

Linköping Studies in Science and Technology. Dissertations.  
No. 1580

# **Methods for Network Optimization and Parallel Derivative-free Optimization**

**Per-Magnus Olsson**



**Linköping University  
INSTITUTE OF TECHNOLOGY**

Financial support has been received from the Swedish Foundation  
for Strategic Research in the Proviking EDOp project.

Department of Mathematics  
Linköping University  
SE-581 83 Linköping, Sweden  
Linköping 2014

Linköping Studies in Science and Technology. Dissertations.  
No. 1580

**Methods for Network Optimization and Parallel Derivative-free Optimization**

Per-Magnus Olsson

*per-magnus.olsson@liu.se*

*www.mai.liu.se*

*Division of Optimization*

*Department of Mathematics*

*Linköping University*

*SE-581 83 Linköping*

*Sweden*

ISBN 978-91-7519-388-5

ISSN 0345-7524

Copyright © 2014 Per-Magnus Olsson

Printed by LiU-Tryck, Linköping, Sweden 2014

*In memory of my parents*



---

## Abstract

This thesis is divided into two parts that each is concerned with a specific problem.

The problem under consideration in the first part is to find suitable graph representations, abstractions, cost measures and algorithms for calculating placements of unmanned aerial vehicles (UAVs) such that they can keep one or several static targets under constant surveillance. Each target is kept under surveillance by a surveillance UAV, which transmits information, typically real time video, to a relay UAV. The role of the relay UAV is to retransmit the information to another relay UAV, which retransmits it again to yet another UAV. This chain of retransmission continues until the information eventually reaches an operator at a base station.

When there is a single target, then all Pareto-optimal solutions, i.e. all relevant compromises between quality and the number of UAVs required, can be found using an efficient new algorithm. If there are several targets, the problem becomes a variant of the Steiner tree problem and to solve this problem we adapt an existing algorithm to find an initial tree. Once it is found, we can further improve it using a new algorithm presented in this thesis.

The second problem is optimization of time-consuming problems where the objective function is seen as a black box, where the input parameters are sent and a function value is returned. This has the important implication that no gradient or Hessian information is available. Such problems are common when simulators are used to perform advanced calculations such as crash test simulations of cars, dynamic multibody simulations etc. It is common that a single function evaluation takes several hours.

Algorithms for solving such problems can be broadly divided into direct search algorithms and model building algorithms. The first kind evaluates the objective function directly, whereas the second kind builds a model of the objective function, which is then optimized in order to find a new point where it is believed that objective function has a good value. Then the objective function is evaluated in that point.

Since the objective function is very time-consuming, it is common to focus on minimizing the number of function evaluations. However, this completely disregards the possibility to perform calculations in parallel and to exploit this we investigate different ways parallelization can be used in model building algorithms. Some of the ways to do this are to use several starting points, generate several new points in each iteration, new ways of predicting a point's value and more.

We have implemented the parallel extensions in one of the state of the art algorithms for derivative-free optimization and report results from testing on synthetic benchmarks as well as from solving real industrial problems.



## **Populärvetenskaplig sammanfattning**

Avhandlingen består av två delar. Den första delen behandlar problem relaterade till obemannade flygfarkoster (UAVer) och handlar om att hitta platser där dessa kan placeras för att filma ett eller flera mål. I fallet med ett mål bildas en kedja där en UAV övervakar målet och sänder filmen till en annan UAV, vilken sänder filmen till en tredje, som i sin tur sänder den till en fjärde o.s.v. Så fortgår det tills filmen når en basstation där en användare tar emot den. I fallet med flera mål vill man t.ex. hitta placeringar av UAVerna så att det krävs så få UAVer som möjligt för att övervaka alla mål samtidigt.

Att placera UAVerna har två delproblem. Det första är att bedöma vad som är lämpliga positioner, samt att avgöra hur bra det fungerar att sända film mellan positionerna, om det överhuvudtaget bedöms som möjligt. Det andra problemet är att hitta effektiva metoder för att beräkna UAVernas placering. I avhandlingen presenteras nya metoder för att beräkna var UAVerna ska placeras, både i fallet med ett mål och i fallet med flera mål. De utvecklade metoderna har implementerats i ett system för obemannade flygfarkoster.

Avhandlingens andra del handlar om optimering av tidskrävande problem, där funktionen som ska optimeras ses som en svart låda, eftersom man inte vet hur beräkningarna görs. Sådana problem uppkommer ofta i samband med utveckling av mekaniska produkter. Ofta görs en mycket stor mängd beräkningar, vars resultat beror på varandra. För att få fram en så bra produkt som möjligt, görs många och noggranna beräkningar, vilket leder till att beräkningarna tar mycket lång tid. Det är inte ovanligt att en enda beräkning av funktionen tar flera timmar. Optimering används för att hitta den bästa produkten, givet att denna ska uppfylla vissa krav. Eftersom man vet så lite om funktionen så använder optimeringsalgoritmen modeller av funktionen, vilka anses giltiga inom ett begränsat område.

Eftersom varje beräkning är mycket tidskrävande är det vanligt att man fokuserar på att minimera antalet gånger funktionen beräknas. Betänker man dock att moderna datorer kan utföra flera beräkningar parallellt inser man att det viktiga är att minska antalet gånger en eller flera punkter beräknas parallellt. Om vi kan beräkna fem punkter parallellt istället för två sekventiellt har vi inte bara sparat tid utan också fått mera information om funktionen.

I avhandlingen presenterar vi olika sätt att minska tiden för att uppnå ett bra resultat genom att utföra beräkningarna parallellt. Vi undersöker hur man kan bygga modellen parallellt, använda flera olika startpunkter samtidigt, samt hur man kan generera flera punkter vilkas värden beräknas parallellt. Vidare diskuterar vi hur man kan försöka att skapa synergieffekter mellan de olika startpunktarna och olika sätt att implementera parallellisering och vad de får för effekter på optimeringsalgoritmen.

Vi har implementerat en algoritm för att lösa problem enligt ovan samt de nya parallella utökningarna. Denna har använts för att lösa såväl testproblem som att lösa industriella problem.



## Acknowledgements

I would like to thank my current supervisors professors Kaj Holmberg and Dag Fritzson as well as my former supervisors professor Patrick Doherty and associate professor Jonas Kvarnström.

Thanks to Fredrik, Martin, Oleg, and Torbjörn for valuable discussions.

I would also like to thank former and current co-workers: David, Elina, Fredrik, Gi-anpaolo, Hongmei, Håkan, Kristian, Mariusz, Martin, Mikael C, Mikael N, Nisse, Olof, Per, Piotr, Spartak, Tommy and Åsa.

A thanks to the administrators Anne Moe and Theresia Carlsson-Roth for guidance in the bureaucratic maze of graduate studies.

Thanks to Anders and Wai at Frontway.

A special thanks to Vadim Engelson at Mathcore/Wolfram for the integration of the developed optimization software into SKF's simulator.

Thanks to the personnel at SKF Engineering & Research Centre in Gothenburg for tolerating my constant questions.

Thanks also to the anonymous interview subjects at SKF worldwide for giving me some of their valuable time.

Last but not least, thanks to friends and enemies for inspiration.



# Thesis Introduction

This thesis consists of two parts, where the first part involves network optimization and the second part involves derivative-free optimization.

The problem under consideration in the first part is to find suitable graph representations, abstractions, cost measures and algorithms for calculating placements of unmanned aerial vehicles (UAVs) such that they can keep one or several static targets under constant surveillance. Each target is kept under surveillance by a surveillance UAV, which transmits information, typically real time video, to a relay UAV. The role of the relay UAV is to retransmit the information to another relay UAV, which retransmits it again to yet another UAV. This chain of retransmission continues until the information eventually reaches a base station where a human operator receives it.

The basic problem of finding placements for the UAVs when there is a single target, is a shortest path problem, which is quite easy to solve. However, that problem does not care about the number of UAVs required to surveil the target. Since there typically is a limited number of UAVs available, combined with a desire to maintain a high quality connection from the surveillance UAV to the base station, this often leads to compromises between the number of UAVs in a chain and the maximum achievable quality. This is a multi-objective optimization problem, that we solve using any of two new algorithms that calculate the complete set of Pareto-optimal solutions.

When there are several targets, which shall be kept under simultaneous surveillance, then the problem of finding suitable UAV positions is a variant of the Steiner tree problem. Due to limitations of the UAVs, there are additional constraints on the Steiner tree, causing the graph to be incomplete and the triangle inequality to not apply. This voids the theoretical guarantees of most Steiner tree algorithms. To solve the problem of finding suitable UAV positions, we adapt the cheapest path heuristic. Once the initial tree is calculated, it can be further improved using a new algorithm that is also presented in this thesis.

Testing of the algorithms has been performed in both virtual environments as well as networks corresponding to real environments. The developed algorithms have been integrated in a system of unmanned aerial vehicles.

The first part of this thesis has been published as the licentiate thesis “Positioning Algorithms for Surveillance Using Unmanned Aerial Vehicles”, thesis number 1476 in Linköping Studies in Science and Technology. There are some minor, mainly typographical, differences in the version presented here, as to that version.

In the second part of this thesis, we investigate algorithms for solving problems where the analytical expression of the objective function is unknown. Such problems occur naturally in simulation driven optimization, which is often used in the development of various mechanical products. The equations used in such simulations are often so complicated that it is impossible to extract an explicit objective function. Since we cannot extract the objective function, we are consequently unable to obtain gradient and Hessian information. Each simulation is equivalent to an evaluation of the objective function, and is often very time-consuming. A single evaluation may take several hours to perform.

---

For solving such problems there are two main types of algorithms available: direct search algorithms and model building algorithms. The first kind evaluates the objective function directly, whereas the second kind builds a model of the objective function, which is then optimized in order to find a new point where it is believed that objective function has a good value. Then the objective function is evaluated in that point. Since the model building algorithms often offer better performance, we concentrate on them in this thesis.

Since the objective function is very time-consuming, it is common to focus on minimizing the number of function evaluations. However, this completely disregards that new computer hardware is typically able to perform several calculations in parallel. If we can perform e.g. five calculations in parallel instead of two in sequence, we not only save time but we also gain more information about the objective function. Thus, it is not decreasing the *number* of function evaluations that is the primary objective, rather it is the *number of times* one or more points are evaluated, that shall be decreased, if this can be done without impairing the objective function value.

With this important distinction clarified, we investigate how parallelization can be used in model building algorithms. Examples of such parallel extensions include using several starting points, generating several new points in each iteration and evaluating each new point in several different models to get a more accurate prediction of the point's value. Additional improvements are storing all evaluated points in a cache, parallel model building, and different variable transformations with the intention of allowing longer steps in more promising direction as well as attempting to make the problem of finding the minimal point easier.

A higher level optimization control has also been developed. It attempts to determine the most promising starting points, terminate the remaining and create new ones from the surviving, but with slightly different parameters, in order to either explore larger parts of the search space or concentrate on finding the optimum, depending on the stage of optimization. This is also an attempt to move in the direction of global optimization. To do this, we again take advantage of parallelization.

In this area, with the wide range of industrial applications, the implementation is very important. We have made a high quality implementation of one of the best known algorithms in the area, which we have extended with the previously described features. Using that implementation, we have performed benchmarking on synthetic test cases as well applied the algorithm on industrial problems. Two companies have used our parallel version of the algorithm, and it has also been incorporated into one of the companies' simulator software.

---

# Contents

<b>I Contributions to Network Optimization</b>	<b>1</b>
<b>1 Introduction</b>	<b>3</b>
1.1 Thesis Contributions . . . . .	6
1.2 Publications . . . . .	7
1.3 Thesis Outline . . . . .	7
<b>2 Related Work</b>	<b>9</b>
2.1 UAVs As Relays . . . . .	9
2.2 Single Target . . . . .	10
2.3 Multiple Targets . . . . .	11
2.4 Area Coverage . . . . .	12
2.5 Exploration . . . . .	13
2.6 Ad-hoc Networks and Wireless Sensor Networks . . . . .	14
<b>3 The Relay Positioning Problems</b>	<b>15</b>
3.1 Problem Setup . . . . .	15
3.2 Definitions of the Single Target Relay Problems . . . . .	16
3.3 Cost Functions . . . . .	17
3.3.1 Transmission Quality . . . . .	19
3.3.2 Position Visibility . . . . .	20
3.3.3 Minimum Free Angle Between Positions . . . . .	20
3.3.4 Minimum Distance to Obstacles . . . . .	22
3.3.5 Surveillance Cost Functions . . . . .	22

---

3.4	Reachability Functions . . . . .	23
3.5	Problem Properties . . . . .	24
3.6	Continuous Solution Methods . . . . .	25
3.7	Summary . . . . .	27
<b>4</b>	<b>Environment Representation and Discretization</b>	<b>29</b>
4.1	Discretization and Graph Creation . . . . .	29
4.2	Fixed-Size Grids . . . . .	31
4.3	Octrees . . . . .	32
4.4	Expanded Geometry Graphs . . . . .	34
4.5	Voronoi Diagrams . . . . .	35
4.6	Discretization Methods Used in Motion Planning . . . . .	35
4.7	Summary . . . . .	36
<b>5</b>	<b>Relay Positioning Algorithms for Single Target Problems</b>	<b>39</b>
5.1	Existing Algorithms for the <b>STR-ParetoLimited</b> Problem . . . . .	40
5.2	A New Label-Correcting Algorithm . . . . .	42
5.2.1	Preliminaries . . . . .	42
5.2.2	Algorithm Details . . . . .	44
5.2.3	Correctness Proof . . . . .	47
5.2.4	Time Complexity . . . . .	48
5.2.5	Improved Preprocessing . . . . .	49
5.2.6	Example . . . . .	49
5.3	A New Dual Ascent Algorithm . . . . .	51
5.3.1	Algorithm Details . . . . .	52
5.3.2	Theoretical Properties . . . . .	53
5.3.3	Example . . . . .	55
5.3.4	Performance Improvements . . . . .	57
5.4	Summary . . . . .	58
<b>6</b>	<b>Relay Positioning for Multiple Targets</b>	<b>61</b>
6.1	Definition of the Multiple Target Relay Problems . . . . .	61
6.2	Relation to Steiner Tree Problems . . . . .	62
6.2.1	Continuous Steiner Trees . . . . .	63
6.2.2	Discrete Steiner Trees . . . . .	64
6.3	Adapting the Cheapest Path Heuristic . . . . .	73
6.3.1	Theoretical Properties . . . . .	74
6.3.2	Extensions . . . . .	76
6.4	Calculating Pareto-optimal Relay Trees For Two Targets . . . . .	77
6.4.1	Determining the Set of Pareto-optimal Relay Trees . . . . .	79
6.4.2	Duplicate Edges in the Relay Tree . . . . .	80
6.5	Improving Relay Trees . . . . .	81
6.5.1	Reduced Trees . . . . .	82

---

6.5.2	Choosing Subtrees for Optimization . . . . .	84
6.5.3	Different Tree Structures . . . . .	87
6.5.4	Collisions Between Trees . . . . .	88
6.5.5	Algorithm Details . . . . .	90
6.5.6	Time Complexity . . . . .	92
6.6	Summary . . . . .	92
<b>7</b>	<b>Implementation and Experimental Results</b>	<b>95</b>
7.1	Software Architecture . . . . .	95
7.2	Problem Setup for Empirical Testing . . . . .	99
7.3	Pareto-Optimal Relay Chains . . . . .	103
7.4	Optimal Chains Using At Most $M$ UAVs . . . . .	106
7.5	Relay Trees . . . . .	110
<b>8</b>	<b>Discussion</b>	<b>119</b>
8.1	Future Research . . . . .	120
<b>II</b>	<b>Contributions to Derivative-free Optimization</b>	<b>131</b>
<b>9</b>	<b>Introduction</b>	<b>135</b>
9.1	Thesis Contributions . . . . .	137
9.2	Presentations . . . . .	137
9.3	Software . . . . .	137
9.4	Thesis Outline . . . . .	138
<b>10</b>	<b>Setting And Problem Description</b>	<b>139</b>
10.1	Frontway . . . . .	139
10.2	SKF . . . . .	140
10.2.1	SKF BEAST . . . . .	140
10.2.2	BEAST Workflow . . . . .	141
10.3	Distinguishing Problem Characteristics . . . . .	143
10.4	Summary . . . . .	146
<b>11</b>	<b>Direct Search Algorithms</b>	<b>147</b>
11.1	Initial Step Length and Termination Criteria . . . . .	147
11.1.1	Initial Step Length . . . . .	147
11.1.2	Termination Criteria . . . . .	148
11.2	Nelder-Mead Simplex . . . . .	148
11.2.1	Nelder-Mead Simplex Pseudocode . . . . .	148
11.2.2	Termination Criteria . . . . .	149
11.2.3	Improvements . . . . .	149
11.2.4	Extensions . . . . .	150

11.3	Subplex . . . . .	150
11.3.1	Subspaces . . . . .	151
11.3.2	Step Length . . . . .	151
11.3.3	Termination Criteria . . . . .	151
11.3.4	Handling of Measurement Errors . . . . .	152
11.3.5	Subplex Pseudocode . . . . .	152
11.4	The r-Algorithm . . . . .	153
11.4.1	r-Algorithm Pseudocode . . . . .	153
11.4.2	Termination Criteria . . . . .	154
11.5	Summary . . . . .	154
<b>12</b>	<b>Model Building Algorithms</b>	<b>157</b>
12.1	Building the Interpolation Model . . . . .	158
12.1.1	Polynomial Bases . . . . .	158
12.1.2	Natural Basis . . . . .	159
12.1.3	Polynomial Interpolation . . . . .	159
12.1.4	Lagrange Polynomials . . . . .	160
12.1.5	Newton Polynomials . . . . .	162
12.1.6	Measurement Errors or Noise in the Objective Function . . . . .	162
12.2	Poisedness . . . . .	163
12.2.1	Measures of Poisedness . . . . .	163
12.3	The Trust Region Framework . . . . .	164
12.3.1	Updating the Trust Region Radius . . . . .	165
12.3.2	Adding a Point to the Interpolation Set . . . . .	166
12.3.3	Updating the Interpolation Set . . . . .	166
12.4	Summary . . . . .	167
<b>13</b>	<b>Existing Model Building Algorithms</b>	<b>169</b>
13.1	The DFO Algorithm . . . . .	169
13.1.1	DFO Pseudocode . . . . .	169
13.1.2	Extensions . . . . .	170
13.2	The UOBYQA Algorithm . . . . .	171
13.2.1	UOBYQA Pseudocode . . . . .	171
13.2.2	Extensions . . . . .	175
13.2.3	Summary . . . . .	175
13.3	The NEWUOA Algorithm . . . . .	175
13.3.1	NEWUOA Overview . . . . .	176
13.3.2	NEWUOA Pseudocode . . . . .	177
13.3.3	The BOBYQA Algorithm . . . . .	180
13.3.4	The LINDCOA Algorithm . . . . .	180
13.3.5	Extensions . . . . .	180
13.4	Summary . . . . .	181

---

<b>14 Radial Basis Functions</b>	<b>183</b>
14.1 Model Building for Radial Basis Functions . . . . .	183
14.2 Optimization Algorithms for Radial Basis Functions . . . . .	184
14.3 Comparison With Other Model Building Algorithms . . . . .	185
14.4 Summary . . . . .	186
<b>15 Parallelism in Algorithms for Derivative-Free Optimization</b>	<b>187</b>
15.1 Parallelization Terms . . . . .	188
15.2 Optimization Runs . . . . .	189
15.3 Building the Initial Model in one Step . . . . .	190
15.4 Non-spherical Trust Region Shapes . . . . .	191
15.4.1 Static Trust Region Shapes . . . . .	192
15.4.2 Dynamic Trust Region Shapes . . . . .	192
15.4.3 Other Trust Region Shapes . . . . .	196
15.5 Generating Several Points in Each Iteration . . . . .	196
15.5.1 Solving the Trust Region Subproblem with Different Radii . . . . .	196
15.5.2 Optimistic Polling . . . . .	197
15.5.3 Algorithm Consequences . . . . .	201
15.6 Towards Global Optimization . . . . .	202
15.6.1 Exploration . . . . .	203
15.6.2 Exploitation . . . . .	204
15.6.3 Connections with Meta-Heuristics . . . . .	206
15.7 Summary . . . . .	206
<b>16 Information Sharing</b>	<b>209</b>
16.1 Evaluating a Point in Several Models . . . . .	209
16.2 Cache of Points . . . . .	210
16.3 Determining Order of Evaluation . . . . .	211
16.3.1 Lowest Expected Value . . . . .	211
16.3.2 Lowest Parent Value . . . . .	212
16.3.3 Fewest Points on the Queue . . . . .	212
16.3.4 Choosing Prioritization Criterion . . . . .	212
16.3.5 Adjusting a Point's Queue Position . . . . .	213
16.3.6 Removing a Point from the Queue . . . . .	213
16.4 Summary . . . . .	213
<b>17 Control of Optimization Algorithms</b>	<b>215</b>
17.1 Separability . . . . .	215
17.2 Guarding Against Ill-Conditioned Models . . . . .	216
17.3 Model Building from Existing Points . . . . .	217
17.4 Controlling the Number of Optimization Runs . . . . .	217
17.4.1 Strategies for Terminating Optimization Runs . . . . .	218
17.4.2 Strategies for Creating New Optimization Runs . . . . .	221

17.5 Summary . . . . .	223
<b>18 Implementation Details</b>	<b>225</b>
18.1 Choice of Algorithm . . . . .	225
18.2 Choice of Third-Party Software . . . . .	226
18.3 Implementation Principles . . . . .	226
18.4 Termination Criteria . . . . .	228
18.5 Implementation of Parallelization . . . . .	229
18.6 Software Architecture . . . . .	229
18.7 Integration With BEAST . . . . .	230
18.7.1 Asynchronous Parallelization . . . . .	231
18.7.2 Constraint Handling . . . . .	232
18.7.3 Implicit Constraints . . . . .	233
18.7.4 Integer variables . . . . .	233
18.7.5 Visualization of Results . . . . .	234
18.8 Summary . . . . .	238
<b>19 Testing on Synthetic Test Cases</b>	<b>241</b>
19.1 Test Case Specifications . . . . .	241
19.2 Test Case and Algorithm Settings . . . . .	243
19.3 Interpreting the Test Results . . . . .	244
19.4 Building the Initial Model in two Steps . . . . .	245
19.5 Effect of a Cache of Evaluated Points . . . . .	246
19.6 Building the Initial Model in One Step . . . . .	249
19.7 Different Values of the Initial Trust Region Radius . . . . .	251
19.8 Non-spherical Trust Region Shapes . . . . .	253
19.8.1 Switching Elliptical Trust Region . . . . .	254
19.8.2 Elliptical Trust Regions . . . . .	255
19.8.3 Circular Level Curves . . . . .	259
19.9 Testing of Parallel Extensions . . . . .	260
19.9.1 Evaluating Points in Several Models . . . . .	260
19.9.2 Solving the Trust Region Subproblem With Different Radii . . . . .	263
19.9.3 Generation of Several Points in a Plus Sign Pattern . . . . .	265
19.9.4 Generation of a Fixed Number of Points . . . . .	267
19.9.5 Several Starting Points . . . . .	270
19.10 Testing of Strategies for Optimization Control . . . . .	275
19.10.1 Keep Single Best Optimization Run . . . . .	275
19.10.2 Dynamic Weight Adjustment . . . . .	277
19.11 Further Analysis and Discussion . . . . .	280
19.12 Testing with Limited Execution Time . . . . .	290
19.13 New Test Cases . . . . .	292
19.14 Summary . . . . .	298

---

<b>20 Testing On Industrial Test Cases</b>	<b>301</b>
20.1 Test Results from Frontway . . . . .	301
20.2 Test Results from SKF . . . . .	302
20.2.1 Testing on a Two Variable Problem . . . . .	302
20.2.2 Testing on a Four Variable Problem . . . . .	303
20.3 Summary . . . . .	303
<b>21 Future Work</b>	<b>307</b>
<b>22 Conclusions</b>	<b>309</b>
<b>Appendix A Terms</b>	<b>311</b>



## **Part I**

# **Contributions to Network Optimization**



# 1

---

## Introduction

Historically, Unmanned Aerial Vehicles (UAVs) have been used for tasks that are considered “dangerous, dirty and dull”. Tasks can be dangerous if they require flying an aircraft over enemy positions. Dirty tasks may require entering areas contaminated by poison or flying into a radioactive cloud with the intention of collecting samples of radioactive dust. A dull activity is something that a human would quickly grow tired of doing.

An example of an activity that is often considered dull is surveillance, which is an essential aspect in a wide variety of applications, for example search and rescue operations, traffic monitoring, forest fire monitoring, law enforcement and military applications. Although mainly labeled as a dull activity, surveillance can also be dangerous, especially if the *surveillance target* is hostile, or if there is limited information about the area around the target. Improving the performance and decreasing the risk of human injuries and casualties are two of the many reasons for using UAVs for surveillance.

The use of unmanned vehicles for surveillance is not new. Such vehicles have been used throughout large parts of the twentieth century and the types of vehicles used vary greatly: from large semi-stationary airships, through UAVs a few meters in size, to micro UAVs weighing less than a kilogram. With advances in technology, the use of UAVs for surveillance as well as for other tasks is likely to increase.

In many cases, the information that is gathered by surveilling the target must be made available to a ground operator at a *base station* as quickly as possible. As the information may include high volume sensor data such as live video, high uninterrupted bandwidth is desirable. The communication equipment and the properties of the communication channel may restrict where the *surveillance UAV* can be placed. Naturally, it must be positioned in such a way that it can surveil the target, but it must also be able to transmit the sensed information to the base station. To maintain good transmission quality for the high-bandwidth communications required when transmitting live video, common require-

## 1. Introduction

---

ments are line-of-sight and limited distance between the surveillance UAV and the base station, corresponding to the maximum communication range [41].

The line-of-sight requirement can be problematic in mountainous or urban areas. While the problem can be mitigated by increasing the UAV's altitude, this option is not always possible for small UAVs, which in some cases are not able to ascend sufficiently. Although larger UAVs might be able to do this, the airspace may be restricted by aviation authorities, which in some cases makes it impossible for the UAV to achieve the required altitude.

If the UAV is able to ascend to the required altitude and that the airspace is available, the distance between the UAV and the target as well as between the UAV and the base station increases. This can adversely affect the quality of sensed information and the maximum communication distance may also be exceeded. Even if the transmission range is sufficient, communicating to and from such an altitude might require significant transmission power, which can be problematic for smaller UAVs. The communication range is also typically limited, and can be quite short, especially when smaller and lower cost UAVs are used. This is because such UAVs might not be able to carry the most powerful and sophisticated communication equipment due to size and weight constraints.

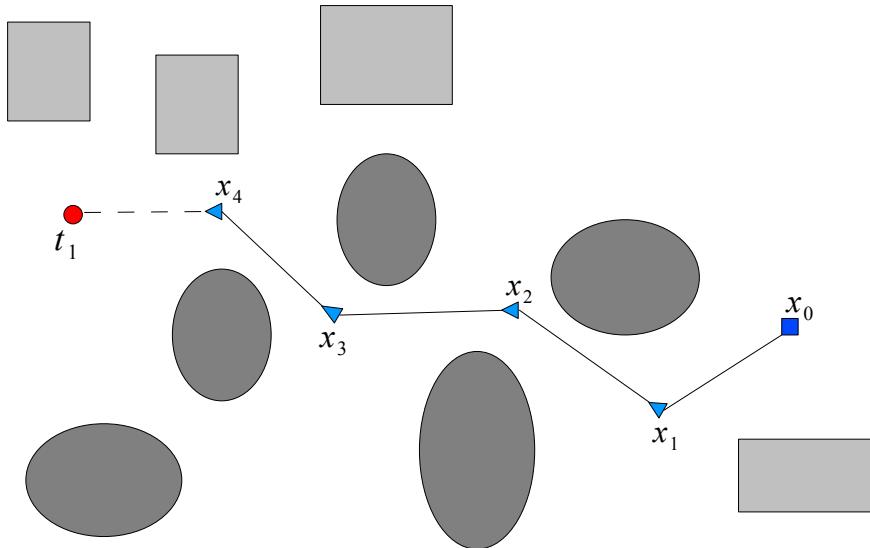
If UAVs are unable to ascend to sufficient altitude, another alternative is to use satellite communication. However, not all organizations have access to such satellites and smaller UAVs might not be able to carry the required equipment.

The above methods for achieving communication between the surveillance UAV and the base station are suitable in some situations, but there are limitations to both methods. An alternative approach is to use one or more communication relays that extend the effective range and forward transmissions around obstacles.

If it is known beforehand where the surveillance target will appear, then the necessary relays can be placed in advance. If the target location is unknown, then a large number of relays can be prepositioned to cover all possible target positions. However, this would probably require many statically placed relays, most of which would not be used, and it limits surveillance to environments where relays are expected to remain for some time.

A more flexible solution is to use UAVs to relay information. This has previously been investigated by several researchers, see e.g. Cerasoli [20] and Pinkney et al. [81]. However, there has been very little research on where the *relay UAVs* should be placed.

This thesis focuses on algorithms for finding suitable positions for such UAVs and gives examples of some of the factors that can be used to distinguish between good and bad positions with regards to UAV placement. For practical reasons, the surveillance UAV is distinguished from the relay UAVs. The surveillance UAV must be equipped with sensors suitable for surveilling the target while the relay UAVs may carry less sophisticated sensors as their task is to relay information. If the relay UAVs are placed correctly, they offer a way to handle both the limited communication range and the line-of-sight requirement. As the distance between the base station and the targets can be quite long, many relay UAVs may be required. For this reason, we aim to describe and develop algorithms that scale well enough to quickly solve problems involving a large number of UAVs in



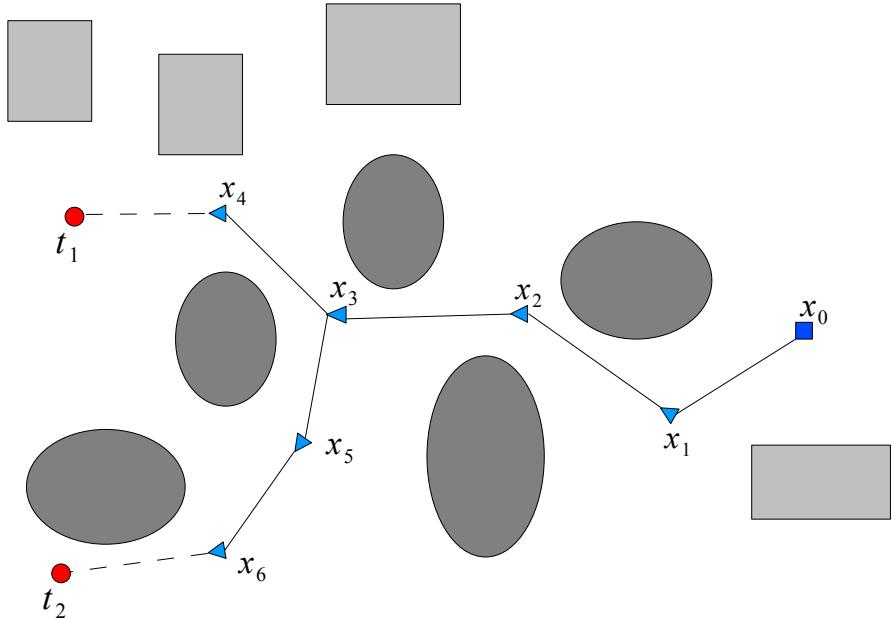
**Figure 1.1:** Example of relay chain with four UAVs. The base station located at location  $x_0$  is connected to the target located at  $t_1$  by the surveillance UAV at  $x_4$  and the relay UAVs at  $x_1-x_3$ .

large areas of operation.

In cases where a single target is surveilled, the UAVs form a *relay chain* (Figure 1.1) between the target and the base station. When there are several targets, calculating separate chains to each would not make the best use of resources. Instead, relays could receive information from several UAVs. This creates a *relay tree*. Such a tree has the root in the base station, the UAVs are the interior nodes and the targets are the leaves. An example is shown in Figure 1.2.

Here we are interested in surveillance of one or more static targets with known locations and as we are looking for positions where the UAVs can be placed, the relay problems are *positioning problems*, not motion planning problems. Algorithms for finding trajectories are not part of this thesis.

The focus in this thesis is on UAVs, especially helicopters that can remain at the same time for a prolonged time. However, the algorithms and concepts presented here work equally well for unmanned ground vehicles (UGVs) as well as for determining placement of other objects used for communication, for example temporary base stations for cellular phone communication in disaster situations.



**Figure 1.2:** A relay tree with two targets.

## 1.1 Thesis Contributions

The main contributions of this thesis include:

- Formalizations of several single and multiple target relay positioning problems, focusing on different objectives and allowing for a large degree of flexibility in modeling surveillance and communication.
- Two different algorithms for solving the different single target relay positioning problems. Both algorithms are based on graph search in a discretized version of the original problem. Each algorithm calculates a set of different relay chains, where each relay chain has a different quality and requires a different number of UAVs. Naturally, a solution requiring a larger number of UAVs is only useful if it has a higher quality. The first algorithm is a label-correcting algorithm that is capable of solving several different relay positioning problems. The second algorithm is focused on the problem of finding the highest quality solution given a limited number of available UAVs.
- A discussion on how the multiple target problems can be modeled to be solved efficiently as well as a theoretical study and discussion of what algorithms are suitable for solving the multiple target problems. We generalize an existing heuristic to fit

the requirements of the multiple target relay positioning problems and present two other algorithms that can be used for different problems involving relay trees for multiple targets. The first of the algorithms for the single target relay problem is used to solve several different multiple target relay positioning problems involving a base station and two targets. The same algorithm is then generalized and used as a heuristic in order to improve existing relay trees with respect to factors such as the number of UAVs required to realize the tree or the quality of the tree.

## 1.2 Publications

Parts of this thesis have previously been presented in the following publications and reports:

- [15] Oleg Burdakov, Patrick Doherty, Kaj Holmberg, Jonas Kvarnström, and **Per-Magnus Olsson**. Positioning unmanned aerial vehicles as communication relays for surveillance tasks. In *Proceedings of the 2009 Conference on Robotics: Science and Systems (RSS)*, pages 257–264, 2009.
- [16] Oleg Burdakov, Patrick Doherty, Kaj Holmberg, Jonas Kvarnström, and **Per-Magnus Olsson**. Relay positioning for unmanned aerial vehicle surveillance. *International Journal of Robotics Research*, 29(8):1069–1087, 2010.
- [17] Oleg Burdakov, Patrick Doherty, Kaj Holmberg, and **Per-Magnus Olsson**. Optimal placement of UV-based communications relay nodes. *Journal of Global Optimization*, 48(4):511–531, 2010.
- [18] Oleg Burdakov, Kaj Holmberg, and **Per-Magnus Olsson**. A dual ascent method for the hop-constrained shortest path with application to positioning of unmanned air vehicles. Technical Report LiTH-MAT-R-2008-07, Linköping University, Department of Mathematics, 2008.
- [34] Patrick Doherty, Jonas Kvarnström, Fredrik Heintz, David Landén, and **Per-Magnus Olsson**. Research with collaborative unmanned aircraft systems. In *Proceedings of the Dagstuhl Workshop on Cognitive Robotics*, 2010.
- [77] **Per-Magnus Olsson**, Jonas Kvarnström, Patrick Doherty, Oleg Burdakov, and Kaj Holmberg. Generating UAV communication networks for monitoring and surveillance. In *Proceedings of the International Conference on Control, Automation, Robotics and Vision (ICARCV)*, 2010.

## 1.3 Thesis Outline

Related work is discussed in Chapter 2. In Chapter 3, problem definitions for the single target relay problems are presented, as well as reachability functions for determining

## **1. Introduction**

---

whether communication and surveillance can take place and cost functions for modeling the cost of such communication or surveillance. Different discretization options are discussed in Chapter 4. Several algorithms for solving relay positioning problems involving a single target are shown in Chapter 5. Chapter 6 defines the multiple target relay problems and presents algorithms suitable for solving the problems. An overview of our implementation and integration into a simulator system as well as experimental results are described in Chapter 7. The conclusions are presented in Chapter 8.

# 2

---

## Related Work

This chapter presents related work in the areas of relay placement and other related areas. Most previous work has been for UGVs, and the amount of work involving UAVs is somewhat limited.

As the problems of interest here are positioning problems, research in motion planning is not part of the related work. The general use of UAVs for surveillance is not included due to limited overlap with the research presented here.

The chapter is structured as follows. Section 2.1 provides an overview of the use of UAVs as relays. Next, Section 2.2 describes different ways to solve relay positioning problems for a single target. Different methods to provide relay trees to multiple targets are the topic of Section 2.3. Section 2.4 describes some different ways in which a UAV has been used to provide coverage of an area. Section 2.5 describes how ground robots have been used for exploration, while at the same time maintaining a connected network. Finally, Section 2.6 describes the relation between ad-hoc networks, wireless networks and the relay positioning problems. That section also provides an overview of how UAVs can be used in conjunction with ad-hoc networks.

### 2.1 UAVs As Relays

The concept of using a UAV as a communications relay in military applications is discussed in Pinkney et al. [81]. The work exclusively considers one UAV acting as a relay between users on the ground and the focus is on different classes of platforms and the different uses of those. Much emphasis is placed on the communications equipment, but no algorithm for UAV placement is presented.

A possible application of UAVs is highway surveillance, and the data link used for transfer of information from a surveilling UAV to a base station has been investigated by

Chen et al. [24]. Although it is mentioned that the UAV moves to keep large areas under surveillance, no algorithms for determining the placement of the UAV or the placement of the base station is performed: the focus is on the data link, including the hardware and its capabilities. An interesting point is that a commercial CDMA network is used for transmitting information from the UAV to the base station. Different bit rates and frame rates are tested to find the highest quality video stream that can be transmitted.

The research performed by Zhan [102] mainly focuses on the performance of communication between relays. However, there is a short discussion about placement of a UAV to enable transmission of information to several users. To communicate with a user, the UAV must be positioned in a circle centered in the user's position. The only described positioning algorithm is to place the UAV in the intersection of several circles. If no such intersection exists, then communication is not possible.

## 2.2 Single Target

In limited cases, a single relay UAV is sufficient to maintain a flow of information to the base station. One such case was investigated by Schouwenaars [86]. A surveillance UAV must fly to a specific position and a single relay UAV is used to maintain a connection with a base station. This problem was formulated as a Mixed-Integer Linear Programming (MILP) problem. The objective was to optimize a cost function while at the same time satisfying certain conditions. Several different conditions were used and presented. One condition was that UAVs were not allowed to fly within a certain distance from obstacles. Another condition was that the distance between the surveillance UAV and the relay UAV was required to be less than a maximum communication range. Similarly, the maximum allowed distance between the relay UAV and a base station was limited. The costs could for example be flight time, fuel consumption or visibility. Both centralized and distributed receding horizon approaches are considered. As the computational complexity increases exponentially with the number of agents, the distributed approach is used. For solving the MILP-problems, the commercial MILP-solver CPLEX was used. The same distributed approach was also used in a scenario with two relay UAVs [87]. It is mentioned that the time required in each iteration is increased when several relay UAVs are used, but the execution time was still within the allotted time interval.

Control behavior for teams of unmanned ground vehicles involving line-of-sight is investigated in Sweeney et al. [90]. In an indoor setting, a lead UGV advances from the base station towards the goal position and incrementally determines where to place relay robots along the way in order to maintain communication with the base station. To enable communication between the robots, line-of-sight (LOS) between them is required and they must be within a certain distance from each other. LOS is estimated in a discretized environment with square grid cells of equal size. Two distances are used: *LOS distance* and *occlusion threshold distance*. If the distance between two robots is less than the LOS distance, then both robots can move freely, assuming that there is line-of-sight

between them. The occlusion threshold distance marks the maximum allowed separation between a pair of robots and cannot be exceeded. If the distance between two robots is more than the LOS distance, the controller of one of the robots is switched off, and that robot remains passive until the other robot comes within LOS distance. By varying the occlusion threshold distance, and how proactive the robots are when trying to maintain line-of-sight, different behaviors are achieved.

A very similar problem is investigated by Nguyen et al. [73]. Several algorithms for positioning UGVs to form a relay chain between the base station and a target are evaluated in terms of energy usage. Initially, all robots are gathered at the base station and then the lead robot advances towards the target. The best-performing algorithm, with respect to energy usage, keeps the other robots at a base station until the lead robot experiences a signal strength below a threshold. When this happens, the lead robot stops and requests a relay UGV to be placed at the lead UGV’s position. The relay moves from the base station to the lead UGV’s position and stays there. This allows the lead UGV to move incrementally towards the target until another position with poor radio signal is encountered. Then a new relay UGV moves from its current position at the base station to the already placed UGV. When it arrives, the already placed relay UGV moves to the position of the lead robot, which is then free to continue. This process is repeated until the target position is reached by the lead UGV. The main disadvantages with the algorithms mentioned above are that they have no theoretical guarantees that the target position is reached, as no *a priori* calculation or evaluation of paths is performed.

Cheng et al. [25] investigated the use of several UAVs for relaying information between a producer and a consumer. They consider the information to be delay-tolerant, which opens up the possibility of transmitting the information from the producer to one UAV, which then flies to the consumer and transmits the information. Two UAVs are used simultaneously. One is delivering information from the producer and one is flying “empty” to the producer to get the next load of information. The authors refer to this procedure as “load-carry-and-deliver”. As we consider the information from the surveillance UAV urgent, this approach cannot be used for the applications discussed in this thesis. However, if the number of relay UAVs is insufficient to form a relay chain, this is one method to get some, albeit delayed, information from the surveillance UAV.

## 2.3 Multiple Targets

Finding suitable positions for UGVs to allow them to find relay chains between a set of sensors (targets) and a base station using potential fields is investigated by Simonetto et al. [88]. The UGVs follow the gradient of the total potential field until a suitable position is found. Several potential field approaches are evaluated, both “standard” and “dynamic”. In the “standard” potential field approach, all robots are influenced by all other robots as well as the sensors. In a “dynamic” approach, the robots change the potential fields to influence other robots to position themselves in configurations that form chains between

the sensors and the base station. Each UGV is affected by all other UGVs and sensors within a certain range. Several different environments are used to evaluate the approaches with respect to connectivity and efficiency. In all environments, the “dynamic” approach performs the best as the “standard” approach causes the robots to spread evenly in the environment. This is disadvantageous as it does not necessarily places robots at locations that are good for forming relay chains.

If a large set of targets must be visited and the number of available UGVs is smaller than the number of targets, then the UGVs must move between targets while at the same time maintaining communication with the base station. One possibility is to create a tree rooted in the base station and spanning all targets and visit one target at a time [72]. Several different tree types are possible, such as depth-limited trees or minimal spanning trees, or trees based on a traveling salesman tour. The trees are evaluated with respect to different criteria, for example average travel distance for each robot. Communication between robots is modeled using a virtual spring-damper model. If the robots are so far away from each other that the signal quality decreases below a threshold, the robots are attracted towards each other. This supposedly avoids the risk of disconnection, although few details are provided.

Another option that also builds on the concept of a tree spanning all targets is to divide the robots into groups, and let each group visit all targets in a subtree. Depending on the number of robots in a group and the number required to visit all targets, it might be possible to visit several targets simultaneously [71]. As the problem is shown to be NP-hard, different heuristics are evaluated with regards to the same criteria as for the sequential traversal method. The heuristics differ in when to split a robot group and the order in which targets are visited when sequential visits are necessary. In our problems, we assume that we have access to enough UAVs to surveil all targets concurrently, therefore the need for route planning between surveillance positions is removed.

## 2.4 Area Coverage

A general investigation of whether using a single relay UAV could improve communication in a simulated urban environment has been performed by Cerasoli [20]. Here the UAV works as a relay between users on the ground and line-of-sight is required between the users and the UAV for communication to take place. The focus of the work is to determine the percentage of the urban area that can be covered with acceptable signal strength, using a single UAV. Eight different positions on a circle, as well as in the middle of the circle are evaluated. The altitude is also varied, between 500, 1000 and 2000 meters. When the UAV is placed at a higher altitude, it has line-of-sight to a larger percentage of the area, and even though the signal strength decreases, better coverage of the area is achieved.

Han et al. [47] performed similar experiments, by investigating the effect of using a UAV to improve the global message connectivity and worst-case connectivity in a net-

work. A series of experiments were performed to investigate the impact that the UAV had on the two types of connectivity. The size of the area was fixed and the number of users was varied between 2–30. As the number of users grew, connectivity improved as more users could communicate without the use of the UAV. Thus, the greatest improvement was achieved when the number of users was small. However, in all cases the UAV could significantly improve both connectivity measures.

The research in the above papers has little in common with the problems that we are interested in. We know the locations of the surveillance UAV and the base station and are consequently not interested in providing coverage of a large area. Neither are we interested in providing communication between arbitrary users.

## 2.5 Exploration

Exploration is one of many application areas for robots. When several robots cooperatively explore an area, it can be very beneficial if they are able to exchange information during the exploration process [83]. The problem is discretized, both temporally and spatially, using a grid. A grid cell is considered explored when it has been visited by a robot. An algorithm based on maintaining an exploration frontier is used to weigh the benefits of exploring unknown territory versus maintaining communication with other robots. A set of possible moves is determined for each robot, and all moves are evaluated with a utility function. The function evaluates each move with respect to different factors, for example, whether it explores a cell on the exploration frontier and whether it maintains communication with other robots. An optimization is performed in each time step, to determine what actions yield the highest total utility for the team. This problem is different from our problems: we are interested in finding a set of locations where robots will be *placed*, rather than finding a sequence of positions that allow robots to explore an environment.

Anderson et al. [5] presents an algorithm for maintaining line-of-sight between groups of ground robots exploring an area. The algorithm is based on several heuristics. First, existing groups of robots are identified, and then groups are connected to each other using a set of relays with the estimated lowest cost to connect the groups. Robots acting as relays are placed one at a time, until a certain confidence threshold is reached. The confidence threshold indicates that any solution found is of sufficiently high quality. Simulated testing indicates that the algorithm performs well, but the algorithm has no theoretical completeness guarantee.

Arkin and Diaz [6] use a behavior-based architecture to allow teams of ground robots with line-of-sight communication to explore buildings and to find stationary objects using only limited knowledge about the area in which the objects are placed. The objective is somewhat different from the objective in this thesis, as the task is to find effective exploration strategies that minimize the time required until all objects are found.

## 2.6 Ad-hoc Networks and Wireless Sensor Networks

Problems that are seemingly very similar to the relay positioning problems are encountered in ad-hoc networks. In such networks, messages are to be delivered in a network where there is no control of the network topology. Routing algorithms for such networks must be able to handle addition and removal of nodes at runtime [55, 62, 67].

