	Algorithm 2	u-value	Rejection of the null-hypothesis	Best algorithm
kroA100	OX	SCX	8	yes	SCX
kroA100	GPX	OX	0	yes	GPX
kroA100	SCX	GPX	10	yes	GPX
kroA100	OX inject LK	OX inject twoOpt	20	yes	OX inject LK
kroA100	OX inject twoOpt	OX	14	yes	OX inject twoOpt
kroA100	SCX LK finish	SCX twoOpt finish	26	no	n/a
kroA100	SCX inject LK	SCX	12	yes	SCX inject LK
kroA100	SCX inject LK	SCX inject 2-opt	54	no	n/a
kroA100	SCX inject LK	SCX inject random	59	no	n/a
kroA100	SCX	OX inject LK	84	yes	OX inject LK
kroA100	SCX inject LK	OX inject LK	59	no	n/a
kroA100	SCX inject LK	GPX	0	yes	GPX
pr439	SCX	OX	0	yes	GPX
pr439	GPX	OX	0	yes	GPX
pr439	GPX	SCX	10	yes	GPX
pr439	OX inject LK	OX inject 2-opt	0	yes	OX inject LK
pr439	SCX inject LK	SCX inject 2-opt	8	yes	SCX inject LK
pr439	SCX inject LK	SCX	22	yes (barely)	SCX inject LK
pr439	SCX inject LK	OX inject LK	44	no	n/a
pr439	SCX	OX inject LK	82	no	n/a
pr439	GPX	SCX inject LK	5	yes	GPX

Table 4.10: Statisical tests

For *Berlin52* there were too many observations that were equal, in particular results that hit the global minimum to reject the null-hypothesis. Instead I decided to focus on the results for the larger instances.
Of the algorithms without injection, GPX is the best, SCX is worse than GPX but better than OX. OX is the worst operator.
The best finish procedure for SCX is statistically LK-search.
For both *kroA100* and *pr439* injection with LK is statistically significant better

than other injection methods for OX, hence we will use this version to compare to other algorithms.
Injected OX is statistically better than standard OX for all instances including *berlin52*, and it SCX injected with LK is statistically not better than SCX injected with twoOpt on the *kroA100* instance, but on the larger instance *pr439*, injection by LK is statistically better.
Injected SCX is statistically better than SCX for *kroA100* and for *pr439* but only barely (the U-value was on the border). Which indicates that injected SCX might be marginally better.
Injected OX is statistically better than SCX, but worse than GPX.
Injected SCX is statistically as good as injected OX and worse than GPX.

Chapter 5

Discussion

In this chapter I will discuss the obtained results. I will use those, together with the analysis from the survey, to present strengths and weaknesses of the three operators. I will then compare them to each other and finally I will present some summarizing thoughts on the use of crossover operators on TSP.

5.1 Comments on the results

5.1.1 The Order operator

The Order operator was the simplest and easiest to implement. The results show that the OX operator has problems with solving moderate and larger sized problems efficiently. On the *Berlin52* case it performs adequately, but on the *kroA100* and *pr439* the results are increasingly worse.
OX seems to be better with a higher number of allowed generations, continously improving on solution quality with increased number. This indicates, that it is capable of escaping local minima traps. This however is probably due to the mutation operator and not the crossover.
OX is one of the most succesful GAs for TSP from before theoretical research suggested key principles like alleles transmission and respectfulness. Since OX