The use of a swarm consisting of several UAVs to improve the range and reliability of an ad-hoc network is investigated by Palat et al. [79]. Good results are achieved, mainly through a large increase in the range using the same transmission power, compared to using a direct ground link.

Brown et al. [14] investigated different scenarios where a UAV was used as a relay node. In the first scenario, there were two groups of radio nodes on the ground. Each group could communicate internally, but could not communicate reliably with the other group. In this scenario, the UAV allows the groups to form a functioning ad-hoc network and both throughput and connectivity were significantly improved.

The second scenario tested whether a UAV could improve throughput between moving radio nodes, including other UAVs. The result was that the UAV greatly improved the connectivity for nodes having initially poor connectivity while the connectivity was somewhat adversely affected for nodes with good initial connectivity. The authors speculate that this depends on a more variable link quality when using UAVs. To determine the subjective quality, they tested web browsing and a real-time voice application via an ad-hoc network. For web browsing, the performance was acceptable when using ad-hoc networks with up to six hops. The voice application worked well with up to three hops.

Significant differences exist between the relay problems and ad-hoc networks. In the relay problems, we are not interested in communication between arbitrary nodes, but in transferring large amounts of information between the surveillance UAV(s) and the base station. For this reason, we are not interested in maintaining connectivity between arbitrary nodes. Instead, we are interested in where the nodes (UAVs) should be positioned so that information can be transmitted to the base station. Furthermore, we have control over the placement of the UAVs and assume that the UAVs will be available for the complete mission.

Wireless Sensor Networks (WSNs) consist of a large number of small sensors that are placed to cover an area [2]. For a survey of routing algorithms in WSNs, the reader is referred to Al-Karaki and Kamal [3]. Although there are some similarities with the problems investigated in this thesis, there are also considerable differences: WSNs must be able to handle frequent sensor failures, and relays are often also sensors and should be placed accordingly. In WSNs, there is also limited control over where the sensors are placed.

# 3

---

## The Relay Positioning Problems

In this chapter, we provide definitions of several different variations of relay positioning problems, and we discuss factors and functions that can be used to determine whether communication and surveillance are possible. We also discuss how the cost of such activities can be modeled. A discussion about continuous solution methods is also included in this chapter.

### 3.1 Problem Setup

We assume that relays are placed in three dimensions. Let  $F \subseteq R^3$  be the region that is free from obstacles, defining the space through which free line-of-sight can be achieved. Let  $U \subseteq F$  be the region where each individual UAV may safely be placed. This region must only include points sufficiently far away from obstacles for the required safety clearances to be satisfied. *No-fly-zones* where UAVs are not permitted may also be excluded from  $U$ . Let  $x_0, t_1 \in R^3 \setminus U$  be the position of a base station and a surveillance target, respectively.

Assume as given two Boolean reachability functions: a *communication reachability function*  $f_{comm}(x, x')$  and a *surveillance reachability function*  $f_{surv}(x, x')$ . The communication reachability function specifies whether communication between two entities at points  $x, x' \in U$  should be considered feasible. It can for example be defined by a limited communication radius and a requirement of free line-of-sight (where all points between  $x$  and  $x'$  must be in  $F$ ), by explicit models of 3D wave propagation, or by any other definition appropriate for the problem at hand. The surveillance reachability function  $f_{surv}(x, x')$  specifies whether a surveillance UAV at  $x \in U$  would be able to surveil a target at  $x' \in R^3 \setminus U$ . This function must take into account suitable minimum and maximum ranges for surveillance as well as sensor-specific limitations such as cameras that

### 3. The Relay Positioning Problems

---

cannot surveil targets in arbitrary directions. For example, a camera mounted on the belly of the UAV cannot surveil targets above the UAV.

Under the assumption that the corresponding reachability function holds, a *communication cost function*  $c_{comm}(x, x')$  determines the cost of communication from  $x$  to  $x'$  and a *surveillance cost function*  $c_{surv}(x, x')$  determines the cost of surveilling a target at  $x'$  from the position  $x$ . The general notion of cost and cost minimization can be used to model a wide variety of quality measures or combinations of such measures. For example, communication costs may be related to transmission power requirements, the risk of interrupted communication or intermittent dropouts, and the risk of detection by adversaries.

Several relay problems for a single target will now be defined. Multiple targets are treated in Section 6.1.

## 3.2 Definitions of the Single Target Relay Problems

A *relay chain*  $\pi$  between the base station position  $x_0$  and the single target position  $t_1$  is defined as a sequence of positions  $[x_0, x_1, \dots, x_k, t_1]$ , where  $\{x_1, \dots, x_k\} \subseteq U$ , such that  $f_{comm}(x_i, x_{i+1})$  for all  $i \in [0 \dots k - 1]$ , and  $f_{surv}(x_k, t_1)$ . The *length* of a chain is defined as the number of agents required to realize the chain, including the base station and all UAVs:  $\text{length}([x_0, x_1, \dots, x_k]) = k + 1$ .

We are often interested in generating relay chains of high quality relative to a problem-specific quality measure. We model such measures in terms of the *cost* of relaying information between positions  $x_i$  and  $x_{i+1}$  and the cost of surveilling the target at point  $t_1$  from a surveillance UAV at point  $x_k$ .

The *cost* of a relay chain  $[x_0, x_1, \dots, x_k]$ , denoted by  $\text{cost}([x_0, \dots, x_k])$ , is defined as

$$\sum_{i=0}^{k-1} c_{comm}(x_i, x_{i+1}) + c_{surv}(x_k, t_1)$$

Given a problem instance as defined above, including the position of the base station and the target, we can now identify a number of interesting single target relay positioning (STR) problems. Some of these problems assume an upper limit on the number of UAVs available, denoted by  $M$ . Setting  $M = \infty$  requires finding all solutions, regardless of length.

**STR-MinLengthMinCost:** Find a relay chain of minimum length among the chains of minimum cost. A solution to this problem is a chain  $s$  such that for all other chains  $c$ ,  $\text{cost}(s) \leq \text{cost}(c)$  and  $\text{cost}(s) = \text{cost}(c) \rightarrow \text{length}(s) \leq \text{length}(c)$ . This corresponds to using the highest quality chain that can be realized with access to an unlimited number of UAVs, with a preference for using fewer UAVs if this is possible without compromising quality.

**STR-MinCostMinLength:** Find a relay chain of minimum cost among the chains of minimum length. A solution to this problem is a chain  $s$  such that for all other chains  $c$ ,  $\text{length}(s) \leq \text{length}(c)$  and  $\text{length}(s) = \text{length}(c) \rightarrow \text{cost}(s) \leq \text{cost}(c)$ . This is useful if minimizing the number of UAVs is strictly more important than maximizing quality.

**STR-MinCostLimited:** Find a relay chain of minimal cost among the chains that use at most  $M$  UAVs. A solution to this problem is a chain  $s$  such that for all other chains  $c$ ,  $\text{length}(s) \leq M + 1$ , and for all other chains  $\text{length}(c) \leq M + 1 \rightarrow \text{cost}(s) \leq \text{cost}(c)$ . This corresponds to a desire to find the highest quality relay chain that can be realized within the given limit on the number of UAVs.

**STR-ParetoLimited:** Find a set of *Pareto-optimal* relay chains that is complete up to a given upper limit on the number of available UAVs. A chain  $s$  is Pareto-optimal for up to  $M$  UAVs if  $\text{length}(s) \leq M + 1$  and for all chains  $c$  of length at most  $M + 1$ ,  $\text{length}(c) < \text{length}(s) \rightarrow \text{cost}(c) > \text{cost}(s)$  and  $\text{cost}(c) < \text{cost}(s) \rightarrow \text{length}(c) > \text{length}(s)$ .

The **STR-ParetoLimited** problem is a bi-objective problem, where each chain represents a different trade-off between the number of UAVs in the chain and the cost of the chain. Each such chain is *Pareto-optimal*, as it cannot be improved in one aspect without a decrease in another aspect [69]. For example, the cost of the chain cannot be improved without also increasing the number of UAVs in the chain.

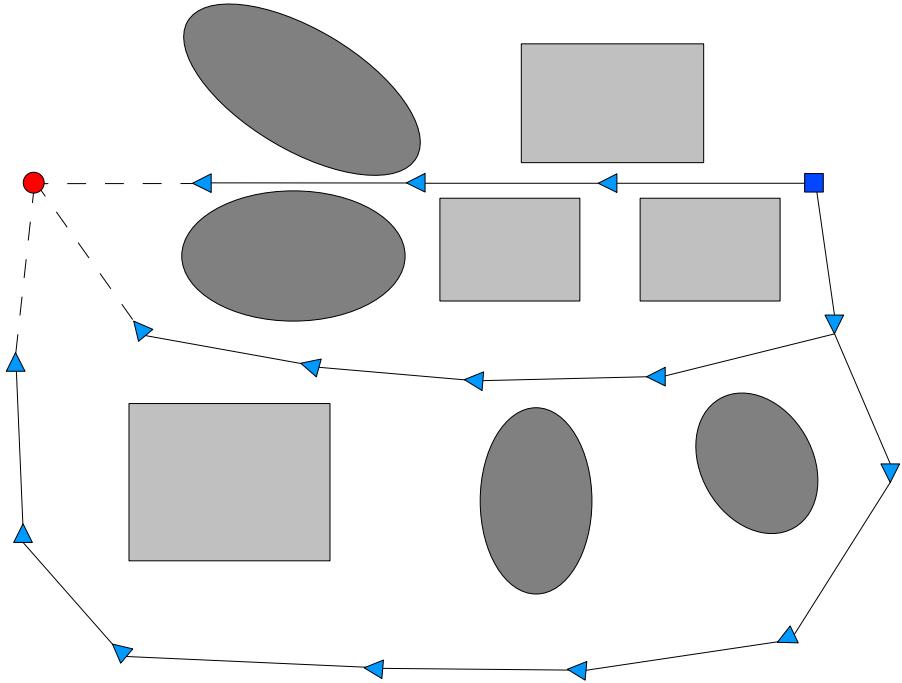
Algorithms for solving the single target relay problems are discussed in Chapter 5.

### 3.3 Cost Functions

The purpose of a cost function is to evaluate pairs of positions with respect to how suited they are to place UAVs for communication or surveillance. A pair of better suited positions has a lower cost.

For two positions  $x, x' \in R^3$ , the communication cost function  $c_{\text{comm}}(x, x')$  models the cost of communicating from  $x$  to  $x'$  and the surveillance cost function  $c_{\text{surv}}(x, x')$  models the cost of surveilling a target located at  $x'$  from position  $x$ . The functions are defined under the condition that  $f_{\text{comm}}$  and  $f_{\text{surv}}$  (see Section 3.4), respectively, hold for the positions  $x$  and  $x'$ . In this section, we first discuss several different measures for communication cost and then we discuss surveillance cost measures.

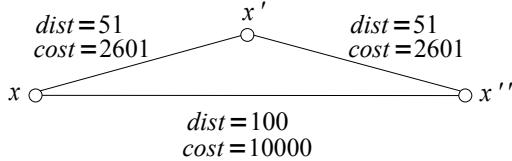
One straightforward measure of a relay chain's quality is the number of UAVs in the chain. However, in many cases it may be advantageous to use a larger number of UAVs if this can improve some other quality measure. An example is shown in Figure 3.1, where positions close to obstacles have higher cost as they offer a lower margin of safety and robustness to external factors. The UAVs in the top relay chain are positioned closer to obstacles and there is a small margin of safety around the UAVs' locations. On the other hand, the bottom relay chain uses a larger number of UAVs, but has a greater margin of safety between the UAVs and obstacles. The middle chain offers an intermediate between the two extremes.



**Figure 3.1:** Three relay chains with different lengths and costs.

Many different factors can be used as cost measures for positions. A cost measure using a combination of factors can be created using for example a weighted sum.

Some of the cost factors suggested in subsequent sections satisfy the *triangle inequality*, defined as  $c_{x,x''} \leq c_{x,x'} + c_{x',x''}$  for all positions  $x, x', x''$ , where  $c_{x,x'}$  denotes the cost of communication from  $x$  to  $x'$ . For such a function, communicating directly from  $x$  to  $x''$  cannot be more expensive than first communicating to  $x'$  and then to  $x''$ . However, even simple cost functions can void the triangle inequality and as we will see, many cost functions of interest here do exactly this. Figure 3.2 shows three positions and the distances between them. Assume that the cost of transmission between positions increases quadratically with distance. An example of such a cost is the transmission power required to achieve a certain signal-to-noise ratio, as will be discussed in Section 3.3.1. With the cost set to the square of the distance,  $c_{x,x'} = c_{x',x''} = 51^2 = 2601$  and  $c_{x,x''} = 100^2 = 10000$ . Thus  $c_{x,x''} = 10000$  and  $c_{x,x'} + c_{x',x''} = 5202$  which shows that relaying information through  $x'$  gives a cheaper path than transmitting directly to  $x''$ . The fact that the triangle inequality is not necessarily satisfied has profound implications for the relay positioning problems as it opens up the possibility of improving the quality of a chain by taking a longer path.



**Figure 3.2:** In some cases, e.g. if the cost increases quadratically with distance, then the triangle inequality no longer applies. This makes a direct transmission from  $x$  to  $x''$  more expensive than two shorter transmissions.

### 3.3.1 Transmission Quality

A surveillance UAV sending a continuous video feed to a base station may not be able to re-transmit lost or faulty data packets. Instead, a continuous stream of video data may be transmitted through the relay chain, where forward error correction [63] is used to recover from errors. However, it is likely that it is impossible to recover from some errors, which decreases the quality of the video stream received at the base station. Such a loss of quality is naturally modeled as a communication cost.

Obviously, the risk of such errors increases as the signal strength decreases. To model this, one can use a cost function inversely related to the *signal-to-noise ratio* (SNR). The SNR is one way of measuring signal strength. When calculating the SNR between two positions, the first step is often to determine the distance between the positions and whether there is *line-of-sight* between the positions. These two factors have a large impact on the communication quality. The SNR is proportional to the transmitter power  $P$  and the antenna gain  $W$  and inversely related to the distance between transmitter and receiver  $d = \|x' - x\|$ . How much the signal-to-noise ratio decreases with distance is approximated by the *path loss exponent*  $\alpha$  [79].

The path loss exponent is a very generic factor that is commonly used to approximate different factors. For example, stronger fading due to lack of line-of-sight yields a higher value of  $\alpha$ . When used to estimate fading outdoors, the value of  $\alpha$  is most often in the 2–5 range, where  $\alpha = 2$  corresponds to propagation in free air, that is, with line-of-sight. As the exact value of  $\alpha$  depends on for example altitude, the amount of particles in the air, atmospheric conditions and the buildings in the environment,  $\alpha = 4$  is commonly used if the exact value is unknown. The reader is referred to Palat et al. [79] for a discussion on different values of  $\alpha$ .

As a high SNR shall yield a low cost, one possible communication cost function based on the SNR is:

$$SNR: c_{comm}(x, x') = \frac{\|x - x'\|^\alpha}{PW}$$

In situations where UAVs with different communication equipment are used, the values of  $P$  or  $W$  can be set to the lowest values of transmitter power and antenna gain,

respectively, to provide a pessimistic estimate of the SNR. More information about calculating transmission signal strength in urban environments, albeit with a focus on cellular phones, is available in Wagen and Rizk [99]. If the environment is known in advance and time allows, a more accurate model of transmission quality can be derived where factors such as reflection and absorption can be taken into account in the cost function  $c_{comm}$  [89, 94].

Another possibility is to set the communication cost to a constant if the distance is below some predetermined value, signifying that the probability of unrecoverable errors caused by a poor signal-to-noise ratio is very low at such distances. At longer distances, the communication cost can for example be related to the inverse of the SNR.

### 3.3.2 Position Visibility

Suppose that a relay UAV is able to relay information from more than one UAV at a time. UAVs can then participate in multiple concurrent surveillance missions. Even if only one mission is initiated at a time, we might still prefer to place relays in locations where they are also likely to be useful in the event that additional missions are initiated in the near future. Such missions may then be able to use fewer additional UAVs by connecting to an existing relay UAV rather than to the base station, which may be considerably further away.

A UAV can theoretically communicate with other UAVs in a sphere whose radius is equal to the maximum communication distance (see Section 3.4). Given line-of-sight requirements, parts of this volume may be obstructed by obstacles, as shown in Figure 3.3.

Let  $V_{ob}(x)$  denote the obstructed volume from position  $x$  within the communication range  $r_{comm}$  and let  $V_{comm} = \frac{4\pi r_{comm}^3}{3}$ . Then the communication cost based on obstructed volume can be calculated as:

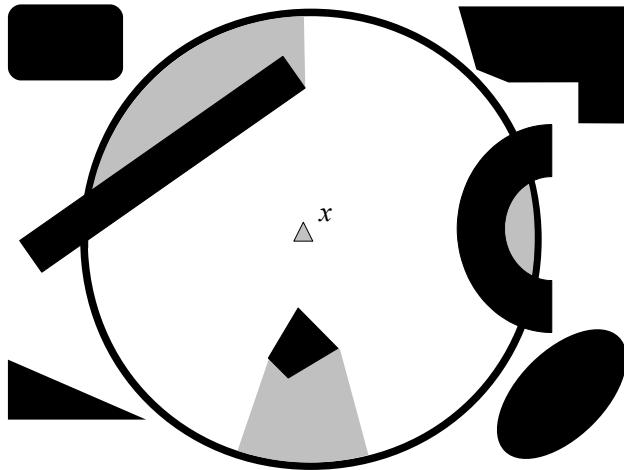
$$Obstructed\ volume: c_{comm}(x, x') = \frac{V_{ob}(x)}{V_{comm}}$$

The position visibility cost measure is especially suitable for the multiple target relay problems, which are discussed in Chapter 6.

### 3.3.3 Minimum Free Angle Between Positions

The *minimum free angle between positions* gives a measure of the ability to perform transmissions between two positions even if wind or other factors affect the UAVs' ability to stay at the designated positions. The minimum angle places the greatest constraints on the UAVs' ability to transmit information and surveil the target the most. For this reason, it is a relevant cost measure when judging the suitability of positions for UAV placement.

In Figure 3.4, a UAV located at  $x$  must not move outside the cone originating in  $x'$  with angle  $\alpha'$  and vice versa. The cost of communication between such positions should be inversely related to the size of the angle, i.e. a small angle should have a large cost. However, this might not be enough, as sufficiently small angles will cause great problems



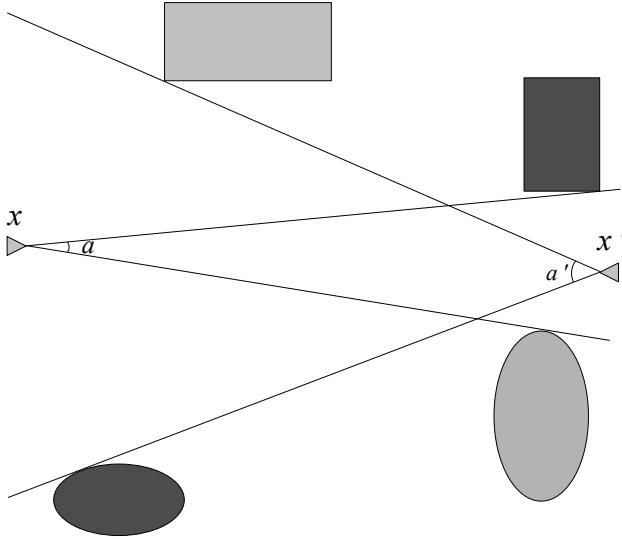
**Figure 3.3:** One possible cost measure is to use the obstructed volume in a sphere as cost. In this two-dimensional example, the cost would be the sum of the obstacles, drawn in black, and the non-visible volume, drawn in grey, inside the circle.

when trying to communicate as the UAVs become very sensitive to external factors such as wind. For example, two UAVs with a minimum free angle of  $5^\circ$  between them may be more than twice as sensitive to disturbances compared to if the minimum free angle was  $10^\circ$ . Therefore, to severely penalize very small angles, a measure such as the inverse of the logarithm of the minimum angle or some other non-linear function can be used as cost. In some cases, we may set the cost to zero or some small constant if the minimum angle is above some threshold value. For example, it might be extremely unlikely that the UAVs cannot stay inside a cone with a  $30^\circ$  angle, so any pair of positions with a minimum angle larger than  $30^\circ$  could have cost zero.

Thus, one possible communication cost function based on the minimum angle between obstacles is:

$$\text{Minimum angle: } c_{\text{comm}}(x, x') = \begin{cases} \frac{1}{\min\{a, a'\}} & \text{if } \min\{a, a'\} < 30^\circ \\ 0 & \text{if } \min\{a, a'\} \geq 30^\circ \end{cases}$$

Which of these functions depending on the minimum angle is most suitable depends on the situation, where factors such as the environment, required safety margin and UAV susceptibility to external factors are taken into consideration.



**Figure 3.4:** Minimum free angle between positions is a possible measure of cost.

### 3.3.4 Minimum Distance to Obstacles

Ideally, a UAV in  $U$  should be so far away from obstacles that there is no risk of collision regardless of external factors such as wind. In practice, this would be very difficult to achieve, given that different vehicles are more or less sensitive to such factors and that some UAVs' control systems may not be capable of precise maneuvering. The set  $U$  would need to be calculated for each type of UAV, and even then, it would be very difficult to prove that no collision can occur. It could also lead to an unnecessary limitation of the search space. Due to the forbidden positions, many missions would require a large number of UAVs, and other missions would be impossible to perform. A reasonable compromise between safety margins and excessive limitation of available positions is to require that all positions in  $U$  have a minimum distance to obstacles, and generally prefer positions that are placed further away from obstacles. Such a requirement could be modeled as a cost, where for example the communication cost  $c_{comm}(x, x')$  would depend on the distance from  $x'$  to the nearest obstacle.

With  $o$  as the closest obstacle from  $x'$ , one possible cost function is:

$$\text{Minimum obstacle distance: } c_{comm}(x, x') = \|o - x'\|$$

### 3.3.5 Surveillance Cost Functions

For the surveillance cost function, it can be relevant to use parameters from the sensors to estimate the quality of the sensed information and use such estimates in the surveillance cost function. If the sensor is a (video) camera, a position from which high-quality

pictures can be taken would have a low surveillance cost. Such a position would have no intervening obstacles between it and the target and be located at an appropriate distance from the target.

It can also be beneficial to take factors other than the physical terrain into consideration when specifying the surveillance cost function. For example, environmental and weather conditions, as well as the position of the sun, can be used. Low costs can be given to positions where the surveillance UAV's position will allow it to take pictures with a minimum of reflections and glare. If the visibility is poor due to for example fog, the surveillance cost could increase faster with increasing distance than if visibility is good. A similar cost function as for SNR (see Section 3.3.1) can be created by using the distance raised to some power  $\alpha$  to signify that image quality decreases with distance, as well as with weather phenomena such as fog.

$$\text{Distance to target: } c_{\text{surv}}(x, x') = \|x' - x\|^\alpha \quad \alpha \geq 1$$

The surveillance cost must be weighed against the communications cost for the rest of the chain. Unless the cost of surveillance is set high enough to influence the algorithms for finding solutions, a high-cost (low quality) surveillance position might be used as the surveillance cost only has a marginal effect on the cost of the complete chain.

## 3.4 Reachability Functions

The purpose of a reachability function is to determine whether communication or surveillance is possible between two positions. The reachability functions are Boolean functions that take as input two positions,  $x$  and  $x'$ . The communication reachability function  $f_{\text{comm}}(x, x')$  holds only if communication between the two positions  $x, x'$  is possible. Similarly, the surveillance reachability function  $f_{\text{surv}}(x, x')$  holds only if a UAV located at  $x$  can surveil a target located at  $x'$ . The factors used in the reachability functions are dependent on the application, and the algorithms to be presented in later chapters for solving the single and multiple target problems are independent of the function. As long as the functions have the above signatures, no further assumptions about the exact formulation of the functions are made.

We assume that the communication between several UAVs does not interfere with each other. This can for example be achieved through using techniques such as time-division multiplexing, frequency-division multiplexing and code-division multiplexing [94]. Time-division multiplexing means that transmitters take turns transmitting. Different frequencies for transmissions are used in frequency-division multiplexing and code-division multiplexing means that different encodings are used.

For the kind of transmissions of interest in this thesis, two common requirements are line-of-sight between sender and receiver and limited maximum transmission range. In particular, such requirements are very common when transmitting high-volume sensor data, such as a live video feed, that requires high uninterrupted bandwidth. Formally, the

### 3. The Relay Positioning Problems

---

line-of-sight requirement holds if  $[x, x'] = \{x \in R^3, x' \in R^3 : \beta x + (1 - \beta)x', \beta \in [0, 1]\} \subseteq F$ . That is, line-of-sight between two positions exists if a straight line between the two positions lies completely in the allowed set  $F$ . Note that we can allow  $F \neq U$  as communication can go through areas where we might not be allowed to place UAVs. The limited communication range is formalized as  $\|x' - x\| \leq r_{comm}$ . It is sometimes desirable to separate the UAV communication range from the UAV surveillance range  $r_{surv}$ . In that case,  $f_{comm}(x, x')$  only holds if  $\|x' - x\| \leq r_{comm}$ , in addition to any other requirements. Similarly,  $f_{surv}(x, x')$  only holds if  $\|x' - x\| \leq r_{surv}$  in addition to any other requirements that are used.

Although line-of-sight is often required for high bandwidth communication, there are communication systems that do not require this. The Multiple Input Multiple Output (MIMO) communication system is an example of this [75, 44]. In fact, having line-of-sight could decrease the performance as objects in the environment are used to “bounce” the radio signals off. MIMO communication systems use several antennas for transmitting and receiving information, and the best performance is achieved if several independent signal paths are used. Such communication systems can also be modeled using the communication reachability function, although the line-of-sight requirement would most likely not be used.

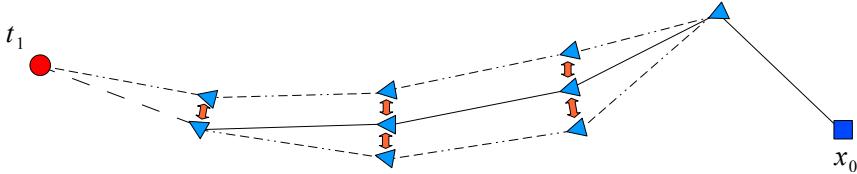
Whereas  $f_{comm}$  depends on the properties of the communication system, the surveillance reachability function  $f_{surv}$  depends on the sensors used for surveillance. For cameras, common restrictions would be line-of-sight and a maximum range requirement, where the information sensed at the maximum range is of sufficient quality to be used, for example, a certain number of pixels per meter of target size for a camera. If several sensors are used to surveil the target, special considerations must be given to whether  $f_{surv}$  holds when one sensor senses the target, or when a certain number of sensors sense it.

The above are just examples of the many different ways to determine whether communication and surveillance is possible or not. No assumptions about the reachability functions are made anywhere else in the solution process and arbitrarily complex functions can be used to determine whether communication and surveillance can be performed, if desired.

## 3.5 Problem Properties

As the target and the base station can be placed arbitrarily and UAVs can be placed anywhere in the set  $U \subseteq R^3$ , the relay problems can be seen as continuous optimization problems. Here we discuss why current methods for continuous optimization are not practical for finding globally optimal solutions for the relay positioning problems.

Assume a given instance of a relay problem in an environment without any obstacles, with the position of a base station  $x_0$  and the target  $t_1$ , as well as a number of UAVs. The reachability functions are based on free line-of-sight. From this, it is possible to calculate a set of feasible relay chains. The number of relay chains in this set is generally



**Figure 3.5:** In an environment without obstacles or any limitations in the reachability functions, it is possible to transform any valid relay chain to any other valid relay chain, as indicated by the arrows.

uncountable as UAVs may be positioned anywhere in a continuous space. It is possible to transform any feasible relay chain to any other feasible chain by moving the UAVs continuously, as exemplified by Figure 3.5.

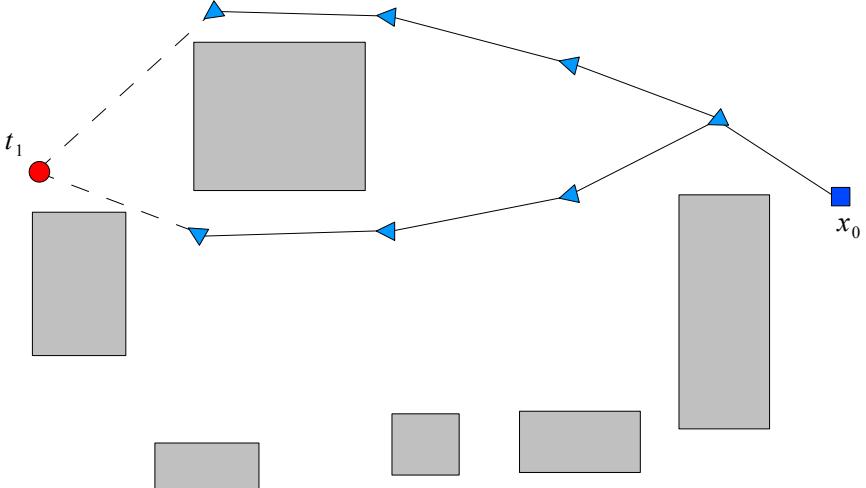
However, we are more interested in environments with obstacles and possibly also restrictions in the reachability functions, e.g. limited communication range. Then discontinuities will occur as it is no longer possible to communicate between arbitrary positions and surveil the target from any position. In such cases, the feasible set consists of several disjoint subsets. Within each such subset, the UAVs in a relay chain may still be moved to form another relay chain. However, it is not possible to transform a chain in one feasible subset to another chain in another feasible subset without going through infeasible points. An example of this can be seen in Figure 3.6, where going from the relay chain on the top to a relay chain on the bottom requires going through the infeasible region in between.

The number of different feasible subsets may grow exponentially with the number of obstacles, which can lead to a large number of subsets in obstacle-rich environments. The large number of subsets leads to many local maxima, as each subset has at least one such point.

## 3.6 Continuous Solution Methods

It is possible to use methods for continuous optimization to calculate relay chains. However, such methods are impractical to use due to the long execution time. In addition, it is an open research question how to formulate a requirement such as line-of-sight in terms of equalities and inequalities. Such a formulation is required in order to apply optimization methods for continuous (non) linear<sup>1</sup> problems. In general, it is an open research question how to solve problems such as the relay problems using methods for continuous (non) linear optimization, especially if we consider that the algorithms are to be used in a setting where a ground operator expects a quick response.

<sup>1</sup>An optimization problem is linear if both the objective function and all constraints are linear, otherwise it is nonlinear.



**Figure 3.6:** The continuous relay problems are computationally intractable as the feasible set is disjoint. A change from the top chain to the bottom chain, or vice versa, requires going through infeasible points.

For each feasible set in which an initial chain has been found, continuous local optimization methods such as line search [74] can be used to find local extrema within that feasible set. Such methods calculate a step length and a direction in which a function value, such as the value of our cost function, improves. A new position is determined by moving a distance equal to the step length from the current position in the calculated direction. At the new position, it is checked whether the reachability function holds. If so, the new position becomes a new starting point for further search and the procedure can be repeated until the UAV is placed at a position that optimizes the function value of the cost function, subject to the constraints in the reachability function. The process is then repeated for each UAV. This can create a locally optimal chain, but if we want to find a globally optimal chain, such a method must be executed for each feasible subset. Due to the large number of subsets, this would take prohibitively long time. Furthermore, for problems requiring finding a set of relay chains, such as **STR-ParetoLimited**, this requires repeating the process for all values of the number of available UAVs up to  $M$ .

Heuristics such as tabu search or simulated annealing lack the guarantee of finding the globally optimal chain. However, they can be used to find an initial relay chain in a feasible set and then continually improve it until a local extremum is found [45].

Another option for solving the relay problems is to use algorithms from the area of computational geometry, where continuous versions of Dijkstra's shortest path algorithm have been developed [70]. Such methods can be used for solving for example the **STR-MinLengthMinCost** problem. A disadvantage of these methods is that they approximate the environment using a set of surfaces and that the time complexity of the algorithm is

dependent on the number of surfaces. Due to high time complexity, the execution time can be very long. In addition, problems such as **STR-MinCostLimited** involve finding several chains of different lengths and costs. The time required to solve such a problem suggests that other representations and methods are more appropriate.

## 3.7 Summary

In this chapter, several relay problems for a single target have been defined. Also, the use of reachability and cost functions have been explained, and several examples of cost functions have been given. The reachability functions determine whether communication and surveillance between a pair of positions is feasible. Therefore, they are often based on factors such as free line-of-sight and a limited maximum distance between the positions.

The purpose of a cost function is to evaluate the relative suitability of pairs of positions with respect to placing UAVs there for communication or surveillance and cost functions are often based on factors such as the distance to obstacles or the distance between the positions. Several of the cost functions void the triangle inequality, which makes it possible to decrease the cost of a relay chain by using a greater number of UAVs.

Obstacles in the environment as well as conditions such as limited communication and surveillance ranges and line-of-sight-requirements in the reachability functions divides the feasible set into several disjoint feasible subsets. Each such feasible subset has at least one local optimum. Determining the global optima using methods for continuous optimization can be very time consuming as such methods need to be executed once for each such subset.

### 3. The Relay Positioning Problems

# 4

---

## Environment Representation and Discretization

One way to decrease the execution time for finding solutions to the relay positioning problems is to use graph search algorithms to solve discrete approximations of the continuous problems. Such algorithms require a graph with nodes corresponding to positions where UAVs can be placed, and edges corresponding to potential communication or surveillance between positions. In this chapter we show how a discretization can be performed in order to construct a graph, and discuss different discretization strategies.

### 4.1 Discretization and Graph Creation

A discretization is performed with the intention of creating a graph consisting of a set of nodes, denoted by  $N$ , and a set of edges, denoted by  $E$ .  $|N|$  denotes the cardinality for the set of nodes, and the notation is used analogously for other sets. A directed graph can be created as follows.

1. Determine a finite set of positions  $U' \subseteq U$  among the free coordinates. These are the positions that will be considered valid placements for relay and surveillance UAVs in the discrete relay problems. Different ways to determine the set of positions  $U'$  are discussed in Sections 4.2–4.5.
2. Associate each position  $x \in U'$  with a unique node.
3. Associate the base station position  $x_0$  with a new node  $n_0$ .
4. Associate the target position  $t_1$  with a new node  $\tau_1$ .
5. For each  $x \in U'$  corresponding to  $n \in N$  and satisfying the communication reachability function  $f_{comm}(x_0, x)$ , create an edge  $e = (n_0, n)$  of cost  $c_{comm}(x_0, x)$  rep-

resenting the possibility of communication between the base station and position  $x$ .

6. For each  $x, x' \in U'$  corresponding to  $n, n' \in N$  and satisfying the communication reachability function  $f_{comm}(x, x')$ , create an edge  $e = (n, n')$  of cost  $c_{comm}(x, x')$ , representing the possibility of communication between positions  $x$  and  $x'$ .
7. For each  $x \in U'$  corresponding to  $n \in N$  and satisfying the surveillance reachability function  $f_{surv}(x, t_1)$ , create an edge  $e = (n, \tau_1)$  of cost  $c_{surv}(x, t_1)$  representing the fact that a surveillance UAV positioned at  $x$  would be able to surveil a target at  $t_1$ .

Then, the graph  $G(N, E)$  represents a discretization of the continuous space. Note that the graph construction algorithm makes no assumptions about the reachability functions  $f_{comm}$  and  $f_{surv}$  other than their signatures.

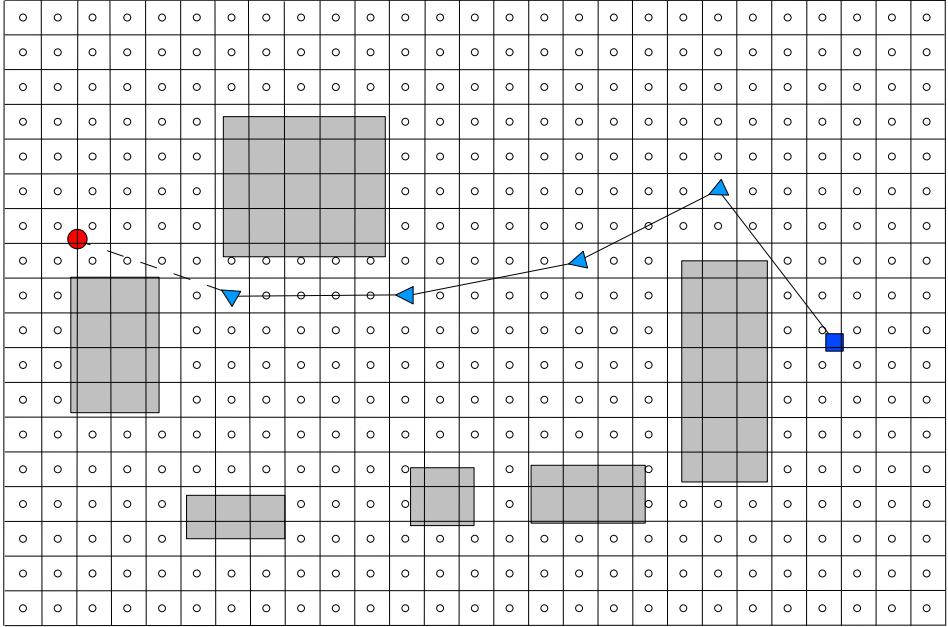
In the discretized space, the *length* of a chain corresponds to the number of edges in the chain. This is also referred to as the *hop count*, where each edge corresponds to one hop. The *cost* of a chain is then defined as the sum of the edge costs along the chain. Thus, a *shorter* chain has fewer edges, while a *cheaper* chain has lower cost.

The discretization itself only allows the application of discrete optimization methods. The problem of finding a set of feasible UAV positions is still not solved. Instead, a reduction to a feasible problem is required. This can be done by considering the following: given a node  $n$ , the set of valid nodes for placing the next UAV in a chain can be identified as exactly the set of nodes for which  $f_{comm}(n, n')$  holds. Let  $N_n \subseteq N$  consist of all such nodes  $n'$ , and analogously,  $N_{n'}$  consists of all nodes all nodes for which  $f_{comm}(n', n'')$  holds. Then, a complete chain from  $n_0$  to  $\tau_1$  can be determined by starting from  $n_0$  and choosing a node in  $N_{n_0}$  as the next node in the chain, and continuing recursively until a node  $n''$  is found for which  $f_{surv}(n'', \tau_1)$  holds. Recall that a UAV placed at such a position can surveil the target located at node  $\tau_1$ .

From this, we know in theory how to choose the set of UAV positions so that a relay chain is formed. Naturally, we are interested in finding a high-quality (i.e. low cost) chain. Finding solutions to such problems is commonly done using graph search algorithms, which will be discussed further in Chapter 5.

In the remainder of this chapter, we discuss different options for choosing the finite set of discrete positions  $U'$  where UAVs may be placed. One of these methods places nodes with the same node density throughout the environment, while others distribute nodes unevenly, typically placing more nodes closer to obstacles. Which discretization method is most suitable depends on several factors, such as the number of nodes and their positions, the reachability functions, and the number of obstacles as well as how the obstacles are distributed in the environment.

Different graphs will affect the length and cost of the solutions. In some cases, a problem instance may require a large number of UAVs as the chain must go around areas with no or few nodes. There are several desirable characteristics in discretization methods that



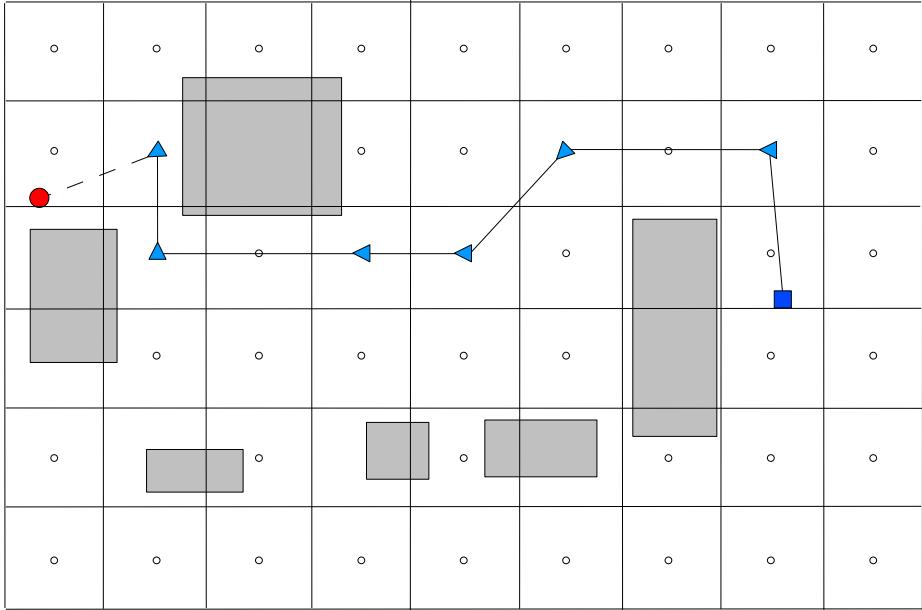
**Figure 4.1:** Nodes created using a medium-resolution grid. The size of the nodes have been exaggerated for illustrative purposes, and the edges are left out to reduce clutter.

minimize the risk for this. To maximize flexibility in choosing the location of the next UAV in the chain, the nodes should have a high degree. The connected nodes should be placed at different angles and distances, providing good coverage of the area using few nodes. If this is not fulfilled, it might not be possible to communicate or surveil through gaps between obstacles, which can lead to long detours. To provide many different positions for the surveillance UAV and provide sensor information of high quality, it is also desirable to have a large set of nodes directly connected to the target node.

## 4.2 Fixed-Size Grids

A simple solution to choosing the set of potential relay and surveillance positions is to use a three dimensional grid with grid cells of equal size. The grid is placed over the terrain and a node is placed in each unobstructed grid cell. The node can be positioned anywhere in the cell, but for illustration purposes they are placed in the middle of each cell. Whether a grid cell is unobstructed can be determined in many different ways. For example, if there are no obstacles within a certain distance from the node position, then the cell could be considered unobstructed.

The resolution is a very important factor when using grids. If it is too low, then UAVs



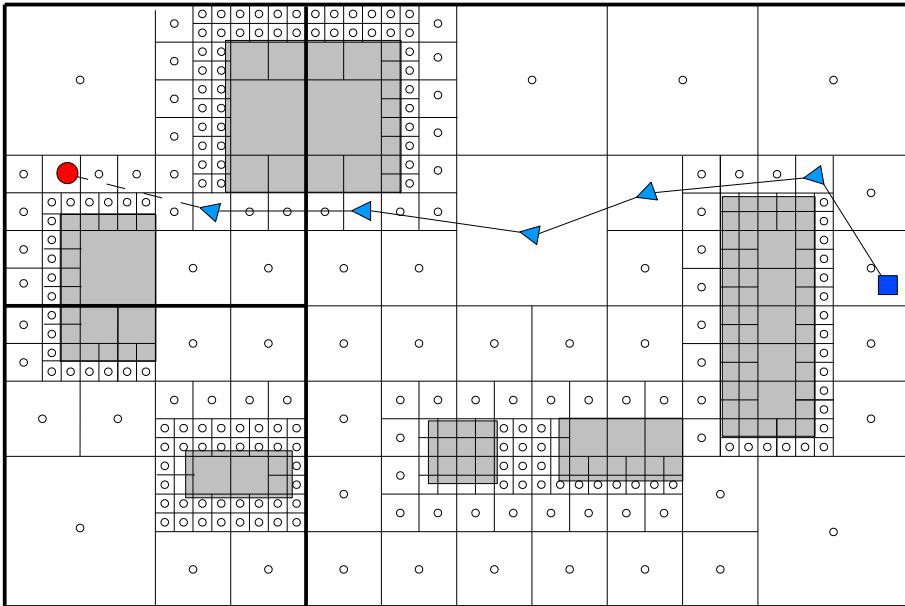
**Figure 4.2:** The low-resolution grid provides few possibilities for UAV placement, which leads to a longer relay chain.

can only be placed at a few places, possibly causing unnecessarily long relay chains. Figure 4.1 shows a graph with nodes created from a medium-resolution grid, which is in contrast to the low-resolution grid displayed in Figure 4.2. In the coarser discretization, 6 UAVs are required, compared to 4 UAVs in the finer discretization as the maximum communication and surveillance range of each UAV can be better utilized.

Some types of terrains might require grids of high resolution to ensure that all positions can be reached, which leads to high memory consumption.

### 4.3 Octrees

Assume that fixed-size grids are used in an environment that mainly consists of a large open area but also has some closely spaced obstacles. To assure that UAVs can be placed between the obstacles, the grid cell size must be quite small. The large number of nodes leads to large memory requirements, and the high resolution is not necessary in the large open area. To decrease the number of nodes and thus the memory requirements, an alternative is to use a data structure with cells of varying size. Examples of such structures are *quadtrees* (for two-dimensional environments) and *octrees* (for three-dimensional environments) [43, 49]. Many other data structures are possible, and an overview of such data structures for different uses is available in Samet [85]. Although data structures such



**Figure 4.3:** Several quadtrees covering an area. The heavy lines mark the edges of the quadtrees.

as octrees are often used to subdivide areas spatially, for example to speed up geometry intersection checks, they can also be used to place nodes. A two-dimensional example is shown in Figure 4.3.

The octree is a hierarchical data structure, with recursive decomposition from the top level with the largest volumes, to the bottom level, which contains the smallest volumes. Volumes that are not further divided are called leaves. Each non-leaf volume in an octree contains eight smaller volumes of equal size, organized in a  $2 \times 2 \times 2$  fashion. Together the smaller volumes cover the exact same volume as the larger volume. Each of the smaller volumes, in turn, either is a leaf or contains eight even smaller volumes, and so on, until the desired number of levels, or the minimum cell size, is reached. Nodes can be placed for example in the middle of each volume or at the corners of the volumes.