is not following these principles, it can introduce many new edges at each generation, (recall the analysis in 2.1.4).
This enables OX to not be trapped in local minima. However as seen in [Jak10] the converging time is considerably slower, as the probability of producing bad offsprings (because of the new edges) are significantly larger, and thus OX needs more generations to produce high-quality offsprings. I believe that this mechanism is reflected in the obtained results.
In most papers I considered for the survey, GAs for TSP used a population size of 10. Thus I would have expected that a population size around this should be preferably for OX. When experimenting, however, I found that solution quality seems to benefit from considering a larger population. Empirically I found a steady improvement in quality until population size reached 100, and a steady decrease in quality when further increasing size. This is the same population size as suggested in [RBSMBT].
The explanation for the increase in performance is probably due that there are more options to work with for OX. with more options, there are both a larger probability for producing good edges in the initialization step and preserving diverse searching points for a longer time. The increased population sized can be viewed as a kind of parallelization of the search, but with interactions between intermediate results.
The decrease, I expect, is caused by the limitation on the generations. With a larger search space, with more candidates to potentially improve, more generations are needed to converge on a good solution. Thus there appears to be a certain 'golden' ration between population size and number of allowed generations.
If those claims are valid, the performance of OX becomes (perhaps not surprisingly) a tradeoff of quality of solution (increasing population size and generations) and execution time.
As evidenced by the very slow performance on a problem of moderate size, *pr439*, it is for most practical purposes not feasible to have a sufficiently large population and allowed number of generations to achieve decent quality of solution for larger problem instances using OX.

5.1.2 OX with Injection

When Figuring out how to use the injection for best effect, it appeared that different strategies worked well on different problems. In general however, injection by an individual improved by Lin Kernighan produced the best results. I tested all three variants on this algorithm. The random restart injection strategy did not improve on the quality of solution. Both the 2-Opt and LK injection strategies improved the quality. Looking closely at the results for LK-injection,

it seems that the improved results are mostly due to the use of LK-search. However in 40% of the cases, OX is capable of improving the solutions produced by the injected LK individual, and decreases the excess percentage by 0.8% in those cases. On average the total results for the LK-injected algorithm are 1% better on average than pure LK-search, and 4% better than standard OX.
When looking at the 2-opt strategy, OX clearly improves on the injected solutions (in 50-60 percent of the cases), by approximately 4% on average. However some of the injected results are not very good, and this causes the end result on *pr439* to be close to that of the standard OX operator. The end result is the second best of all OX-algorithms though, and not far behind the LK-injection strategy (except on the *pr439* instance) based on fitness value. The running time is considerably slower though.
It is still clear that working on a moderate-large instance, *pr439*, (except for the LK-injection strategy) is slow work compared to other algorithms as it spends between 20 and 40 minutes on achieving solutions of subpar quality.
It can be argued that the injection should be performed at other times, but when I experimented with moving the injection time in generations, and the number of injections, I found no noticeable difference.

Although, perhaps because of the chosen method, there was no clear benefit from using injection as a diversity maintaining strategy. The results support the idea that if given decent candidates as input (fitness wise), crossover operators are capable of generating new and better solutions using these candidates. It is an encouraging result especially since OX is by no means optimized for this purpose.

5.1.3 The SCX operator

The SCX operator is quite easy to implement, requiring a little more work than the OX operator, and given the testresults, SCX seems to be a preferred choice compared to OX.
I was unable to reproduce the results claimed in [Ahm10] (by a factor of 10 in excess percentage). However with the addition of a 'finishing' procedure, I was able to reproduce similar results, and for *kroA100*, even better results. For the smaller instances SCX seems to provide good quality of solutions, as stated by the author. On the larger instances including *pr439* SCX performsnot as well. It was not as effective at improving good solutions as expected, which was thought to be a strength of SCX.
The overall impression from the test results is that SCX is better than the early crossover operators and many other suggested operators, but appeearently it does not scale well.

The main argument for using SCX (by the author) was that it retained approximately 71% of edges on each crossover and then introduced good new edges. It seems that the operator seems to find a tradeoff, complying with both watson and Radcliffe et al. [RSJJ94]. As argued earlier the principles in [RSJJ94] is the most important in order to create good offspring, as SCX adheres to these but not completely. This is a possible explanation of why it is better than an algorithm like OX, but fails to outperform GPX.
In SCX the problem is viewed as one-directional, that is, the order is only considered from left to right. However since we are working on a bi-directional problem, we might miss on preserving/adding good edges.

example: This example is taken from *berlin52*, Consider the optimal sequence:

xx	40	37	38	48	yy.