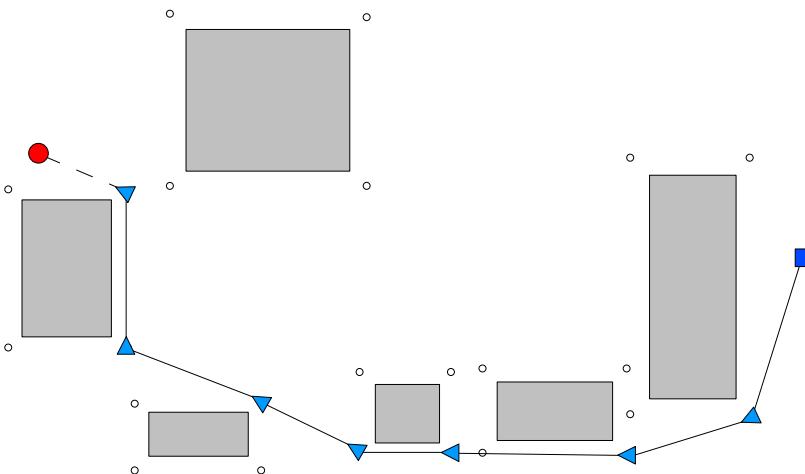
Octrees are designed to cover cubic areas, and if the volume to be covered is not cube shaped, there are two possibilities. Either an octree covering a larger volume must be used, or the area must be divided into several cubic volumes.

Using a data structure with grid cells of varying size has the advantage of a decreased number of nodes and edges and therefore lower memory consumption. A disadvantage is the comparatively low concentration of nodes at positions far from obstacles, yielding fewer possible UAV locations.

## 4.4 Expanded Geometry Graphs

In an *expanded geometry graph*, obstacles in the environment are expanded a certain distance “outward”, hence the name [64]. The exact distance varies, but is often related to the size of the UAVs and the required safety distance. Nodes are then placed at the corners of the expanded objects. Figure 4.4 shows an expanded geometry graph. If obstacles are large, the distance between the nodes at its corners may exceed the maximum communication or surveillance range. In such cases, additional nodes might have to be placed along the sides of the obstacles to provide connections. The same problem can occur also occur in other cases, for example when obstacles are further from each other than the communication range or when the distance between the target position  $t_1$  and the closest obstacle is longer than the surveillance range.

As nodes are placed close to the corners of obstacles, this approach is suitable for urban terrains, where the node placement also ensures that relay chains can use any gaps between buildings. However, this approach to discretization is somewhat brittle: if the terrain is so dense that the expanded obstacles collide, then no nodes can be placed. In addition, the approach is not suitable in terrain involving large open areas as no nodes are placed at such locations. The lack of nodes forces the relay chain to take long detours around such areas. Figure 4.4 shows an example of this.

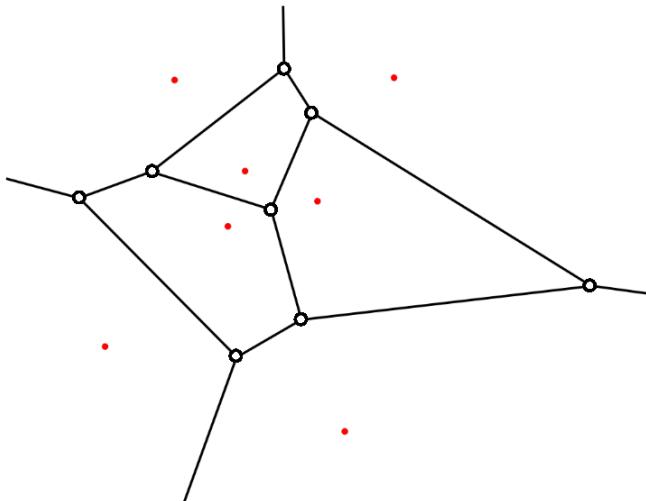


**Figure 4.4:** An expanded geometry graph with nodes placed at corners of objects.

## 4.5 Voronoi Diagrams

Another possibility is to use a graph created by a *Voronoi diagram* [7, 76]. The Voronoi diagram is a discretization that takes as input a set of obstacles. Then, the environment is divided into convex cells such that each cell contains exactly one obstacle and every point in a cell is closer to the obstacle in it than to any other obstacle. Figure 4.5 displays a Voronoi diagram, where the boundaries between the cells are drawn as lines. The boundaries are the positions that offer maximum distance between the obstacles. Nodes can then be placed where three or more cell boundaries meet [13]. This causes the number of nodes to be a function of the number of obstacles. Where there are few obstacles, there will be few nodes. If there were no obstacles, only a single cell would be created. This would generate very few nodes, in many cases making it impossible to find solutions to the relay problems. Therefore, this approach is best suited for environments with a large number of obstacles. To remedy the problem of too few nodes, one option is to also place nodes on the cell boundaries [11].

Voronoi diagrams have been used in many areas, for example UAV path planning [21, 9, 13, 68] and UGV path planning in environments with many buildings [11].



**Figure 4.5:** An Voronoi diagram with points as obstacles for illustrative purposes. A cell is created for each obstacle. The lines mark the boundaries between the cells. Nodes are placed where three cells meet.

## 4.6 Discretization Methods Used in Motion Planning

The research area of motion planning is concerned with finding paths from a start position to a goal position. Given a potentially high-dimensional space, discretizations are

commonly used to make the problems tractable. Furthermore, algorithms commonly use discretizations created specifically for motion planning, rather than the general methods discussed so far. Examples of such methods are Rapidly Exploring Random Trees (RRTs) [59, 60, 80] and Probabilistic Road Maps (PRMs) [57, 60]. RRT algorithms construct a new tree every time motion planning is performed, while PRMs produce a graph in advance and add start and goal nodes each time a new motion planning request is performed.

Different methods for selecting the positions in RRTs and PRMs have been tried. Both simple methods such as sampling randomly within a distance and more advanced methods such as Gaussian sampling [12], bridge sampling [53], and sampling on the borders of obstacles [4] have been used.

In basic RRTs, each node is only connected to the predecessor in the tree and one or more successors. The main use of RRTs is to provide graphs for motion planning, and in many cases, a relatively small number of nodes of comparatively low degree is sufficient to provide coverage of an area.

In PRMs, a graph rather than a tree is created by connecting each node to all other nodes between which the correct reachability function holds, instead of just a subset of these nodes. Just like with RRTs, a relatively small number of nodes is often sufficient to cover an area.

A problem with using RRTs and PRMs to provide discretizations for the relay positioning problems is that they place nodes without any consideration to the number of hops that are required to reach the goal, as this is not a big concern in motion planning. Instead, the main concern is providing reasonably short paths with regards to distance, without any concern about how many nodes are used in the paths. This is very different from the relay positioning problems, where the number of hops in a path is very important, as the number of UAVs required to realize the path is proportional to the number of nodes in the path.

## 4.7 Summary

The continuous relay problems are computationally intractable as they typically have a disjoint feasible set and a large number of extreme points. In order to make the problems solvable in acceptable time, a discretization can be performed and the discrete problem is solved instead. A desirable characteristic of the graph is a large number of nodes at different distances and angles from each other. It is also advantageous that the nodes have high degree as this offers more options for communication and surveillance.

In this chapter, several different discretization methods have been discussed. The simple grid places nodes uniformly in the environment, while other methods such as octrees place more nodes closer to obstacles. Methods from motion planning, such as rapidly exploring random trees and probabilistic road maps can also be used, where the latter is more suitable due to higher node degree. In dense urban environments, the expanded geometry graph can be useful.

While all desirable characteristics may be fulfilled in some cases by a single discretization method, it is likely that a combination of methods will yield better results. As an example, an expanded geometry graph or a Voronoi diagram can be combined with a grid to provide a graph with a higher density of nodes at positions close to obstacles while at the same time maintaining a minimum node density further away from obstacles.

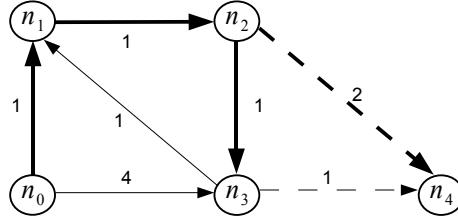
#### 4. Environment Representation and Discretization

## Relay Positioning Algorithms for Single Target Problems

Once a discrete graph has been created, for example using any of the methods discussed in Chapter 4, graph search algorithms are used for solving the single target relay problems. This chapter discusses different algorithms for calculating relay chains, and experimental results for testing the algorithms with a fixed-size, three-dimensional grid are available in Chapter 7.

Both the **STR-MinLengthMinCost** and **STR-MinCostMinLength** (see Section 3.1) problems can be solved by a common shortest path algorithm such as Dijkstra's [32]. Alternatively, the A\* algorithm can be used if a suitable heuristic is available [48]. Both problems require that optimization is first performed with respect to one factor and when the optimum of that factor is found, optimization is performed with respect to another factor. Though this may sound like a two-step process, it is commonly done using lexicographic ordering. For example, the **STR-MinLengthMinCost** problem can be solved for each node in  $N$  by using compound costs of the form  $\langle cost(\pi), len(\pi) \rangle$  with  $\langle cost(\pi_1), len(\pi_1) \rangle < \langle cost(\pi_2), len(\pi_2) \rangle$  if  $cost(\pi_1) < cost(\pi_2)$  or ( $cost(\pi_1) = cost(\pi_2)$  and  $len(\pi_1) < len(\pi_2)$ ). Thus, chains are optimized first in order of cost and then in order of number of hops (UAVs) required. The **STR-MinCostMinLength** problem is solved analogously.

Since finding a relay chain from  $n_0$  to  $\tau_1$  in a graph is equivalent to finding a path from  $n_0$  to  $\tau_1$  in the same graph, we will use the terms interchangeably. As efficient algorithms for the **STR-MinLengthMinCost** problem exists, the remainder of this chapter will explore algorithms for the **STR-MinCostLimited** and **STR-ParetoLimited** problems, beginning with the latter.



**Figure 5.1:** An example graph labeled with edge costs. The corresponding MLMC-tree is presented by heavy lines.

## 5.1 Existing Algorithms for the STR-ParetoLimited Problem

**STR-ParetoLimited** is closely related to the *all hops optimal path problem* (AHOP) graph search problem, which has previously been applied in network routing problems [46]. For each  $k = 1, 2, \dots, L$ , find a cheapest path from  $n_0$  to  $\tau_1$  among those paths whose lengths do not exceed  $k$ . We call such paths AHOP solutions for the length  $k$ . From the definition of Pareto-optimality given in Section 3.2, it follows that any Pareto-optimal solution of length  $k$  is also an AHOP solution for length  $k$ . However, the reverse is not true: a Pareto-optimal solution of length  $k$  must also have minimal length among all paths of minimal cost. This is illustrated by the example in Figure 5.1. For  $\tau_1 = n_4$ , there is only one path of length 2, namely  $\pi_1 = n_0 \rightarrow n_3 \rightarrow n_4$  with  $cost(\pi_1) = 5$ . This path is an AHOP solution for  $k = 2$ , since there is no cheaper path of length at most 2. Since there is also no equally cheap path of length strictly less than 2, it is also Pareto-optimal. There is one path of length 3,  $\pi_2 = n_0 \rightarrow n_1 \rightarrow n_2 \rightarrow n_4$  with  $cost(\pi_2) = 4$ . Again, this is both an AHOP solution for  $k = 3$  and a Pareto-optimal solution. However, consider the path  $\pi_3 = n_0 \rightarrow n_1 \rightarrow n_2 \rightarrow n_3 \rightarrow n_4$ , also of cost 4. Both  $\pi_2$  and  $\pi_3$  are AHOP solutions for  $k = 4$ , since they are of equal and minimal cost and are sufficiently short, but only  $\pi_2$  is Pareto-optimal, since it is shorter.

Though the AHOP-problem and the **STR-ParetoLimited** problem are not identical, there is a close correspondence. Suppose that we have generated a complete set  $S$  of Pareto-optimal paths and require an AHOP solution of length of at most  $k$ . If such solutions exist, the path in  $S$  having the greatest length  $k' \leq k$  will be one of them. Conversely, suppose that we have generated a complete set of AHOP solutions. Then, a complete set of Pareto-optimal solutions can be found by grouping the AHOP paths by cost and selecting a path of minimum length in each group. Thus, the discretized **STR-ParetoLimited** problem can be solved by using an AHOP algorithm and selecting the Pareto-optimal solutions from the AHOP solutions.

As a solution to the **STR-ParetoLimited** problem commonly contains several chains, common shortest path algorithms such as Dijkstra's algorithm or A\* cannot solve the

---

```

1 for each  $n \in N$  do  $g_0(n) \leftarrow +\infty$ ,  $p_0(n) \leftarrow \text{nil}$ 
2  $g_0(n_0) \leftarrow 0$ 
3 for  $k = 1, \dots, L$  do
4   for each  $n \in N$  do
5      $g_k(n) \leftarrow g_{k-1}(n)$ ,  $p_k(n) \leftarrow p_{k-1}(n)$ 
6     for each  $n \in N$  do
7       for each  $n' \in n_+$  do
8          $c \leftarrow g_{k-1}(n) + c_{n,n'}$ 
9         if  $c < g_k(n')$  then
10           $g_k(n') \leftarrow g(n)$ ,  $p_k(n') \leftarrow n$ 

```

**Figure 5.2:** Algorithm 1 – for solving the AHOP problem, by Balakrishnan and Altinkemer [8].

problem by themselves in a single execution. An early method for calculating the set of solutions to the AHOP problem is the Bellman-Ford algorithm [29]. Given a start node, the algorithm finds paths requiring at most  $k$  edges during iteration  $k$ . If a cheaper path is found later, the more expensive path is discarded. This was later modified by Lawler [61], whose algorithm in iteration  $k$  stores the cheapest path for each node, using at most  $k$  edges, without overwriting any paths found in earlier iterations. Lawler’s algorithm was further improved by Balakrishnan and Altinkemer [8], who introduced an upper bound on the number of edges so that only paths with length  $k \leq L$  are calculated. This modification allows the algorithm to solve the AHOP problem more efficiently. The pseudocode of Balakrishnan and Altinkemer’s algorithm is shown in Figure 5.2 and from now on, we will refer to it as Algorithm 1.

The following terminology is used for all algorithms. For all nodes  $n$  in  $N$ ,  $g_k(n)$  is the cost to reach node  $n$  from  $n_0$  in at most  $k$  steps and  $p_k(n)$  is the path predecessor of node  $n$  when doing so. The function  $g(n)$  is the cost of the cheapest path to  $n$  found so far, regardless of the number of steps. The predecessors of a node  $n$  are denoted by  $n_-$ , the successors are denoted by  $n_+$  and the cost of going from  $n$  to  $n'$  is  $c_{n,n'}$ .

Algorithm 1 has a time complexity of  $O(L|E|)$ . It is natural to require that  $L \leq |N| - 1$ , since no cheapest path can be longer than  $|N| - 1$ . Therefore, the time complexity of the algorithm can also be expressed as  $O(|N||E|) \subseteq O(|N|^3)$ . A complete solution to the **STR-ParetoLimited** problem is calculated for each node by Algorithm 1. Even if the cheapest path to the goal node has been found, and the length of the path is  $\ll L$ , the algorithm performs  $L$  outer iterations. This can lead to a large number of unnecessary calculations, especially if the number of nodes is large.

To remedy the long execution time, we have developed a new algorithm for the **STR-ParetoLimited** problem, presented in Section 5.2.

## 5.2 A New Label-Correcting Algorithm

Our first new algorithm (Algorithm 2) is based on the AHOP version of the Bellman-Ford algorithm (Figure 5.2) [61], and just like that algorithm, it is a label-correcting algorithm. It improves the performance of Algorithm 1 as it is able to avoid a large percentage of the unnecessary calculations. Part of the improvement stems from a preprocessing step, involving the calculation of a tree consisting of paths to all nodes. Each such path has the least number of hops among those that have the lowest cost. Such paths correspond directly to solutions to the **STR-MinLengthMinCost** problem and will be called *Minimum Length Minimum Cost paths (MLMC-paths)*. A tree of MLMC-paths from  $n_0$  to all nodes in  $N$  is called an *MLMC-tree* and can be generated by Dijkstra's algorithm using compound path costs as previously described. The paths making up the MLMC-tree provide an upper bound on the length of the Pareto-optimal paths to each node in the graph. Calculations proceed in a manner similar to Algorithm 1, but the length of the MLMC-path to each node bounds the number of iterations for each node.

### 5.2.1 Preliminaries

To present the basic ideas of our algorithm and justify its correctness, we need the following definitions, which are illustrated by Table 5.1 for the graph in Figure 5.1.

$k$	$n_0$	$n_1$	$n_2$	$n_3$	$n_4$	$N_k$	$N_k^*$
0	$0^*$	-	-	-	-	$\{n_0\}$	$\{n_0\}$
1	-	$1^*$	-	4	-	$\{n_1, n_3\}$	$\{n_1\}$
2	-	-	$2^*$	-	5	$\{n_2, n_4\}$	$\{n_2\}$
3	-	-	-	$3^*$	$4^*$	$\{n_3, n_4\}$	$\{n_3, n_4\}$

**Table 5.1:** Costs of  $k$ -hop Pareto paths, and the sets  $N_k$  and  $N_k^*$ , for the graph in Figure 5.1. The costs corresponding to MLMC-paths are marked by asterisks.

A node  $n$  is called a  *$k$ -hop Pareto node* if there exists a Pareto-optimal path of length  $k$  from  $n_0$  to  $n$ . We call such a path  *$k$ -hop Pareto-optimal*. For any  $k$ , the set of all  $k$ -hop Pareto nodes is denoted by  $N_k$ . As an example,  $n_4$  is both a 2-hop and a 3-hop Pareto node, showing that a node may be a member of multiple  $N_k$  sets. Although  $n_4$  can be reached in 4 hops with the minimal cost 4, it is not a 4-hop Pareto node as the resulting path is not Pareto-optimal. The costs for the Pareto-optimal paths are presented in Table 5.1.  $N_0 = \{n_0\}$  as the start node is the only node reachable in zero steps. The  $N_k^*$  sets store the set of nodes that occur at exactly  $k$  hops from  $n_0$  in the MLMC-tree. Let  $k_{max}^*$  denote the height of the MLMC-tree, yielding  $0 \leq k \leq k_{max}^*$  for the  $N_k^*$  sets. No Pareto-optimal path can be longer than  $k_{max}^*$  as longer paths must be at least as expensive. The  $N_k^*$  sets partition  $N$ , and as  $N_k^* \subseteq N_k$ , any path of depth  $k$  in the MLMC-tree is

$k$ -hop Pareto-optimal. Intuitively,  $N_0^* = N_0 = \{n_0\}$ .

Though the MLMC-tree may not be unique, the partitioning does not depend on the choice of tree, since the shortest path (in terms of number of hops) is chosen if several paths to the same node have the same cost.

Our algorithm exploits the following properties of Pareto-optimal paths: any Pareto-optimal path of length  $k$  must consist of a sequence of nodes occurring in  $N_0, N_1, \dots, N_k$ . That is, any non-empty Pareto-optimal path to a node  $n'$  must consist of an extension of a shorter Pareto-optimal path to a predecessor  $n$ .

### Theorem 5.1

Let  $n' \in N_k$ . Suppose that

$$\pi' = n_0 \rightarrow \dots \rightarrow n \rightarrow n'$$

is a  $k$ -hop Pareto path. Then  $n \in N_{k-1}$ , and the path  $\pi$  composed by the first  $k-1$  hops of this path is a  $(k-1)$ -hop Pareto path from  $n_0$  to  $n$ .

**Proof:** By the definition of  $k$ -hop Pareto-optimality of  $\pi'$ , there is no path  $\pi''$  from  $n_0$  to  $n$  such that:

$$\text{length}(\pi'') < \text{length}(\pi) \quad \text{and} \quad \text{cost}(\pi'') \leq \text{cost}(\pi)$$

or such that:

$$\text{length}(\pi'') = \text{length}(\pi) \quad \text{and} \quad \text{cost}(\pi'') < \text{cost}(\pi).$$

□

Theorem 5.1 implies that lines 7–10 of Algorithm 1 only have to be executed for nodes  $n \in N_{k-1}$ . The outgoing edges from other nodes cannot yield Pareto-optimal paths, nor can they result in updated node costs. The set  $N_{k-1}$  can be generated during iteration  $k-1$  and used in the next iteration.

Our new algorithm uses the following identification of  $k$ -hop Pareto nodes: we can find a cheaper path to a node  $n$  using  $k$  hops than we could using  $k-1$  hops only if  $n \in N_k$ . That is, only if there is a Pareto-optimal path to  $n$  using  $k$  hops.

$$n \in N_k \iff g_{k-1}(n) > g_k(n). \quad (5.1)$$

Since an MLMC-tree is available, execution of lines 7–10 in Algorithm 1 can be skipped for the nodes  $n' \in N_k^*$ , as they are known to be  $k$ -hop Pareto nodes, and the costs  $g_k(n')$  of the corresponding  $k$ -hop Pareto-optimal paths are already available in the MLMC-tree. Both of these facts are used to limit the number of calculations in our new algorithm.

The difference between the two algorithms is illustrated in Table 5.1, which shows all the values of  $g_k(n)$  produced by our new algorithm. In contrast, Algorithm 1 produces the values of  $g_k(n)$  for every node  $n$  and for each  $k = 1, \dots, 4$ . Furthermore, it makes the same calculations for producing identical values, for instance,  $g_1(n_3) = g_2(n_3) = 4$  and  $g_3(n_3) = g_4(n_3) = 3$ . For the same node, our algorithm calculates  $g_1(n_3)$  only, and  $g_3(n_3)$  is provided by the MLMC-tree.

$k$ (path length)	$g_k$ (cost)	$p_k$ (predecessor)
1	4	$n_0$
3	3	$n_2$

**Table 5.2:** Reachability records for node  $n_3$  after execution of Algorithm 2 on the graph in Figure 5.1.

## 5.2.2 Algorithm Details

The algorithm incrementally generates and updates a set of *reachability records* for each node. This set can be represented as one table for each node, as seen in Table 5.2, which displays the reachability records for the node  $n_3$  in the graph in Figure 5.1. Each reachability record  $\langle k, g_k, p_k \rangle$  associated with a node indicates that the node can be reached from the start node  $n_0$  in  $k$  hops at a cost of  $g_k$  using the predecessor  $p_k$ . A complete path from  $n_0$  to  $n$  can always be reconstructed by considering the reachability record of the predecessor  $p_k$  for  $k - 1$  hops and continuing recursively until  $n_0$  is reached. As the reachability records are traversed from  $n$  to  $n_0$ , the path needs to be reversed after retrieval. In our application, the number of hops corresponds to the number of agents required to realize the chain.

The first row of the table corresponds to using the minimal number of hops and the following rows correspond to using an increasingly larger number of hops, giving a path with progressively lower cost, until the least-cost path with the largest number of hops, which is found on the last row. For any number of hops smaller than the smallest  $k$  in the table, no chain can be found. For example, Table 5.2 shows that no chain of length  $k = 0$  could be found, regardless of cost. Other “missing” values of  $k$  indicate that even if chains of length  $k$  exist, they are not Pareto-optimal. For example, the fact that Table 5.2 contains no record for  $k = 2$  means that a chain of length 2 would be at least as expensive as a chain of length 1, the nearest smaller value of  $k$ . Thus, using a chain of length 2 would be pointless. However, using a chain of length 3 could be a useful alternative, as this would reduce the cost to 3 (in other words, increase the quality of the chain).

After termination, each reachability record is guaranteed to correspond to a Pareto-optimal path. Conversely, if there is a Pareto-optimal path of length  $l$  between  $n_0$  and  $n$ , then  $n$  is guaranteed to have a reachability record for  $k = l$ . Thus, the fact that a target node  $\tau_1$  is associated with a reachability record  $\langle k, g_k, p_k \rangle$  corresponds exactly to the existence of a Pareto-optimal relay chain between  $n_0$  and  $\tau_1$  of length  $k$  and cost  $g_k$ , allowing  $\tau_1$  to be reached from the base station node using  $k - 1$  intermediate UAVs, of which one is the surveillance UAV.

**Preprocessing.** Our algorithm uses a preprocessing step (line 0) consisting of calculating the MLMC-tree, using for example Dijkstra’s algorithm. In the tree, the chain to

---

```

0 Calculate MLMC-tree and extract  $k_{max}^*$  and all  $N_k^*$ 
1 for each  $n \in N \setminus \{n_0\}$  do  $g(n) \leftarrow +\infty$ 
2 for each  $n \in n_0_-$  do           // Incoming edges...
3    $E \leftarrow E \setminus \{(n, n_0)\}$       // ...are removed
4    $N_0 \leftarrow \{n_0\}$ 
5 for  $k = 1, \dots, \min\{M + 1, k_{max}^* - 1\}$  do
6   for each  $n' \in N_k^*$  do
7     for each  $n \in n'_-$  do           // Incoming edges...
8        $E \leftarrow E \setminus \{(n, n')\}$     // ...are removed
9      $N_k \leftarrow N_k^*$ 
10    for each  $n \in N_{k-1}$  do
11      for each  $n' \in n_+$  do
12         $c \leftarrow g_{k-1}(n) + c_{n,n'} \quad // To n' through n in k hops$ 
13        if  $c < g(n')$  then
14           $g(n') \leftarrow c \quad // Lowest cost so far$ 
15           $g_k(n') \leftarrow c \quad // Lowest cost in k hops$ 
16           $p_k(n') \leftarrow n \quad // Predecessor for k hops$ 
17         $N_k \leftarrow N_k \cup \{n'\}$ 

```

**Figure 5.3:** Algorithm 2 – MLMC-tree-based label-correcting algorithm.

each node is the longest, and thus cheapest, to each node in the graph. We then retrieve the height  $k_{max}^*$  of this tree. This limits the number of relays required for any optimal relay chain for this graph, and also the depth to which the graph needs to be searched. For all  $k$ , we also extract the set  $N_k^*$  of all nodes of depth  $k$  in the tree.

For all nodes, we create an initial reachability record corresponding to the cheapest Pareto-optimal relay chain for each node. This allows us to abort execution individually for each node, as no cheaper path can be found for lengths longer than the length of the MLMC-path. In Table 5.2, the initial record corresponds to the row where  $k = 3$ , and signifies that no path longer than three steps can decrease the path cost.

**Initialization.** Initially,  $g(n) = \infty$  is set for all nodes. Then the algorithm constructs and uses a sequence of sets  $N_k$ , each of which is characterized, according to Theorem 5.1, by the fact that any  $k$ -hop Pareto-optimal path must consist of a  $(k-1)$ -hop Pareto-optimal path to a node  $n \in N_{k-1}$  followed by a single outgoing edge from  $n$ .

Lines 2–4 provide initial values for the initial node  $n_0$ . Since  $N_0^* = \{n_0\}$ , a reachability record with  $g_0(n_0) = 0$  must have been created during the preprocessing step, which indicates that it can be reached with cost 0 in 0 hops. Since the value  $g(n)$  where  $n \in N_k^*$  is not used in iteration  $k$  or any subsequent iteration, no value of  $g(n_0)$  is assigned. Clearly no chain ending in  $n_0$  can improve the value  $g_0(n_0) = 0$ , so all incoming edges to  $n_0$  can be removed (lines 2–3). Finally, line 4 prepares for the first iteration by

setting  $N_0 = \{n_0\}$ , indicating that any 1-hop Pareto-optimal path must consist of a path to  $n_0$  in 0 hops (the empty path) followed by a single outgoing edge. This line handles all paths of length 0.

**Main Part.** The longest and thus cheapest Pareto-optimal path to each node was found during the preprocessing, and the main part of the algorithm begins by calculating the shortest (and thus most expensive) Pareto-optimal path. Then paths are found in order of increasing path length and decreasing cost.

Each iteration of lines 5–17, considers paths of length  $k \geq 1$ , up to a maximum of  $\min\{M + 1, k_{\max}^* - 1\}$ . The upper bound reflects the fact that: (i) we allow at most  $M$  hops (UAVs), yielding a total path length of up to  $M + 1$  when edges to the base station and target are included, (ii) no paths of length greater than  $k_{\max}^*$  can be Pareto-optimal, due to the meaning of  $k_{\max}^*$ , and (iii) any Pareto-optimal path of length exactly  $k_{\max}^*$  was already generated during the preprocessing.

Recall that any node  $n' \in N_k^*$  occurs at depth  $k$  in the MLMC-tree. Reachability records for strictly shorter paths to  $n'$  were already created in earlier iterations, and a record for a path of length  $k$  was created during the preprocessing. Records for paths of length strictly greater than  $k$  cannot be created, as this would correspond to finding a path, which is longer but strictly cheaper than the one of length  $k$ , which was by definition a cheapest path. Thus, no new paths ending in any such  $n'$  can be Pareto-optimal, and we can remove all incoming edges to  $n'$  without affecting correctness or optimality (lines 6–8). However, we need to consider longer paths going *through* these nodes (line 9, explained further below).

In lines 10–17, we consider all potentially Pareto-optimal relay chains of length  $k$ . Any such chain must consist of a path to a node  $n \in N_k$  followed by a single outgoing edge from  $n$ . We consider each such path in turn, determining its destination  $n'$  and calculating its cost  $c$  (lines 10–12). If the path has lower cost than the cheapest path found previously (with a cost of  $g(n')$ ) we create a new reachability record with  $g_k(n')$  set to the new path cost and  $p_k(n')$  set to the new predecessor  $n$  (lines 13–16). Note that though reachability records are sparse, we know that any node in the set  $N_{k-1}$  does have a reachability record for  $k - 1$  due to the construction of such sets.

What remains is to prepare for the next iteration by constructing  $N_k$  according to its definition and characterization (relation 5.1). That is, we should construct  $N_k$  in such a way that any  $(k + 1)$ -hop Pareto-optimal relay chain necessarily consists of a path of length  $k$  to a node  $n \in N_k$  followed by a single outgoing edge from  $n$ . It is clear that no Pareto-optimal chain would use  $k$  hops to reach a node  $n$  if it could be reached in  $k - 1$  hops without incurring additional costs. This is stated in relation (5.1) which also covers the cases when  $n$  cannot be reached at all with paths of length  $k - 1$ . Thus, it is only necessary to consider nodes whose costs were decreased by allowing paths of length  $k$ . This is achieved in line 9, for nodes in  $N_k^*$  where we found a cheapest path of length  $k$  during the preprocessing, and in line 17, for nodes where we found a path of length  $k$  and of lower cost in this iteration.

For each node, Algorithm 2 produces a complete set of Pareto-optimal paths. In the preprocessing stage, when the MLMC-tree is created, it generates the longest and cheapest Pareto-optimal path, which is a path of minimum length among the paths of minimum cost. It then finds the shortest and most expensive Pareto-optimal path, and continues in order of increasing length and decreasing cost.

### 5.2.3 Correctness Proof

To prove that the variables and sets calculated by Algorithm 2 are correct, we use the notation presented in Section 5.2.1.

#### Theorem 5.2

*After iteration  $k$  of Algorithm 2, the calculated  $N_k$  sets defines the set of all  $k$ -hop Pareto nodes. For each  $n' \in N_k$ , the variable  $g_k(n')$  defines the cost of a  $k$ -hop Pareto-optimal path from  $n_0$  to  $n'$  and  $p_k(n')$  defines the predecessor of  $n'$  in this path. For each  $n' \in N$ ,  $g(n')$  defines the cost of a cheapest path using at most  $k$  hops from  $n_0$  to  $n'$  such that  $n' \in N_k^*$  with  $k' > k$ .*

**Proof:** In order to distinguish  $N_k$ ,  $g(n')$ ,  $g_k(n')$  and  $p_k(n')$  from the sets and variables generated by Algorithm 2, we will use the notation  $N_k^A$ ,  $g^A(n')$ ,  $g_k^A(n')$ ,  $p_k^A(n')$  for those that are created by Algorithm 2. The claims in the theorem can be formulated as follows:

1.  $N_k^A = N_k$ ,
2.  $g_k^A(n') = g_k(n')$ ,  $\forall n' \in N_k$ ,
3.  $p_k^A(n') = p_k(n')$ ,  $\forall n' \in N_k$ ,
4.  $g^A(n') = g_k(n')$ ,  $\forall n' \in N$  such that  $n' \in N_{k'}^*$ , with  $k' > k$ .

We will prove by induction that the claims in the theorem hold for  $k = 0, \dots, \min\{M + 1, k_{max}^* - 1\}$ . All line numbers refer to the pseudocode in Figure 5.3.

The claims 1–4 hold for  $k = 0$  as the start node  $n_0$  is the only node that is reachable in zero steps and the cost of such a path is  $g_0(n_0) = 0$  with  $p_0(n_0) = \text{nil}$ . As only  $n_0$  is reachable in  $k = 0$  steps,  $N_0 = N_0^* = \{n_0\}$  applies. For all other nodes  $n' \in N$ ,  $g(n') = \infty$ . In Algorithm 2, the cost  $g_0^A(n_0) = 0$  and the predecessor  $p_0^A(n_0)$  are assigned during execution of the MLMC-tree. All  $N_k^*$  sets are extracted after the execution of the MLMC-tree (line 0). As  $n_0$  is the only node reachable in zero step,  $N_0^A = N_0^* = \{n_0\}$  must apply. The cost  $g(n') = \infty$ ,  $\forall n' \in N \setminus \{n_0\}$  is set in line 2.

Assume that the claims hold for some  $k = i < \min\{M + 1, k_{max}^* - 1\}$ .

We will show that the claims 1–4 apply for  $k = i + 1$ . We begin by proving that  $N_{i+1}^A$  correctly defines the set of all  $(i + 1)$ -hop Pareto nodes. Line 9 implies that if  $n' \in N_{i+1}^*$ , then  $n' \in N_{i+1}^A$ . To prove that  $N_{i+1}^A$  correctly defines the set of all  $(i + 1)$ -hop Pareto-nodes, we must prove that i) for all  $n' \in N_{i+1}^A \setminus N_{i+1}^*$ ,  $n'$  is an  $(i + 1)$ -hop Pareto-node, and ii) if  $n' \notin N_{i+1}^A$ , then  $n'$  is not an  $(i + 1)$ -hop Pareto-node. We begin by proving the former.

For node  $n' \in N_{i+1}^A \setminus N_{i+1}^*$ , the existence of an  $(i + 1)$ -hop path from  $n_0$  to  $n'$  with a cost of less than the cost of a cheapest path to  $n'$  with at most  $i$  hops, is equivalent to  $n'$  being an  $(i + 1)$ -hop Pareto node, by (5.1). By the induction hypothesis, at the end of iteration  $k = i$ , the value of  $g^A(n')$  is the cost of the cheapest path using at most  $i$  hops. The case when  $n'$  is added to  $N_{i+1}^A$  on line 17 corresponds to the case of finding an  $(i + 1)$ -hop path with a cost less than  $g^A(n')$ . Thus  $n'$  is an  $(i + 1)$ -hop Pareto node according to (5.1).

To prove ii), consider  $n' \notin N_{i+1}^A$ . By i),  $n' \notin N_{i+1}^*$ . Assume by contrary that  $n' \in N_{i+1} \setminus N_{i+1}^*$ . By Theorem 1 and (5.1) there should exist some  $n \in N_i$  such that  $g_i(n') > g_{i+1}(n) + c_{n,n'}$ . By claims 1 and 4,  $n \in N_i^A$ . Then  $c < g(n')$  in line 13 must hold at least once. This implies that  $n' \in N_{i+1}^A$ , which contradicts our assumption that  $n' \in N_{i+1} \setminus N_{i+1}^*$ . Thus ii) holds.

For any  $n' \in N_{i+1}^A$ , the for-loop in lines 10–17 assures that the equality

$$g_{i+1}^A(n') = \min_{n \in n'_- \cap N_i} \{g_i^A(n) + c_{n,n'}\}.$$

holds. The right hand side is equal to  $g_{i+1}(n')$  by its definition. Thus claim 2 holds for  $k = i + 1$ . As any change in  $g_{i+1}^A(n')$  is associated with a change in  $p_{i+1}^A(n')$ , claim 3 also holds.

To prove claim 4, we consider the two cases iii)  $n' \in N_{i+1} \setminus N_{i+1}^*$  and iv)  $n' \notin N_{i+1}$ . Note that claim 4 does not refer to  $n' \in N_{i+1}$ . For case iii), lines 14–15 implies that  $g^A(n')$  changes at iteration  $i + 1$  only if  $g_{i+1}^A(n')$  changes, and if  $g_{i+1}^A(n')$  changes at iteration  $i + 1$ , then  $g^A(n') = g_{i+1}^A(n')$ . Then as claims 1 and 2 were earlier shown to hold for  $k = i + 1$ , claim 4 holds for  $n \in N_{i+1} \setminus N_{i+1}^*$ . For case iv),  $g^A(n')$  has the same value at the end of iterations  $i$  and  $i + 1$ . Also,  $g_i(n') = g_{i+1}(n')$ . Assuming that  $n' \in N_k^*$  with  $k' > i + 1$ , then by claim 4 for  $i = k$ , the value of  $g^A(n')$  is equal to  $g_i(n')$  at the end of iteration  $i$ . The same holds for  $g^A(n')$  at the end of iteration  $i + 1$ . Thus claim 4 applies for both case iii) and iv). This completes the proof for  $k = i + 1$ .

As the claims hold  $k = 0$ , by induction they hold for  $k = 0, \dots, \min\{M + 1, k_{\max}^* - 1\}$  as this is the exact number of performed iterations.  $\square$

See [17, 15, 16] for additional details and proofs.

### 5.2.4 Time Complexity

The label-correcting algorithm presented here performs a calculation of the MLMC-tree, which can be performed in  $O(|E| + |N| \log |N|)$  time using Dijkstra's algorithm (line 0). Clearing the costs on line 1 takes  $O(|N|)$  time, and lines 2–4 requires at most  $|E|$  time and the time complexity for the preprocessing is  $O(|E| + |N| \log |N|)$ . At most  $k_{\max}^* - 1$  outer iterations are performed, where each iteration will go through  $|E| \leq |N|^2$  edges and the time complexity of the main part of the algorithm is  $O((k_{\max}^* - 1)|E|) \subseteq O((k_{\max}^* - 1)|N|^2)$ . Thus the time complexity for the complete algorithm is

$O(k_{\max}^* |N|^2) \subseteq O(|N|^3)$ . The time complexity is the same as for the original Bellman-Ford, but typically,  $k_{\max}^* \ll |N|$  applies for our problems. Also,  $|E| \ll |N|^2$  often applies as each node is only connected to a small subset of nodes.

### 5.2.5 Improved Preprocessing

As can be seen on line 6 in Figure 5.3, the number of main iterations is limited by  $\min(M + 1, k_{\max}^* - 1)$ . From that, it is obvious that no unnecessary iterations are performed in the main part of Algorithm 2.

However, if the graph consists of a very large number of nodes and the graph topology is such that  $k_{\max}^* \gg M + 1$ , then some unnecessary calculations will be performed during the preprocessing step, for nodes that will not be reached in the main part of the algorithm. Iterations for such nodes can be removed by introducing a depth-limit in the calculation of the cheapest path tree. When the depth of a node reaches  $M + 1$ , the node cannot be part of a relay chain unless the node is the target node. This fact is exploited on line 5 in Algorithm 2, and no calculations starting in the node will be performed. As the value of  $k_{\max}^*$  is retrieved *after* the calculation of the MLMC-tree, a depth limit of  $M + 1$  is the strongest depth limit of the tree calculation that can be used without affecting the algorithm's properties.

### 5.2.6 Example

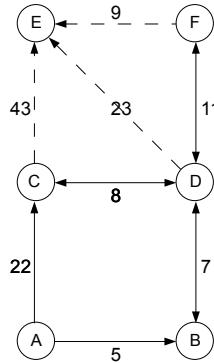
We will now show how the algorithm works in a small example, displayed in Figure 5.4a, with  $n_0 = A$ ,  $\tau_1 = E$  and  $M = \infty$ . Communication edges are solid and surveillance edges are dashed.

During the preprocessing, the MLMC-tree marked by the heavy edges in Figure 5.4b is calculated. The reachability records  $\langle \text{hops}, \text{cost}, \text{predecessor} \rangle$  are displayed next to each node.

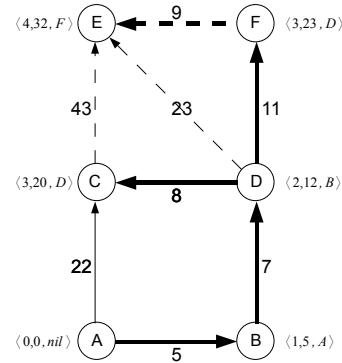
For node  $E$ , the chain is  $A \rightarrow B \rightarrow D \rightarrow F \rightarrow E$ . From the MLMC-tree, the  $N_k^*$  sets are determined:  $N_0^* = \{A\}$ ,  $N_1^* = \{B\}$ ,  $N_2^* = \{D\}$ ,  $N_3^* = \{C, F\}$ ,  $N_4^* = \{E\}$ . As  $k_{\max}^* = 4$  the number of iterations is limited to 3. Initially, the node cost  $g$  is set for all nodes,  $g(A) = 0$ , and  $g(n) = \infty$  for all other nodes, and any incoming edges to the start node are removed. The graph after these modifications is displayed in Figure 5.4c.  $N_0 = N_0^* = \{A\}$  is set before the first iteration.

At the start of the first iteration, all incoming edges to the nodes in  $N_1^* = \{B\}$  are removed. All outgoing edges of the nodes in  $N_0 = \{A\}$  are treated. A new chain of length 1 and cost 22 is found to node  $C$ . As  $g(C) = \infty$ , the new chain is stored and  $g(C)$  is updated (Figure 5.4d). The set of nodes for treatment in the next iteration,  $N_1$ , is constructed during this iteration and consists of the nodes  $B$  and  $C$ .  $N_1 = \{B, C\}$ .

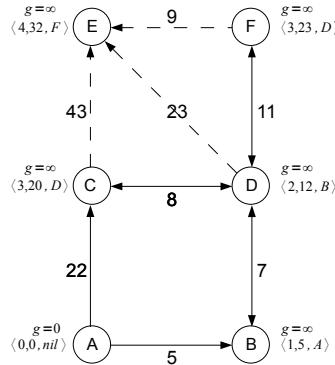
In the second iteration, all incoming edges to the nodes  $N_2^* = \{D\}$  are removed. Next, the outgoing edges of nodes in  $N_1 = \{B, C\}$  are treated, and a chain of length 2 and cost 65 to node  $E$  is found and stored:  $A \rightarrow C \rightarrow E$ . As the edge from node  $B$  to  $D$



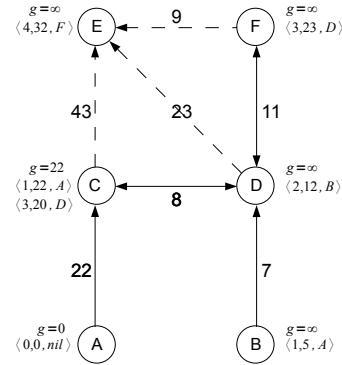
(a) The initial graph.



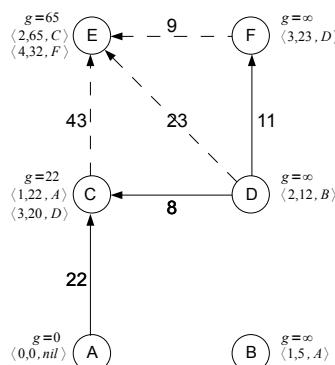
(b) MLMC-tree calculated during preprocessing.



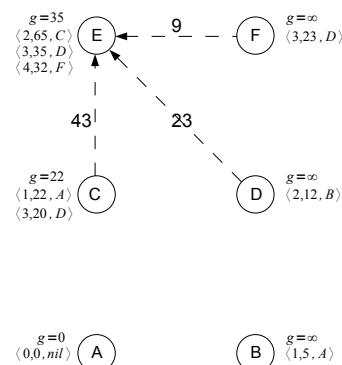
(c) Before the first iteration, with node costs set.



(d) After the first iteration.



(e) After the second iteration.



(f) After the third iteration.

**Figure 5.4:** Example showing the execution of the label-correcting algorithm.

Length	Cost	Chain
2	65	$A \rightarrow C \rightarrow E$
3	35	$A \rightarrow B \rightarrow D \rightarrow E$
4	32	$A \rightarrow B \rightarrow D \rightarrow F \rightarrow E$

**Table 5.3:** The chains to node  $E$  for the graph in Figure 5.4.

was removed, no new chain involving node  $B$  can be found.  $N_2 = \{D, E\}$ . Figure 5.4e displays the graph at this stage.

Iteration 3 is the final iteration as  $k_{max}^* = 4$ . Incoming edges to  $N_3^* = \{C, F\}$  are removed. All nodes in  $N_2 = \{D, E\}$  are treated and a new chain of length 3 and cost 35 to node  $E$  is found:  $A \rightarrow B \rightarrow D \rightarrow E$ . Node  $E$  has no outgoing edges, thus no new chain from it can be found.  $N_4 = \{E\}$ . The final graph is displayed in Figure 5.4f. All chains to node  $E$  and their costs are displayed in Table 5.3.

## 5.3 A New Dual Ascent Algorithm

The **STR-MinCostLimited** problem corresponds directly to the shortest path problem with hop-constraints [31], where “shortest” in this case means “lowest cost”. This problem does not require finding all chains, only a sufficiently short chain. To solve the problem, label-correcting algorithms such as Algorithm 1 are often used, and a single sufficiently short chain is extracted from the solution. However, doing so requires calculating a potentially large set of relay chains, and discarding all but one. As Algorithm 2 performs fewer calculations and should have a shorter execution time, we limit our discussion to that algorithm. During preprocessing, the longest chain is calculated, and then the shortest and most expensive chain is calculated in the first iteration. In later iterations, progressively longer and less expensive chains are calculated. Thus, if the longest chain has only marginally too many hops, calculations must continue until all shorter chains have been calculated. There is no advantage to the fact that the first chain was “almost” feasible.

We therefore present another algorithm (Algorithm 3) that begins with calculating the cheapest chain and then calculates more expensive chains, which are shorter. This type of algorithm is based on the dual ascent approach to optimization. From a mathematical perspective, it tries to maximize the piece-wise affine Lagrangian dual function by traversing a dual segment at a time.

As mentioned previously, common shortest path algorithms such as Dijkstra’s cannot be used to calculate a complete set of relay chains directly. However, such algorithms can be used to calculate a single relay chain, provided that one exists. Any shorter chain is more expensive, and for a cheapest path algorithm to find such a chain, the edge costs

must be increased. If all edge costs are increased by an equal amount, the cost of a longer path is increased proportionally more than a shorter path. If all edge costs are increased by  $\epsilon$ , a chain of length  $k$  has its cost increased by  $k\epsilon$ . The cost of another relay chain, of length  $m > k$ , is increased by  $m\epsilon$ . Thus, the cost of the longer chain is increased by  $(m - k)\epsilon$  more than the cost of the shorter chain. This is the basic idea behind the dual ascent algorithm, which repeatedly calculates a MLMC-tree and an edge cost increase  $\epsilon$ . As all edge costs are increased by  $\epsilon$ , progressively shorter paths are found as the costs of longer chains are increased more than the costs of shorter chains.