We consider two candidates where the sequence is close to the optimal:

p1:	40	38	37	48
p2:	40	37	48	38

From the cost-matrix I know that the edge $40 \leftrightarrow 38$ is better than $40 \leftrightarrow 37$, that $37 \leftrightarrow 38$ is extremely cheap and that $37 \leftrightarrow 48$ is slightly worse than $38 \leftrightarrow 48$. Both parents thus have edges present in the global optimum. However since SCX is one directional and always starts with p1, the offspring will be build as:

40	38	37	48

, although if p1 had been reversed (or read from the end) the resulting sequence in the offspring would have been:

48	38	37	40

, which is the optimal sequence.

Another problem related to this, is when two local minima are similar (not identical), If the order of the second candidate is reversed compared to the first, the crossover would basically only use the first candidate for building the offspring.

Consider this small example:
In here the example from the survey section that GPX solves is revisited.
The two parents have the following genes:

p1:	2	1	9	8	4	6	10	3	5	7
p2:	7	10	3	5	6	4	1	9	8	2

The optimal solution is:

opt:	2	1	9	8	4	6	5	3	10	7

Remember that the following holds for the edges:
$8 \leftrightarrow 4$ *is smaller than* $8 \leftrightarrow 3$,
$4 \leftrightarrow 6$ *is smaller than* $4 \leftrightarrow 3$,

$2 \leftrightarrow 1$ *is smaller than* $2 \leftrightarrow 8$,
$8 \leftrightarrow 4$ *is smaller than* $1 \leftrightarrow 4$,
$6 \leftrightarrow 5$ *is smaller than* $6 \leftrightarrow 10$,
$6 \leftrightarrow 10$ *is smaller than* $6 \leftrightarrow 3$,
$10 \leftrightarrow 7$ *is smaller than* $5 \leftrightarrow 7$,
Given those edges the offspring constructed by SCX, will be exactly p1, however if p2 had been reversed, the following offspring would have been created:
offspring: | 2 | 1 | 9 | 8 | 4 | 6 | 5 | 3 | 10 | 7 |, *which is the optimal solution.*

The strength of this operator is when the edges used in the global optimum are present in the population. If the candidates chosen for mating contain one such edge, it will be in the offspring (unless as shown in the above example, the candidate containing such an edge is in reverse order).
If the global minimum edges are not present, SCX tends to be stuck in a local minimum. When adding 2-Opts in the search process, the overall performance improves, I believe it is mainly due to this mechanism.
An algorithm using SCX as the crossover operator should focus on this particular strength, thus it aught to find a way to introduce edges that are present in the global optimum, through, for instance, local search.

5.1.4 SCX with injection

The three injection strategies were used in the same manner as for OX.
The random restart injection strategy resulted in results that were close to that of the standard SCX algorithm. The differences were found to be statistically insignificant.
The 2-opt injection produced results that were very similar to standard SCX, and did not result in any noticeable improvements on the quality of solutions.
The Lin-Kernighan injection strategy gave (except for the *berlin52* instance) an improvement of 0.4% and 1% excess percentage on *kroA100* and *pr439* respectively. The injected results were used on average 1% of the following generations.

As for OX, the idea of a random restart injection to preserve diversity, does significantly not affect the fitness of the achieved solution.
The idea of introducing a good 'building block' in the form of a local search improved search point seems to work well for SCX. Demonstrating, as for OX, that crossover can use good local minima to create new and better local minima. Where LK-search usually generate a good search point, the 2-opt strategy are

more inconsistent. Sometimes the 2-opt enhanced injected individual is a very bad local minima, and SCX has trouble using it as a building block. This is probably the reason why the LK-injection strategy is better than 2-opt strategy for this algorithm.