### 5.3.1 Algorithm Details

The pseudocode for the dual ascent algorithm is presented in Figure 5.5.

The parameter  $l$  is the number of acceptable hops and is initialized to  $M + 1$ , with the one added for the last hop between the surveillance UAV and the target. The parameter  $\alpha$ , representing the extra cost that is added to each edge, is initialized to  $\alpha_0$ , which is commonly set to 0 to ensure that all paths are found (line 1). Other values of  $\alpha_0$  are possible, but at the risk of decreasing the path length too much, thus finding a more expensive chain than required. The chain between  $n_0$  and  $\tau_1$  is calculated through repeated calculations of the MLMC-tree, using modified edge costs  $c'_{n,n'} = c_{n,n'} + \alpha$  (line 3). From the MLMC-tree, the chain  $\pi$  from  $n_0$  to  $\tau_1$  is retrieved, together with the depth  $q_n$  and the optimal chain cost  $y_n$ , for all nodes. The depth  $q_n$  is the number of hops required to reach node  $n$  from  $n_0$  in the MLMC-tree, and the optimal path cost  $y_n$  is the minimum cost required to reach node  $n$  from  $n_0$  given this value of  $\alpha$  (line 3).

If the chain between  $n_0$  and  $\tau_1$  is short enough, i.e. the number of hops is less than  $l$ , then a valid solution has been found and the algorithm terminates (line 4).

The next step in the algorithm (line 5) is to find the edges that could potentially be

```

1   $\alpha \leftarrow \alpha_0, l \leftarrow M + 1$ 
2  loop
3      Calculate MLMC-tree from  $n_0$  using costs  $c'_{n,n'} = c_{n,n'} + \alpha$ .
      From the tree, obtain  $\pi$  and  $y_n, q_n$  for all  $n \in N$ 
4      if  $length(\pi) \leq l$  then return  $\pi$  // Feasible solution
5       $H \leftarrow \{(n, n') \in E : q_{n'} \geq q_n + 2\}$ 
6      if  $H = \emptyset$  then return failure // No chain can be shortened
7      for  $\{n, n'\} \in H$  do
8           $\epsilon_{n,n'} \leftarrow \frac{c'_{n,n'} + y_n - y_{n'}}{q_{n'} - q_n - 1}$ 
9           $\epsilon \leftarrow \min \epsilon_{n,n'}$ 
10          $\alpha \leftarrow \alpha + \epsilon$ 
```

**Figure 5.5:** Algorithm 3 – dual ascent algorithm.

included in the MLMC-tree in order to decrease the length of a chain. The condition for an edge that could be added to the MLMC-tree is that the number of hops required to reach the edge end node  $q_{n'}$  is at least two more than the number of hops required to reach the edge start node  $q_n$ . If the difference in the number of hops was one, the edge could already be included in the tree and would not make any path in the tree shorter, thus the difference in depth must be at least 2. The set  $H$  consists of all edges with a depth difference of  $\geq 2$  between the nodes. If  $H = \emptyset$  then we already have a tree of minimum depth and no edge can be included in order to decrease the length of any path in the tree, and the algorithm terminates (line 6).

An iteration through  $H$  is performed (lines 7–8), with the intention to calculate the edge cost increase that should be applied to all edge costs in order to decrease the length of at least one path in the tree. Since we know that the edges in  $H$  are not already included in the tree, going through  $n$  to reach  $n'$  is more expensive than the current path to  $n$ , i.e.  $y_n + c'_{n',n} > y_{n'}$  holds for all edges in  $H$ . Making the path to  $n'$  through  $n$  equally expensive requires increasing the cost by  $y_n + c'_{n',n} - y_{n'}$ , more than the cost of the shorter path. The cost increase must be distributed over  $q_{n'} - (q_n + 1) = q_{n'} - q_n - 1$  edges, yielding an increase per edge of  $\frac{c'_{n',n} + y_n - y_{n'}}{q_{n'} - q_n - 1}$ .

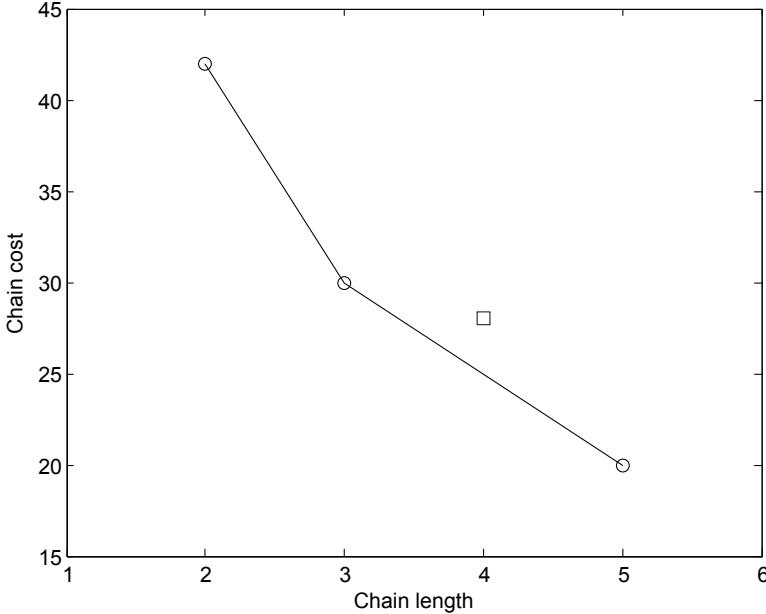
To ensure that no chain on the problem instance's convex hull (see Section 5.3.2) is missed when the edge cost increase is applied to all edge costs, the minimum  $\epsilon_{n,n'}$  is used and it is added to the value of  $\alpha$ , on lines 9–10. This will increase the cost of the shorter path by exactly the amount required to make the shorter path equally expensive as the longer path, and a shorter path will be preferred in such situation due to use of the compound cost (see page 39). This concludes one iteration of the algorithm, and the process of calculating the MLMC-tree is then repeated, using the new value of  $\alpha$  (line 3).

The edge(s) yielding the minimal  $\epsilon$  is normally the set of edges that will be included in the MLMC-tree and the set of edges ending in same nodes as the included edges will leave the tree. When several edges in  $H$  have the same end node, only one of the edges is included in the tree.

Although the dual ascent algorithm is intended to create solutions to the **STR-MinCost-Limited** problem, it can be modified to solve the **STR-ParetoLimited** problem. For this problem, line 4 does not return the chain  $\pi$ , but rather it is yielded as a partial solution and execution continues until  $H = \emptyset$ .

### 5.3.2 Theoretical Properties

The dual ascent algorithm follows the problem instance's convex hull and finds all Pareto-optimal solutions on the hull. Figure 5.6 shows the Pareto-optimal relay chains marked by circles, and the convex hull of a particular problem. Any Pareto-optimal chain not on the convex hull of the problem instance, such as the relay chain marked by a square, will not be found. Thus, the algorithm is not optimal for the **STR-MinCostLimited** problem, as it occasionally will find a slightly shorter chain than the longest feasible. However, Pareto-optimal chains that are not on the convex hull can be found using a branch-and-bound



**Figure 5.6:** The convex hull of a problem instance is marked by a solid line and the Pareto-optimal relay chains on the hull are marked with circles. The square marks a Pareto-optimal chain that will not be found by the dual ascent algorithm.

post-processing step [18]. As the cheapest and the shortest chains are on the convex hull, they will always be found.

**Monotonicity With Respect to  $\alpha$ .** We will now prove that increasing the value of  $\alpha$  yields shorter but more expensive chains.

### Theorem 5.3

Let  $n \in N$  be an arbitrary node distinct from, but reachable from, the start node  $n_0$ . Consider the paths  $\pi_1$  and  $\pi_2$  generated from  $n_0$  to  $n$  in the MLMC-tree for two distinct values of  $\alpha$ , denoted by  $\alpha_1$  and  $\alpha_2$ . If  $0 \leq \alpha_1 < \alpha_2$ , then  $\text{cost}(\pi_1) \leq \text{cost}(\pi_2)$  and  $\text{length}(\pi_1) \geq \text{length}(\pi_2)$ . That is, increasing  $\alpha$  cannot increase the length or decrease the true cost of an optimal path from  $n_0$ .

**Proof:** Here we make use of two distinct path cost measures: the true cost of a path  $\pi$  is defined in terms of the communication and surveillance costs only, and is denoted by  $\text{cost}(\pi)$ . As the edge costs are incremented by  $\alpha$ , we define the modified cost of  $\pi$  given a value of  $\alpha$  to be  $\text{cost}_{\text{mod}}(\pi, \alpha)$ . Obviously,  $\text{cost}_{\text{mod}}(\pi, \alpha) = \text{cost}(\pi) + \alpha \cdot \text{length}(\pi)$ .

It is clear that  $\text{cost}_{\text{mod}}(\pi_1, \alpha_1) \leq \text{cost}_{\text{mod}}(\pi_2, \alpha_1)$ , since  $\pi_1$  is a cheapest path for  $\alpha = \alpha_1$ . Analogously,  $\text{cost}_{\text{mod}}(\pi_2, \alpha_2) \leq \text{cost}_{\text{mod}}(\pi_1, \alpha_2)$  must apply.

Suppose that  $\text{cost}(\pi_2) < \text{cost}(\pi_1)$ , so that using a larger  $\alpha$  resulted in a decrease in true path cost. If the paths were of equal length, changing  $\alpha$  would always change the modified costs of  $\pi_2$  and  $\pi_1$  by the same amount, and  $\pi_2$  would have been strictly preferred over  $\pi_1$  for all values of  $\alpha$ . As  $\pi_1$  was returned for  $\alpha = \alpha_1$ , this cannot be the case. Suppose instead that  $\text{length}(\pi_2) > \text{length}(\pi_1)$ . Then the modified cost of  $\pi_2$  would increase by a greater amount than the modified cost of  $\pi_1$  when  $\alpha$  is increased. Thus, if  $\pi_1$  was returned for a smaller value of  $\alpha$ ,  $\pi_2$  cannot be preferred for a larger value of  $\alpha$ . The case where  $\text{length}(\pi_2) < \text{length}(\pi_1)$  remains. But if  $\pi_2$  is shorter and has a lower true cost, then its modified cost must be less for any positive value of  $\alpha$ , and we cannot have  $\text{cost}_{\text{mod}}(\pi_1, \alpha_1) \leq \text{cost}_{\text{mod}}(\pi_2, \alpha_1)$ . Thus, it must be the case that  $\text{cost}(\pi_2) \geq \text{cost}(\pi_1)$ .

Suppose that  $\text{length}(\pi_2) > \text{length}(\pi_1)$ , so that using a larger  $\alpha$  resulted in an increase in path length. We know that  $\text{cost}_{\text{mod}}(\pi_1, \alpha_1) \leq \text{cost}_{\text{mod}}(\pi_2, \alpha_1)$ . When  $\alpha$  is increased from  $\alpha_1$  to  $\alpha_2$ , the modified cost of  $\pi_2$  must increase by a strictly greater amount than the modified cost of  $\pi_1$ , since  $\pi_2$  is longer and modified costs increase in proportion to length. Thus,  $\text{cost}_{\text{mod}}(\pi_1, \alpha_2) < \text{cost}_{\text{mod}}(\pi_2, \alpha_2)$ , which contradicts  $\text{cost}_{\text{mod}}(\pi_2, \alpha_2) \leq \text{cost}_{\text{mod}}(\pi_1, \alpha_2)$ . Thus  $\text{length}(\pi_2) \leq \text{length}(\pi_1)$ .  $\square$

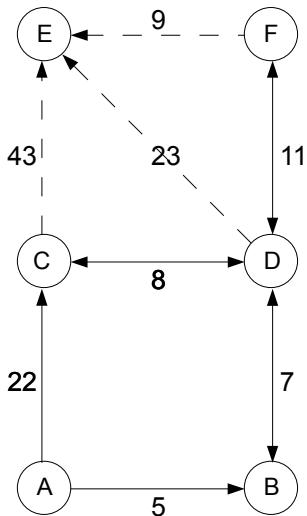
**Convergence and Time Complexity.** Let  $T_D$  denote the tree depth of the MLMC-tree, calculated as  $T_D = \sum_{n \in N} q_n$ .  $T_D$  is bounded from below by  $|N| - 1$  as the tree with the least depth is star shaped with one node in the middle having depth 0 and the other  $|N| - 1$  nodes having depth 1, yielding  $T_D = |N| - 1$ . The tree with the maximum depth has two nodes with degree one, and all other nodes have degree two, i.e. a single path. In that case, the first node has  $q_n = 0$ , the second node  $q_n = 1$  and so on, yielding  $T_D = \frac{|N|(|N|-1)}{2}$ . As this is the maximum possible depth we have  $|N| - 1 \leq T_D \leq \frac{|N|(|N|-1)}{2}$ . From this it is apparent that the number of possible iterations is bounded by  $|N|^2$ . In each iteration, each of the  $|E| \leq |N|^2$  edges is checked for inclusion in the MLMC-tree, and the MLMC-tree is calculated. Such a tree can be computed in  $O(|E| + |N| \log |N|)$  time. Therefore, the maximum time complexity is  $O(|N|^2(|E| + |E| + |N| \log |N|)) \subseteq O(|N|^4)$ . In most cases, this is a severe overestimation as graphs requiring  $|N|^2$  iterations are not common on the relay problems.

From Figure 5.5, it is evident that only edges with  $q_{n'} \geq q_n + 2$  can enter the MLMC-tree. When such an edge is included in the MLMC-tree, it will decrease the depth of node  $n'$  and all nodes in the subtree rooted in  $n'$ . As the depth will not be increased for any node,  $T_D$  will decrease. As  $T_D$  is decreased by at least one in every iteration, the algorithm converges.

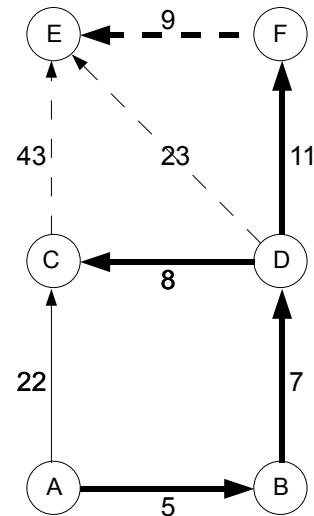
For additional proofs, the reader is referred to [18].

### 5.3.3 Example

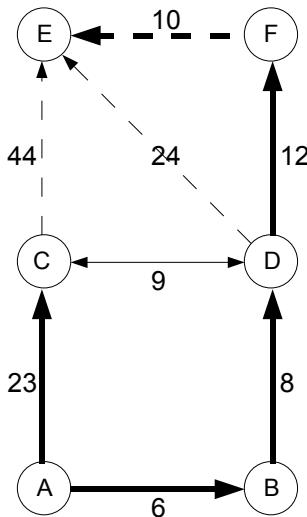
We will now show how the dual ascent algorithm works for finding a chain from  $n_0 = A$  to  $\tau_1 = E$  with a length of at most three hops. The start value  $\alpha_0 = 0$  and the initial graph is displayed in Figure 5.7a.



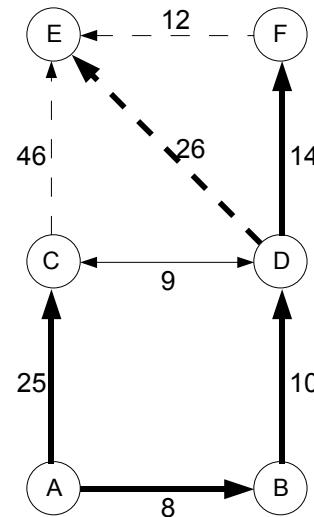
(a) Initial graph.



(b) The MLMC-tree calculated in iteration 1 is marked by heavy edges.



(c) In iteration 2, a new MLMC-tree is calculated after  $\alpha = 1$  has been added to all edge costs.



(d) A sufficiently short chain is found in iteration 3.

**Figure 5.7:** The dual ascent algorithm is applied to finding a chain from A to E of at most three hops.

In iteration 1, the MLMC-tree-is calculated and the cheapest chain,  $\pi_1 = A \rightarrow B \rightarrow D \rightarrow F \rightarrow E$ ,  $cost(\pi_1) = 32$  and  $length(\pi_1) = 4$ , is found (Figure 5.7b). The set  $H = \{(A, C), (D, E)\}$ . Calculating  $\epsilon$  for  $(A, C)$  with the variables  $c_{A,C} = 22, y_A = 0, y_C = 20, q_A = 0, q_C = 3$  yielding  $\epsilon_{A,C} = \frac{22+0-20}{3-0-1} = 1$ . Analogously, the edge  $(D, E)$  yields  $\epsilon_{D,E} = \frac{23+12-32}{4-2-1} = 3$ . The cost increase  $\alpha = \epsilon_{A,C} = 1$  is applied to all edge costs.

In the second iteration, the new edge costs are used when the MLMC-tree is calculated.  $\pi_1$  is still the cheapest chain ( $cost_{mod}(\pi_1) = 36$ ) and the new MLMC-tree is displayed in Figure 5.7c. The edge  $(A, C)$  enters the MLMC-tree and the edge  $(D, C)$  leaves it. The length of the chain to  $C$  is decreased, but the chain to  $E$  is not affected. In this iteration,  $H = \{(C, E), (D, E)\}$ , which yields  $\epsilon_{C,E} = \frac{44+23-36}{4-1-1} = 15.5$  and  $\epsilon_{D,E} = \frac{24+14-36}{4-2-1} = 2$

All edge costs are increased by  $\epsilon = \epsilon_{D,E} = 2$  before calculating the new MLMC-tree in iteration 3 (Figure 5.7d). A new chain  $\pi_2 = A \rightarrow B \rightarrow D \rightarrow E$  is found. As  $cost_{mod}(\pi_1) = cost_{mod}(\pi_2) = 44$  and  $length(\pi_2) = 3$ ,  $\pi_2$  will be preferred as it requires fewer hops. The chain  $\pi_2$  is sufficiently short and the algorithm terminates.

### 5.3.4 Performance Improvements

The basic dual ascent algorithm as displayed in Figure 5.5 allows for several performance optimizations, which have been used in our empirical testing.

**Optimized Generation of MLMC-trees.** For all iterations except the first, the generation of an MLMC-tree can be optimized by making use of the fact that an MLMC-tree has already been calculated, albeit with different modified edge costs.

First, the cost of each edge in the previously created tree is increased by the recently calculated  $\epsilon$ , and the cost of each node  $n$  is increased by  $q_n\alpha$  (that is, the node cost is increased in proportion to its depth). This ensures that the cost  $y_n$  of any node  $n$  correctly reflects the new modified cost of the path from  $n_0$  to  $n$ .

After the increase in edge costs, the tree is no longer an MLMC-tree. For some nodes, it is possible to find cheaper paths in the graph than those that are currently included in the tree. Any such paths must include at least one of the edges that yielded the current value of  $\epsilon$ . We can therefore begin “repairing” the tree starting at the source nodes of those edges, rather than from the root node  $n_0$ , by initializing Dijkstra’s algorithm with a priority queue containing exactly those source nodes. As the tree is traversed in the standard manner, only those nodes to which we find a cheaper path than in the previous calculation of the tree are added to the priority queue. Thus, parts of the tree where the chain is already optimal will not be visited, speeding up the calculation of the tree. This way of starting from an existing tree is sometimes referred to as Ford’s algorithm [56].

**Search Space Limitation.** As seen in the example in Section 5.3.3, the dual ascent algorithm decreases  $T_D$  in each iteration, but the chain between  $n_0$  and  $\tau_1$  might not be affected. Depending on the difference in edge costs and which edge determines  $\epsilon_{n,n'}$ ,

there can be many iterations that do not affect the chain between  $n_0$  and  $\tau_1$ . However, to increase the chance that the relevant chain is affected, the search space can be limited. To guarantee that no relevant chains are missed, the desired effect is to remove the parts of the graph that are irrelevant for the current target.

An example of such irrelevant nodes are the nodes that require too many steps from  $n_0$ . Such nodes can be removed, and to determine the number of steps required to reach the node from  $n_0$ , a *shortest path tree* starting in  $n_0$  can be used. This tree is calculated by using for example Dijkstra's algorithm and setting all edge costs to 1. The output of such a calculation is the minimum number of hops required to reach each node, denoted by  $q_n^{SP} \forall n \in N$ . After the shortest path tree has been calculated, all nodes not fulfilling  $q_n^{SP} \leq M + 1$  are removed from  $N$ , as they require too many hops to be reached. Naturally, if it is not possible to reach a node using the maximum number of steps allowed, it cannot be part of a valid path between  $n_0$  and  $\tau_1$  and it can be safely removed.

The same pruning of the search space can be performed by calculating another shortest path tree, this time rooted in  $\tau_1$ . Let  $q_n^{SPT}$  denote the number of hops required to reach node  $n$  from node  $\tau_1$ . After calculation of the shortest path tree rooted in  $\tau_1$ , the value of  $q_n^{SPT}$  is set for all reachable nodes. As both  $q_n^{SP}$  and  $q_n^{SPT}$  are available, the condition of  $q_n^{SP} + q_n^{SPT} \leq l$  can be applied. Nodes not satisfying this condition can be removed in order to increase the performance of the algorithm, without affecting the optimality or completeness properties. Obviously, if  $q_n^{SP}$  hops are required to reach node  $n$  from  $n_0$ , a valid path between  $n$  and  $\tau_1$  may require no more than  $l - q_n^{SP}$  hops.

Limiting the search space in this way can potentially improve performance more for small values of  $M$ , as larger parts of the search space are discarded. Thus, it is not certain that a lower value of  $M$  yields slower execution, instead the search space limitation may yield large improvements when solving the **STR-MinCostLimited** problem.

This way of improving performance has also been suggested as a preprocessing step for the NP-hard problem of finding shortest paths fulfilling both a limit on the number of hops and a limit on the path cost [39].

## 5.4 Summary

Of the single target relay problems defined earlier, efficient algorithms exist for the **STR-MinLengthMinCost** and **STR-MinCostMinLength** problems. In this chapter, new algorithms for solving the **STR-MinCostLimited** and the **STR-ParetoLimited** problems have been presented. The **STR-ParetoLimited** problem is solved using a new label-correcting algorithm. The algorithm uses a preprocessing step consisting of calculating an MLMC-tree from the start node to all nodes in the graph. Each path in such a tree consists of the path having the fewest steps from the set of cheapest paths. The tree provides an upper bound on the path length to each node, and this is used to terminate calculations for each node in the main part of the algorithm. This avoids many unnecessary calculations, which improves the execution time significantly, as demonstrated in Chapter 7.

The **STR-MinCostLimited** problem can also be solved by the label-correcting algorithm, but as it only requires finding a sufficiently short path, other methods may be more efficient. This problem can be solved using a dual ascent algorithm, which repeatedly calculates an MLMC-tree. After each calculation of the tree, it is checked whether the path from the base station node  $n_0$  to the target node  $\tau_1$  is sufficiently short. If so, the path is a feasible solution and the algorithm terminates. If not, the algorithm calculates and applies a cost increase to all edge costs. The cost increase is calculated in such a way that at least one path in the tree is shortened, and the process of repeatedly calculating the MLMC-tree and the edge cost increase is continued until a sufficiently short path is found or until no path in the tree can be shortened.



# 6

---

## Relay Positioning for Multiple Targets

We have previously presented algorithms for calculating relay chains for surveillance of a single target. In this chapter, we will discuss how simultaneous surveillance of several targets can be modeled. We also discuss algorithms for positioning relays in such cases.

Assume that there are several targets that must be surveilled. In such a case, one option is to use the algorithms already described to calculate several chains, creating one independent relay chain to each target. If the UAVs are only capable of relaying a single stream of information, then this is the best that we can do. However, if the UAVs can handle several streams of information, then we can take advantage of this and calculate a *relay tree* where some UAVs relay information from several surveillance UAVs. This synergy may allow us to decrease the number of UAVs required to surveil the targets. We will now go on and define problems for surveillance of multiple targets and then discuss algorithms for solving them.

### 6.1 Definition of the Multiple Target Relay Problems

For problems involving several targets, the same assumptions as for single targets are made (see Section 3.1), but instead of a single target, assume as given a set  $T = \{t_1, \dots, t_l\} \subseteq R^3 \setminus U$  of  $l$  surveillance target positions.

A *relay tree* between  $x_0$  and the target positions  $\{t_1, \dots, t_l\}$  is a set of relay chains that together form a tree structure. As an example, consider Figure 1.2 on page 6, where there are two chains:  $[x_0, x_1, x_2, x_3, x_5, x_6, t_1]$  and  $[x_0, x_1, x_2, x_3, x_4, t_2]$ . Note that these chains may share positions, corresponding to a UAV being used in multiple chains. For each target  $t_i \in T$ , there is a relay chain beginning in  $x_0$  and ending in  $t_i$ . Let  $L$  be the number of UAVs required to realize the tree and let the non-target positions in the tree be denoted by  $[(x_0, \dots, x_L)]$ . In Figure 1.2, there are six such UAV positions:  $x_1 - x_6$ .

## 6. Relay Positioning for Multiple Targets

---

Also, let  $x^-$  denote the unique predecessor of position  $x$ . For example, in Figure 1.2 the predecessor of  $x_5$  is  $x_3$ :  $x_5^- = x_3$ . Then, the cost of a relay tree is

$$\sum_{i=1}^L c_{comm}(x_i^-, x_i) + \sum_{i=1}^l c_{surv}(t_i^-, t_i)$$

Let the *length* of a relay tree be the number of agents required to realize the chain, that is  $L + 1$  for  $L$  UAVs and the base station.

With the necessary definitions in place, we define the following multiple target relay problems. Some of the targets assume a limit on the number of available UAVs, and just like in the single target case, we use the letter  $M$  to denote this limit. Setting  $M = \infty$  requires finding all solutions, regardless of length.

**MTR-MinLengthMinCost:** Find a relay tree of minimum length among the trees of minimum cost. A solution to this problem is a tree  $s$  such that for all other trees  $t$ ,  $cost(s) \leq cost(t)$  and  $length(s) = length(t) \rightarrow cost(s) \leq cost(t)$ . This corresponds to using the highest quality tree that can be realized with access to an unlimited number of UAVs, with a preference for using fewer UAVs if this is possible without compromising quality.

**MTR-MinCostMinLength:** Find a relay tree of minimum cost among the trees of minimum length. A solution to this problem is a tree  $s$  such that for all other trees  $t$ ,  $length(s) \leq length(t)$  and  $length(s) = length(t) \rightarrow cost(s) \leq cost(t)$ . This is useful if minimizing the number of UAVs is strictly more important than maximizing quality.

**MTR-MinCostLimited:** Find a relay tree of minimal cost among the chains that use at most  $M$  UAVs. A solution to this problem is a tree  $s$  such that  $length(s) \leq M + 1$  for all other trees  $t$ , and  $length(t) \leq M + 1 \rightarrow cost(s) \leq cost(t)$ . This corresponds to a desire to find the highest quality relay tree that can be realized within the given limit on the number of UAVs.

**MTR-ParetoLimited:** Find a set of *Pareto-optimal* relay trees that is complete up to a given upper limit on the number of available UAVs. A tree  $s$  is Pareto-optimal for up to  $M$  UAVs if  $length(s) \leq M + 1$  and for all trees  $t$  of length at most  $M + 1$ ,  $length(t) < length(s) \rightarrow cost(t) > cost(s)$  and  $cost(t) < cost(s) \rightarrow length(t) > length(s)$ .

As we will see, these problems are difficult to solve quickly. Therefore, we will focus on finding algorithms suitable for calculating approximate solutions to the **MTR-MinLengthMinCost** and **MTR-MinCostMinLength** problems in the rest of this thesis. Solving these problems requires minimization of the total cost or the length of the tree. This makes the multiple target positioning problems similar to *Steiner tree* problems.

## 6.2 Relation to Steiner Tree Problems

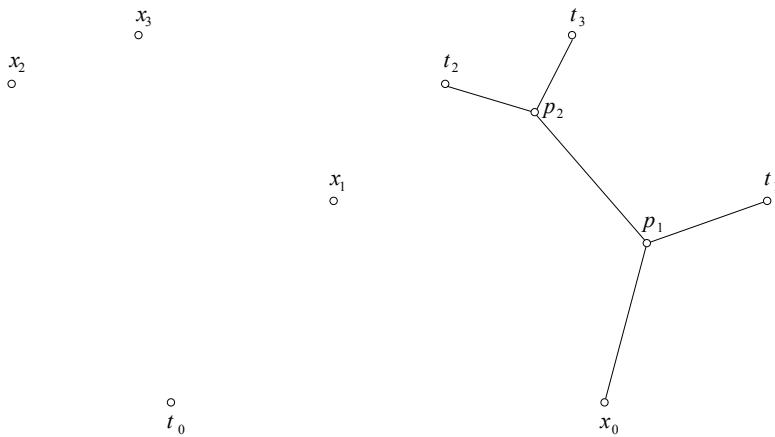
The Steiner tree problem is a well-known optimization problem that exists in both continuous and discrete variants. Steiner tree problems are in general NP-hard and occur in

practical applications such as VLSI routing, telecommunication and transportation, and have attracted considerable research efforts. A comprehensive list of applications and algorithms is beyond the scope of this thesis, and we refer to the surveys by Winter [100], Du et al. [38, 37] and Hwang et al. [54].

Common for both the continuous and discrete variants is the term *terminals*, denoted by  $Z$ . This is the set of positions that the tree must span, in our case consisting of the base station and all targets. In the continuous problems, this is a set of points, and in the discrete problems it is a set of nodes. In the general Steiner tree problem, the terminals are not required to be leaves in the tree.

### 6.2.1 Continuous Steiner Trees

The continuous Steiner tree problem consists of connecting a given set of terminals  $Z$  by lines of minimum total length so that any two terminals are connected either directly or via other terminals and lines. In the context of the relay problems,  $Z = T \cup \{x_0\}$ . Any non-terminal point in the Steiner tree, where two or more lines meet, is called a *Steiner point*. Figure 6.1a displays a two-dimensional Steiner tree problem with  $Z = \{x_0, t_1, t_2, t_3\}$ . A solution with two Steiner points,  $p_1$  and  $p_2$ , is displayed in Figure 6.1b. In the continuous Steiner tree problem, there is no need to specify the positions where Steiner points can be located, as all positions can be used.



(a) A continuous Steiner tree problem with four terminals.

(b) A solution to the problem in Figure 6.1a, with two Steiner points:  $p_1$  and  $p_2$ .

**Figure 6.1:** A two-dimensional continuous Steiner tree problem.

To be able to solve the general multiple target relay problems, the following requirements all must be handled: i) three-dimensional environments, ii) realistically sized environments, iii) environments with obstacles, iv) cost functions that do not obey the triangle-

inequality, since we may sometimes achieve higher quality relay trees through using a larger number of UAVs and longer transmission paths, and v) limited maximum length of lines, corresponding to a limited communication range.

There are algorithms can handle some of the requirements individually, but not all requirements simultaneously. An example of such an algorithm is the algorithm by Fampa and Anstreicher [42], that can handle three-dimensional environments. Such algorithms can be of interest under very particular circumstances, such as when the reachability functions do not use a range limitation, the cost functions obey the triangle inequality and there are no obstacles in the environment. Similarly, there are algorithms that calculate a Steiner tree in a plane and are able to handle obstacles [101]. Such algorithms are only of interest if the placement of UAVs can be restricted to a plane.

In particular, the requirement that algorithms must be able to handle arbitrary cost functions prohibits the use of current algorithms for continuous Steiner trees. Therefore, we discretize the search space and solve discretized approximations of the multiple target relay problems.

### 6.2.2 Discrete Steiner Trees

Given a graph  $G(N, E)$ , the discrete Steiner tree problem consist of finding a minimum cost tree  $T_G$  that spans the terminals  $Z \subseteq N$ , in such a way that all terminals are connected by edges in  $E$ . In the continuous Steiner tree problem, Steiner points can be placed anywhere. In the discrete Steiner tree problems, the set of possible positions are the nodes where we can place UAVs in the discretization discussed in Chapter 4. We will later describe how graphs for use in the discrete Steiner tree problems can be created, as well as different variants of the discrete Steiner tree problem, in both undirected and directed graphs. However, we first introduce some common notation.

**Notation.** Given a Steiner tree  $T_G$ , its nodes and edges are denoted by  $N(T_G)$  and  $E(T_G)$ , respectively. We follow the notation in e.g. [97, 93, 26] and use the term *Steiner nodes* to denote the non-terminal nodes  $S(T_G) = N(T_G) \setminus Z$ . Each Steiner node corresponds to the use of a UAV. As the number of UAVs is denoted by  $L$  (see Section 6.1),  $L = |S(T_G)|$  applies and a tree is feasible if  $L \leq M$ . As for relay chains,  $|E(T_G)| = |N(T_G)| - 1$ . Thus, as the number of targets is fixed for each problem, and as there must be a single base station, minimizing the number of nodes or edges/hops in a relay tree is equivalent to minimizing the number of UAVs required to realize the tree.

In some nodes in  $N(T_G)$ , a path “splits” into two or more, which represents the fact that they receive and relay information from several UAVs. As an example, consider  $x_3$  in Figure 1.2 on page 6. Such nodes are called *join nodes* and have two or more outgoing connections. We denote the set of join nodes in  $T_G$  by  $J(T_G)$ . Conceptually, the join nodes correspond to the Steiner points in the continuous problem.

**Related Problems.** The Steiner tree problem in an undirected graph is similar to the minimum spanning tree problem as both problems require that nodes in a graph are connected, and that the resulting tree has minimum cost. A difference is that  $Z = N$  for the minimum spanning tree, as all nodes must be connected in the minimum spanning tree problem. The Steiner tree problem requires that the set of terminals  $Z$  are connected and that the cost of the tree is minimized. This makes the general Steiner tree problem more difficult than the minimum spanning tree problem.

While many Steiner tree problems are known to be NP-hard, there are some exceptions. The aforementioned minimum spanning tree is known to be optimally solvable in polynomial time [29]. For  $|Z| = 1$ , the problem is trivial as only a single node is involved. When  $|Z| = 2$ , the problem is the shortest path problem, known to be optimally solvable in polynomial time. Thus, both the cheapest path problem and the minimum spanning tree are special cases of the more general Steiner tree problem. The case when  $|Z| = 3$  is also possible to solve optimally in polynomial time, which is discussed further in Section 6.5.

**Discretization.** Creating directed graphs for the multiple target problems can be done using the same method as for a single target, with a few modifications. Replace steps 4 and 7 in the method described in Section 4.1 with the corresponding steps below.

- 4'. For each target position  $t_i \in \{t_1, \dots, t_l\}$  a new target node  $\tau_i$  is created. Let  $\mathcal{T} = \{\tau_1, \dots, \tau_l\}$  be the set of all target nodes and let  $Z = \mathcal{T} \cup \{n_0\}$  be the set of terminals.
- 7'. For each  $\tau_i \in \mathcal{T}$  corresponding to  $t_i$  and for each  $x \in U'$  corresponding to  $n \in N$  and satisfying  $f_{surv}(x, t_i)$ , create a directed edge  $e = (n, \tau_i)$  of cost  $c_{surv}(x, t_i)$  representing the fact that a surveillance UAV at  $x$  would be able to surveil the target at  $t_i$ .

Undirected graphs can be created in a similar manner as directed graphs. The main difference is in the reachability and cost functions. If such a function is asymmetric, then this is almost certainly due to the fact that the activity that the function is modeling is asymmetric in nature. For example, if a camera is mounted on a UAV's belly and can only see below the UAV, then the surveillance reachability function holds only if the UAV is located above the target. However, it is known in which direction any given surveillance edge will be used, as exactly one of its endpoints is a surveillance target. An undirected surveillance edge can therefore be given a cost corresponding to this specific direction of surveillance.

Most communication functions are symmetric, and many of those that are not can be adjusted to become symmetric. For example, consider the communication cost function based on obstructed volume (Section 3.3.2). The cost is based on the position  $x$  only. Let the nodes  $n$  and  $n'$  correspond to positions  $x$  and  $x'$  respectively. The cost of the undirected edge between the two nodes can be set to  $c_{x,x'} = \frac{V_{ob}(x) + V_{ob}(x')}{2V_{comm}}$ . An asymmetric cost function can often be used like this to determine edge costs in an undirected graph.

**The Steiner Minimum Tree Problem in Undirected Graphs.** The Steiner Minimum Tree (SMT) in an undirected graph is defined as:

*Given an undirected graph  $G = (N, E)$ , an edge cost function  $c : E \rightarrow R^+$  and a non-empty subset  $Z \subseteq N$  of terminals. Find a tree  $T_G$  such that there is a path between every pair of terminals and the total cost( $T_G$ ) =  $\sum_{e \in E(T_G)} c(e)$  is minimized.*

As previously mentioned, some restricted variations of the Steiner tree problems such as the cheapest path problem and the minimum spanning tree problem are optimally solvable in polynomial time. As the general SMT is NP-hard, the time required for calculating optimal solutions is often prohibitively long. For this reason, heuristic algorithms are in many cases used in practice [100].

The majority of heuristics for the SMT can be broadly divided into three categories: path heuristics, tree heuristics and vertex heuristics. The path heuristics are based on cheapest path algorithms. The tree heuristics are based on constructing a tree spanning all terminals, which is then improved to decrease the cost. Vertex heuristics attempt to first find good Steiner nodes, and then calculate a tree spanning the terminals and the Steiner nodes. For an overview of heuristics for the SMT problem, the reader is referred to Hwang et al. [54].

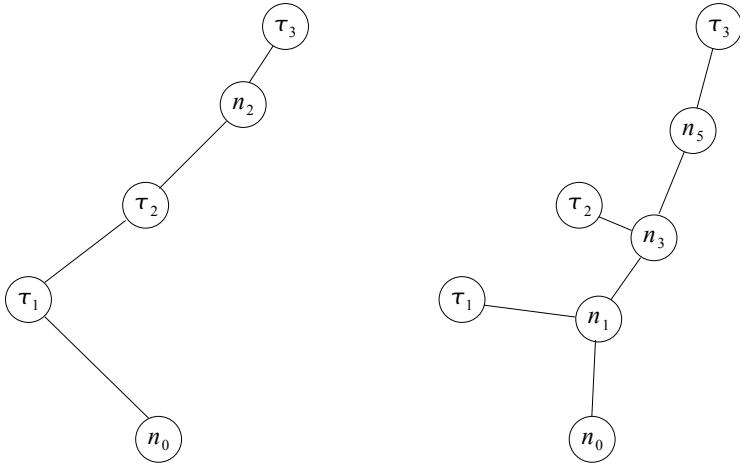
Heuristics are often chosen on basis of the *approximation ratio*  $\rho$ . The approximation ratio guarantees that the cost of the solution calculated by the heuristic is no more than  $\rho$  times the cost of the optimal solution. The currently<sup>1</sup> best known approximation ratio for the general Steiner tree problem in an undirected graph is  $\rho = \ln(4) + \epsilon < 1.39$  [19].

However, in the SMT problem, any terminal can be a join node. In the multiple target relay problems, there is a big difference between the target nodes and the base station node, which may be connected to an arbitrary number of nodes. No target node can be a join node as all targets must be leaves in the relay tree. That is, each target node must be connected to the rest of the tree via a single edge. This is because the targets are arbitrary objects and cannot be used to relay information. The single edge between an inner node and the target node corresponds to a surveillance UAV (the inner node) surveilling a target. Figure 6.2a shows an example which is a valid solution to the SMT but is not a valid relay tree. The target node  $\tau_1$  is connected directly to the target node  $\tau_2$ , which in turn is connected to the node  $n_2$ , which is surveilling the target node  $\tau_3$ . Transferring information from the surveillance UAV at  $n_2$  would require the cooperation of the targets  $\tau_1$  and  $\tau_2$ . The Steiner tree in Figure 6.2b is a valid relay tree as each target node is connected to the rest of the tree by a single edge.

As the SMT problem does not distinguish between terminals and allows any terminal to be connected to multiple nodes, it cannot be used to model the multiple target relay problems. However, the multiple target relay problems can be modeled as several other Steiner tree problems.

---

<sup>1</sup>A previous version mentioned  $\rho = 1 + \frac{\ln(3)}{2} \approx 1.55$  from the paper by Robins and Zelikovsky [82].



**(a)** In a general Steiner tree, terminals may be connected via several edges.

**(b)** As each target node is connected to the rest of the tree with a single edge, this Steiner tree is a valid relay tree.

**Figure 6.2:** Solutions to Steiner tree problems are not necessarily solutions to the multiple target relay problems.

**Terminal Steiner Trees.** The *terminal Steiner tree problem* uses an undirected graph, but requires all terminals to be leaves in the solution. This fits our problem well although it is unnecessarily restrictive as it does not allow several UAVs to communicate directly to the base station. This restriction can be circumvented by adding an extra node, which is connected to the base station via an edge with cost zero. Then the extra node is used as a terminal instead of the base station. This allows us to model the multiple target relay problems as a terminal Steiner tree problem.

A generalization of the terminal Steiner tree problem is the *partial terminal Steiner tree problem* (PTSTP), which takes two sets of nodes as input: one set in which all nodes must be leaves and one set of nodes that do not have to be leaves. This fits our problem better as the targets must be leaves while the base station node does not have to be a leaf. Just like the terminal Steiner tree problem, the PTSTP uses an undirected graph. Currently there are only two published algorithms for the PTSTP [51, 52].

The currently best known approximation ratio for the terminal Steiner tree problem is  $2\rho - \frac{\rho}{3\rho-2} \approx 2.14$  (assuming an approximation ratio of  $\rho = 1.39$  for the SMT problem). The reason for this approximation ratio is that the algorithm first constructs a Steiner tree and then modifies it to become a terminal Steiner tree, which may increase the cost of the original tree by a certain factor. If edge costs are either 1 or 2, the approximation

ratio is lowered<sup>2</sup> to 1.42 [66]. For the partial terminal Steiner tree, the lowest known approximation ratio is  $2\rho - \frac{\rho}{3\rho-2} - \epsilon = 2.14 - \epsilon$  (assuming  $\rho = 1.39$ ), where  $0 \leq \epsilon \leq \rho - \frac{\rho}{3\rho-2}$  [52].

The algorithms for (partial) Steiner trees require that the triangle inequality applies and that the graph is *complete*. A graph is complete if edges between all pairs of nodes exist. Below we discuss the implications of these requirements in detail.

In the basic Steiner tree problem, it is possible to make the triangle inequality apply by changing the cost of any edge that is more expensive than the cheapest path between the nodes. The cost of such an edge is set to the cost of the cheapest path. This causes the triangle inequality to apply, and a solution can be calculated as in the normal case. Then it is checked whether any edge with a changed cost is included in the solution. If so, the edge is replaced by the cheapest path that had the actual cost. However, this method cannot be used in the terminal Steiner tree problems, as shown by Figure 6.3. Here we want to calculate a partial terminal Steiner tree where  $\{\tau_1, \tau_2\}$  must be leaves and  $n_0$  does not have to be a leaf. Assume that the algorithm with an approximation ratio of  $2.14 - \epsilon$  is used.

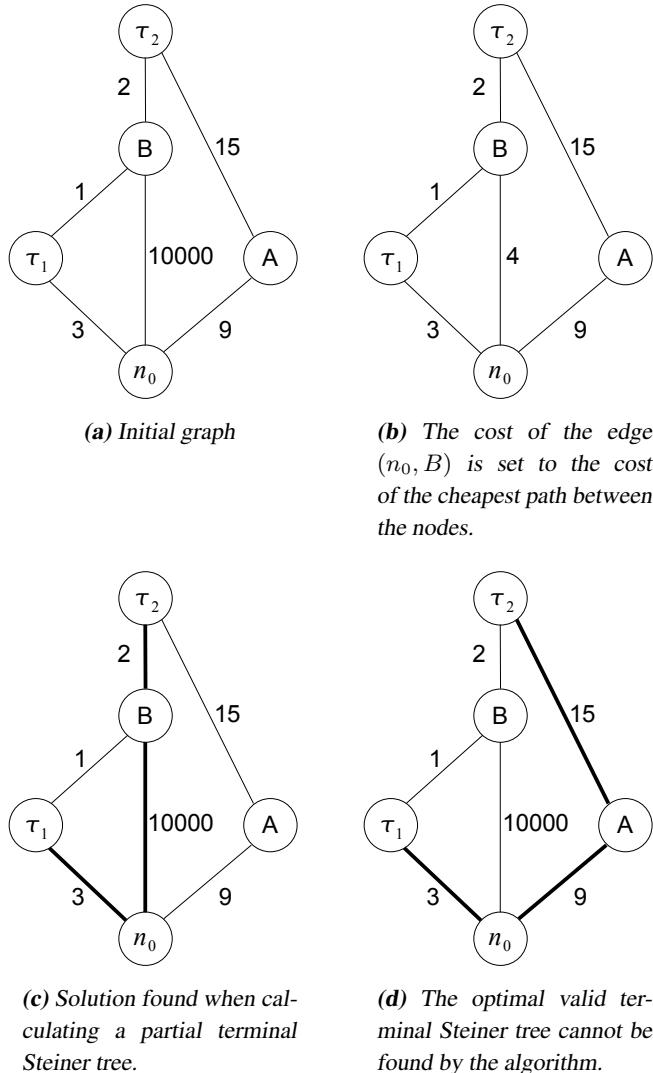
Figure 6.3a shows the original graph and Figure 6.3b shows the graph after the cost of the edge  $(n_0, B)$  is set to the cost of the cheapest path between  $n_0$  and  $B$ , i.e. 4. Then the algorithm is executed, which is only guaranteed to find a tree with a cost within a factor  $2.14 - \epsilon$  of the optimal cost. The optimal tree has cost 9 and the tree found by the algorithm may thus cost up to  $9 * (2.14 - \epsilon) = 19.26 - 9\epsilon \leq 19.26$ . Assume that the optimal tree is found by the algorithm. However, when edge  $(n_0, B)$  is replaced by the cheapest path between the nodes, the tree is no longer a valid partial terminal Steiner tree as the path goes through a terminal. Instead, the real edge  $(n_0, B)$  must be used. However, this edge has a cost of 10000, giving a total tree cost of 10005 (Figure 6.3c).

Furthermore, the algorithm can never find the optimal partial terminal Steiner tree, depicted in Figure 6.3d, as it has cost 27 and is outside the guaranteed approximation ratio. This shows that this method for making the triangle inequality apply cannot be used for (partial) terminal Steiner trees and consequently not for calculating solutions to the multiple target relay problems.