5.1.5 The GPX operator

The hybrid algorithm integrating the GPX operator, delivers comparatively good results. On the small *berlin52* instance the optimal solution is found in every test run. On *kroA100* the optimal solution was found in 50% of the runs, although all solutions where within 0.5 excess percentage. When used on the larger instance *pr439* the global optimum was not found, but the average excess percentage was 0.6%. These results are significantly better than the other tested algorithms, and are close to the claims (quality wise) made by Whitley et al. in [WHH10].
It was to be expected that the hybrid algorithm would perform well, as it makes heavy use of the LK-search routine. In order to gauge the effectiveness of GPX I looked at whether it did indeed improve on the solutions with each generation, thus demonstrating the value of the GPX crossover operator.
I found that the GPX is indeed able to reassemble local minima to a new local minimum (the tunneling claim by whitley in [WHH09] and [WHH10]) and to find the global optimum in this way. On *kroA100* LK-search was unable to get closer than 0.3% excess percentage, and on average the LK-produced solutions where around 1.3% excess. The GPX had to use between 2 and 3 generations to recombine the produced local minimas to produce the global optimum.
This effect was larger on the *pr439* instance, where LK-search produced results at least 1.0% excess percentage over global minimum (1.9% on average). GPX was able to reduced this to an excess percentage of 0.5%. These findings demonstrate the claimed effectiveness of GPX by [WHH09][WHH10].
The strengths of GPX is directly tied to its adherence to the principles of alleless transmission and respectfulness. It is very good at preserving good edges from both parents, and recombine them in the offspring.
One contributing factor is that the problem is considered both-directional. It respects the fact that it is working on a problem that is best represented as a graph, and thus avoids the trap that SCX suffers from in 5.1.3.
In the following I will present two examples. The first one revisits the example from the survey 2.1, which is the same as in 5.1.3, the principle of combining good building blocks is shown. The first parent contains a good solution in the first part, and the second parent do that in the second part.
Afterwards I present an example of the effectivity on the problem instance of *berlin52*, the example was taken from a run of the GPX-algorithm on the prob-

lem during my tests.
Example:
The two parents have the following genes:

p1:	2	1	9	8	4	6	10	3	5	7
p2:	2	8	9	1	4	6	5	3	10	7

The optimal solution is:

opt:	2	1	9	8	4	6	10	3	5	7

Using GPX, the string is two partition components one containing the sequence between 2 and 4 and one containing the sequence between 6 and 7. This will result in the choice of p1 inside the first partition component and p2 inside the second component. Thus the optimal solution would have been optained.
A visual representation was given in 2.1

A real life example is taken from the *berlin52* instance:
Real life example:
In this example, Two solutions were selected for mating.
Parent1 had a fitness value of 7777, and Parent2 had a fitness value of 7658.
In the following image, the union graph of the two parents are shown, as well as how to partition the graph. For simplicity, surrogate edges are compressed.
In the offspring, the edges of parent2 are followed in the largest component, and the edges of parent1 are followed in the smallest component. Together this gives a fitness of 7544. Which is the optimal result for this instance (given my distance function).

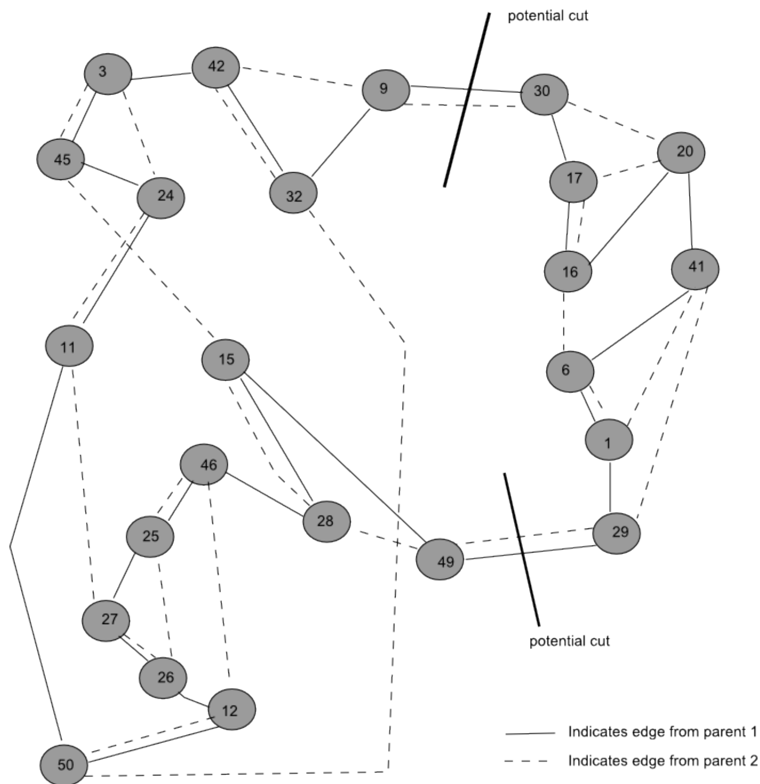


Figure 5.1: a real life example of the GPX operator

Since GPX adheres to principles of [RSJJ94], the hybrid algorithm needs to apply measures for preserving diversity and for introducing new edges into the population. The author explains the need for diversity, in [WHH10], to keep good edges from solutions that have relative low fitness value, and thus would be lost in an elitist survival selection procedure. Diversity is maintained by combining elitist selection with using the strategy of diversity selection. From the testresults this seems to allow it to escape the local minima the other algorithms get stuck in.

The use of double bridge moves as mutation operator is a good choice for this algorithm (it was originally introduced in conjunction with LK-search, when constructing chained-Lin Kernighan). By breaking up the paths at 3 different places and recombining them, 4 new edges are introduced. there is a chance that a suboptimal subpath needed more than the maximum depth of lk-search number of swaps, in order to be improved, now can be handled by the algorithm, thus allowing LK-search to turn a suboptimal sequence into a better

(sub)optimal sequence. Although the combined solution might be worse it will have introduced an important building block that can be used by GPX in the next generation.

Since breaking such a subpath in a large problem instance, would have a low probability it might take an extended number of generations to be able to reach the global optimum if it depends on the double bridge move in order to escape a local minimum.

The implementation of GPX included some tricky sub-algorithms and datastructure choices, including the need for a working implementation of the LK-heuristic. Although it should be easy to reproduce if given a detailed algorithmic description, it is definitely more difficult to implement than the other operators. A final remark should be made, that even though I fail to reach the same running rimes as the authors, this is most likely due to the unoptimized version of Lin Kernighan I use.

5.1.5.1 Summarizing thoughts on the tested operators

The testing results showed that GPX was the most effective algorithm followed by SCX and then OX. It also showed that injecting local optima into SCX and OX improved performance.
This indicates that operators adhering to the principles of [RSJJ94], yield the best results, as GPX adheres fully, while SCX does it partly and OX does not focus on it.
Another common property is that crossover seems to be more effective when given local optima as starting points. It is especially clear in the operator where it is designed with this in mind (GPX), but both OX and SCX benefits from it as well which can be seen in the injected versions of both. Local search algorithms like LK-search will be a good choice for providing these local optima.
In cases where local search cannot escape a local optimum or if some edges appearing in the global optimum is missing, a mutation operator is necessary. A mutation operator like double bridge move is effective in the current case because of the inherent limitation in LK-search (as described in the previous section 5.1.5.
Diversity is important in multi-modal fitness landscapes like TSP as argued in [FOSW09] and [WHH10], in the hybrid-GPX algorithm it was incorporated using diversity selection, which, in the testresults, appeared to meet its purpose. Unfortunately it was not clear whether injection had an impact on the

performance due to diversity. Injecting unimproved random solutions, seemed to be detrimental to overall performance. However the appearent succesful use of diversity selection in GPX indicates that diversity measures are important, but that they may vary depending on used operator and overall algorithm.
From the survey and the tests, the following design features seems to be important for the performance of a good GA/hybrid algorithm for tsp:

- Full adherence to principles of alleless transmission and respectfulness
- The use of local search heuristics to ensure there are good candidates for mating during crossover
- An efficient diversity scheme, helping to ensure that edges in the global optimum is present in the population
- Adding an operator capable of exploration, like a mutation operator, that is capable of breaking the algorithm out of a local minimum that neither the crossover or the local search heuristic can escape from.

If the purpose is not to solve large TSPs, and/or in circumstances where extremely fast running times is crucial, one might consider alternative options. Based on the tests of SCX and OX, especially their ability to hit the global optimum at least one in ten runs, combined with the claim that they are easy to implement, they might be an interesting choice. Especially if the financer is reluctant to spend many ressources on it.