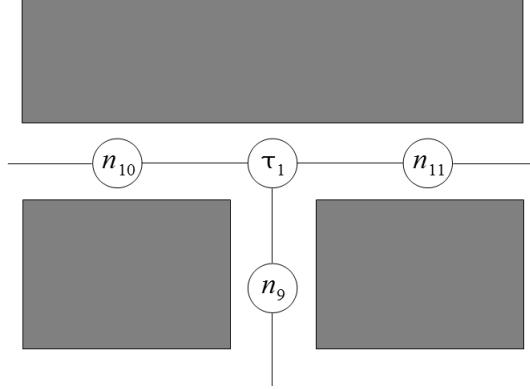
Another option is to only consider instances of the terminal Steiner tree problems where, for each terminal, there is a non-terminal with exactly the same set of neighbors [36]. In such cases, it is possible to transform a terminal Steiner tree problem in which the triangle inequality does not hold into a problem where it does hold. However, this restriction cannot be used in general in the multiple target relay problems. Suppose that when each target node  $\tau_i$  is created, an extra step is performed, with the intention of placing a node  $s_i$  so that it gets the same set of neighbors as  $\tau_i$ . There must then be some distance between  $\tau_i$  and  $s_i$ , as it must be possible to locate a UAV at  $s_i$  without colliding with the target or any other object in the environment. Also, all nodes that could *surveil* the target  $\tau_i$  must be able to *communicate with*  $s_i$ . As the communication

---

<sup>2</sup>This was more relevant in a previous version, when  $\rho = 1.55$ , and is included here for completeness.



**Figure 6.3:** Setting the cost of any edge that is more expensive than the cheapest path between the nodes to the cost of the cheapest path, causes the triangle inequality to apply. However, this method cannot be used in the terminal Steiner tree problems.



**Figure 6.4:** In some cases it is impossible to place another node so that it gets the same set of neighbors as a target node. This is especially evident in urban environments.

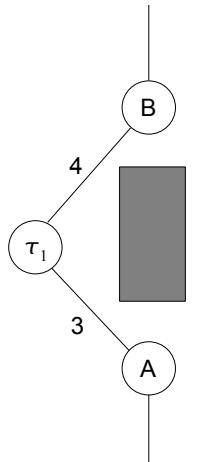
and surveillance reachability functions may be completely different, this may prevent  $s_i$  from getting the desired set of neighbors, especially if we consider the terrain around  $\tau_i$ . The terrain is most likely to cause problems in urban environments, as exemplified by Figure 6.4. Regardless of where  $s_i$  is placed, it will not get the same set of neighbors as  $\tau_i$ , when common requirements such as free line-of-sight and limited range are used in the reachability functions.

Even if we assume that a cost function obeying the triangle inequality is used, the requirement of a complete graph is difficult to fulfill. The graphs that are used in the relay problems are the results of discretizations of real environments. As such, it is very unlikely that the graphs are complete as there almost certainly are obstacles preventing this. Even if there are no obstacles, the reachability functions generally do not allow the creation of a complete graph due to limited maximum range and similar limitations.

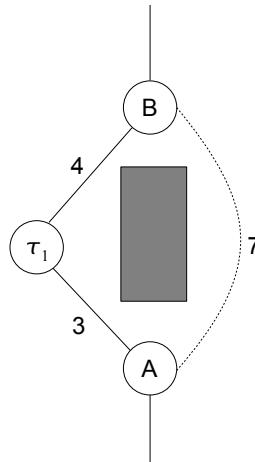
Instead, an incomplete graph must be made complete. One method is to let any missing edge be replaced by the cheapest path between the nodes [36]. This is similar to the above example of making the triangle inequality apply, but the difference is that in that example, there were edges with too high cost, while here there are edges missing.

The idea is to first replace the missing edges to make the graph complete, and then execute an algorithm that requires a complete graph. After this is done, any added edges that are used in the solution are replaced by the cheapest path. However, we will show that this method cannot be used in the relay problems as this may cause a path that originally did not pass through a target node to do exactly that.

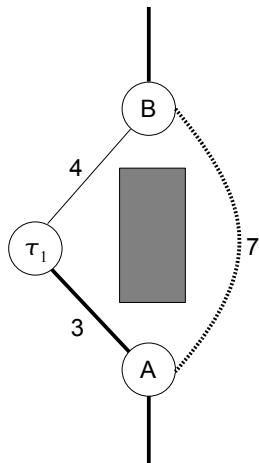
A part of a graph is displayed in Figure 6.5a: the terminal  $\tau_1$  is the only path between the nodes  $A$  and  $B$ , which in turn are connected to other nodes that are not displayed. As part of making the graph complete, a new, “pseudo edge” between  $A$  and  $B$  is inserted (Figure 6.5b). The new edge consists of the cheapest path between  $A$  and  $B$ , in this case  $A \rightarrow \tau_1 \rightarrow B$  and the edge cost is the cost of that path. After the graph completion



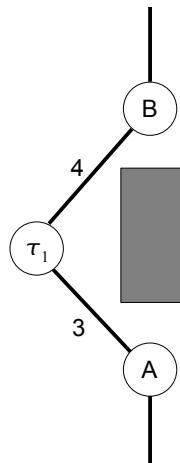
**(a) Initial graph**



**(b)** A “pseudo edge” between nodes  $A$  and  $B$  is added as part of making the graph complete.



**(c)** The heavy edges are included in a Steiner tree.



**(d)** The “pseudo edge” is replaced by the real edges, making the tree illegal.

**Figure 6.5:** In the relay problems, the graph cannot be made complete by adding extra edges consisting of the cheapest paths as this may cause the tree to become invalid.

process is performed, a (partial) terminal Steiner tree is calculated. In the tree, both the new edge  $(A, B)$  and the edge  $(A, \tau_1)$  are included (Figure 6.5c). When the “pseudo edge” is replaced by the real edges,  $\tau_1$  is no longer a leaf and the resulting tree is not a valid (partial) terminal Steiner tree (Figure 6.5d). Therefore, for the problems of interest here, this method to make the graph complete cannot be used.

The algorithms for the terminal Steiner tree problem and the partial terminal Steiner tree problem require that the triangle inequality holds and that the graph is complete. Unless both these requirements hold, no constant approximation ratio for the terminal Steiner tree problem can be given unless  $NP = DTIME(|Z|^{O(\log \log |Z|)})$  [36]. For the PTSTP, two open research questions are whether there exists an approximation algorithm if the triangle inequality does not hold, and if there exists a constant approximation ratio [52].

Neither a complete graph nor a cost function that obeys the triangle inequality can be guaranteed in our problems. Therefore, the algorithms lose their greatest advantages, namely that a solution is guaranteed and that the cost of the solution is guaranteed to be within a certain factor from the cost of the optimum solution. However, it is possible to use algorithms for the *directed Steiner tree* to solve our relay problems.

**Directed Steiner Trees.** A directed Steiner tree is a possible solution to the problems discussed above, because the algorithms do not require complete graphs or that the cost function obeys the triangle inequality. As a directed Steiner tree uses a directed graph, we can make sure that all targets are leaves by choosing to make all target nodes without edges. The directed Steiner tree problem requires a root node and a set of terminals as input. Naturally, in our case the root node corresponds to the base station node  $n_0$  and the set of terminals is the target nodes  $\mathcal{T}$ .

The directed Steiner tree problem provides a flexible way to rewrite the relay problems discussed here, as well as other Steiner tree problems. However, the increased flexibility comes with a price, as the heuristics for directed Steiner trees have high time complexities, high error bounds and long execution times [22, 84, 104]. The algorithm by Charikar et al. [22] in some cases requires more than a minute to solve problems with  $|N| = 100$  and  $|E| = 400$ . To improve this long execution time, Hsieh et al. [50] presented an algorithm that had shorter execution time and yielded trees with similar costs, when tested. However, the algorithm requires  $O(|T||N|^2 + h|T||N|)$  memory space, where  $h \geq 1$  is the height of the tree. We are interested in solving relay problems in discretizations of real environments, where the number of nodes may be in the tens of thousands and the number of edges in the tens of millions. As the memory requirements for such problems would be prohibitive, such algorithms cannot be used for solving the multiple target relay problems.

**Summary.** Finding a relay tree is a kind of Steiner tree problem as the total cost of the tree must be minimized. However, the Steiner minimal tree problem does not distinguish between the target nodes and the other nodes in the tree. This allows a target node to be

connected to the rest of the tree with several edges, which in the context of the relay problems correspond to using a target to relay information. As this is generally not possible, other ways to model the multiple target relay problems have been investigated.

As existing algorithms for the terminal Steiner trees problems require a complete graph and that the triangle inequality applies, they cannot be used to calculate relay trees. The common ways to make the graph complete cannot be used in the relay problems as they can create invalid relay trees, in which a target node is connected to the rest of the tree with several edges.

Algorithms for solving the directed Steiner tree problem do not have the same requirements regarding a complete graph and the triangle inequality as they use a directed graph. However, the algorithms for solving such problems have either long execution times or high memory requirements.

Considering this, we investigated whether heuristics for the SMT could be modified to calculate relay trees. It turns out that we could adapt the cheapest path heuristic for this purpose.

## 6.3 Adapting the Cheapest Path Heuristic

The cheapest path heuristic for calculating Steiner trees has been used in both undirected and directed graphs [91, 95]. The heuristic starts with a single node and repeatedly executes a cheapest path algorithm to find paths to the unconnected terminals. During execution of the heuristic, a Steiner tree is incrementally built, where each iteration adds a path from the nodes in the current tree to an unconnected terminal.

The algorithm in its original form is not suitable for calculating relay trees, as it allows target nodes to be connected to several other nodes. However, the algorithm can be modified to fulfill this requirement. The algorithm needs to be changed so that it is guaranteed that each target node is connected to the rest of the tree using a single edge.

To calculate the cheapest paths efficiently, we use a cheapest path algorithm that can handle sets of start nodes and goal nodes, such as Dijkstra's algorithm. In the context of relay problems, the start nodes are the nodes in the tree  $N(T_G)$  and the set of goal nodes is the set of target nodes to which no path yet has been found,  $\mathcal{T} \setminus N(T_G)$ .

Figure 6.6 shows the pseudocode of our modified cheapest path heuristic for calculating relay trees. The relay tree is initialized with the base station node  $n_0$  and an empty set of edges (line 1). The predecessor of  $n_0$  is set to *nil*.  $|\mathcal{T}|$  iterations are performed, as this is the number of targets, and each iteration connects a single target node (line 3). Line 4 clears the priority queue  $Q$ . The cost  $g(n)$  is initialized for all nodes not in the relay tree (lines 5–6). The set of start nodes is initialized in lines 7–9. For each node, the cost is set to zero and the node is inserted into the priority queue.

Each iteration continues until a path to an unconnected terminal is found (line 16) or the priority queue is empty (line 11). The latter means that the complete graph has been searched and no unconnected terminal has been found. Thus, not all targets can be

connected to the tree, and the algorithm returns with failure (line 11). If  $Q$  is not empty, the least cost node  $n$  is extracted (line 12).

It is checked whether this node is one of the unconnected target nodes (line 13). If so, this means that a cheapest path  $p$  to an unconnected target node has been found. The path is retrieved by the function `Retrieve-Path( $n$ )`, and is added to the relay tree, thereby connecting a previously unconnected target node to the relay tree (lines 14–15). This ends one iteration, and the algorithm continues with the next unconnected target node (line 15).

If the node was not an unconnected target node, it is checked whether  $n$  is a target node (line 17). This check is necessary because  $n$  may be a target node that is already included in  $T_G$ . If  $n$  is not a target node, then each neighbor node  $n'$  is considered with the intention of finding a cheaper path to  $n'$  (lines 18–22). If a cheaper path is found, the cost and the predecessor of node  $n'$  are updated and  $n'$  is inserted into  $Q$ . If  $n'$  already was in  $Q$ , its position is updated due to the decreased cost. The function `Extract-tree` extracts the complete relay tree so that it can be returned to the user (line 23).

The cheapest path heuristic as described here expands a single tree until all targets have been connected. There are other slightly different heuristics based on repeated execution of a cheapest path algorithm that share the same name and error bound [54].

### 6.3.1 Theoretical Properties

Algorithm 4 is based on the cheapest path heuristic which is sound and complete for the general Steiner tree problem in both undirected and directed graphs [91]. The pseudocode in Figure 6.6 is an efficient implementation of the cheapest path heuristic with the change that no outgoing connections are created from target nodes (line 17). This corresponds exactly to the requirement in the multiple target relay problems that each target must be a leaf in the tree.

**Soundness.** The heuristic is sound as each target is connected with a single edge to the rest of the tree. The single connection is assured in line 17 which only considers the outgoing edges of non-target nodes. Furthermore, with the exception of the root node  $n_0$ , each node in the tree only has one predecessor, and thus a tree structure is created. When a cheaper path to a node has been found (line 19), the predecessor node is updated (line 21). Thus, after execution of the algorithm in Figure 6.6, a tree has been created which connects all terminals and each target node is connected to the rest of the tree with a single edge.

**Completeness.** Consider a problem with target nodes  $\tau, \tau' \in \mathcal{T}$ . If there is a path from  $n_0$  to  $\tau$  that does not require going through  $\tau'$ , then it does not matter that the outgoing edges of  $\tau'$  are not considered, a valid path will be found anyway. If this applies for all  $\tau \in \mathcal{T}$ , then a valid relay tree will be found, as a path to each such target is found. If the path to  $\tau$  requires going through  $\tau'$ , then the problem cannot be solved as  $\tau'$  must be

---

```

1   $T_G \leftarrow \langle n_0, \emptyset \rangle$ 
2   $p(n_0) = nil$ 
3  for  $k=1, \dots, |\mathcal{T}|$  do           // Connect one more target
4       $Q \leftarrow \emptyset$                   // Clear priority queue
5      for each  $n \in N \setminus N(T_G)$  do
6           $g(n) \leftarrow \infty$ 
7      for each  $n \in N(T_G)$  do        // Initialize start nodes
8           $g(n) = 0$ 
9          Insert( $Q, n$ )
10     loop
11     if  $Q = \emptyset$  then return failure // Not all target nodes are reachable
12      $n \leftarrow \text{Extract-Min}(Q)$ 
13     if  $n \in \mathcal{T} \setminus N(T_G)$  then   // Found unconnected target node
14          $p \leftarrow \text{Extract-Path}(n)$ 
15          $T_G \leftarrow T_G \cup p$             // Add path to tree
16         exit loop                  // Continue with next target
17     if  $n \notin \mathcal{T}$  then
18         for each  $n' \in n_+$  do        // Treat neighbor nodes
19             if  $g(n') > g(n) + c_{n,n'}$  then
20                  $g(n') \leftarrow g(n) + c_{n,n'}$ 
21                  $p(n') \leftarrow g(n)$ 
22                 Insert( $Q, n'$ )
23 Extract-tree( $\mathcal{T}$ )                // Extract the tree

```

**Figure 6.6:** Algorithm 4 – Modified cheapest path heuristic for calculating a relay tree.

an interior node in the tree. No such tree is a valid relay tree. In that case, the algorithm is not required to return a solution.

**Time Complexity.** The algorithm has a time complexity of  $O(|\mathcal{T}|(|E| + |N| \log |N|))$  as  $|\mathcal{T}|$  executions of a cheapest path algorithm are performed, each one with a time complexity of  $O(|E| + |N| \log |N|)$ .

**Approximation Ratio.** The approximation ratio is  $|\mathcal{T}|$  in both directed and undirected graphs [95]. Despite this approximation ratio, the cheapest path heuristic has been known to be competitive with more advanced methods in directed graphs [50, 95].

### 6.3.2 Extensions

The basic algorithm as shown in Figure 6.6 can easily be extended to find relay trees for a variety of restrictions on how UAVs may be used. These extensions require the addition of a Boolean function  $\text{Acceptable-Connection}(n, n')$  that holds if certain conditions are fulfilled, discussed further below. The function is added to the if-statement in line 19, which will then be:

**if**  $g(n') > g(n) + c_{n,n'}$  **and**  $\text{Acceptable-Connection}(n, n')$  **then.**

Next we will give some examples of the specific restrictions and how these are implemented using the function  $\text{Acceptable-Connection}$ .

**Relay UAVs Have Limited Capacity.** If the relay UAVs can only relay a limited amount of information, then an important question is whether all information streams require the same amount of bandwidth or if it differs. If all information streams require equal amounts of bandwidth, then each relay UAV must be able to handle a certain number of connections. Adding a new information stream requires that all UAVs between the first relay UAV and the base station can handle the additional stream of information. If information streams instead require different amounts of bandwidth, then the same UAVs must be able to handle the additional bandwidth.

To implement this requirement, the function  $\text{Acceptable-Connection}$  could check whether  $n' \in \mathcal{T} \setminus N(T_G)$ , i.e. if  $n'$  is an unconnected target. If this is true, then a surveillance UAV may be placed at  $n$  if all nodes in the path  $q$  between  $n$  and  $n_0$  can handle an additional stream of information. This is controlled by retrieving the path  $q$  and checking each node in  $q$  for available relay capacity. If all nodes have available capacity, then the information can be relayed back to the base station and the target  $n'$  can be surveilled from a surveillance UAV at  $n$ . Otherwise, a relay chain to  $n'$  with the surveillance UAV at some other position must be found.

**Surveillance UAVs May Not Relay.** If surveillance UAVs have limited energy, then restricting the amount of information that they transmit is one way to extend the time that a target can be surveilled. Naturally, each surveillance UAV must transmit information from its own surveillance, but should not relay any other information.

This restriction is different from the restriction that relay UAVs have limited capacity. Here the surveillance UAVs cannot relay any information at all, while relay UAVs still have unlimited capacity.

If a surveillance UAV cannot be used to relay information, then each surveillance UAV must have one connection to a predecessor (relay UAV or base station) and one or more connections to surveillance targets. As  $n$  is in  $T_G$ , it has a predecessor and it is sufficient to check if the currently treated node  $n' \in n_+$  is a target node. If so, then the edge  $(n, n')$  is a surveillance edge and may be added to the tree. If not, the edge corresponds to a communication edge and is not allowed. This requirement can be implemented in the function  $\text{Acceptable-Connection}$  which holds if  $n' \in \mathcal{T}$ .

**Limited Target Distance.** A surveillance UAV can only surveil multiple targets within a limited distance from each other. Therefore, when a surveillance UAV is already surveilling one or more targets and it is checked whether it can surveil an additional target, the distance between the targets currently under surveillance is calculated and compared to the maximum allowed distance. If the distance is below the limit, then the additional target may also be surveilled.

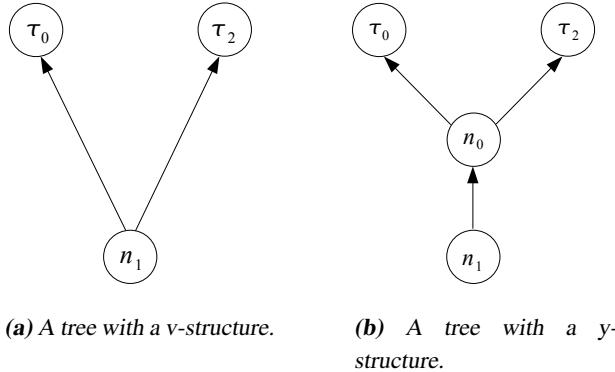
These requirements are checked by the function `Acceptable-Connection`. The neighbor node  $n'$  is an unconnected target if  $n' \in \mathcal{T} \setminus N(T_G)$ . Retrieve the possibly empty set  $E_{surv,n}$  consisting of all edges  $(n, \tau_i) \in E(T_G), \forall \tau_i \in \mathcal{T}$ . Then, all  $\tau_i$  in  $E_{surv,n}$  are target nodes and arbitrary restrictions regarding which targets that can be surveilled can be applied.

## 6.4 Calculating Pareto-optimal Relay Trees For Two Targets

Trees calculated with the modified cheapest path heuristic often have good quality, but there might still be room for improvement. One possible option is to incrementally improve the tree by optimizing subtrees. For example, we can choose subtrees with a root node and two leaves and perform local optimization on such trees. In this section we present a method that calculates Pareto-optimal relay trees for the case of a base station and two targets (leaves). In a later section, this algorithm will be generalized and used to optimize subtrees in larger relay trees.

In optimal Steiner trees, there can be at most  $|Z| - 2$  join nodes, and therefore a relay tree with  $|Z| = 3$  has *at most* one join node. Obviously, it must have one join node, as the paths to  $\tau_1$  and  $\tau_2$  both originate in  $n_0$ . The fact that an optimal Steiner tree with three terminals has exactly one join node allows us to solve the **MTR-ParetoLimited** problem optimally. We can first calculate all Pareto-optimal relay chains to each node, and then connect them so that all Pareto-optimal Steiner trees are created. A Steiner tree with a single join node has either a *v-structure* or a *y-structure*. If  $n_0$  is the only common node in the paths to  $\tau_1$  and  $\tau_2$ , then the tree has a v-structure (Figure 6.7a). Otherwise, the paths have at least one more common node, and the tree has a y-structure (Figure 6.7b).

For two targets, both the **MTR-MinCostMinLength** and the **MTR-MinLengthMinCost** problems can be solved optimally with a time complexity of  $O(|E| \log |E|) \subseteq O(|N|^2 \log |N|^2)$  through the algorithm by Chen [23]. However, for the local optimizations that will be described in Section 6.5, we are more interested in solving the **MTR-ParetoLimited** problem as this allows us to choose other trees than the cheapest or shortest. We solve the **MTR-ParetoLimited** problem through executing Algorithm 2 three times, and then calculating all possible valid Steiner trees. Between each pair of nodes, there are at most  $|N|$  different Pareto-optimal paths. As there are three terminals in the tree, this yields at most  $|N|^3$  Pareto-optimal trees in each node. Determining this for all  $|N|$  nodes yields a time complexity of  $O(|N|^4)$ . Executing Algorithm 2 three times is in  $O(|N|^3)$  and selecting



**Figure 6.7:** Structures of trees with one join node.

the best relay tree is in  $O(|N|)$ . Thus the total time complexity is  $O(|N|^4)$ . However, in practice there are far fewer than  $|N|$  Pareto-optimal paths to each node, and the time complexity is a severe overestimate.

As the Steiner tree must connect all three nodes, only nodes that are reachable from both  $n_0$ ,  $\tau_1$  and  $\tau_2$  are of interest as join nodes. If no such node exists, then no tree can connect the base station and the target nodes and the problem cannot be solved.

We refer to this method of calculating relay trees for a base station node and two target nodes as Algorithm 5. Figure 6.8 displays the pseudocode of the algorithm. For each execution of Algorithm 2, a set of reachability records is created in each reachable node. Thus, after three executions of Algorithm 2 (lines 1–3), three sets of reachability records exist in the nodes that are reachable from both  $n_0$ ,  $\tau_1$  and  $\tau_2$ . Then, an iteration through all reachable nodes, except  $\tau_1$  and  $\tau_2$ , is performed. In each node, the set of relay trees is calculated by repeatedly selecting and combining a reachability record from each set (lines 4–5), described in more detail in Section 6.4.1.

- 1 Execute Algorithm 2 starting in  $n_0$ .
- 2 Execute Algorithm 2 starting in  $\tau_1$ .
- 3 Execute Algorithm 2 starting in  $\tau_2$ .
- 4 **for** each  $n \in N \setminus \mathcal{T}$  **do**
- 5     Construct relay tree(s) with join node  $n$   
        from the reachability records in  $n$ .
- 6 Return the set of Pareto-optimal relay trees.

**Figure 6.8:** Algorithm 5 – Solving the two targets relay problems through multiple executions of Algorithm 2.

Special care must be taken if Algorithm 5 is used in a directed graph. The executions of Algorithm 2 in lines 2–3 will use the nodes’ incoming edges instead of outgoing. This is because in the final tree, the edges will be used to provide paths to the target nodes, not from them.

### 6.4.1 Determining the Set of Pareto-optimal Relay Trees

We will now show how the reachability records are combined to create relay trees.

Assume that after Algorithm 2 has been executed three times, once starting in the base station node and once in each target node, the set of reachability records displayed in Table 6.1 exists for some node  $n \in N \setminus \mathcal{T}$ . For example, there are three paths from  $n_0$  to  $n$ .

$k$	$g_k$	$p_k$
1	50	$n_0$
2	32	$n_1$
4	21	$n_2$

$k$	$g_k$	$p_k$
1	13	$\tau_1$

$k$	$g_k$	$p_k$
1	19	$\tau_2$
2	15	$n_{23}$

**Table 6.1:** Reachability records for executions starting in nodes  $n_0$ ,  $\tau_1$  and  $\tau_2$ , respectively, for some node  $n$ .

By repeatedly selecting and combining Pareto-optimal paths from the different sets, information about  $3 \times 1 \times 2 = 6$  trees is created. All six relay trees have node  $n$  as join node, and we calculate the total costs and lengths by adding the costs and lengths of the individual chains.

By looking at the resulting information about the trees (Table 6.2), it is evident that some trees are inferior. For example, there is no reason to use the tree with total path length 5 and total cost 60 as it is more expensive than the tree with the same length and cost 53. The inferior trees can be removed by evaluating the trees in a manner similar as for the Pareto-optimal relay chains, see Section 3.2.

Path length					
to $n_0$	to $\tau_1$	to $\tau_2$	total	Cost	Pareto-optimal
1	1	1	3	82	Yes
1	1	2	4	78	No
2	1	1	4	64	Yes
2	1	2	5	60	No
4	1	1	5	53	Yes
4	1	2	7	49	Yes

**Table 6.2:** Information about the different relay trees joining in node  $n$ .

The above process finds all Pareto-optimal trees with join node  $n$ . The process is then repeated for all nodes  $N \setminus \mathcal{T}$ , as no target node can be a join node. The complete set of Pareto-optimal trees, possibly with different join nodes, can be found as follows. Start with an empty set of Pareto-optimal trees. Then for each non-target node  $n$ , consider the trees that could be generated with  $n$  as a join node according to the reachability records in that node. For each such tree, check whether there is already a better (in the Pareto-optimal sense) tree in the current set of trees. If not, the tree is added to the set and any trees dominated by the new tree are removed. The set of Pareto-optimal trees together form the solution to the **MTR-ParetoLimited** problem. Table 6.3 shows such a set.

Total path length	Total cost	Join node
3	82	$n$
4	64	$n$
5	51	$n''$
7	49	$n$
8	47	$n''$

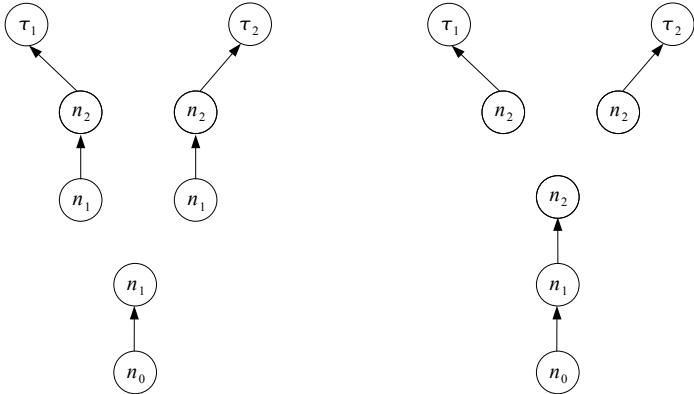
**Table 6.3:** The set of Pareto-optimal relay trees.

#### 6.4.2 Duplicate Edges in the Relay Tree

In rare cases, it may happen that two or more paths in a relay tree coincide in more nodes than just the join node, i.e. there are also common edges. Figure 6.9a shows an example where three paths have been calculated to the join node  $n_1$ . As our calculations simply sum the costs and lengths of all paths involved, the cost of the edge  $(n_1, n_2)$  will be counted twice. This tree will then be considered at least as expensive, and use one hop more, than the tree in Figure 6.9b. The latter tree has  $n_2$  as join node.

It might seem like all trees need to be checked for duplicate edges and subtract the extra cost and hop of any such edges. However, regardless of whether the graph is directed or undirected, nothing needs to be done. When iterating through all non-target nodes to find the join node for the tree, a tree that includes the edge  $(n_1, n_2)$  and its cost only once will be found. In this example, the tree has join node  $n_2$ .

Assume that the tree in Figure 6.9a has been found. In a directed graph, the edge  $(n_1, n_2)$  is guaranteed to exist as it has been used in the paths for  $\tau_1$  and  $\tau_2$ . The cost of the edge and the hop required to use the edge has been included in the tree twice. Naturally, using the edge once cannot be more expensive than using it twice. In addition, using the edge once also gives a tree with one less hop. Therefore, a cheaper tree with one less hop must exist (Figure 6.9b). Such a tree will be found when iterating through all non-target nodes to find the join node of the tree. The same reasoning applies in an undirected graph, where  $c_{n_1, n_2} = c_{n_2, n_1}$  obviously holds. Therefore, it must be the case



**(a)** The paths that are combined to form a relay tree are displayed separately for clarity. The join node is  $n_1$ .

**(b)** The tree with  $n_2$  as join node will be preferred as it will be cheaper.

**Figure 6.9:** Duplicate edges can occur when reachability records are combined to form a tree, but there always exists a cheaper relay tree without the duplicates.

that  $c_{n_1, n_2} \leq 2c_{n_2, n_1}$ , with equality for cost zero. In other words, it will never be more expensive to use the edge once than to use it twice. Similarly, a cheaper and shorter tree must exist and will be found when determining the join node of the tree.

## 6.5 Improving Relay Trees

Algorithm 5 creates Pareto-optimal relay trees with one join node. This can also be used to optimize larger trees with one join node and is used in a heuristic improvement algorithm that can be applied to optimizing relay trees once an initial relay tree has been calculated [77].

The algorithm works by performing a series of local optimizations of the existing tree. In each such optimization, a subtree with one join node is chosen for optimization. Then a set of candidate subtrees is calculated to replace it. Each such candidate subtree also has one join node. From the set of candidates, the best subtree according to some *optimization criterion* (described further below), is chosen to replace the existing subtree. Then, the new subtree is compared to the existing subtree and a replacement is performed only if the new subtree is better than the existing subtree with respect to the optimization criterion. As the number of hops and the cost of the complete tree is the sum of the costs/hops of all subtrees, any improvement of a subtree improves the complete tree. When a replacement has been performed, the new and improved tree can be displayed to the user.

The process of continually optimizing a relay tree can be performed until no better tree is found, or until the available time runs out, if there is a time limit.

**Optimization Criterion.** Subtrees are optimized with respect to a certain optimization criterion. Examples of such criteria are minimizing the number of hops in the tree or finding the least cost tree, regardless of the number of hops, or the cheapest tree with a limit on the number of hops in the tree.

The algorithm permits the use of different optimization criteria, and the criteria can also be changed during the course of optimizing the relay tree. For example, assume that we want the least cost feasible tree. If the original tree is infeasible, then the optimization criterion of minimizing the number of hops in the tree is used until a feasible tree is found. At that time, the optimization criterion is changed to finding the least cost tree with a limit on the number of UAVs, to assure that the tree remains feasible. If the initial tree was feasible, the optimization criterion would be to find a least cost tree with the restriction that the tree must remain feasible.

The above optimization criteria correspond to some extent to the three first multiple target relay problems, and are typically used when trying to calculate approximate solutions to these problems. This algorithm is similar to an anytime algorithm [103], in the sense that it continually improves a solution as time goes on.

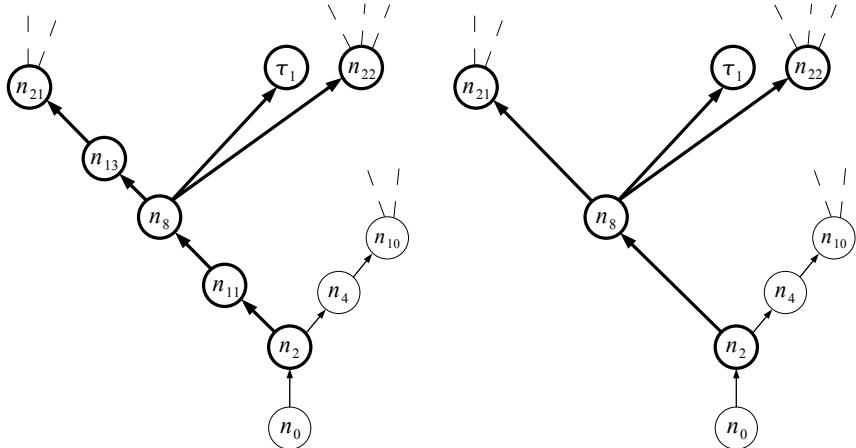
**Notation.** Let a subtree of  $T_G$  be denoted by  $T_s$ , with  $r(T_s)$  denoting the root node and  $L(T_s)$  denoting the set of leaves of the subtree. A subtree that is a candidate for replacing  $T_s$  is denoted by  $T'_s$ . Naturally, each candidate subtree  $T'_s$  has the same root node and the same set of leaves as the subtree it is intended to replace:  $r(T_s) = r(T'_s)$  and  $L(T_s) = L(T'_s)$ . The set of join nodes in each subtree  $T_s$  consists of exactly one join node and we use the variable  $J(T_s)$  interchangeably for the set of join nodes in  $T_s$  and the join node itself.

### 6.5.1 Reduced Trees

Only a few nodes are necessary to characterize the structure of  $T_G$ . The set of such nodes is referred to as *key nodes*. These nodes are the terminals together with the join nodes.

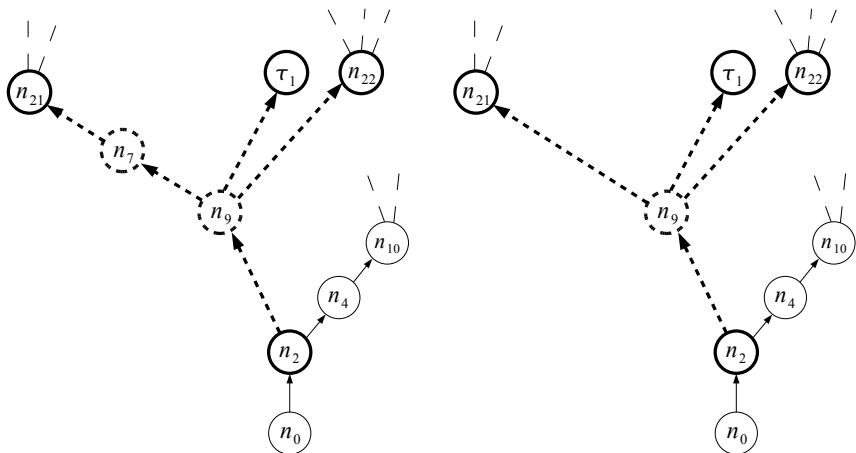
To allow for quickly determining the subtrees for optimization, a *reduced tree* is created. The reduced tree is created by finding all paths in the tree that start in one key node and end in another key node. Each such path is then replaced by a single edge. Thus, the reduced tree consists of only key nodes and maintains the same topology as the relay tree.

Figure 6.10a displays a part of a relay tree, and the corresponding reduced tree is displayed in Figure 6.10b. It is clear that the reduced tree retains the same topology as the original tree. An optimization of the subtree with  $J(T_s) = \{n_8\}$ ,  $r(T_s) = n_2$  and  $L(T_s) = \{n_{21}, \tau_1, n_{22}\}$ , marked by the heavy lines, is performed. A better subtree is found and the previous subtree is replaced. The tree after replacement is displayed in Figure 6.10c, and the corresponding reduced tree is displayed in Figure 6.10d. The new subtree is marked by heavy dashed lines in both figures.



(a) A part of a relay tree.

(b) The reduced tree corresponding to the tree in Figure 6.10a.



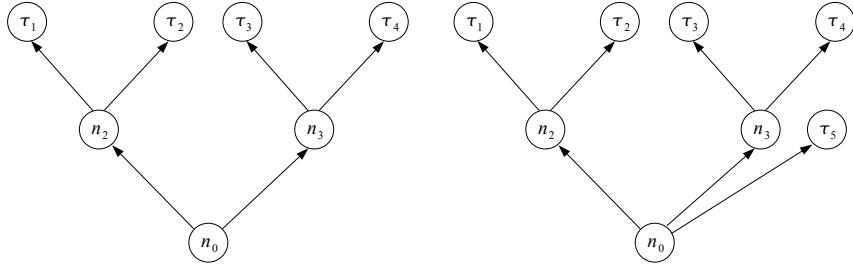
(c) The new subtree, marked by heavy dashed lines, replaces the old subtree.

(d) The reduced tree corresponding to the tree in Figure 6.10c.

**Figure 6.10:** Part of a relay tree and the corresponding reduced tree, before and after optimization. The subtree marked by heavy lines is replaced by the tree marked by heavy dashed lines.

### 6.5.2 Choosing Subtrees for Optimization

In each iteration, the algorithm chooses a join node  $J(T_s)$  whose subtree  $T_s$  will be optimized. The reduced tree is used to determine the root node  $r(T_s)$  and set of leaves  $L(T_s)$  of  $T_s$ . The leaves are all the immediate successors of  $J(T_s)$  in the reduced tree. The reason why all leaves are chosen is discussed further below. In most cases,  $r(T_s)$  is the immediate predecessor of  $J(T_s)$ . Such a subtree has a y-structure (Figure 6.7b), and subtrees with such structure are chosen whenever possible.



(a) In this tree, only two optimizations are needed.

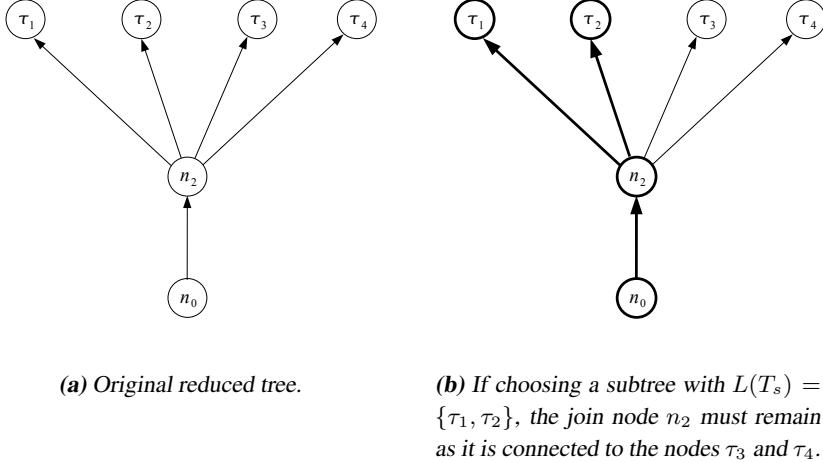
(b) This tree requires three subtree optimizations to allow that all parts of the tree have been subject to optimization.

**Figure 6.11:** Two different cases can occur when  $n_0$  is a join node.

If  $J(T_s) = n_0$ , there are two possibilities (Figure 6.11). If no immediate successor of  $n_0$  in the reduced tree is a target, then no optimization with  $n_0$  as join node needs to be performed. For an example of this, consider the tree in Figure 6.11a. In that tree, there are three join nodes,  $n_0, n_2$  and  $n_3$ . However, only two subtree optimizations need to be performed. These are the trees with  $n_2$  and  $n_3$  as join nodes, respectively. Together these optimizations are sufficient to optimize all parts of the tree. If any of  $n_0$ 's immediate successors in the reduced tree is a target, then an optimization of a subtree with  $n_0$  as join node needs to be performed. Consider Figure 6.11b for an example of this. In addition to the two optimizations with  $n_2$  and  $n_3$  as join nodes, a third optimization with  $J(T_s) = r(T_s) = n_0$  needs to be performed. The leaves in that subtree would be  $\{n_2, n_3, \tau_5\}$  and the subtree has a v-structure (Figure 6.7a). The third optimization is required so that all subtrees have been subject to optimization.

The fact that each join node is chosen in some iteration, together with the structure of the chosen subtrees permits the replacement of all non-terminal nodes.

The reason for including all immediate successors of the join node is exemplified in Figure 6.12. The reduced tree has one join node,  $n_2$  (Figure 6.12a). If a subset of a join node's leaves were chosen for optimization, then several optimizations with the same join node would need to be performed, to allow that all paths in the tree are optimized. Consider Figure 6.12b, where the subtree with  $r(T_s) = n_0, J(T_s) = n_2$  and  $L(T_s) = \{\tau_1, \tau_2\}$  is to be optimized. This subtree is marked by the heavy lines. When performing



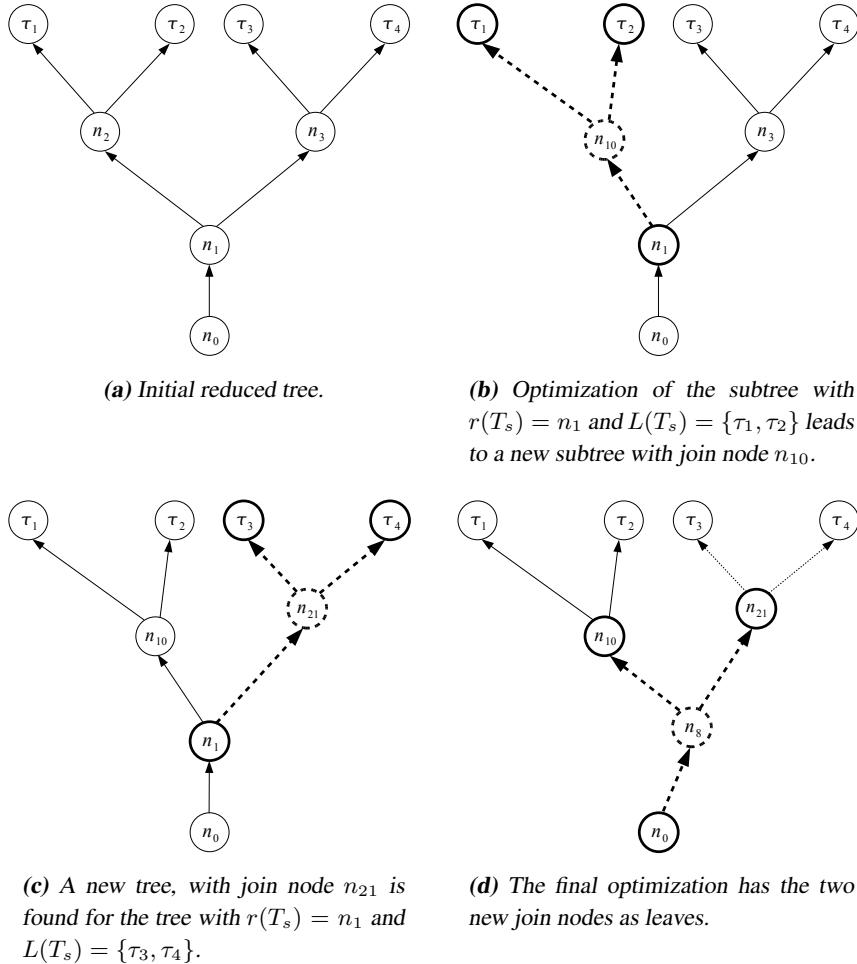
**Figure 6.12:** Including the complete set of a join node's leaves in a subtree optimization potentially allows a better result when the subtree is optimized.

this optimization, the join node  $n_2$  cannot be replaced as it is connected to the leaves  $\tau_3$  and  $\tau_4$ . In that case, only the paths to and from  $n_2$  can be optimized. If instead the complete set of leaves is chosen when optimizing the subtree, then the join node can also be replaced. This potentially permits better trees and is the reason why we include all immediate successors of the join node in subtree optimizations.

It is possible that the complete tree would be chosen for optimization in one iteration. However, for that to happen, the tree would need to have a very specific structure, i.e. all targets would be connected directly to  $n_0$  without any intermediate join nodes.

We will now give an example of how join nodes are chosen and how this affects the tree over several iterations. Consider the reduced tree in Figure 6.13a. For example, let the subtree with  $r(T_s) = n_1, J(T_s) = n_2$  and  $L(T_s) = \{\tau_1, \tau_2\}$  be chosen in the first iteration. During optimization of that subtree, only the root and leaf nodes are fixed: the join node and all other nodes in the subtree can be replaced. Assume that during this optimization, a subtree with the join node  $n_{10}$  replaces the old subtree (Figure 6.13b). In the second iteration, an optimization of the subtree with  $r(T_s) = n_1, J(T_s) = n_3$  and  $L(T_s) = \{\tau_3, \tau_4\}$  is performed. This finds a new subtree with join node  $n_{21}$  (Figure 6.13c). The final optimization of the tree involves the subtree with  $r(T_s) = n_0, J(T_s) = n_1$  and  $L(T_s) = \{n_{10}, n_{21}\}$ . This subtree's leaves are the resulting middle nodes from the previous optimizations. Thus, all paths in the tree have been subject to optimization at least once (Figure 6.13d).

A successful optimization of one subtree can allow for further optimization of other subtrees. The condition is that at least one node in the optimized subtree is a non-leaf node in another subtree. The reason for this is that the root node and leaves are fixed in each optimization, and cannot be replaced. As an example, consider Figure 6.13d where



**Figure 6.13:** By choosing subtrees with y-structures for optimization whenever possible, all paths in the tree are subject to optimization.

a new subtree has replaced the old subtree. Here the join node has changed from  $n_1$  to  $n_8$ . As subtrees are chosen to be partially overlapping, this opens up the possibility of further optimizing subtrees involving the node  $n_8$ . Here there are two such subtrees, one subtree with  $r(T_s) = n_8$ ,  $J(T_G) = n_{10}$  and  $L(T_s) = \{\tau_1, \tau_2\}$  and another subtree with  $r(T_s) = n_8$ ,  $J(T_G) = n_{21}$  and  $L(T_s) = \{\tau_1, \tau_2\}$ . If any of these trees are optimized and the join node is replaced, it opens up the possibility of optimizing the tree with  $r(T_G) = n_0$  again. This shows an example of the fact that the number of subtree optimizations for a tree is not fixed, but depends on how the optimization of different subtrees affects other subtrees.

The order in which subtrees are optimized can be chosen in many different ways, for example starting with the subtrees furthest from the root node and progressing towards the root node.

### 6.5.3 Different Tree Structures

All subtrees that are chosen for optimization have one join node, and such trees can have either a y-structure or a v-structure. The y-structure is the most common and occurs in a majority of cases. In such a subtree, the root node is either  $n_0$  or a join node. The leaves are either join nodes or targets, or a combination thereof. A basic example of such a tree is displayed in Figure 6.7b on page 78. The subtree marked by heavy lines in Figure 6.10d displays another example of such a tree. The root node is  $n_2$ , the join node is  $n_8$  and the leaves are  $\{n_{21}, \tau_1, n_{22}\}$ . The root node, the join node and two of the leaves are join nodes in  $T_G$ . This example shows a generalized y-structure with more than two leaves.

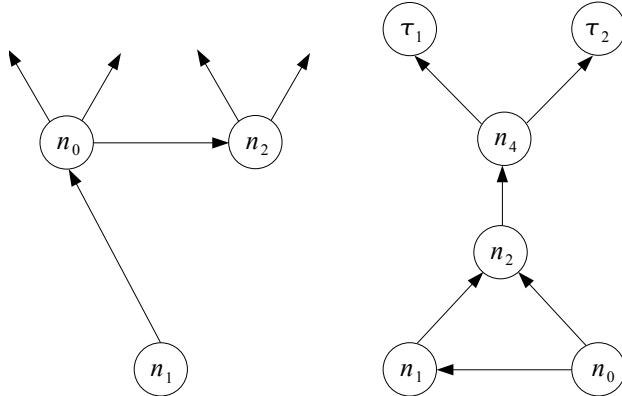
A subtree chosen for optimization can also have a v-structure (Figure 6.7a on page 78). This occurs only when the join node of the subtree coincides with  $n_0$ . The leaves of the subtree can be join nodes or targets or a combination.

It is possible that the subtree  $T_s$  has one structure and the candidate subtree  $T'_s$  that replaces  $T_s$  has a different structure. This poses no problem as the root node and the set of leaves is the same for the two trees. Therefore, the new subtree can be inserted into  $T_G$ . The candidate  $T'_s$  can have either a y-structure, a v-structure, an l-structure or an x-structure.

The l-structure (Figure 6.14a) occurs when a leaf in  $T_s$  becomes the join node in  $T'_s$ . Obviously, this structure can only occur when the new join node is a non-target node, as the join node must be able to connect the other leaves.

On rare occurrences, when treating a subtree with three or more leaves, it can happen that the paths from the root node to the leaves split and join several times (Figure 6.14b). This is no longer a valid tree as at least one node has multiple predecessors, due to several paths joining in the node. Each node with multiple predecessors is a conflict that must be resolved, so that each node only has one predecessor. The possible exception to this is if the subtree's root node is  $n_0$ , which of course has no predecessor.

Solving a conflict involves evaluating each path with respect to the chosen optimization criterion, and removing all but the best path. However, it is not necessary to evaluate



(a) A tree with an l-structure.

(b) An invalid tree with an x-structure.

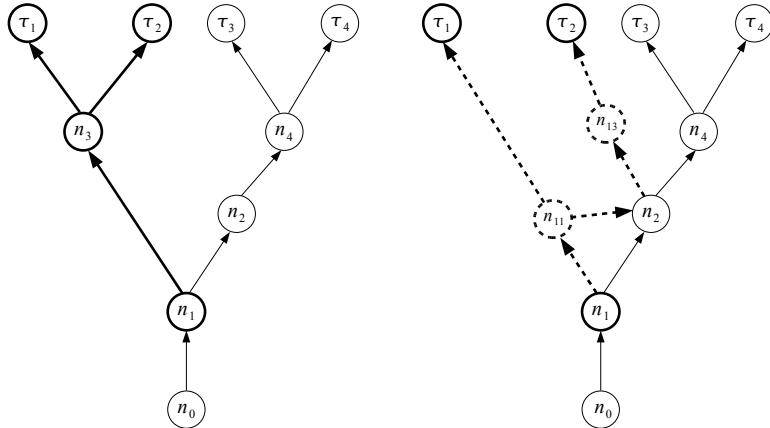
**Figure 6.14:** The candidate subtree  $T'_s$  can have any of the above structures as well as any of the structures in Figure 6.7.

the complete paths: it is sufficient to evaluate the paths from the last common node to the conflicting node. The best path is kept and all other paths are removed. As the best path is kept, the quality of the subtree does not deteriorate. After the worse paths have been removed for all conflicts, the result is a valid subtree.

The number of join nodes can change during the course of optimization. If  $T_s$  has a y-structure and  $T'_s$  has either a v-structure or an l-structure, then a join node has been removed and the number of join nodes decreases by one. If  $T_s$  has a v-structure and  $T'_s$  has either a y-structure or an l-structure, then a join node has been added and the number of join nodes increases by one. A change in the number of join nodes does not affect the number of subtree optimizations that is performed in the basic algorithm. However, if the algorithm is changed to continue to perform optimizations of subtrees that have been optimized previously, the changing number of join nodes will have an effect on the number of subtree optimizations that are performed. See Section 6.5.5 for a discussion about this change.

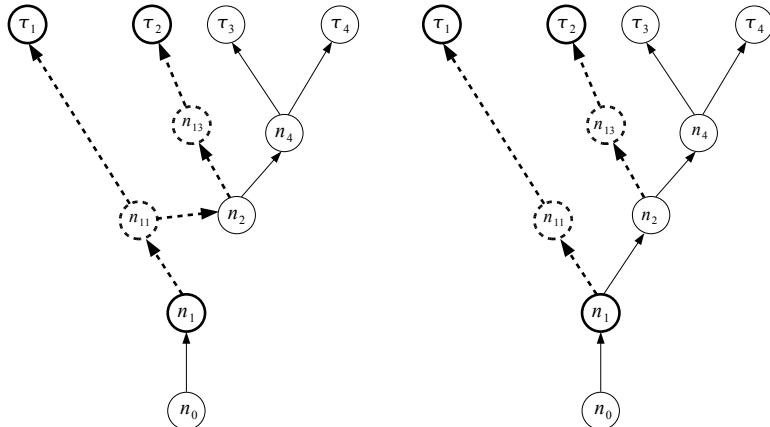
#### 6.5.4 Collisions Between Trees

When optimizations are performed, it is possible that the new subtree “collides” with other parts of the tree. That is, one or more nodes in the new subtree are also part of the tree that is not currently being optimized. Figure 6.15 shows an example of a tree collision and the different situations that can occur. The subtree marked by heavy lines in Figure 6.15a is to be replaced by the subtree marked by heavy dashed lines in Figure 6.15b. However, the new subtree has node  $n_2$  in common with the part of the tree that is not currently being optimized.



**(a)** The subtree marked by heavy lines is optimized.

**(b)** Node  $n_2$  is part of both the old unoptimized tree and the new subtree.



**(c)** If the new path in to  $n_2$  is better, parts of the old tree are removed.

**(d)** If the old path to  $n_2$  is better, parts of the new subtree are removed.

**Figure 6.15:** Collision between the new subtree and the rest of the relay tree.

Just like the conflicts within the subtree, the collision causes the relay tree to violate the definition of a tree and must be corrected. There are several different ways to solve this: a simple way is to mark nodes in  $N(T_G) \setminus N(T_s)$  as not being allowed in  $N(T'_s)$ . However, this might lead to lower quality trees as some nodes are removed from use.

Another alternative that potentially gives better trees, but requires an additional step, is to allow all nodes, and check whether the new subtree  $T'_s$  contains one or more nodes that are part of the tree that is not being optimized. For each such node, we must determine which path to the node to use.

Consider Figure 6.15b, with the colliding node  $n_2$ . Here we must determine whether it is better to use the old or the new path to  $n_2$ . To do this, a traversal is performed from  $n_2$  towards  $n_0$ , but stopping at the first node in the tree that must remain even if the path to  $n_2$  is removed. Such an unchanging node can be either a join node or  $n_0$  if no join node is encountered. The traversal is performed in both the original tree and in  $T'_s$ . In  $T'_s$ , the path is  $n_{11} \rightarrow n_2$  and in the original tree, it is  $n_1 \rightarrow n_2$ . It does not matter if the first unchanging nodes are different. It is never necessary to remove more than the path to such a node, as the join nodes have several successors and must thus remain in the tree even if one successor is removed, and the base station node  $n_0$  is never removed. If the path in  $T'_s$  is better according to the optimization criterion, then the path in the old tree is removed, and the path in  $T'_s$  remains (Figure 6.15c) and vice versa (Figure 6.15d).

If there are several collisions, they can be handled sequentially, and the updated tree is used when determining the paths to the join nodes, thus the currently best tree is used at all times.

### 6.5.5 Algorithm Details

To optimize subtrees, Algorithm 5 is generalized. The pseudocode for the new algorithm, Algorithm 6, is displayed in Figure 6.16. The preference relation  $T'_s \prec T_s$  holds if  $T'_s$  is better than  $T_s$  with respect to the optimization criterion.

As described in Section 6.5.2, choosing a subtree for optimization is a matter of selecting a join node and extracting the corresponding subtree. We keep track of the set of join nodes whose subtree has not been optimized. This set of untreated join nodes of  $T_G$  is denoted by  $J_u(T_G)$ . Initially  $J_u(T_G) = J(T_G)$ .

Systematically optimizing the complete tree requires optimizing all subtrees. This corresponds to  $|J_u(T_G)|$  subtree optimizations (line 1). Each subtree optimization begins with choosing a join node  $J(T_s)$  (line 2). Once a join node has been chosen, it is removed from  $J_u(T_G)$  as it may not be chosen again (line 3).

The subtree  $T_s$  in which  $J(T_s)$  is the join node is extracted from the reduced tree (lines 4). Then Algorithm 2 is executed once starting in the subtree's root node and once in each of the subtree's leaf nodes (lines 5–7). Each execution creates a set of reachability records in each reachable node.

Similar to Algorithm 5, only nodes reachable from both the root node and the subtree's leaves are of interest when the join node is determined. Target nodes are excluded from

---

```

1 for  $k = 1, \dots, |J_u(T_G)|$  do
2    $J(T_s) \leftarrow \text{Choose-join-node}(J_u(T_G))$ 
3    $J_u(T_G) \leftarrow J_u(T_G) \setminus J(T_s)$ 
4    $T_s \leftarrow \text{Extract-subtree}(J(T_s))$ 
5   Execute Algorithm 2 starting in  $r(T_s)$ 
6   for each  $n \in L(T_s)$  do
7     Execute Algorithm 2 starting in  $n$ 
8   for each  $n \in N \setminus \mathcal{T}$  do
9     Calculate-cost-for-subtrees( $n$ )
10   $T'_s \leftarrow \text{Choose-best-subtree}$ 
11  if  $T'_s \prec T_s$  then
12     $T_G \leftarrow T_G \setminus T_s$            // Remove old subtree
13     $T_G \leftarrow T_G \cup T'_s$           // Insert new subtree
14  Yield  $T_G$                          // Yield improved tree

```

**Figure 6.16:** Algorithm 6 – Algorithm for optimizing existing relay trees.

the calculation of subtrees, as the target nodes must be connected to the tree with a single edge, and join nodes must have several connections to the tree (lines 8–9). The next step is to choose the best candidate subtree, according to the optimization criterion (line 10). If the chosen candidate subtree is better than the existing subtree, the existing subtree is removed and replaced by the candidate subtree (lines 11–13). This also updates the reduced tree. As the relay tree has improved, it is yielded to the ground operator (line 14).

The algorithm described by the pseudocode in Figure 6.16 optimizes all subtrees of  $T_G$  once, but does not return to previously optimized subtrees to optimize them again. The possible benefit of doing so is discussed in Section 6.5.2. Allowing the re-optimization of subtrees requires two changes in the algorithm. The first change is to replace the **for**-loop in line 1 with a **while**-loop that is performed as long as  $J_u(T_G) \neq \emptyset$ . The second change is performed if the **if**-statement in line 11 holds. In that case, all non-target key nodes of  $T'_s$  are added to  $J_u(T_G)$  if they are not already in  $J_u(T_G)$ . The reason for this is that e.g. a non-target leaf in  $T'_s$  will be a join node in some other subtree that can potentially be improved. These changes will cause the optimization to continue as long as there are join nodes of subtrees that have the possibility of improvement.

If any of the extensions in Section 6.3.2 were used when calculating the initial tree, then the restrictions specified by the extension must be met when optimizing the tree. Such restrictions can be handled by modifying the function **Choose-best-subtree** to choose a subtree that satisfies all restrictions.

Algorithm 6 improves the relay tree, and the amount of improvement depends on the initial tree. In some cases, the modified cheapest path heuristic can give trees where all subtrees treated by Algorithm 6 are already optimal and when this happens, no improvement can be made. Experimental results are available in Section 7.5.

### 6.5.6 Time Complexity

We will now go through the pseudocode of Algorithm 6 to determine its time complexity. The number of iterations in the for-loop is  $|J_u(T_G)| = |J(T_G)|$ , as this is the number of subtrees that requires optimization to optimize the complete tree. Choosing the join node  $J(T_s)$  requires  $O(|J_u(T_G)|) \subseteq O(|N|)$  time (line 2). Removing  $J(T_s)$  from  $J_u(T_G)$  is  $O(1)$  (line 3). Extracting the subtree  $T_s$  is in  $O(|N|)$  (line 4). Lines 5–7 are discussed below. Calculating the cost of the subtrees requires combining all reachability records in each node. Let the maximum number of terminals in a subtree be denoted by  $b$ , where  $b \leq Z$ . There are at most  $|N|$  Pareto-optimal paths between each pair of nodes, and thus there can be at most  $|N|^b$  reachability records that need to be combined. Thus the time complexity of calculating all subtrees for  $N$  nodes is in  $O(|N|^{b+1})$  (line 8–9). Choosing the best subtree is  $O(|N|)$ . Removing  $T_s$  and inserting  $T'_s$  are both  $O(|N|)$  (lines 12–13).

The way we choose subtrees will cause Algorithm 2 to be executed twice for each join node, once when it is join node and once as a leaf in another subtree. The only exception to this is if  $n_0$  is a join node, in which case one less execution of Algorithm 2 is necessary as  $n_0$  is never a leaf in any subtree. In addition, Algorithm 2 must be executed once for each terminal. In total, this requires at most  $|Z| + 2|J(T_G)|$  executions of Algorithm 2. This is the total number of executions of Algorithm 2 that will be performed in lines 5–7 for  $|J(T_G)|$  iterations. As  $|J(T_G)| \leq |Z| - 2$ , this yields at most  $3|Z| - 4$  executions of Algorithm 2. Thus the time complexity for executing Algorithm 2 is  $O((3|Z| - 4)|N|^3) \subseteq O(|Z||N|^3)$ .

This yields a total time complexity of  $O(|Z||N|^3 + |N|^{b+1})$ . While this time complexity may be perceived as high, the maximum number of terminals in a subtree is commonly very low and the number of Pareto-optimal paths to each node is often far less than  $N$ .

## 6.6 Summary

The multiple target relay problems are variants of the NP-hard Steiner tree problem. There are both continuous and discrete versions of the Steiner tree problems. Algorithms for solving the multiple target relay problems must be able to handle large three-dimensional environments with obstacles. As it is not certain that the triangle-inequality applies, no existing algorithms for the continuous Steiner problem can be used to solve our problems. Therefore, we discretize the environment and formulate our problem as a discrete Steiner tree problem.

Most discrete Steiner tree problems allow a target node to be connected to several other nodes. In the context of the relay problems, this means that a target is used to relay information. This is unrealistic and means that most Steiner tree problems are not sufficiently expressive for our needs. Upon further investigation, we found that the Steiner tree problem in a directed graph and the (partial) terminal Steiner tree problem in an undirected graph are sufficiently powerful to model the relay problems.

The algorithms for calculating a Steiner tree in a directed graph have long execution

time as well as high memory consumption, making them difficult to use in our setting. The few published algorithms for the partial terminal Steiner tree problem make strong assumptions about the graph, such as that the graph is complete and that the triangle inequality holds for the cost function. Neither of these requirements is guaranteed to be fulfilled in the multiple target relay positioning problems. Therefore, the algorithms must be extensively modified to guarantee solutions to any of the multiple target relay problems, if this is at all possible. Furthermore, due to the lack of complete graphs and cost functions obeying the triangle inequality, it is unlikely that any bound on the error can be given.

Instead, modifications of existing heuristic algorithms are investigated, and it is possible to modify the cheapest path heuristic to fit our needs. Using the modified cheapest path heuristic, we are able to calculate approximate solutions to the **MTR-MinCost-MinLength** and the **MTR-MinLengthMinCost** problems.

For problems involving a base station and two targets, the label-correcting algorithm from Section 5.2 is extended and used to solve all multiple target relay problems defined here. Furthermore, this algorithm is further generalized and used to improve existing relay trees, calculated by e.g. the cheapest path heuristic. The algorithm incrementally performs local optimizations of subtrees. This process can continue as long as the relay tree can be improved or for a predetermined time. The use of different optimization criteria allows that the relay tree is optimized with respect to different objectives, such as finding the tree requiring the fewest UAVs or the cheapest tree that can be realized given a limit on the number of available UAVs.

There are algorithms for finding hop-constrained Steiner trees [96, 30] as well as the bi-criteria Steiner tree problems [65, 98]. Such algorithms could possibly be used to calculate approximate solutions to hop-constrained and bi-criteria relay trees. How much modification such algorithms require to be useful in practical relay problems is a matter for future research.



---

# Implementation and Experimental Results

The algorithms for solving the relay problems have been implemented in an existing infrastructure. The software architecture and the user interfaces are briefly described in this chapter together with the experimental results.

## 7.1 Software Architecture

For several years, the Artificial Intelligence and Integrated Computer Systems division [1] at the Department of Computer and Information Sciences at Linköping University has developed an experimental infrastructure for collaborative unmanned aerial systems [33, 34]. This infrastructure has several inter-linked components, such as a number of UAV systems, including two Yamaha RMAX UAVs (Figure 7.1a), several micro UAVs (Figures 7.1b–7.1d) and several software systems for experimentation. One of these systems is used for research in the area of delegation-based cooperation among UAVs.

The algorithms in this thesis, including the necessary infrastructure, has been designed and implemented as an extension to this system. This allows us to empirically test the algorithms and allows other parts of the system to use the algorithms.

The software uses a modular and distributed architecture where new services can be added through servers. Servers can be complex pieces of software such as path planners or less complex, such as an interface for manually controlling the camera onboard a UAV. For communication between the components of the system, the Common Object Request Broker Architecture (CORBA) [28] middleware is used. CORBA permits the system to be distributed over several computers. This allows both computers on the ground and on board UAVs to be used simultaneously.

The algorithms have been implemented as a *relay server* that encapsulates all algorithms and interfaces necessary to specify and solve relay problems. Figure 7.2 shows a

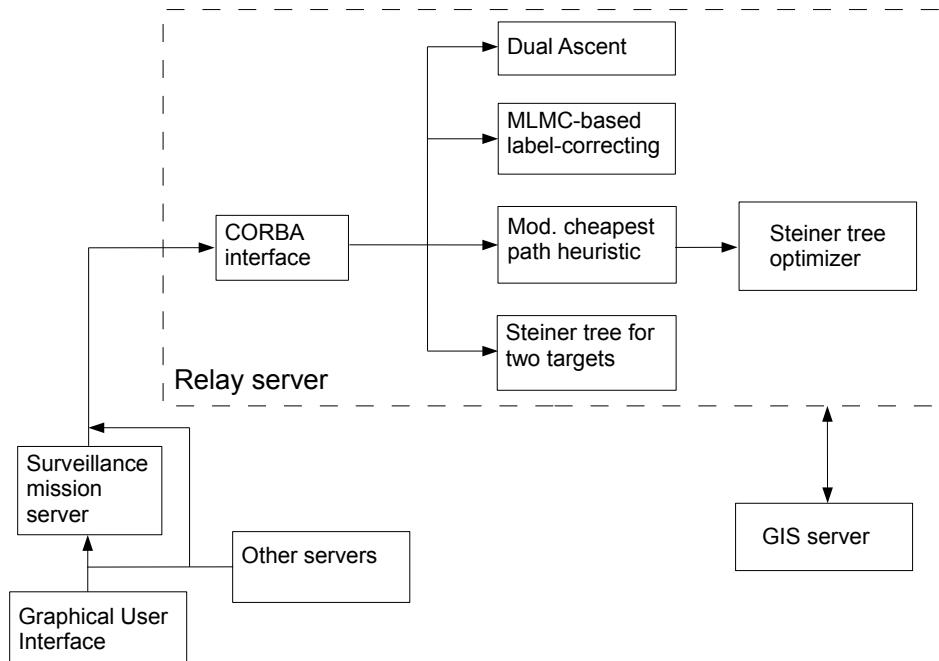


**Figure 7.1:** Some of the UAVs that are part of the experimental infrastructure for UAV research. All except the Yamaha RMAX are designed and built in-house.

diagram of the software architecture of the relay server.

Calls from outside the relay server go through the CORBA interface. Using this interface, problem parameters such as the positions of the base station targets and the number of UAVs are set, and the values are stored internally. From the interface, the appropriate function for calculating relay chains and trees can be called, and the result is returned through the CORBA interface. Missions can be specified from the Graphical User Interface (GUI), which then calls the surveillance mission server that delegates tasks to specific UAVs. The delegation is very briefly described in the use case below. The CORBA interface can also be called from other servers in the system.

The GUI allows users to set problem-specific parameters such as the communication range, the number of allowed UAVs and more. It also has a two-dimensional visualizer that displays the environment from a top-down perspective. Figure 7.3 shows the interface after solving a single target relay problem. In the visualizer, the base station is marked by a yellow square near the bottom left corner, and the target is marked by a red circle. Each relay chain is visualized in a distinct color, and the position of each UAV is marked by a circle. This is the common GUI when relay problems are specified and solved, but there is also an optional three-dimensional visualizer available (Figure 7.4 on page 100).



**Figure 7.2:** The major components in the relay server (dashed rectangle) and the main connections to other parts of the UAV infrastructure.

Other information about e.g. the environment that is required to perform calculations of relay chains or trees is retrieved from other servers in the system, most often the GIS server, which stores information about the environment. With the required information available, the graph is created internally in the relay server.

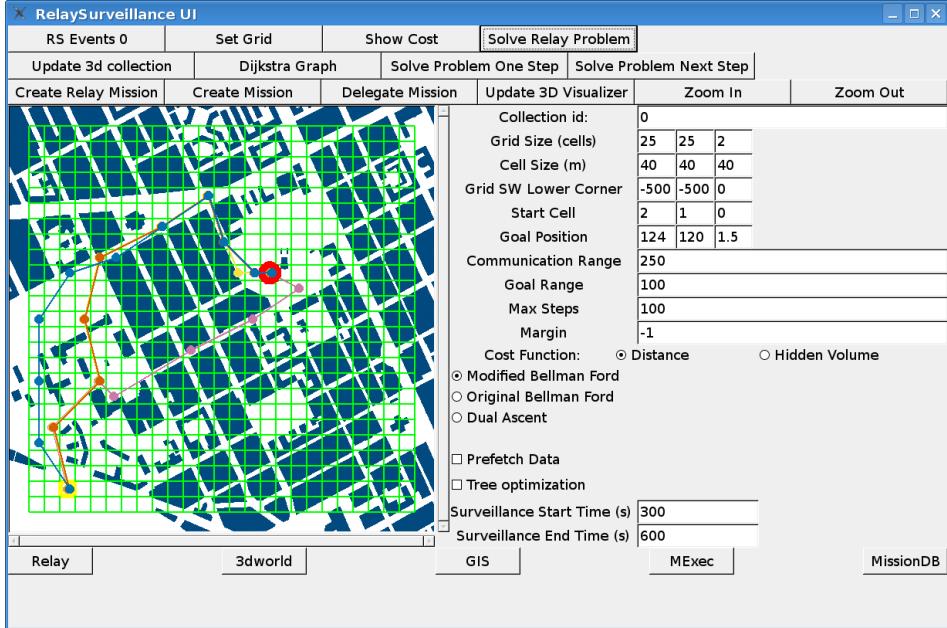
As described earlier, the algorithms and the interface presented here are part of a infrastructure, which is typically operated by one or more human ground operators. The system allows the ground operator to instantiate a variety of missions, such as traffic monitoring, photogrammetry and surveillance. Here we describe a use case where the users set up a surveillance mission.

When a surveillance mission is in the planning stage, the ground operator sets problem parameters such as the position of the base station and one or more targets, reachability and cost functions as well as discretization. Depending on whether there is a single target or if there are several targets, different options for choosing algorithms are presented.

The ground operator also decides the objective of the calculations. For both single and multiple target surveillance, he can choose whether to minimize the number of UAVs required for surveillance to find the minimum cost relay chain/tree or find a chain/tree of minimal cost. For single target problems, the option to find several different relay chains, i.e. all Pareto-optimal chains, is also available. For multiple target problems, he can also choose whether to continue to optimize the relay tree after the initial tree has been

## 7. Implementation and Experimental Results

---



**Figure 7.3:** Screen shot from the graphical user interface after solving a single target relay problem. The map in the left part of in the interface displays four different relay chains between the base station in the lower left hand corner and the target in the middle.

calculated.

In the current prototype system, a broadcast is issued with the intention of finding a set of UAVs that are available for performing a relay mission. The UAVs in the area respond to the request for participation. Naturally, if some UAVs already have a mission or are unable to participate in a relay mission for other reasons, such as lacking appropriate sensors or communication equipment, they respond that they are unable to participate. UAVs able to participate preliminarily accept the request for participation.

At this point all necessary problem parameters have been set and the appropriate algorithm is used to calculate the solution. Once the solution has been calculated, it is displayed to the ground operator. If the solution cannot be realized using the available number of UAVs, more UAVs must be found or some parameter must be changed to find a solution requiring fewer UAVs.

Otherwise, a task specification tree [35] is created corresponding to the selected relay chain or tree. The surveillance mission service then calls the delegation system [58], which attempts to delegate this tree to an appropriate set of UAVs supporting the required roles. If all constraints associated with the mission can be satisfied, including timing constraints, the UAVs accept their tasks. When a sufficient number of UAVs can take part in the mission, the ground operator accepts the final delegation of roles and restrictions.

Once the delegation process is finished and each UAV has been assigned a role in the mission, each UAV uses its own path planner to find a flyable trajectory to its designated position. Due to the decentralized nature of the path planning step, it can be done in parallel. Now everything is set up for executing the mission and the surveillance can begin as soon as all UAVs have arrived at their positions.

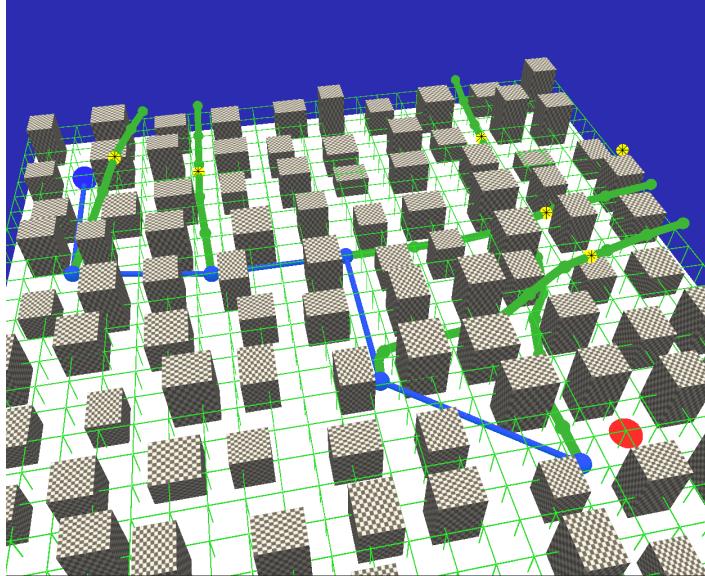
## 7.2 Problem Setup for Empirical Testing

For testing, we used directed graphs constructed from grids and several different environments. All environments have the same size,  $1000 \times 1000 \times 80$  meters. The grid cell size was varied between 10 and 40 meters. The resolution in the horizontal direction has a greater impact on the probability of finding good paths, and this is reflected in the choice of grid cell sizes. All testing was performed on a standard PC with a 2.4 GHz Core 2 Duo CPU and 2 GB RAM.

For all testing, the testing used reachability functions based on free line-of-sight and a range of 100 meters for both communication and surveillance. Two different cost functions were used: one based on distance and the other on obstructed volume. The distance cost function has a constant cost of 300 up to 60 meters, corresponding to the assumption that communication within this distance will have comparatively constant, but not perfect, quality. After 60 meters, the cost increases with the square of the distance. This tests the case where a wide variety of Pareto-optimal relay chains is generated for any target. The second cost function is based on obstructed volume (see Section 3.3.2). This generally results in considerably fewer Pareto-optimal chains, testing the performance of the algorithms for this end of the spectrum as well.

**Randomized Urban.** The first environment used in testing is an urban environment with semi-random placement of 100 tall buildings, as shown in Figure 7.4. To reduce clutter, the figure displays a sparse discretization and only the “lowest” level of grid cells. In this figure, a specific relay chain is visualized. The base station is in the upper left corner and is connected by dark lines representing communication links to dark spheres denoting intended positions for relay and surveillance UAVs. The target is in the lower right corner and is visible from the last UAV in the chain. Subtasks have been delegated to several UAVs, marked by a dark stars on lighter spheres. In the figure, the simulated UAVs have used their path planners to generate individual flight paths (indicated by lighter lines) and are in the process of flying to their intended positions.

**Urban With Boulevards.** The second simulated environment randomizes the buildings roughly in four blocks, and leaves space for two broad boulevards that cross in the middle. The buildings in this environment are more diverse in size and may intersect each other, creating an urban environment where buildings share common walls.



**Figure 7.4:** Randomized urban environment.

**Randomized Dense Urban.** The third environment also places buildings semi-randomly. However, it generates 625 smaller buildings, creating an environment where each node has fewer neighbors.

**Stockholm.** The Stockholm environment is an area of central Stockholm extracted from OpenStreetMap [78]. In this environment, the buildings form city blocks of irregular shapes and different sizes. The many houses make this a very dense area, and therefore, considerably fewer nodes are created, compared to the other environments. As the map does not contain any building heights, all buildings are considered infinitely high when testing.

Figure 7.3 shows a part of the Stockholm environment.

**Revinge.** The Revinge environment is a 3D model of an emergency services training facility in the south of Sweden (Figure 7.5). It predominantly consists of open areas with some buildings scattered throughout the environment.

**Properties of the Test Environments.** Some information about the graphs is displayed in Table 7.1. The average number of nodes and edges are shown for each world and discretization. Similarly, the minimum, average and maximum node degrees are also displayed. As expected, the average and maximum node degree increases with decreasing grid cell size, as there are more nodes for which the reachability functions hold. The



**Figure 7.5:** The Revinge emergency services training ground.

numbers are for testing of the single target relay problems. The numbers for the multiple target problems are slightly higher as there are several target nodes and edges connecting these nodes to the rest of the graph.

Recall that that  $k_{max}^*$  is the maximum number of hops required in an MLMC-tree, i.e. no minimum cost path will require more than  $k_{max}^*$  hops. The value of  $k_{max}^*$  is commonly larger for the cost function based on distance than the cost function based on obstructed volume. For the cost function based on distance, the cost increases faster than linearly. Therefore it is possible to decrease the cost of a path by exchanging a long edge for several shorter edges. This allows a larger set of Pareto-optimal paths and increases the value of  $k_{max}^*$ .

The values of  $k_{max}^*$  in the dense urban and the Stockholm environments are considerably higher than for the other environments. This is especially evident for the coarser discretizations. Due to the low node connectivity, relay chains are in some cases forced to take long detours to reach their targets. The values of  $k_{max}^*$  in Table 7.1 are the average of  $k_{max}^*$  for the different environments and discretizations and are from the testing calculations of Pareto-optimal relay chains.

All algorithms for single target relay problems tested here calculate paths to all reachable nodes from the base station. Therefore, Table 7.1 includes the number of reachable nodes and edges only. There can be small graphs that are not reachable from the main graph due to obstacles.

## 7. Implementation and Experimental Results

---

World	Cell size (meters)	$ N $	$ E $	Node degree Min/avg/max	Avg. $k_{max}^*$ Vol.	Dist.
Randomized urban	40×40×40	1,022	21,815	5/21/35	14.1	23.6
	33×33×40	1,791	53,767	6/30/47	14.9	24.1
	25×25×25	3,796	257,711	11/67/113	13.9	20.1
	20×20×20	7,846	1,102,063	21/140/221	13.5	19.0
	15×15×20	13,852	3,741,793	30/270/434	12.4	17.7
	12×12×20	22,008	9,286,019	40/421/680	12.5	18.1
	10×10×20	31,807	18,753,049	61/589/948	12.3	17.8
Urban with boulevards	40×40×40	989	27,455	1/27/42	12.7	22.1
	33×33×40	1,781	62,547	1/35/50	12.6	23.2
	25×25×25	3,775	339,148	2/89/135	12.0	18.4
	20×20×20	7,941	1,456,997	7/183/278	13.3	17.6
	15×15×20	13,813	4,850,503	11/351/531	12.8	16.9
	12×12×20	21,885	11,970,877	11/546/831	14.4	18.1
	10×10×20	31,965	24,452,326	15/764/1,154	12.7	17.4
Randomized dense urban	40×40×40	660	3,176	1/4/15	27.0	41.2
	33×33×40	1,493	23,888	1/16/42	19.3	24.7
	25×25×25	2,975	62,134	2/20/57	16.8	22.7
	20×20×20	4,107	167,347	3/40/99	15.1	20.2
	15×15×20	10,852	876,771	7/80/220	13.3	18.1
	12×12×20	17,325	2,206,373	11/127/373	13.2	18.7
	10×10×20	25,235	4,545,123	15/180/510	13.0	18.0
Stockholm	40×40×40	467	7,046	1/15/37	21.2	34.0
	33×33×40	976	16,538	3/16/47	22.7	25.4
	25×25×25	2,263	104,242	1/46/130	19.5	25.8
	20×20×20	4,860	455,256	1/92/261	17.4	22.7
	15×15×20	8,707	1,496,489	1/171/498	16.7	20.7
	12×12×20	13,647	3,691,925	1/270/775	15.3	20.2
	10×10×20	20,003	7,581,444	1/379/1,085	15.1	20.0
Revinge	40×40×40	1,170	38,903	1/33/42	13.9	23.0
	33×33×40	2,049	86,351	1/42/51	13.9	24.0
	25×25×25	4,480	479,167	2/106/137	13.7	20.0
	20×20×20	9,341	2,047,203	7/219/280	13.4	18.5
	15×15×20	16,273	6,784,873	11/416/532	12.3	17.3
	12×12×20	25,663	16,762,197	1/653/832	12.4	18.0
	10×10×20	37,307	33,976,220	1/910/1,157	12.3	17.5

**Table 7.1:** Information about worlds and discretizations in empirical testing of algorithms for single target relay problems.

## 7.3 Pareto-Optimal Relay Chains

To solve **STR-ParetoLimited** problems, the new MLMC-tree-based label-correcting algorithm (Algorithm 2) was used, and for comparison, we used the truncated version of Bellman-Ford (Algorithm 1). Testing was performed in the environments described above, and in each environment, we used the seven discretizations described in Table 7.1. For each discretization, we randomly generated 100 combinations of start and goal positions, and to ensure that all Pareto-optimal chains were found,  $M = \infty$  was used.

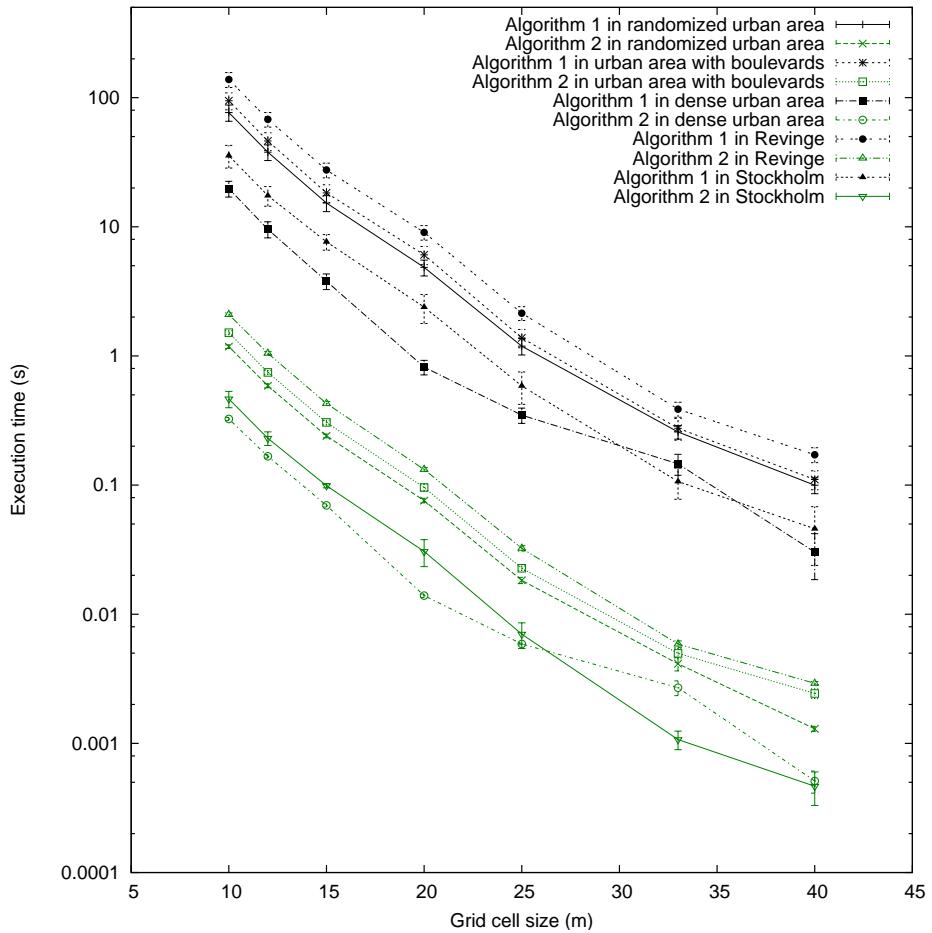
Figures 7.6 and 7.7 display the average execution times for generating all Pareto-optimal chains, using cost function based on obstructed volume and distance, respectively. Times for Algorithm 1 are displayed in black and times for Algorithm 2 are in green. The standard deviation in execution time is indicated using error bars.

From Figures 7.6 – 7.7 it is evident that Algorithm 2 is able to decrease the execution time more when the cost function based on obstructed volume is used. This cost function commonly generates few different Pareto-optimal chains. After this set of chains is found, no calculations are necessary and thus this cost function offers great potential for improving the execution time. The speedups are in the order of 45–75 times except in the Stockholm environment where the speedups are 75–100 times.

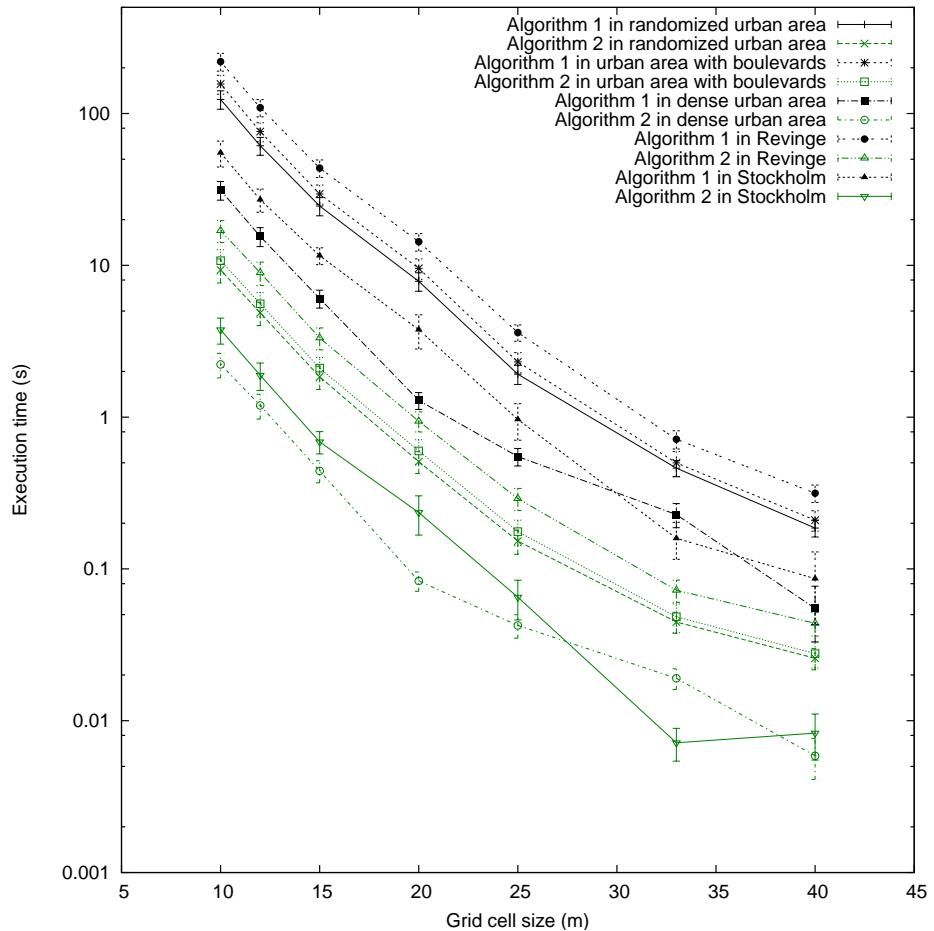
The cost function based on distance (Figure 7.7) allows for a greater set of Pareto-optimal chains, often 4–10 chains. Many more calculations need to be performed and the calculations performed by Algorithm 2 cannot be terminated as early. Even in these more difficult circumstances, Algorithm 2 achieves a speedup in the order of 10–22 in the Stockholm environment. For the other environments the speedup is 7–10 for the two coarsest discretizations (33 and 40 m grid cell size) and 12–16 for the other.

When run in dense urban environments, the execution times of the algorithms vary more depending on the discretization, for both cost functions. For Algorithm 2 in the Stockholm environment, the execution time for test cases in the discretization with 33 m grid cells is lower than the execution time for the discretization with 40 m grid cells. This is due to fewer solutions being created for the finer discretization. On average, the number of solutions is two for the discretization with 33 m grid cells and six for the discretization with 40 m grid cells. The smaller number of solutions can also be seen for Algorithm 1, where the increase in execution time from 40 m to 33 m grid cells is quite small. For both algorithms, the dense Stockholm environment also causes a greater standard deviation of the execution times, as compared to the other environments.

The conclusion of testing of algorithms for the **STR-ParetoLimited** problem is that our new algorithm (Algorithm 2) offers far better performance in the tested environments. For the cost function based on distance, the speedup is less than for the cost function based on obstructed volume, as the number of Pareto-optimal chains typically is larger. However, Algorithm 2 consistently outperforms Algorithm 1, in many cases by an order of magnitude.



**Figure 7.6:** Test results for testing of algorithms for the **STR-ParetoLimited** problem, with cost function based on obstructed volume.



**Figure 7.7:** Test results for testing of algorithms for the **STR-ParetoLimited** problem, with cost function based on distance.

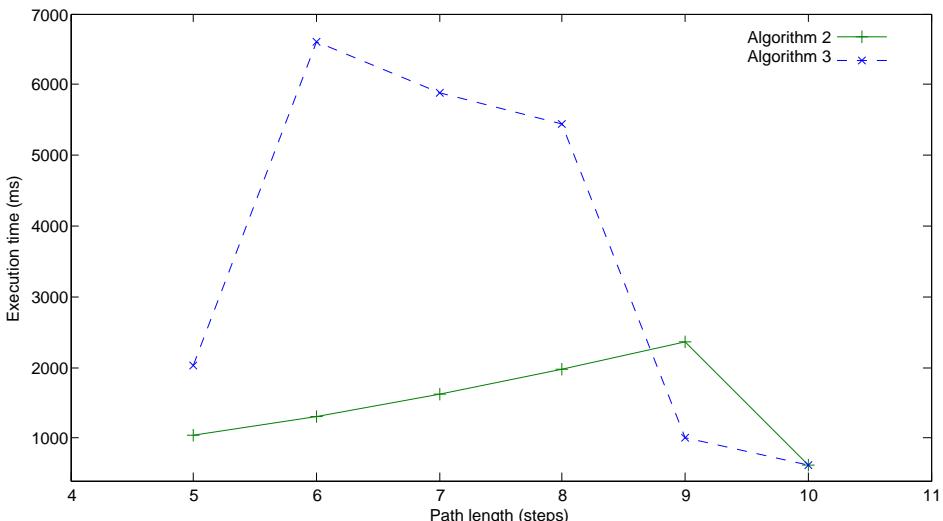
## 7.4 Optimal Chains Using At Most $M$ UAVs

To test the performance of the dual ascent algorithm (Algorithm 3) for solving the **STR-MinCostLimited** problems, the same environments as for **STR-ParetoLimited** were used. Both Algorithm 1 and Algorithm 2 are able to solve this problem, but as Algorithm 2 is the fastest algorithm available, it is used for comparison. We ran separate tests for different values of  $M$ , simulating a user requesting relay chains of different lengths.

For the **STR-MinCostLimited** problem, the more interesting test cases occur when the cost function based on distance is used, as it often produces several Pareto-optimal chains. For this reason, this cost function was used in testing. For each discretization, we used randomly generated 100 combinations of start and goal positions.

The test results are available in Table 7.2, in which the first columns display information about the discretizations. The last four columns display, for each algorithm, the percentage of test cases where one of the tested algorithms is faster than the other, and the average speedup when this happens. Only test cases with a speedup larger than 1% are displayed here. For test cases with a smaller speedup, the algorithms are considered equally fast. The numbers in the table represent the test cases with a speedup greater than 1%.

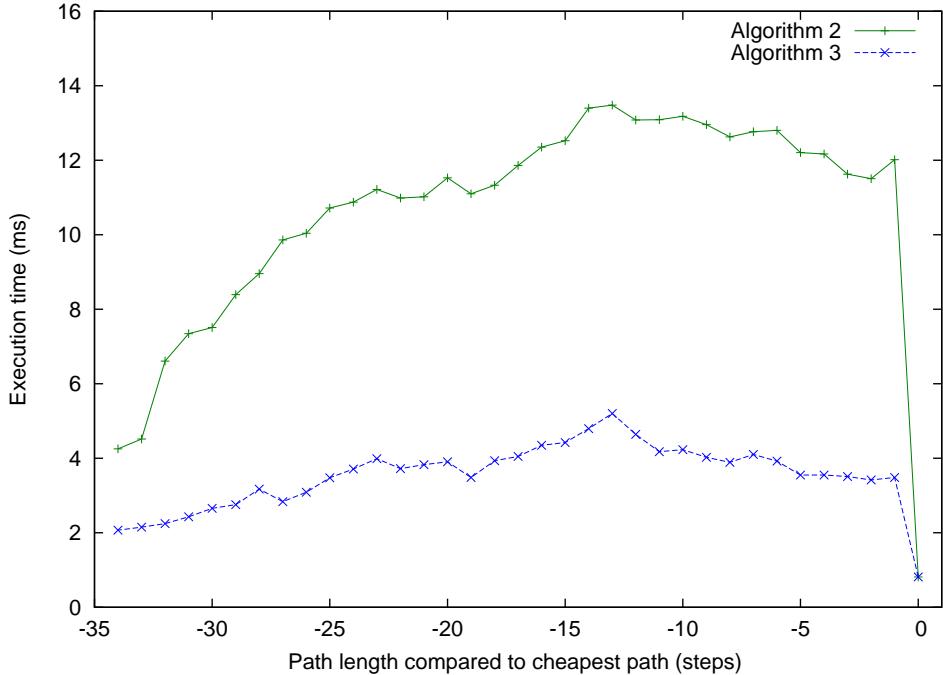
Figure 7.8 exemplifies the difference in the order in which the paths are calculated. Times for the MLMC-based label-correcting algorithm (Algorithm 2) are in green and times for the dual ascent algorithm (Algorithm 3) are in blue. Algorithm 2 first calculates



**Figure 7.8:** Timing results for Algorithm 2 and Algorithm 3 for a specific test case, showing the times for when chains of specific lengths are found. Data from urban environment with  $12 \times 12 \times 20$  grid cells.