A real life example where this would be feasible could be the following case: A small internetbased company delivers to a number of (different) customers every day using a single truck. To minimize the delivery time, the shortest round trip is desired. This should be calculated every morning and given to the driver.
Using a quick implementation of SCX, focus can be spend on specifying an effective cost function, and thus the company can relatively quick get a good tsp-solver for their purpose.

5.2 Potential of genetic algorithms

How do GA/hybrid GAs perform compared to other algorithm types used on TSP? As explaned earlier I was unable to hit the fast running times claimed by GPX in [WHH10], But it seems that in certain cases a good crossover operator

can improved on performance of local-search algorithms, proving the original idea by Radcliffe and Surry [RSJJ94]. In [WHH10], a GPX hybrid algorithm was claimed to improve on Chained Lin-Kernighan.
Keld Heldsgaun [Hel00] presented an implementation of the LK-heuristic that, by his claimed results, far outperform those results stated for GPX and chained LK in [WHH10]. One could argue that a combination of this very fast LK-implementation used in a hybrid algorithm might produce even better results.
Whitley, one of the authors of the GPX, recently tried to do this in [Hel09], where he was able to improve the performance of the LK-Helsgaun on 'clustered instances' where it was known to struggle.
Other known approaches includes cutting-planes and branch & bound techniques. Those were out of the scope of this project, so I have not obtained results from those.
It seems clear however that pure GAs, in the way that they are traditionally presented in the field of evolutionary computation, are infeasible compared to other methods. This would seem to indicate that the focus within the evolutionary computing community working on tsp's, should be on constructing/inventing crossover operators that support and supplement state-of-the-art local search techniques, integrating it into hybrid algorithms.

5.3 Conclusion

In this project I have considered a large number of different crossover operators for the tsp-problem, that I have been able to find. I have tried to compare them and to provide an overview of the performance of all operators.
I selected 3 of those crossover operators, that seemed to be among the best. The Order, Sequential Constructive and General Partition crossover operator. Those were implemented and thoroughly analysed using statistical and numerical tests and qualitative observations of the solution process.
SCX was found to be better than OX, and GPX was found to be the best operator based on quality of solutions. GPX was considerably faster than the other tested algorithms.
Considered to be important for multimodal fitness landscapes, I tested the effect of some diversity measures. GPX preserves diversity through diversity selection, which seems to work well. I tried using injection strategies on the algorithms. Injection by itself did not enhance performance, but if injection were done with solutions improved by local search heuristics, the performance were significantly improved.
In the last chapter I have suggested some general guidelines for design of crossover algorithms for tsp based on my findings in literature and the performed tests.

The results of the qualitative study of the three operators combined with principles established from the literature study, indicates that the traditional crossover concepts does not work optimally on tsp.
Attention should be spend on constructing 'hybrid-algorithms' combining search heuristics, that quickly can generate good search points, and a crossover operator that can recombine these search point to obtain new better search points. It has been shown in studies that this approach can improve on the solutions obtained by current state-of-the-art algorithms.

Bibliography

[ACR03] David Applegate, William Cook, and André Rohe. Chained lin-kernighan for large traveling salesman problems. *INFORMS J. on Computing*, 15(1):82–92, January 2003.

[Ahm10] Zakir H. Ahmed. Genetic algorithm for the traveling salesman problem using sequential constructive crossover operator. *International Journal of Biometrics & Bioinformatics (IJBB)*, 3, 2010.

[AS] CWI Amsterdam Alexander Schrijver. On the history of combinatorial optimization (till 1960). http://homepages.cwi.nl/~lex/files/histco.pdf.

[CS96] S. Chen and S. Smith. Commonality and genetic algorithm, technical report cmu-ri-tr-96-27, 1996.

[Dav85] Lawrence Davis. Applying adaptive algorithms to epistatic domains. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'85, pages 162–164, San Francisco, CA, USA, 1985. Morgan Kaufmann Publishers Inc.

[FOSW09] Tobias Friedrich, Pietro S. Oliveto, Dirk Sudholt, and Carsten Witt. Analysis of diversity-preserving mechanisms for global exploration*. *Evol. Comput.*, 17(4):455–476, December 2009.

[GL85] D. E. Goldberg and R. Lingle. Alleles, loci, and the traveling salesman problem. In *Proc. of the International Conference on Genetic Algorithms and Their Applications*, pages 154–159, Pittsburgh, PA, 1985.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[GR] Universität Heidelberg Gerhard Reinelt. Tsp-lib. http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/.

[Hel00] Keld Helsgaun. An effective implementation of the lin-kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126:106–130, 2000.

[Hel09] K Helsgaun. General k-opt submoves for the lin–kernighan tsp heuristic, math. programming comput., v1, pp. 119-163,, 2009.

[Hol75] J.H. Holland. Adaptation in natural and artificial systems. 1975.

[HWH12] Doug Hains, Darrell Whitley, and Adele E. Howe. Improving lin-kernighan-helsgaun with crossover on clustered instances of the tsp. In Carlos A. Coello Coello, Vincenzo Cutello, Kalyanmoy Deb, Stephanie Forrest, Giuseppe Nicosia, and Mario Pavone, editors, *PPSN (2)*, volume 7492 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 2012.

[Jak10] Koji Jakudo. A survey on crossover for tsp - japanese presentation. http://www.iba.t.u-tokyo.ac.jp/~jaku/pdf/rindoku101109_slide.pdf, 2010.

[KG11] Daniel Karapetyan and Gregory Gutin. Lin-kernighan heuristic adaptations for the generalized traveling salesman problem. *European Journal of Operational Research*, 208(3):221–232, 2011.

[OSH87] I. M. Oliver, D. J. Smith, and J. R. C. Holland. A study of permutation crossover operators on the traveling salesman problem. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application*, pages 224–230, Hillsdale, NJ, USA, 1987. L. Erlbaum Associates Inc.

[RBP05] Shubhra Sankar Ray, Sanghamitra Bandyopadhyay, and Sankar K. Pal. New genetic operators for solving tsp: Application to microarray gene ordering. In Sankar K. Pal, Sanghamitra Bandyopadhyay, and Sambhunath Biswas, editors, *PReMI*, volume 3776 of *Lecture Notes in Computer Science*, pages 617–622. Springer, 2005.

[RBSMBT] M. Rajabi Bahaabadi, A. Shariat Mohaymany, M. Babaei, and AWT_ TAG. An efficient crossover operator for traveling salesman problem. *Iran University of Science & Technology*, 2.

[RSJJ94] Nicholas J. Radcliffe, Patrick D. Surry, Eh Jz, and Eh Jz. Fitness variance of formae and performance prediction, 1994.

[SS11] Monica Sehrawat and Sukjvir Singh. "modified order crossover (ox) operator". Computer Science & Engineering, Intl. Journal, page 2019, 2011.

[Uni] George Washington University. 2-opt, and lk descriptions. http://www.seas.gwu.edu/~simhaweb/champalg/tsp/tsp.html.

[WHH09] Darrell Whitley, Doug Hains, and Adele Howe. Tunneling between optima: Partition crossover for the traveling salesman problem. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, GECCO '09, pages 915–922, New York, NY, USA, 2009. ACM.

[WHH10] Darrell Whitley, Doug Hains, and Adele E. Howe. A hybrid genetic algorithm for the traveling salesman problem using generalized partition crossover. In Robert Schaefer, Carlos Cotta, Joanna Kolodziej, and Günter Rudolph, editors, *PPSN (1)*, volume 6238 of *Lecture Notes in Computer Science*, pages 566–575. Springer, 2010.

[WRE+98] J. Watson, C. Ross, V. Eisele, J. Denton, J. Bins, C. Guerra, and D. Whitley A. Howe. The traveling salesrep problem, edge assembly crossover, and 2-opt. In *of Lecture Notes in Computer Science*, pages 823–832. Springer, 1998.

[WSF89] L. Darrell Whitley, Timothy Starkweather, and D'Ann Fuquay. Scheduling problems and traveling salesmen: The genetic edge recombination operator. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 133–140, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.