World	Cell size (meters)	Algorithm 2 faster		Algorithm 3 faster	
		% paths	Speedup	% paths	Speedup
Randomized urban	40×40×40	0	0	100	2.96
	33×33×40	2	1.09	98	2.34
	25×25×25	26	1.48	74	2.03
	20×20×20	48	3.28	52	1.83
	15×15×20	84	11.6	16	1.63
	12×12×20	87	25.5	13	2.18
	10×10×20	90	33.9	10	2.01
Urban with boulevards	40×40×40	0	1.01	100	3.16
	33×33×40	1	1.04	99	2.46
	25×25×25	22	1.32	78	2.03
	20×20×20	49	2.87	51	1.85
	15×15×20	82	8.42	18	1.69
	12×12×20	74	23.7	26	1.51
Randomized dense urban	40×40×40	3	1.02	97	3.43
	33×33×40	1	1.09	99	2.70
	25×25×25	7	1.28	93	1.95
	20×20×20	34	1.68	66	1.83
	15×15×20	70	6.39	30	1.61
	12×12×20	80	16.4	20	1.98
	10×10×20	85	18.3	15	1.91
Stockholm	40×40×40	2	1.06	98	3.18
	33×33×40	5	1.08	95	2.56
	25×25×25	7	1.54	93	2.60
	20×20×20	32	1.74	68	2.08
	15×15×20	57	4.81	43	2.03
	12×12×20	72	13.5	28	1.87
	10×10×20	79	18.7	21	1.94
Revinge	40×40×40	1	1.01	99	2.17
	33×33×40	7	1.13	93	1.57
	25×25×25	54	1.43	46	1.35
	20×20×20	92	2.78	8	1.28
	15×15×20	97	11.1	3	1.22
	12×12×20	98	24.1	2	1.19
	10×10×20	99	28.0	1	1.40

**Table 7.2:** Speedup and test case information for testing of the **STR-MinCost-Limited** problem.

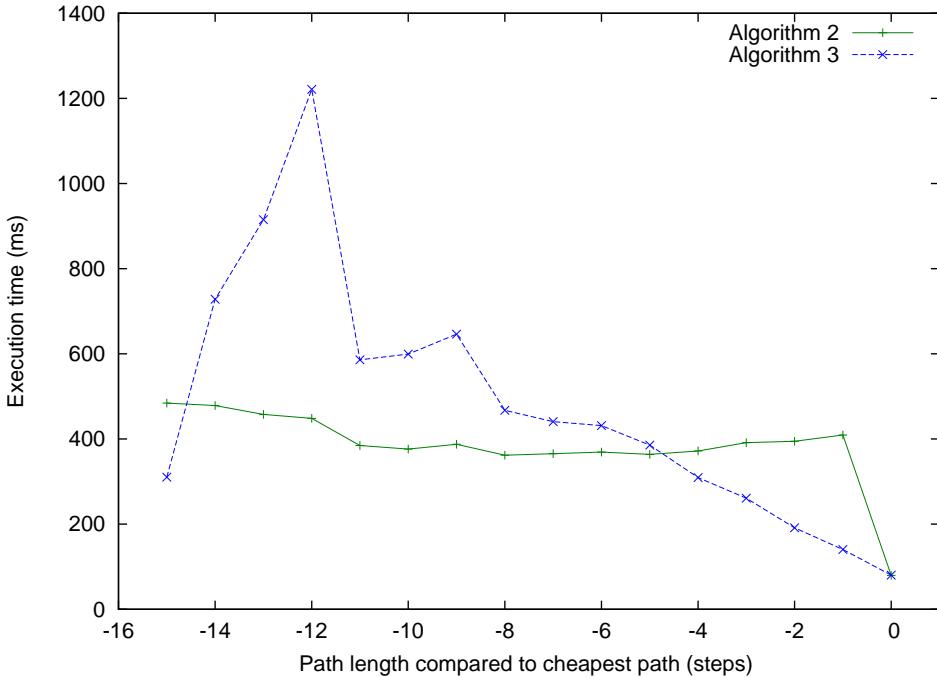


**Figure 7.9:** Timing results for testing of algorithms for the **STR-ParetoLimited** problem in the Stockholm environment with  $40 \times 40 \times 40$  m grid cells.

the longest (length 10) and cheapest chain during preprocessing, and all remaining chains are calculated in order of increasing length. Thus, the maximum execution time is for the chain of length 9. Algorithm 3 also finds the longest path first, and then goes on to find incrementally shorter paths. In general, longer paths can be generated quite quickly as they require fewer iterations. The chain of length 5 was found in shorter time than the chain of length 6 as separate tests were run for different values of  $M$ , and the search space was more effectively constrained when calculating the shorter chain.

We now give three examples of how the execution time is affected by the discretization in the Stockholm environment. Figure 7.9 displays the average execution times for a discretization with  $40 \times 40 \times 40$  m grid cells. As the lengths of the longest and cheapest chains vary, relative path lengths have been used. As an example, assume that the longest chain has a length of 20 hops. Then, the time reported for a relative path length  $-5$  corresponds to a path length of  $20 - 5 = 15$  hops. Naturally, not all test cases have the same number of Pareto-optimal chains and this affects the shape of the curves for execution times. For example, the execution times for relative length  $-10$  are averaged over considerably fewer (and on average more difficult) test cases than the results of relative length 0. For this reason, the curves at that end of the spectrum have a more uneven appearance.

Figure 7.10 displays execution times for the discretization with  $20 \times 20 \times 20$  m grid

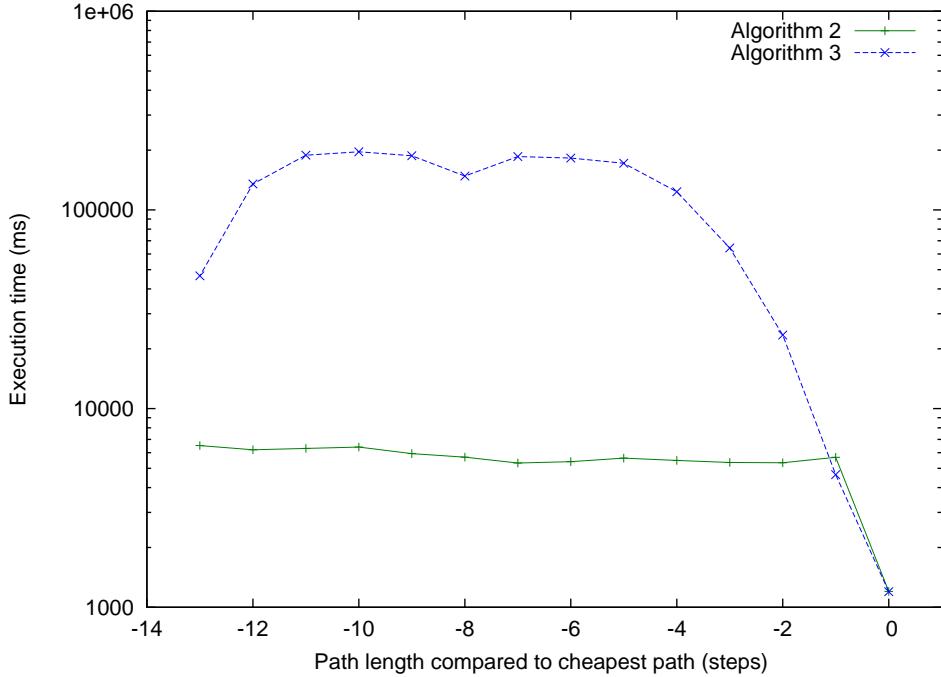


**Figure 7.10:** Timing results for the **STR-ParetoLimited** problem in the Stockholm environment with  $20 \times 20 \times 20$  m grid cells.

cells. Algorithm 3 is faster for paths slightly shorter than the cheapest path, and then the gap between the times for Algorithm 2 and Algorithm 3 increases as shorter paths are desired. For some path lengths, e.g. -14 to -5, Algorithm 2 is the faster as Algorithm 3 must perform a large number of iterations. Although each iteration is quite fast due to the constrained search space, Algorithm 2 is still faster.

Figure 7.11 displays execution times for the discretization with  $10 \times 10 \times 20$  m grid cells. Due to the large difference in execution time, this figure uses a log-scale. It is easy to see that Algorithm 2 provides a more consistent performance than Algorithm 3 in this environment. Here the pruning of the search space that Algorithm 3 performs is not sufficient to yield good performance. In many of the iterations, edges that do not affect the path from  $n_0$  to  $\tau_1$  get included in the tree, and therefore a large number of iterations must be executed to find a shorter path. The greater number of iteration of course leads to a longer execution time.

In graphs with few nodes and edges, the dual ascent algorithm outperforms Algorithm 2, in many cases being 2–3 times faster. As the number of nodes and edges increase, Algorithm 2 offers better performance than Algorithm 3. Although Algorithm 3 constrains the search space, this is not always enough as the number of calculations of MLMC-trees is quite large in some cases. For example, consider the Stockholm environ-



**Figure 7.11:** Timing results for testing of algorithms for the **STR-ParetoLimited** problem in the Stockholm environment with  $10 \times 10 \times 20$  m grid cells.

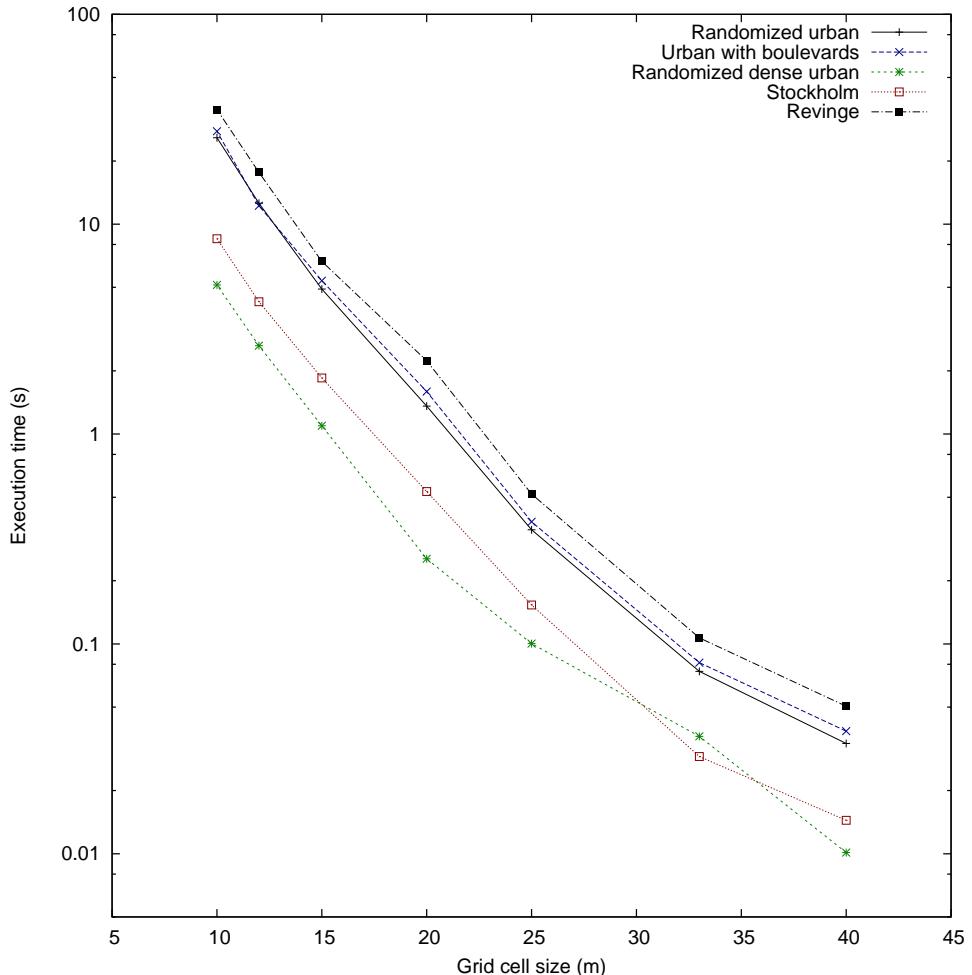
ment, in which the dual ascent algorithm offers the better performance in a majority of the cases, for the four coarsest discretizations. As the number of nodes and edges increase, Algorithm 2 offers much better performance, and the same trend is evident for the other testing environments.

From the test results is the evident that the dual ascent algorithm is more suitable in dense environments, where the number of nodes and edges is low, and Algorithm 2 is better suited for environments where there are many nodes and edges.

## 7.5 Relay Trees

To test the algorithms for multiple target relay positioning problems, we used the same environments and discretization as earlier, but this time we used nine targets distributed in three clusters with three targets in each cluster. Each randomized cluster had to be at least 300 meters from the base station and within each cluster, the targets were at most 100 m from each other. The cost function based on distance, as described in the beginning of this chapter, was used.

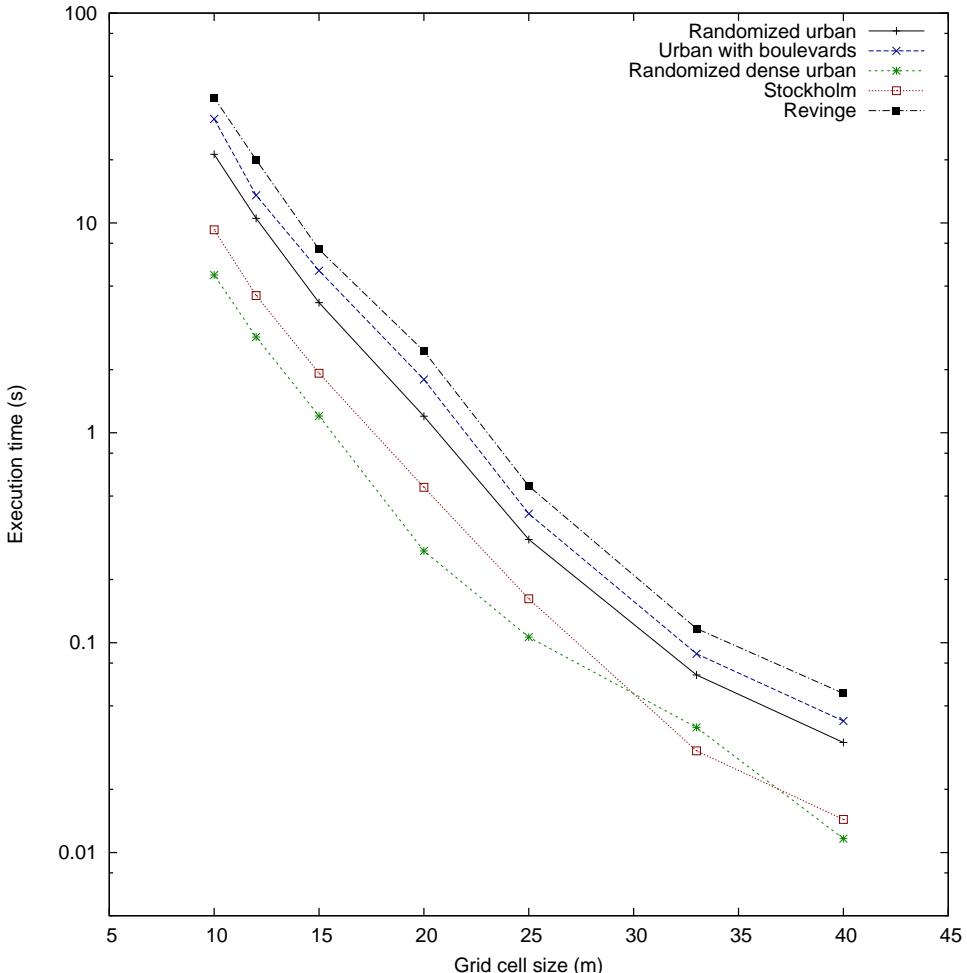
The initial relay trees were generated using Algorithm 4. We ran separate tests for the **MTR-MinLengthMinCost** and the **MTR-MinCostMinLength** problems. For the



**Figure 7.12:** Execution times for Algorithm 4 for calculating the initial relay tree for the **MTR-MinLengthMinCost** problem.

former, we used compound costs (see page 39) of the form  $\langle c, l \rangle$ , i.e. first the cost was minimized and in case of equal cost of several paths, the shortest path was chosen. The latter problem used the reversed prioritization. Execution times for **MTR-MinLength-MinCost** in the different environments are displayed in Figure 7.12 and execution times for **MTR-MinCostMinLength** are displayed in Figure 7.13. From the figures, we can see that execution times for the both problems are quite similar, and that there is large difference in calculation time for the different environments. There is sometimes an order of magnitude difference for the different environments but the same discretization. The reason is that the size of the graphs, especially the number of edges, differ considerably in the different environments. A smaller graph gives fewer possibilities when calculating

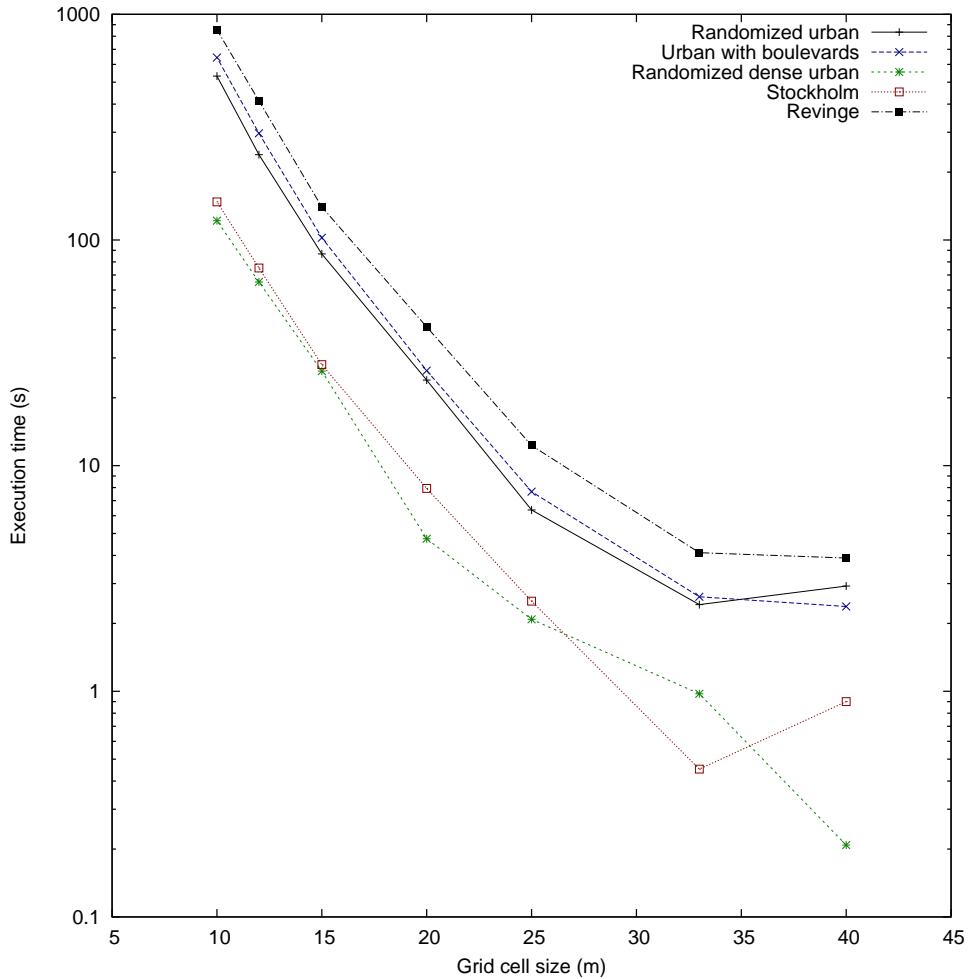
relay trees, which leads to a shorter execution time.



**Figure 7.13:** Execution times for calculating the initial relay tree using Algorithm 4, for the **MTR-MinCostMinLength** problem.

After the initial tree was calculated, we used Algorithm 6 to improve the trees further. The algorithm was set to first optimize subtrees further away from the root node, and then subtrees progressively closer to the root node were chosen. Optimization was performed until no improved subtree could be calculated. The execution times for Algorithm 6 for the **MTR-MinLengthMinCost** problem are available in Figure 7.14 and times for the **MTR-MinCostLimited** problem are available in Figure 7.15.

The average improvements of the primary and secondary factors (UAVs and cost for the **MTR-MinCostMinLength** problem and cost and UAVs for the **MTR-MinLengthMinCost** problem), are displayed in the rightmost part of Table 7.3. The numbers are the

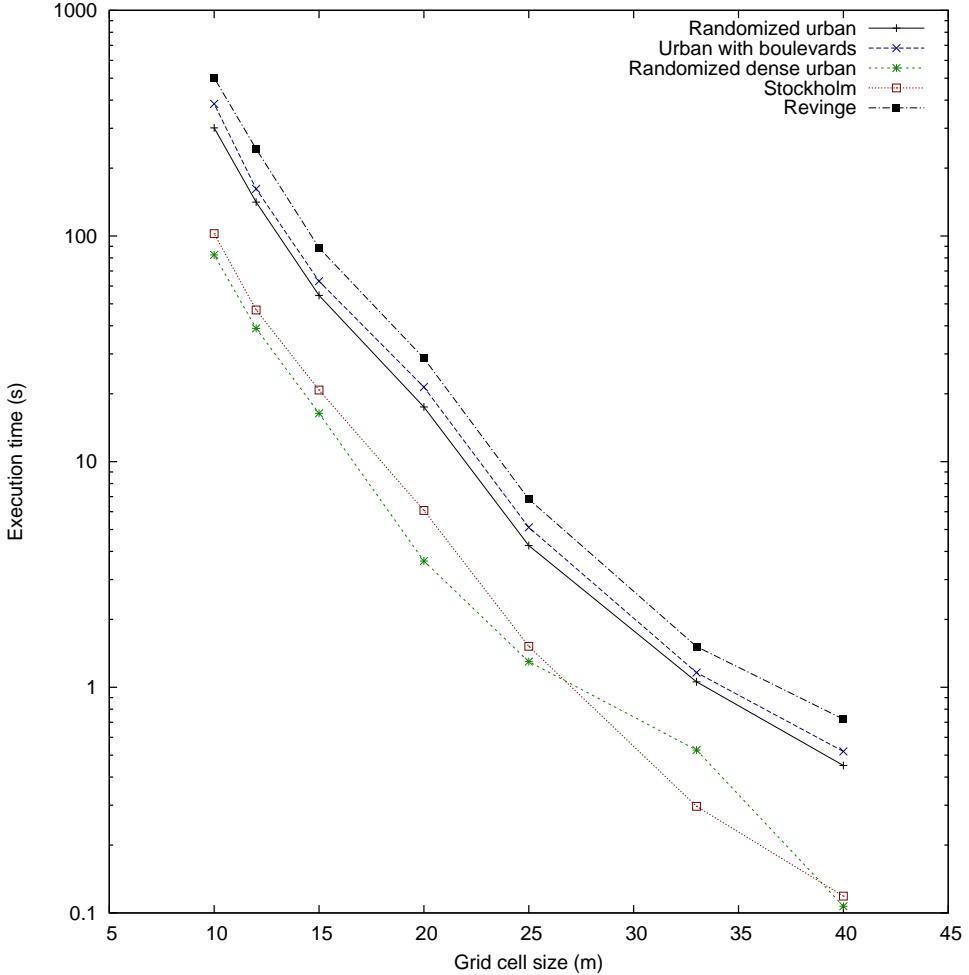


**Figure 7.14:** Execution times for Algorithm 6 for optimizing the relay tree, for the **MTR-MinLengthMinCost** problem.

improvement in percent over the initial tree.

The improvement in the number of UAVs is often in the 5–8% range, and the cost can often be decreased by more than 5%. When focusing on improving the cost, the improvement is often in the 4–6% range and the improvement in the secondary factor (number of UAVs) is slightly lower. From this, it is evident that focusing primarily on improving the number of UAVs in the tree yields a greater improvement as compared to focusing primarily on decreasing the cost. Also, the improvement is greater in larger graphs, which is to be expected as there are more opportunities to improve an existing subtree due to having more nodes to choose from.

When interpreting these numbers, it is important to remember that the cheapest path



**Figure 7.15:** Execution times for optimizing the relay tree using Algorithm 6, for the **MTR-MinCostMinLength** problem.

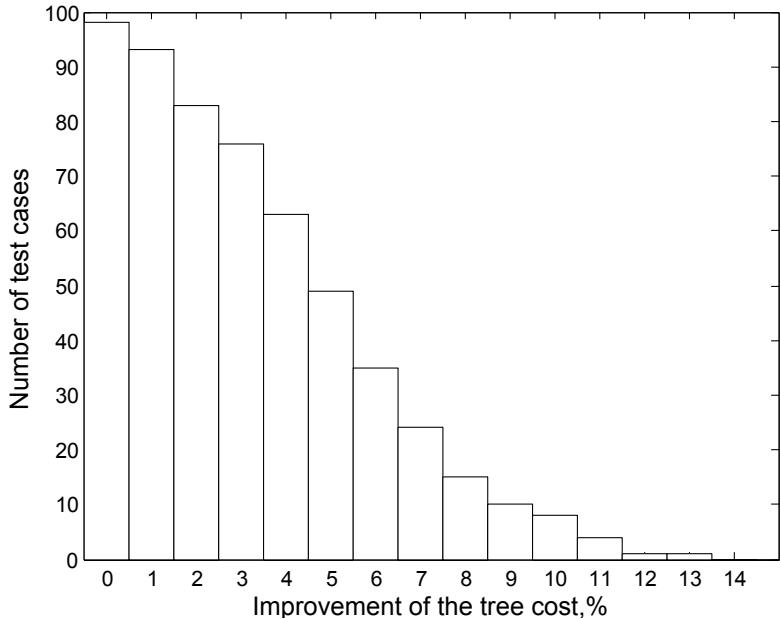
heuristic has proven to be competitive with more advanced heuristics in directed graphs [50, 95]. In the testing performed by Hsieh et al. [50], all results were within 5.5% from the optimal cost. While the graphs were much smaller than the ones used here, it is an indication that the cheapest path heuristic often finds high-quality solutions.

Figure 7.16a shows the distribution of cost improvement for the Revinge environment with  $20 \times 20 \times 20$  m grid cells. The y-axis displays the number of test cases that achieved a given improvement. Although the average improvement was a little more than 5%, one tree was improved by more than 13%. Figure 7.16b displays the improvement in the number of UAVs in the tree for the same test case. The average improvement in the number of UAVs is less than the improvement of the tree cost, but the maximum

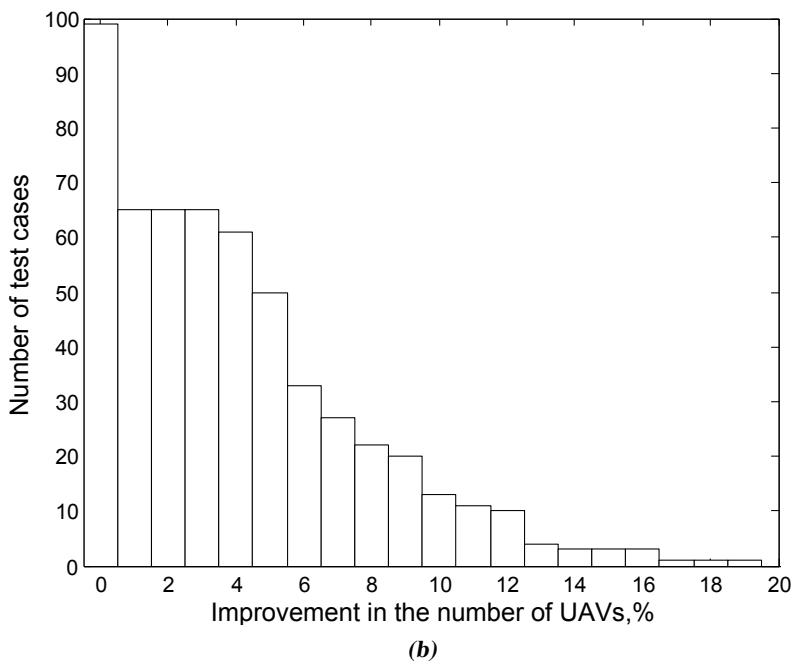
World	Cell size (meters)	Avg. improvement, %			
		MCML		MLMC	
		UAVs	Cost	Cost	UAVs
Randomized urban	40×40×40	6.3	8.1	3.2	2.9
	33×33×40	7.0	1.3	4.8	4.9
	25×25×25	7.0	9.2	4.4	3.7
	20×20×20	7.0	8.8	5.2	4.8
	15×15×20	7.7	5.0	5.9	5.9
	12×12×20	8.7	6.3	5.8	5.9
	10×10×20	9.0	5.8	6.2	6.0
Urban with boulevards	40×40×40	6.9	6.5	3.7	3.0
	33×33×40	6.1	4.3	4.8	4.8
	25×25×25	6.2	10.1	4.7	4.0
	20×20×20	7.4	8.4	4.9	4.5
	15×15×20	7.9	4.4	5.8	5.6
	12×12×20	8.3	4.1	6.3	6.3
	10×10×20	7.9	8.5	5.9	5.1
Randomized dense urban	40×40×40	3.0	2.7	2.1	0.6
	33×33×40	5.1	3.1	3.5	3.7
	25×25×25	6.6	4.5	3.9	3.6
	20×20×20	6.9	4.4	4.6	5.6
	15×15×20	9.7	2.9	5.4	6.4
	12×12×20	8.7	4.7	5.2	5.8
	10×10×20	10.1	6.9	6.3	7.8
Stockholm	40×40×40	4.2	5.3	2.0	0.5
	33×33×40	3.9	5.6	3.3	2.6
	25×25×25	3.7	6.0	3.4	3.4
	20×20×20	5.1	5.5	3.7	3.1
	15×15×20	6.2	6.4	4.7	4.6
	12×12×20	6.6	3.9	4.5	4.3
	10×10×20	6.5	3.0	4.4	4.8
Revinge	40×40×40	7.8	8.2	4.0	3.8
	33×33×40	7.0	3.8	5.0	4.6
	25×25×25	6.7	6.9	4.7	3.1
	20×20×20	6.3	9.1	5.3	4.6
	15×15×20	7.3	6.5	5.0	4.9
	12×12×20	8.2	4.1	5.4	4.7
	10×10×20	8.2	4.9	6.1	5.2

**Table 7.3:** Improvement in primarily the number of non-terminal nodes (UAVs) and secondarily the tree cost for the **MTR-MinCostMinLength** and the reverse prioritization for the **MTR-MinLengthMinCost** problem, as calculated by Algorithm 6.

## 7. Implementation and Experimental Results

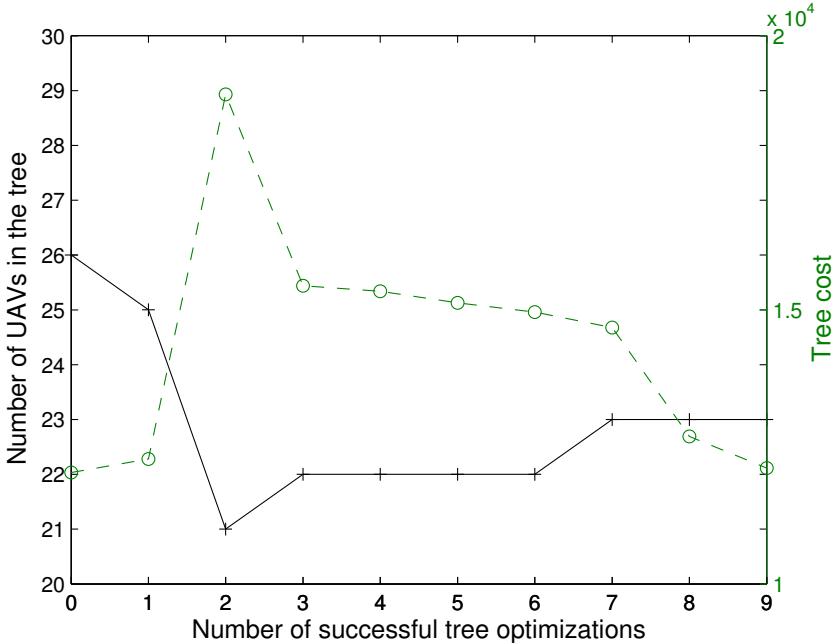


(a)



(b)

**Figure 7.16:** Data for the Revinge environment with  $20 \times 20 \times 20$  m grid cells. The y-axis displays the number of test cases with a given improvement.



**Figure 7.17:** Minimization of the number of hops is performed until a feasible tree is found, using at most 23 UAVs. Once a feasible tree is found, the optimization criterion is changed to finding the least cost feasible tree.

improvement is larger.

To conclude we give an example for a certain test case from the randomized urban environment with grid cells measuring  $33 \times 33 \times 40$  meters. Here Algorithm 6 is used to find the cheapest feasible tree. The original tree was calculated using Algorithm 4. Figure 7.17 shows how the number of UAVs and the tree cost change during optimization. The solid black curve is the number of UAVs in the tree and the dashed green curve is the tree's cost. The ground operator has set the number of UAVs available  $M = 23$ . As the first tree uses 26 UAVs, it is infeasible and the optimization criterion is to minimize the number of UAVs in the tree. The second successful optimization finds a tree using 21 UAVs. The optimization criterion is then automatically changed to finding the least cost feasible tree, i.e. a tree using at most 23 UAVs. Each new tree decreases the cost, until no lower cost tree is found, after the ninth subtree optimization. In total, the number of UAVs is decreased from 26 to 23 and the cost is only marginally increased. The complete optimization took less than 10 seconds and each improved tree is available to the user as soon as it is calculated.

## 7. Implementation and Experimental Results

# 8

---

## Discussion

In this thesis, we have provided definitions for relay positioning problems for both single and multiple targets. We have also suggested a solution process that enables us to model such problems with a large degree of flexibility. This process includes discretizing the environment and applying graph search algorithms to find relay chains and relay trees representing the locations where the UAVs should be placed. Graph search algorithms allow a great deal of flexibility, e.g. when choosing what factors to use for evaluating suitability of UAV placement.

We have presented two new algorithms for calculating relay chains for surveillance of a single target. One algorithm is tailored towards calculating a chain using at most a given number of UAVs. The other algorithm calculates a set of Pareto-optimal relay chains, and lets the ground operator choose between the different alternatives.

The problem of simultaneous surveillance of several targets is considerably more difficult since it adds further restrictions to the already NP-hard Steiner tree problem. Very few existing algorithms are applicable to our problem, as we require that all surveillance targets are leaves in the tree. Several of the algorithms that can model this requirement cannot be used due to memory requirements or assumptions about the graph. To quickly generate relay trees for such situations, we modify the cheapest path heuristic. Due to its flexible nature, we can use it to solve problems involving a large variety of restrictions and extensions. Once a relay tree has been calculated, we apply another algorithm to continually improve the tree. The algorithm performs local optimizations of the tree and each such improvement yields an improvement of the complete tree. Once such an improved tree has been found, it can be immediately displayed to the ground operator. The process of improvement can be performed until the algorithm can no longer improve the tree or until a certain time has passed.

The algorithms have been implemented and integrated into the UAS infrastructure

developed at our lab. We have described the implementation as well as the testing that has been performed. The algorithms have been tested in both simulated environments as well as discretizations of real environments.

Although we are able to solve some problems optimally or close to optimally, there are still potential for improvement and expansion in many areas. Some of the areas are discussed in the next section.

### 8.1 Future Research

A discretization created from a three-dimensional grid was used for testing of the algorithms presented here and several other discretization options were also discussed. It would be interesting to make a structured evaluation of different discretization approaches for different environments to determine which, or more likely which combinations of, discretizations that are most suitable for different environments. For example, in an urban environment, it appears likely that a grid combined with an expanded geometry graph will provide a good compromise between memory requirement and quality of relay chains and trees.

In the **STR-MinCostLimited** problem, an optimization is performed in order to find the least-cost chain given a limit on the number of hops in the chain. This is an example of a bi-objective problem where one objective function can be specified freely while other is fixed. The dual ascent algorithm can be generalized to handle problems where both objectives can be specified freely, something that can be very useful in the context of relays. As an example: when micro-UAVs are used as relays it can be necessary to explore the trade-offs between transmission quality and tolerance to wind drift for different positioning alternatives. The kind of problem suggested here is in its general form the NP-hard constrained shortest path problem [10, 39], which is often solved using Lagrangian relaxation, based on the same dual function as the dual ascent method presented here, but with other dual search techniques.

In this thesis, we assumed that the targets were stationary and that their positions were known. Obvious extensions are lifting one or more of these assumptions, and to provide relay chains to a moving target or relay trees to several moving targets. This increases the complexity of the problem considerably. As an example, it might be impossible to maintain constant communication between the surveillance UAV and the base station as the UAVs have to temporarily move behind buildings and outside the communication range. For such situations, an approach where it is guaranteed that information can flow from the surveillance UAV to the base station at certain time intervals, e.g. every fifth second, might be sufficient.

---

## Bibliography

- [1] AIICS, 2011. <http://www.ida.liu.se/divisions/aiics/>. Accessed January 14, 2011.
- [2] Ian F. Akyildiz, Weilian Su, Yogesh Sankarasubramaniam, and Erdal Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38(4):393–422, 2002.
- [3] Jamal N. Al-Karaki and Ahmed E. Kamal. Routing techniques in wireless sensor networks: a survey. *IEEE Wireless Communications*, 11(6):6–28, December 2004.
- [4] Nancy M. Amato and Yan Wu. A randomized roadmap method for path and manipulation planning. In *IEEE International Conference on Robotics and Automation (ICRA)*, 1996.
- [5] Stuart O. Anderson, Reid Simmons, and Dani Goldberg. Maintaining Line of Sight Communications Network between Planetary rovers. In *Proceedings of the 2003 IEEE/RSJ International Conference of Intelligent Robots and Systems (IROS)*, pages 2266–2272, 2003.
- [6] Ronald C. Arkin and Jonathan Diaz. Line-of-sight constrained exploration for reactive multiagent robotic teams. In *7th International Workshop on Advanced Motion Control*, pages 455–461, 2002.
- [7] Franz Aurenhammer. Voronoi diagrams - a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3):345–405, 1991.
- [8] Anantaram Balakrishnan and Kemal Altinkemer. Using a hop-constrained model to generate alternative communication network design. *ORSA Journal of Computing*, 4:192–205, 1992.

## Bibliography

---

- [9] Randal Beard, Derek Kingston, Morgan Quigley, Deryl Snyder, Reed Christiansen, Walt Johnson, Timothy McLain, and Michael A. Goodrich. Autonomous vehicle technologies for small fixed-wing UAVs. *Journal of Aerospace Computing, Information, and Communication*, 2(1):92–108, January 2005.
- [10] John E. Beasley and Nicos Christofides. An algorithm for the resource constrained shortest path problem. *Networks*, 19(4):379–394, July 1989.
- [11] Paul Blaer. Robot path planning using generalized Voronoi diagrams. [http://www1.cs.columbia.edu/~pblaer/projects/path\\\_planner/voronoi.html](http://www1.cs.columbia.edu/~pblaer/projects/path_planner/voronoi.html), 2009. Accessed October 13, 2009.
- [12] Valérie Boor, Mark H. Overmars, and A. Frank van der Stappen. Gaussian sampling for probabilistic roadmap planners. In *Proceedings of the International Conference on Robotics and Automation (ICRA)*, 2001.
- [13] Scott A. Bortoff. Path planning for UAVs. In *Proceedings of the American Control Conference*, pages 364–368, vol.1, 2000.
- [14] Timothy X. Brown, Brian Argrow, Cory Dixon, Sheetalkumar Doshi, Roshan-George Thekkekunnel, and Daniel Henkel. Ad hoc UAV ground network (AUGNet). In *Proceedings of the AIAA 3rd “Unmanned Unlimited” Technical Conference*, 2004.
- [15] Oleg Burdakov, Patrick Doherty, Kaj Holmberg, Jonas Kvarnström, and Per-Magnus Olsson. Positioning unmanned aerial vehicles as communication relays for surveillance tasks. In *Proceedings of the 2009 Conference on Robotics: Science and Systems (RSS)*, pages 257–264, 2009.
- [16] Oleg Burdakov, Patrick Doherty, Kaj Holmberg, Jonas Kvarnström, and Per-Magnus Olsson. Relay positioning for unmanned aerial vehicle surveillance. *International Journal of Robotics Research*, 29(8):1069–1087, 2010.
- [17] Oleg Burdakov, Patrick Doherty, Kaj Holmberg, and Per-Magnus Olsson. Optimal placement of UV-based communications relay nodes. *Journal of Global Optimization*, 48(4):511–531, 2010.
- [18] Oleg Burdakov, Kaj Holmberg, and Per-Magnus Olsson. A dual ascent method for the hop-constrained shortest path with application to positioning of unmanned air vehicles. Technical Report LiTH-MAT-R-2008-07, Department of Mathematics, Linköping University, 2008.
- [19] Jaroslaw Byrka, Fabrizio Grandoni, Thomas Rothvoß, and Laura Sanit  . An improved LP-based approximation for steiner tree. In *Proceedings of the 42nd ACM symposium on Theory of computing*, pages 583–592. ACM, 2010.

- 
- [20] Carmen Cerasoli. An analysis of unmanned airborne vehicle relay coverage in urban environments. In *Proceedings of MILCOM 2007*. IEEE, 2007.
  - [21] Phillip R. Chandler, Meir Pachter, and Steven Rasmussen. UAV cooperative control. In *Proceedings of the American Control Conference*, pages 50–55, 2001.
  - [22] Moses Charikar, Chandra Chekuri, To-yat Cheung, Zuo Dai, Ashish Goel, Sudipto Guha, and Ming Li. Approximation algorithms for directed Steiner problems. In *SODA '98: Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms (SODA)*, pages 192–200, 1998.
  - [23] Nang-Ping Chen. New algorithms for the Steiner tree on graphs. In *IEEE Symposium on Circuits and Systems*, pages 1217–1219, 1983.
  - [24] Yu Ming Chen, Liang Dong, and Jun-Seok Oh. Real-time video relay for UAV traffic surveillance systems through available communication networks. In *Wireless Communications and Networking Conference (WCNC)*, 2007.
  - [25] Chen-Mou, Cheng, Pai-Hsiang Hsiao, H. T. Kung, and Dario Vlahh. Maximizing throughput of UAV-relaying networks with the load-carry-and-deliver paradigm. In *Wireless Communications and Networking Conference (WCNC)*, 2007.
  - [26] Sunil Chopra and M. R. Rao. The Steiner tree problem ii: Properties and classes of facets. *Mathematical Programming*, 64(1):231–246, 1994.
  - [27] Gianpaolo Conte, Maria Hempel, Piotr Rudol, David Lundström, Simone Duranti, Mariusz Wzorek, and Patrick Doherty. High accuracy ground target geo-location using autonomous micro aerial vehicle platforms. In *Proceedings of the AIAA Guidance, Navigation, and Control Conference and Exhibit*, 2008.
  - [28] CORBA., 2009. <http://www.omg.org/gettingstarted/corbafaq.htm>. Accessed April 19, 2010.
  - [29] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms*. MIT Press, 1990.
  - [30] Alysson M. Costa, Jean-Francois Cordeau, and Gilbert Laporte. Fast heuristics for the Steiner tree problem with revenues, budget and hop constraints. *European Journal of Operational Research*, 190(1):68–78, 2008.
  - [31] Geir Dahl and Luis Gouveia. On the directed hop-constrained shortest path problem. *Operations Research Letters*, 32(1):15–22, January 2004.
  - [32] Edsger Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

## Bibliography

---

- [33] Patrick Doherty, Patrik Haslum, Fredrik Heintz, Torsten Merz, Per Nyblom, Tommy Persson, and Björn Wingman. A Distributed Architecture for Autonomous Unmanned Arial Vehicle Experimentation. In *Proceedings of the 7th International Symposium on Distributed Autonomous Systems*, pages 221–230, 2004.
- [34] Patrick Doherty, Jonas Kvarnström, Fredrik Heintz, David Landén, and Per-Magnus Olsson. Research with collaborative unmanned aircraft systems. In *Proceedings of the Dagstuhl Workshop on Cognitive Robotics*, 2010.
- [35] Patrick Doherty, David Landén, and Fredrik Heintz. A distributed task specification language for mixed-initiative delegation. In *Proceedings of the 13th International Conference on Principles and Practice of Multi-Agent Systems (PRIMA-2010)*, 2010.
- [36] Doratha E. Drake and Stefan Hougardy. On approximation algorithms for the terminal Steiner tree problem. *Information Processing Letters*, 89:15–18, 2004.
- [37] Ding-Zhu Du, Bing Lu, Hung Ngo, and Panos M. Pardalos. *Steiner tree problems*. Kluwer Academic Publishers, 2001.
- [38] Ding-Zhu Du, J. MacGregor Smith, and J.H. Rubenstein. *Advances in Steiner Trees*. Kluwer Academic Publishers, 2000.
- [39] Irina Dumitrescu and Natashia Boland. Improved preprocessing, labeling and scaling algorithms for the weight-constrained shortest path problem. *Networks*, 42(3):135–153, 2003.
- [40] Simone Duranti, Gianpaolo Conte, David Lundström, Piotr Rudol, Mariusz Wzorek, and Patrick Doherty. LinkMAV, a prototype rotary wing micro aerial vehicle. In *Proceedings of the 17th IFAC Symposium on Automatic Control in Aerospace*, 2007.
- [41] Joel M. Esposito and Thomas W. Dunbar. Maintaining wireless connectivity constraints for swarms in the presence of obstacles. In *Proceedings of the International Conference on Robotics and Automation (ICRA)*, pages 946–951, 2006.
- [42] Marcia Fampa and Kurt M. Anstreicher. An improved algorithm for computing Steiner minimal trees in Euclidean d-space. *Discrete Optimization*, 5(2):530–540, 2008.
- [43] R. A. Finkell and J. L. Bentley. Quad trees: a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974.
- [44] David Gesbert, Mansoor Shafi, Da shan Shiu, Peter J. Smith, and Ayman Naguib. From theory to practice: an overview of MIMO space-time coded wireless systems. *IEEE Journal on Selected Areas in Communications*, 21(3):281–302, 2003.

- 
- [45] Fred W. Glover and Gary A. Kochenberger. *Handbook of Metaheuristics*. Springer, 2003.
  - [46] Roch Guérin and Ariel Orda. Computing shortest paths for any number of hops. *IEEE/ACM Transactions on Networking*, 10(5), 2002.
  - [47] Zhu Han, A. Lee Swindlehurst, and K. J. Ray Liu. Smart deployment/movement of unmanned air vehicle to improve connectivity in MANET. In *Wireless Communications and Networking Conference (WCNC)*, 2006.
  - [48] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost graphs. *IEEE Transactions on Systems Science and Cybernetics*, 4(2), 1968.
  - [49] Donald Hearn and M. Pauline Baker. *Computer Graphics with OpenGL. 3rd Edition*. Prentice Hall, 2003.
  - [50] Ming-I Hsieh, Eric Hsiao-Kuang Wu, and Men-Feng Tsai. FasterDSP: A faster approximation algorithm for directed Steiner tree problem. *Journal of Information Science and Engineering*, 22:1409–1425, 2006.
  - [51] Sun-Yuan Hsieh and Huang-Ming Gao. On the partial terminal Steiner tree problem. *Journal of Supercomputing*, 41(1):41–52, 2007.
  - [52] Sun-Yuan Hsieh and Wen-Hao Pi. On the partial-terminal Steiner tree problem. In *Proceedings of the International Symposium on Parallel Architectures, Algorithms, and Networks*, pages 173–177, 2008.
  - [53] David Hsu, Tingting Jiang, John Reif, and Zheng Sun. The bridge test for sampling narrow passages with probabilistic roadmap planners. In *Proceedings of the International Conference on Robotics and Automation (ICRA)*, 2003.
  - [54] Frank K. Hwang, Dana S. Richards, and Pawel Winter. *The Steiner Tree Problem*. North-Holland, 1992.
  - [55] David B. Johnson and David A. Maltz. *Dynamic Source Routing In Ad Hoc Wireless Networks*. Kluwer Academic Publishers, 1996.
  - [56] J.R. Ford Jr. Network flow theory. Technical report, The Rand Corporation, 1956. Technical Paper P-923.
  - [57] Lydia E. Kavraki, Petr Svestka, Jean-Claude Latombe, and Mark H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE Transactions on Robotics and Automation*, 12(4):566–580, 1996.
  - [58] David Landén, Fredrik Heintz, and Patrick Doherty. Complex task allocation in mixed-initiative delegation: A UAV case study (work in progress). In *Proceedings*

- of the 13th International Conference on Principles and Practice of Multi-Agent Systems (PRIMA–2010)*, 2010.
- [59] Steven M. LaValle. Rapidly-exploring random trees: A new tool for path planning. Technical Report Technical Report 98-11, Computer Science Dept., Iowa State University, 1998.
  - [60] Steven M. LaValle. *Planning Algorithms*. Cambridge Press, 2006.
  - [61] Eugene L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, 1976.
  - [62] Qun Li and Daniela Rus. Sending messages to mobile users in disconnected ad-hoc wireless networks. In *Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 44–55, 2000.
  - [63] Shu Lin and Daniel J. Costello. *Error Control Coding (2nd Edition)*. Prentice Hall, 2004.
  - [64] Tomás Lozano-Pérez and Michael A. Wesley. An algorithm for planning collision-free paths among polyhedral obstacles. *Communications of the ACM*, 22(10):560–670, 1979.
  - [65] Madhav V. Marathe, R. Ravi, Ravi Sundaram, S. S. Ravi, Daniel J. Rosenkrantz, and Harry B. Hunt. Bicriteria network design problems. *Journal of Algorithms*, 28(1):142–171, 1998.
  - [66] Fábio Viduani Martinez, José Coelho de Pina, and José Soares. Algorithms for terminal Steiner trees. *Theoretical Computer Science*, 389:133–142, 2007.
  - [67] Martin Mauve, Jörg Widmer, and Hannes Hartenstein. A survey on position-based routing in mobile ad hoc networks. *IEEE Network*, 15(6):30–39, November/December 2001.
  - [68] Timothy W. McLain and Randal W. Beard. Cooperative rendezvous of multiple unmanned air vehicles. In *AIAA Guidance, Navigation and Control Conference*, 2000.
  - [69] Kaisa Miettinen. *Nonlinear Multiobjective Optimization*. Kluwer Academic Publishers, 1999.
  - [70] Joseph S. B. Mitchell. Geometric shortest paths and network optimization. In J.R. Sack and J. Urrutia, editors, *Handbook of Computational Geometry*, chapter 15. North Holland, 1999.
  - [71] Alejandro R. Mosteo and Luis Montano. Concurrent tree traversals for improved mission performance under limited communication range. In *Proceedings of*

---

*the 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2009.

- [72] Alejandro R. Mosteo, Luis Montano, and Michail G. Lagoudakis. Guaranteed-performance multi-robot routing under limited communication range. In Hajime Asama, Haruhisa Kurokawa, Jun Ota, and Kosuke Sekiyama, editors, *Distributed Autonomous Robotic Systems 8*, pages 491–502. Springer Berlin Heidelberg, 2009.
- [73] Hoa G. Nguyen, Narek Pezeshkian, Michelle Raymond, A. Gupta, and Joseph M. Spector. Autonomous communication relays for tactical robots. In *Proceedings of the 11th International Conference on Advanced Robotics (ICAR)*, 2003.
- [74] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization, Second Edition*. Springer, 2006.
- [75] Claude Oestges and Bruno Clerckx. *MIMO Wireless Communications: From Real-World Propagation to Space-Time Code Design*. Academic Press, 2007.
- [76] Atsuyuki Okabe, Barry Boots, Kokichi Sugihara, and Sung Nok Chiu. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. Wiley, 2000.
- [77] Per-Magnus Olsson, Jonas Kvarnström, Patrick Doherty, Oleg Burdakov, and Kaj Holmberg. Generating UAV communication networks for monitoring and surveillance. In *Proceedings of the International Conference on Control, Automation, Robotics and Vision (ICARCV)*, 2010.
- [78] OpenStreetMap., 2010. <http://www.openstreetmap.org>. Accessed January 15, 2010.
- [79] Ramesh Palat, Annamalai Annamalai, and Jeffrey Reed. Cooperative relaying for ad-hoc ground networks using swarm UAVs. In *Proceedings of MILCOM 2006*, 2006.
- [80] Per Olof Pettersson. Sampling-based path planning for an autonomous helicopter.
- [81] Frank J. Pinkney, Dan Hampel, and Stef DiPierro. Unmanned aerial vehicle (UAV) communications relay. In *Proceedings of MILCOM 1996*. IEEE, 1996.
- [82] Gabriel Robins and Alexander Zelikovsky. Improved Steiner tree approximation in graphs. In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms (SODA)*, pages 770–779, 2000.
- [83] Martijn N. Rooker and Andreas Birk. Multi robot exploration under the constraints of wireless networking. *Control Engineering Practice*, 15(4):435–445, 2007.

## Bibliography

---

- [84] Steven Roos. Scheduling for ReMove and other partially connected architectures. Technical Report 1-68340-44(2001)-05, Laboratory of Information Technology and Systems, Delft University of Technology, 2001.
- [85] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan-Kaufmann, 2006.
- [86] Tom Schouwenaars. *Safe Trajectory Planning of Autonomous Vehicles*. PhD thesis, Massachusetts Institute of Technology, February 2006.
- [87] Tom Schouwenaars, Andrew Stubbs, James Paduano, and Eric Feron. Multivehicle path planning for nonline-of-sight communication. *Journal of Field Robotics*, 23(3/4):269–290, 2006.
- [88] Andrea Simonetto, Paul Scerri, and Katia Sycara. A mobile network for mobile sensors. In *Proceedings of the 11th International Conference on Information Fusion*, 2008.
- [89] Vinay Sridhara and Stephan Bohacek. Realistic propagation simulation of urban mesh networks. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 51(12):3392–3412, 2007.
- [90] John Sweeney, TJ Brunette, Yuandong Yang, and Roderic Grupen. Coordinated teams of reactive mobile platforms. In *Proceedings of the 2002 IEEE International Conference on Robotics and Automation (ICRA)*, 2002.
- [91] Hiromitsu Takahashi and Akira Matsuyama. An approximate solution for the Steiner problem in graphs. *Mathematica Japonica*, 24, 1980.
- [92] UAS Technologies Sweden AB, 2011. [www.uastech.com](http://www.uastech.com).
- [93] Martin Thimm. On the approximability of the Steiner tree problem. In J. Sgall, A. Pultr, and P. Kolman, editors, *Lecture Notes in Computer Science 2136: Mathematical Foundations of Computer Science 2001*, pages 678–689. Springer Berlin / Heidelberg, 2001.
- [94] David Tse and Pramod Viswanath. *Fundamentals of Wireless Communication*. Cambridge University Press, 2005.
- [95] Stefan Voß. Worst-case performance of some heuristics for Steiner’s problem in directed graphs. *Information Processing Letters*, 48:99–105, 1993.
- [96] Stefan Voß. The Steiner tree problem with hop constraints. *Annals of Operations Research*, 86(0):321–345, 1999.
- [97] Stefan Voß. Modern heuristic search methods for the Steiner tree problem in graphs. In Ding-Zhu Du, J. MacGregor Smith, and J.H. Rubenstein, editors, *Advances in Steiner Trees*, pages 283–323. Kluwer Academic Publishers, 2000.

- 
- [98] Mirko Vujošević and Milan Stanojević. A bicriterion Steiner tree problem on graph. *Yugoslav Journal of Operations Research*, 13(1):25–33, 2003.
  - [99] Jean-Frédéric Wagen and Karim Rizk. Radiowave propagation, building databases, and GIS: anything in common? A radio engineer’s viewpoint. *Environment and Planning B: Planning and Design*, 30(5):767–787, September 2003.
  - [100] Paweł Winter. Steiner problems in networks: A survey. *Networks*, pages 129–167, 1987.
  - [101] Martin Zachariasen and Paweł Winter. Obstacle-avoiding euclidean Steiner trees in the plane: An exact algorithm. In M.T. Goodrich and C.C. McGeoch, editors, *Lecture Notes in Computer Science 1619: Algorithm Engineering and Experimentation*, pages 286–299. Springer, 1999.
  - [102] Pengcheng Zhan. *Optimizing Wireless Throughput: Methods and Applications*. PhD thesis, Brigham Young University, 2007.
  - [103] Shlomo Zilberstein. Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3):73–83, 1996.
  - [104] Leonid Zosin and Samir Khuller. On directed Steiner trees. In *SODA ’02: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 59–63, 2002.

## Bibliography

---

## **Part II**

# **Contributions to Derivative-free Optimization**



---

*Derivatives are the most useful attributes a function can be blessed with when it comes to optimization.*

Andrew Conn, Nicholas Gould and Philippe Toint.  
*Trust-region Methods*. MPS-SIAM, 2000.



# 9

---

## Introduction

A common problem during product development is to minimize or maximize a certain aspect of a product, while still fulfilling other important requirements. A typical example is to minimize the weight of a car but still have a high strength of the chassis so that the car is safe in case of an accident. Naturally, such calculations are not performed by hand. Instead sophisticated computer software is used. Such software typically solves a huge number of equations, in order to provide an accurate result in the end. In some cases the result of one simulation is used as input to another simulator, which in turn performs an equally great number of calculations. Together, this results in a problem where the function's analytical expression is unknown. Such problems are not limited to production of mechanical products: they occur in very diverse areas: selecting parameters for facial recognition, choosing capacities of components in paper mills, in finance, in electronics and more. The interested reader is referred to the book by Conn et al. [17] for more examples of applications.

The complexity of the simulations has some important implications. The first is that since the function's analytical expression is unknown, we are unable to obtain gradient and Hessian information. Another implication is that each simulation, which corresponds to an evaluation of the objective function, typically takes a long time. Whereas most problems in academic benchmarking take at most a few milliseconds, a single simulation of an industrial problem may take several minutes or hours to perform. In some cases a few hours is not sufficient for a simulation, instead several days or even weeks are required!

Naturally, to find good solutions to the problems, optimization algorithms in software are used in order to automate this process. Since the objective function is very time-consuming, it is common to focus on minimizing the number of evaluations. However, this completely disregards that new computer hardware is typically able to perform several

calculations in parallel. If we can perform e.g. five calculations in parallel instead of two in sequence, we not only save time but we also gain more information about the objective function. This should make it evident that minimizing the *number* of function evaluations is not the primary objective. More important is minimizing the *number of times* one or more points are evaluated in parallel.

For solving problems where derivatives are unavailable, algorithms for *derivative-free optimization* are used. They can be broadly divided into two categories: direct search methods and model building methods. The former evaluates the objective function directly while the latter creates a model of the objective function and uses it in calculations. The idea is that the model acts as a surrogate for the real objective function and that calculations involving the model are considerably cheaper to perform than a function evaluation. The model is then optimized using some sort of optimization algorithm. The optimization algorithm returns a point, typically an optimum of the model, and the objective function is then evaluated in that point, i.e. a simulation is performed. Then the model is updated with the new point, and the algorithm continues with the next iteration.

In this thesis, we investigate how model building algorithms for derivative-free optimization can be parallelized. We scrutinize such algorithms, beginning when the initial model is created. How we can parallelize the algorithm to handle several concurrent starting points is discussed next. Then we investigate different ways how to generate several points for evaluation in each iteration. Naturally, if there are several starting points, we let each one generate a point for evaluation, but we also discuss different ways how to generate several points from each starting point. We also investigate how to create synergy between the starting points by sharing information. Saving function evaluations by storing all points in a cache is also discussed, together with different ways to prioritize the order in which the generated points should be evaluated. Different ways to implement the parallelization and the consequences that this has on the algorithm are also discussed.

Implementation is a very important but often neglected part of optimization software. It is common to make a Matlab implementation of the algorithm in order to run a few test cases for a scientific paper. Needless to say, such implementations are rarely suitable for non-academic use. We are proud to say that we have implemented a complete algorithm with the extensions mentioned above, for use in industrial settings. This means that we have spent a considerable amount of time trying to make the algorithm numerically robust, documenting not only the implementation itself but also the different test cases constructed to test our implementation. Furthermore, we have supplied documentation about the different settings and suggested suitable default values so that it shall be relatively easy for the user to use our software. Our implementation has been used by several companies and is included in one of the companies' internal simulator software.

## 9.1 Thesis Contributions

The main contributions in this part of the thesis include:

- A survey of derivative-free algorithms suitable for optimization of time-consuming objective functions in industrial settings. This includes both direct search algorithms such as the classic Nelder-Mead Simplex algorithm as well as more advanced model building algorithms, and some of the theory behind the latter.
- A description of and examples of how model building algorithms for derivative-free optimization can be parallelized with regards to computer resource use and information sharing. We make very few assumptions about the algorithm and the intention is that our findings shall be applicable in most model building algorithms.
- An industrial strength implementation of a parallelized version of one of the state of the art algorithms for derivative-free optimization. A description of the implementation as well as test results on synthetic test cases, comparing the results of using and not using the parallel extensions.
- A display of how the implemented algorithm has been applied in industrial test cases. The algorithm has been used on such diverse applications as optimization of components in a paper mill and the optimization of rolling bearings.

## 9.2 Presentations

Parts of this thesis have previously been presented as follows:

- **Per-Magnus Olsson.** Parallelization of Algorithms For Derivative-free Optimization. Presented at *21st International Symposium on Mathematical Programming (ISMP)*. August 19–24, 2012 in Berlin, Germany.
- **Per-Magnus Olsson.** Parallel Extensions of Algorithms For Derivative-free Optimization. Presented at *The Fourth International Conference on Continuous Optimization (ICCOPT)*. July 27–August 1, 2013 in Lisbon, Portugal.

## 9.3 Software

The software described in this thesis has been incorporated in the software SKF BEAST 11.2 released in December 2013. See Section 10.2.2 for more information about SKF BEAST and Section 18.7 for more information about the specific implementation of our software in that simulator.

## 9.4 Thesis Outline

This part of the thesis is organized as follows. Chapter 10 gives the problem setting and gives a description of the problems that we are interested in solving. Chapters 11–14 provides an overview of previous work in the area of derivative-free optimization as well as some theory. Direct search methods are presented in Chapter 11. The theory of model building algorithms with linear or quadratic models is presented in Chapter 12 and examples of such algorithms are given in Chapter 13. Radial Basis Functions are another kind of model building algorithms and are presented in Chapter 14.

Chapter 15 explains different ways in which parallelization of model building algorithms can be performed. How information can be shared between the now parallel parts of the algorithms, as well as how one can attempt to create synergy between them is described in Chapter 16. Control of optimization algorithms is discussed in Chapter 17. Section 18 gives details about our particular software implementation, including implementation of parallelization, as well as integration into the industrial simulator SKF BEAST.

Results from empirical testing on synthetic test cases and discussion about test results are available in Chapter 19, and results from testing on industrial test cases are available in Chapter 20. Some possible directions for future research are presented in Chapter 21. Finally, conclusions are presented in Chapter 22.

In Appendix A, we list the terms used in the thesis and is recommended that the reader first familiarizes himself with the terms before reading the rest of the thesis.

# 10

---

## Setting And Problem Description

Here we describe the problems under consideration and the consequences for the choice of optimization methods. In general, we are interested in solving advanced, typically non-linear, optimization problems that occur in real world applications. During this project, we have cooperated with several companies to get their input on what kinds of problems that they encounter.

### 10.1 Frontway

One of the companies that we have cooperated with is Frontway [25], which is a SME that specializes in consulting for the process industry, especially the pulp and paper industry. Among their products, Frontway has developed their own proprietary simulation software PaperFront<sup>©</sup> that is used to model quality, material- and energy consumption in a pulp and paper mill.

In PaperFront<sup>©</sup>, the infrastructure of pulp and paper mills with pumps, towers and chests are handled as well the recirculation and handling of many components in a flow stream. Up to 100 components can be handled, divided in solids, solubles etc. Input of raw material and output of the finished product is simulated and the simulation uses a model that the user creates through adding predefined blocks that represent the tanks, pumps, etc. The model is created via a CAD-like tool. The user can then connect the different blocks and set parameter values for them to simulate the flow of water, pulp and more. When the model has been specified, PaperFront<sup>©</sup> can be used simulate the consumption of water, use of electricity etc. for a specified production program. The software can be used to model a single production line or a complete plant with intricate connections between the different production lines. One thing that makes PaperFront<sup>©</sup> special is the connection to a relational database that allows the handling of multiple settings and evaluations.

A common application is to calculate the sizes of tanks, capacities of pumps and other components in a production line, given a production program, i.e. a list of quantities of products that must be produced within a certain time. Since the components (tanks, pumps, etc.) are expensive, it is important to select tank sizes and pump capacities that are sufficient for the intended production, including some margin of safety, but not excessively large since the cost would be prohibitive. So far, this has been done by choosing a tank size, pump capacity etc. and let PaperFront<sup>©</sup> determine whether the production program is possible to fulfill. If so, then the tank size and/or pump capacity is decreased until PaperFront<sup>©</sup> fails to find a way to produce the required quantities. The lowest tank volume and pump capacity that can be used while still fulfilling the production program, are interpreted as sufficient and a certain margin of safety is added to these to values to get the final values. The final values are then used when ordering the components that will be used. Naturally, as the number of tanks, pumps and other components increase, performing this testing by hand becomes cumbersome, and the user might also have difficulty doing a good job since all components have different costs.

## 10.2 SKF

Our main partner has been the Swedish company SKF [76]. SKF is a large multinational company that is probably best known for its bearings, but is also involved in lubrication, seals, mechatronics and more.

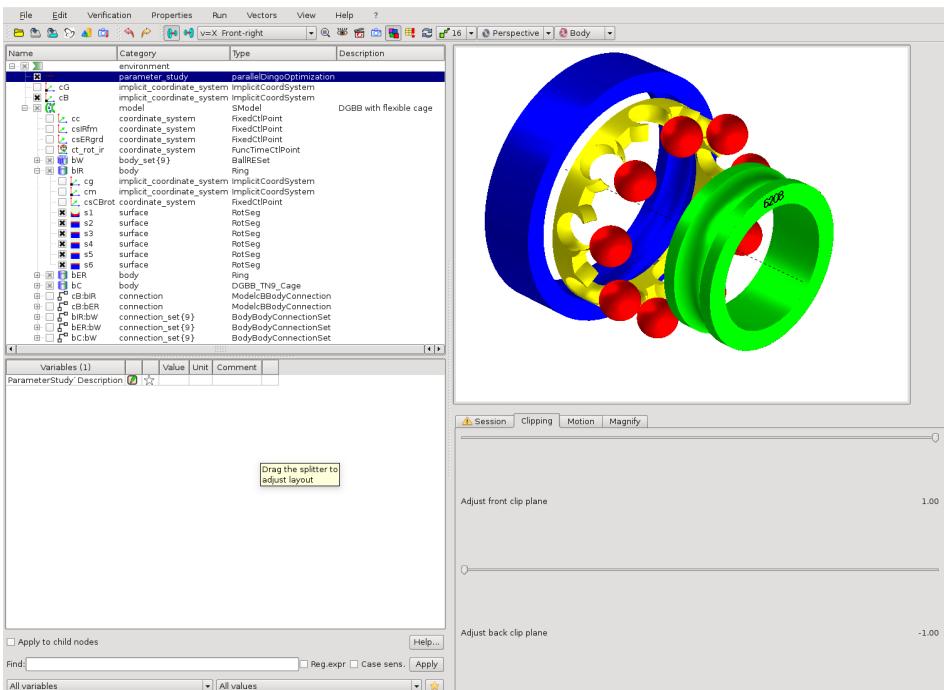
SKF develops a wide range of products, and to help with the development of products, the employees use a large variety of tools. Below we will briefly describe one of their most advanced tools.

### 10.2.1 SKF BEAST

To help the development of new products SKF has developed a simulation tool called BEAST (BEAring Simulation Tool) [24]. It is a virtual test rig for testing the dynamics of different kinds of mechanical products such as bearings and other machine elements. BEAST is specialized in detailed contact calculations, something that is very important for evaluation of bearings. Typically BEAST is used for dynamic simulations, where one or more moving objects are subject to one or more forces.

To specify a problem in BEAST, the user uses the accompanying tool Beauty. Figure 10.1 shows the Beauty user interface. Typically the user specifies the problem in several steps, and the first step is to specify the model. It is possible to import CAD models as well as to specify the model from smaller components. Throughout the process, the user can set parameters regarding the physical shape and dimensions, as well as lubrication parameters and much more.

Once the model has been specified, it is time to set up the specific test case by specifying forces, rotations, the time from standstill until all forces have been applied etc. The user can also control the simulation itself, e.g. setting the time step in the simulation, how



**Figure 10.1:** The Beauty user interface.

much information should be saved to log files and how often the data should be saved. The last thing to do is to run a verification of the model, to make sure that all required data has been set and that everything required running the simulation has been specified.

When the model and the simulation parameters have been completely specified, the user submits the problem to BEAST for simulation. Since dynamic multi-body contact simulations require a lot of computational power, the calculations are typically distributed over several computers. Once the simulation is completed, the result is returned to the user and he can then analyze the result, using the Beauty software.

Although the BEAST software is capable of simulating a great number of machine elements, for simplicity we will typically refer to the product as a rolling bearing in this thesis.

## 10.2.2 BEAST Workflow

With some knowledge about how a problem is specified, we will now present the engineer's workflow when using the BEAST tool chain for development and evaluation of new products. Although we use BEAST as an example of a tool, this sequence of work is descriptive of how many mechanical engineers work and the following description is not limited to how SKF engineers work.

The engineer working with Beauty and BEAST typically follows a certain workflow when investigating a new product. After setting up the problem according to the previous section, he often performs some initial tests to determine whether the product performs as expected. During the development and evaluation of a new product, he might do many iterations to achieve a product that performs satisfactorily.

The next step depends on what the engineer wants to investigate and he might not perform all of the following steps, but here we describe the complete process.

As an example, assume that the engineer wants to investigate what factors affect the expected life length of a bearing the most. In that case, a Design of Experiments (DoE) phase is typically performed next. The purpose of DoE is to gather data about how one or more factors/variables affect the output and then fit some simplified model to the data. A classic reference in the DOE area is Fisher [22] and a more modern book is by Montgomery [46]. In DoE, the variables are given different values and a simulation is run for each of the different values.

Several of the classical techniques for minimizing the influence of external factors on the results, such as replication, blocking and randomization, are unnecessary since BEAST is deterministic [51]. Each set of variable values correspond to one point in the design space. The initial points can of course be placed in many different ways and it is beneficial if the points are placed in such a way that they are space-filling. This means that they cover the search space efficiently and also capture unexpected events easier than other designs. There are many space-filling designs, e.g. Latin Hypercube, Sobol Sequence and maximin design [79].

Once all simulations required by the Design of Experiments have been performed, the information can be analyzed by the user, and he may also use the built-in tools in Beauty. Typically some of the parameters are more important than others, as will be visible from the results of the simulations. To further investigate the effects of these variables and to sort out the interactions between them, a full factorial design can be performed on the most important variables.

In a full factorial design, the user chooses a value in addition to the standard value and then all combinations of variable values are tested, for a total of  $2^n$  tests. This yields more information and it might be the case that some variable was not as important as previously believed. If the user considers such variables as not important enough, they can be removed. After this stage there are typically only a small number of variables left.

The last thing to do is to perform optimization of the remaining variables within each variable's range. The optimization algorithm that is implemented in BEAST is the Nelder-Mead Simplex algorithm (see Section 11.2). The goal of the optimization is to find the combination of variable values that yields the best objective function value. When optimization has been performed for one set of variable values, it is common to use these new values and iterate and try different experiments settings, i.e. vary the applied forces and so on. The reason for this is to see if the behavior of the product is satisfactory in a variety of situations and not only in the previously tested cases. This might lead to new constraints on the variables, if the new simulations show that the product performs poorly

in some case.

Of course, testing all possible cases that could occur during real world use would be too time-consuming, so the engineer must use his experience and judgment to determine some representative cases and perform simulations of those. Once the optimization has been performed to satisfactory results, this part of the design phase is finished.

## 10.3 Distinguishing Problem Characteristics

We performed interviews with some BEAST users in order to get more information about how they work and also what their wishes were. The interviews and discussions with the BEAST team and Frontway employees led to the following compilation of consequences for the optimization problem and thus for the algorithm, that must be considered. While some of the terminology is BEAST specific, many other simulators have similar terms and work in a similar way.

- In both simulators, each function evaluation is the result of a simulation run. In BEAST, each simulation simulates the bearing for a specified time period. The bearing starts from a standstill at  $t = 0$ , forces are applied and the rotational speed increases until it has reached the desired rotational speed. The time for this is specified by the user and once it has passed, the simulation continues for a predetermined time, to allow the bearing to reach a steady state. Once this time has passed, the simulation is finished and the result is returned to the user.

During a BEAST simulation, thousands of differential equations are solved in order to determine the forces between components, component locations, any vibrations and so on. Of course, the equations also affect each other over time and together this means that there is no explicit objective function that can be derived in order to get information about the gradient and Hessian. Since we have very limited knowledge about what goes on inside the BEAST simulation, the objective function is essentially a black box. As the simulation typically is performed with a very fine discretization, both in the time domain and in the meshes that are used to represent the components in the simulation, each function evaluation can be very time consuming. The exact calculation time also depends on the complexity of the bearing as well as the simulation settings that the user can change in order to investigate different things. It is not uncommon for a function evaluation to take several hours.

In PaperFront<sup>©</sup>, due to the complicated recirculation of water, steam, and other materials between within the production line and between different production lines, we cannot extract a single objective function, in order to apply common optimization algorithms. Thus we treat the calculations in PaperFront<sup>©</sup> as a black box, just like in BEAST.

Different ways to fulfill the production program are calculated in PaperFront<sup>©</sup>, and it will typically take quite some time until a valid production program has been

found. For a single production line, this is often around 30-45 minutes.

- When designing a mechanical product such as a rolling bearing, there is not a single objective function that determines whether a product is good. Instead there are many different predefined *performance criteria* that together decide the merit of the product. In BEAST, the user can choose whether to optimize a single or several of these criteria, and he can also define his own criterion. If a single predefined criterion is used, then its value remains as it is. If several criteria are used or if the user defines his own, then this is handled by creating a single objective function as a weighted product of the used criteria. Each performance criteria is normalized to [0,1] using minimum and maximum required values, which must be set by the user. The minimum required value must be exceeded for the product to be considered valid. The maximum required function value is a value that is judged to be sufficiently good; no effort is made to improve a performance criterion that meets or exceeds its maximum required value.

To get the final objective function value, the normalized performance criteria values are multiplied with each other. The effect of creating an objective function in this way is that if any of the performance criteria does not reach its minimum required value, then the objective function value becomes zero. If all performance criteria meets or exceeds their maximum values, then the objective function value becomes one. Thus, the objective function value of a valid product is in the interval (0, 1], where higher is better.

If some predefined performance criterion is used, then the acceptable values may be either large negative or large positive, depending on the criterion. If the user defines his own criterion, the range is typically (0, 1].

Since some of these are to be minimized and some are to be maximized, the user marks whether each criterion shall be minimized or maximized. For each criterion that shall be maximized, it is done so according to  $\max = -\min$ , since the algorithm performs minimization only. Therefore, the final problem is a minimization problem.

When using PaperFront<sup>©</sup> for the applications discussed here, the objective function is to minimize the cost for the used components.

- Since the optimization in BEAST is performed at the end of the design process, there are typically few variables in the optimization problem. There are almost never more than 10 variables and there are fewer than five in most cases.

In the case with Frontway, the number of variables is the same as the number of tanks, pumps etc. combined. For the cases that have been discussed, the number of variables are about the same as in BEAST.

- The user typically has specified a range for each variable. Some of these are physical in nature, such as the size of a component cannot be negative, while others may

be limitations set by the user, representing his knowledge that a certain variable must be within a certain interval in order for the final product to perform satisfactory. Also, the user has the possibility to specify arbitrary constraints, which must be fulfilled. In the BEAST simulator, all kinds of constraints are handled by penalty functions<sup>1</sup>. This means that if a point does not fulfill all requirements, then it is assigned a constant, poor value. This will typically cause a performance criterion to not meet its minimal required value and thus will the objective function get a zero value.

The effect for the optimization is that there are typically constraints in the problem, both bounds and arbitrary constraints, but that they are all handled by penalty functions. Also, a point that violates several constraints will get the same value as a point that violates a single constraint. A problem with this way of handling constraints is that since all points that does not satisfy all the constraints are assigned the same value, we cannot determine how much a point violates a constraint, or how many constraints that are violated. In effect, this merges the objective function and the constraint handling into a black box.

Since no optimization has been used previously in PaperFront<sup>©</sup>, there is no constraint handling. Previously, if no way to fulfill the required production program was found by PaperFront<sup>©</sup>, then the indata values was considered invalid and therefore ignored by the user.

- SKF has a large number of computers allocated to BEAST users since the objective function is time-consuming to evaluate and it is common to do parameter studies using e.g. full factorial tests, where each simulation can be performed in parallel. The user is able to request a number of computers to perform the calculations, and if possible the request is fulfilled. Also, the user can request that each simulation is distributed over a certain number of computers. Since there is a large amount of computational power available, the optimization algorithm should try to take advantage of this via parallelization.

While Frontway is a considerably smaller company, they still have access to a fair number of computers that can be used to calculations. For this reason, parallelization is important in their case also.

- The calculation time for performing a simulation may vary considerably, even for the same product. This is because some combinations of variables may trigger vibrations or similar events, which require longer calculation times. However, if the objective function is evaluated in points that are close to each other, the time required for each evaluation will typically be approximately the same.

---

<sup>1</sup>Although the functions are referred to as penalty functions, they do not conform to the common definition of a penalty function used in the optimization community since they are discontinuous at the edge of the feasible region (see Section 18.7.2). Nonetheless, we will use the penalty function term here.

- Since the final product is mechanical with finite precision, it is desirable to find an optimum that is robust to small manufacturing imperfections. Even though the production process is able to maintain very small tolerances for large series of products, it is always good to have some margin for errors. This together with the fact that the final objective function is typically multimodal and might contain high-frequency information, may encourage exploration of large parts of the search space.

Motivated by the previously described problem characteristics, we assume that  $f$  and its derivatives are unavailable and we will treat the objective function as a black box. Since any constraints specified by the user are handled by penalty functions, this leads to the following unconstrained optimization problem:

$$\begin{aligned} & \min f(x) \\ & x \in R^n \\ & \nabla f \text{ unavailable} \end{aligned}$$

In the remainder of this part of the thesis we will focus on derivative-free algorithms for solving the above optimization problem that are suitable for parallelization.

### 10.4 Summary

We are interested in solving advanced non-linear optimization problems in an industrial setting. Such problems often occur when a simulator, e.g. the BEAST or PaperFront<sup>©</sup>, is used to simulate and test different kinds of products, without reducing the production or the availability of the production line. The problems and the environment that we are interested in have some distinguishing characteristics: each evaluation of the objective function is a simulation run, which typically takes a long time. Since the simulated equations are very complicated, it would be extremely difficult, if at all possible, to extract an explicit objective function. Thus no derivative information is available. In most of these problems, the number of variables is very small, since optimization is performed at the end of the design process where many variables have been fixed. Since constraints are handled internally by penalty functions, we have an unconstrained optimization problem.

The fact that the objective function is time-consuming to evaluate and that gradients are unavailable, limit what kinds of algorithms that are suitable. Since there is a large amount of computational power available, we will in the rest of this thesis focus on algorithms for derivative-free optimization that can be parallelized, in order to use the available computational power effectively.

# 11

---

## Direct Search Algorithms

Algorithms for derivative-free optimizations can be broadly divided into two categories: algorithms that do not build a model of  $f$  and model building algorithms that build a model of  $f$ .

This and the next few chapters will describe both types of algorithms. In this chapter non-model building algorithms, also called direct search algorithms, are described. Typically, this kind of algorithms require little more than series of evaluation of  $f$ . Several of the algorithms presented here are based on the Nelder-Mead simplex algorithm (see Section 11.2) and as such they can benefit from tuning the parameters used in that algorithm. Naturally, the parameters affect the performance of the algorithm, but finding good values of the comparatively few parameters in the Nelder-Mead Simplex algorithm is in many cases less cumbersome than determining the values of parameters for the model building algorithms described in Chapter 12.

### 11.1 Initial Step Length and Termination Criteria

For all optimization algorithms and especially for the algorithms discussed in this section, the termination criteria and the initial step length must be determined.

#### 11.1.1 Initial Step Length

The initial step length is not only used in the first step, but is in some cases used throughout the execution of the algorithm. In other cases, the step lengths in subsequent iterations are a fraction of the initial step length. Therefore it can be very important to choose a suitable step length: a too short step length can cause many unnecessary evaluations

of the objective function and a too long step length may cause the algorithm to bypass regions with good objective function values.

### 11.1.2 Termination Criteria

Typically the algorithm is allowed to run for a certain number of evaluations, but other choices are of course possible. If the authors have pointed out that they use a certain termination criteria, this is discussed in the corresponding algorithm's section.

In the pseudocode of the algorithms in this thesis, we often abstract the different termination criteria in a function `Termination-Criteria` that returns true if the used termination criterion is fulfilled, otherwise it returns false.

## 11.2 Nelder-Mead Simplex

The Nelder-Mead Simplex algorithm is a well-known algorithm for optimization of non-linear functions [50]. The algorithm takes its name from the two authors and the fact that it uses a simplex, formed by the points in the algorithm. A simplex in  $n$ -space is the convex hull of  $n + 1$  linearly independent points. Thus in 2-space it can be represented by a triangle, in 3-space by a tetrahedron etc. The points that make up the simplex must be linearly independent, otherwise the simplex collapses into a smaller subspace, i.e. it loses (at least) one dimension.

The Nelder-Mead Simplex is a direct-search method that evaluates the objective function in one point in each iteration. Depending on the objective function value in that point and the objective function values in the simplex's other points, an action is decided upon. Thus the algorithm does not require any information about derivatives, nor does it build a model of the objective function. The fact that it requires so little information makes it easy to apply to a wide variety of problems.

When contrasted with other derivative-free optimization methods, it is apparent that the Simplex method and its successors focus on replacing the worst point rather than improving the best point.

### 11.2.1 Nelder-Mead Simplex Pseudocode

The pseudocode for the basic Nelder-Mead Simplex algorithm is available in Figure 11.1. On line 1, let  $Y$  consist of  $n + 1$  linearly independent points and use these points to create an initial simplex. Lines 2–6 form the main part of the algorithm, that is repeated until execution is aborted, see Section 11.2.2 for different options for the termination criteria. The first thing to do in each iteration is to find the point  $x_k$  that has the worst objective function value (line 3). Once this is done, the centroid  $\bar{x}$  of the remaining points is determined on line 4. Then the direction between  $x_k$  and the centroid  $\bar{x}$  is calculated (line 5). The final thing to do, on line 6, is to replace the  $x_k$  by the reflected *candidate*

- 1 Create an initial simplex with  $n + 1$  points.
- 2 **while** Termination-criteria **not met**
- 3      $x_k \leftarrow \operatorname{argmax} f(x_i) x_i \in Y$
- 4      $\bar{x} \leftarrow \frac{1}{n+1} \sum_{i \in Y \setminus \{x_k\}} x_i$
- 5      $\Delta x \leftarrow x_k - \bar{x}$
- 6     Replace  $x_k$  in  $Y$  by  $x_k^+ \leftarrow \bar{x} + \Delta x$

**Figure 11.1:** Nelder-Mead Simplex algorithm.

point  $x_k^+ \leftarrow \bar{x} + \Delta x$ . Formally:  $Y \leftarrow Y \setminus \{x_k\}$ ,  $Y \leftarrow Y \cup \{x_k^+\}$ . Then a new iteration starts at line 2.

### 11.2.2 Termination Criteria

The function Termination-criteria can be defined in many different ways. Nelder and Mead terminate the algorithm when the standard deviation of the simplex values falls within a certain tolerance  $f_{tol}$ :  $\sqrt{\frac{1}{n} \sum_{i=0}^n (f(x_i) - \bar{f})^2} < f_{tol}$ , where  $\bar{f} = \frac{1}{n+1} \sum_{i=0}^n f(x_i)$ . Another possible termination test is  $(f_h - f_l) / (1 + |f_l|) < f_{tol}$ , where  $f_h$  is the point with the highest objective function value and  $f_l$  is the point with the lowest function value, as suggested by Gill et al. [28].

### 11.2.3 Improvements

In the basic algorithm above, the simplex never changes size, it only moves on the objective function surface. The inability to take longer or shorter steps may lead to unnecessary slow convergence. However, there have been many different suggestions for improving this, and some are listed below:

1. If the new point  $x_k^+$  is the best point, double the step length:  
 $x_k^+ \leftarrow \bar{x} + 2\Delta x$  (expansion).
2. If  $x_k^+$  is not better than the second worst point, half the step length:  
 $x_k^+ \leftarrow \bar{x} + 0.5\Delta x$  (shrinkage).
3. If  $x_k^+$  is the worst of all points, half the step direction and reverse the step direction:  
 $x_k^+ \leftarrow \bar{x} - 0.5\Delta x$  (contraction).
4. If all previous improvements fail, let the best point remain and move all other points towards it.

It should be noted that there is little theory behind these modification, instead they are rules of thumb that are believed, but not guaranteed, to improve convergence towards

an optimum. For example, there is no theory behind the doubling or halving of the step length. For monotonous functions, these values could be considered as a form of binary search, but it must be remembered that the functions of interest here are unlikely to be monotonous.

The Nelder-Mead Simplex algorithm requires memory in the order of  $O(n^2)$  as there are  $n + 1$  points, with each point being an  $n$ -vector.

### 11.2.4 Extensions

The Complex (Constrained Simplex) is a further development of the Simplex algorithm that allows the optimization of objective functions subject to constraints [8]. The Complex algorithm works in a similar manner as the Simplex algorithm, but there are some differences. At least one point, that is guaranteed to be feasible, must be given by the user. The other points are created through randomization. Within the bounds on each coordinate value, the value is randomized with an equal probability for all values. As the points are created in this way, they are guaranteed to satisfy the bound constraints, but they are not guaranteed to satisfy any other constraints. If a point does not satisfy a constraint, it is moved halfway towards the centroid of the remaining points. The process of moving the point continues until a feasible position has been found.

If, at any time during execution, a point does not satisfy a bound constraint, then the point is moved into the feasible region, and placed a small distance  $\delta$ , e.g.  $10^{-6}$  from the violated constraint. Placing the point inside the feasible region by a small distance has been questioned by other researchers, who instead place the point exactly on the constraint. Nonetheless, this method for handling constraints has been used in other algorithms, for example in the NLOpt optimization library [37].

## 11.3 Subplex

The Subplex (Subspace-searching simplex) method is designed especially for problems with high-dimensionality [63]. It decomposes the problem into several low-dimensional subspaces, and then applies the Nelder-Mead Simplex method (see Section 11.2) in each one of these.

In addition to the variables used in the Simplex algorithm, it adds several new variables:  $\psi$  is used in the Simplex algorithm to control the accuracy of the subspace searches,  $\omega$  is used for adjusting the scale of the step sizes and  $ns_{min}$  and  $ns_{max}$  determine the minimum and maximum subspace dimensionality, respectively. The variables must satisfy  $0 < \psi < 1$  and  $0 < \omega < 1$ . The variables  $ns_{min}$  and  $ns_{max}$  must fulfill  $1 \leq ns_{min} \leq ns_{max} \leq n$  and  $ns_{min} \cdot \lceil n/ns_{max} \rceil \leq n$ .

### 11.3.1 Subspaces

The rationale behind the Subplex algorithm is the claimed inefficiency of the Simplex algorithm when applied to high-dimensional problems. Therefore Subplex divides the search space into a number of subspaces, where each subspace should have a low dimensionality to mitigate this inefficiency. However, it must also be determined which variables that should make up each subspace, which is discussed further below.

The number of subspaces is denoted by  $n_{subs}$ . Naturally, the dimensionality of each subspace,  $ns_1, ns_2, \dots, ns_{subs}$  must satisfy  $ns_{min} \leq ns_i \leq ns_{max} \forall i$ .  $\sum_{i=1}^{n_{subs}} ns_i = n$  must also apply.

To determine the subspaces, the *progress vector*  $\Delta x$  is used. The progress vector is a vector of  $n$  scalars, one for each variable. Each value in  $\Delta x$  measures the difference in objective function value caused by the corresponding variable. While the word gradient is not used, the description of the progress vector sounds very much like a gradient. To group the variables into subspaces,  $\Delta x$  is first sorted in order of decreasing values. The vector is then searched from beginning to end for gaps in the magnitude of the values. The values in  $\Delta x$  that are similar in size are grouped together to form a subvector, as long as the dimensional constraints are satisfied. Each subvector forms one subspace and the Simplex algorithm is then applied in each subspace.

The original creator of the algorithm believes that there will be significant gaps in magnitude of function values and that most of these values with the same magnitude can be fitted into one subspace. Also, it is believed that most of the improvement in the objective function will occur in the subspace with the largest value of the function values.

Typically,  $ns_{min} \leftarrow \min(2, n)$  and  $ns_{max} \leftarrow \min(5, n)$  are used. Just like for the variables in the Simplex algorithm, no theoretical justification of these numbers is presented.

### 11.3.2 Step Length

The step length determines the scale and orientation of the initial simplexes used in the Simplex algorithm. As execution progresses, the step length is changed, depending on the values in the progress vector  $\Delta x$ . If the values in  $\Delta x$  are small, then the step length is decreased. On the other hand, if the values in  $\Delta x$  are large, then the step length is increased in order to take larger steps towards better objective function values.

The orientation of each simplex is also determined when the step lengths are determined. The idea is to use information in the progress vector to determine favorable directions and then the step directions are changed as necessary.

### 11.3.3 Termination Criteria

The Subplex algorithm uses two termination criteria: one for each execution of the Simplex algorithms (inner criterion) and one for the Subplex algorithm itself (outer criterion).

For the inner criterion, none of the termination criteria suggested for the basic Simplex algorithm is used. Instead, it uses a criterion based on the distance between the simplex vertices. This is measured using the distance between the points giving the best and worst objective function values. When this distance has been sufficiently reduced, the Simplex algorithm terminates.

For determining when the Subplex algorithm itself should terminate, the outer termination test uses the closeness of successive  $x$  iterates and the step variable,  $step$ . The formula for determining termination is  $\frac{\max(\|\Delta x\|_\infty, \|step\|_\infty \cdot \psi)}{\max(\|x\|_\infty, 1)} \leq f_{tol}$ .

### 11.3.4 Handling of Measurement Errors

In some cases, objective function values are subject to measurement errors or noise. Such errors can lead to rejection of good points or inclusion of bad points into a simplex. This is problematic for functions based on the movement of a simplex, as the movement is based on the function values of all simplex vertices. Thus it is important that the function values are correct, otherwise the simplex might move in the wrong direction. Another problem is that the simplex-based methods can converge to a false extremum in case there is a large measurement error in the function value.

An obvious solution is to perform several function value evaluations for each vertex position, and for example take the average value of these. However, in high-dimensional problems, this may require many evaluations, and if each evaluation can take considerable time, this might not be feasible. The Subplex method mitigates the problem of measurement error by repeatedly performing objective function evaluations at the currently best vertex, each time a new simplex is generated. Over time, this creates a set of function values for the same point. Then, the minimum, maximum or mean of these values is used, depending on the nature of the measurement error. Though somewhat expensive, this procedure is said to decrease the risk of getting stuck at false extrema.

### 11.3.5 Subplex Pseudocode

A very high-level pseudocode of the Subplex method is available in Figure 11.2.

```
1 while Termination-criteria not met
2     Set step sizes
3     Set subspaces
4     for each subspace do
5         Execute Nelder-Mead Simplex
```

**Figure 11.2:** The Subplex algorithm. Adapted from [63].

## 11.4 The r-Algorithm

The r-algorithm was developed by Naum Shor and first published in Russian [74] and later described in English [73]. The r-algorithm takes its name from the Russian word “raznost”, which means difference. This difference is the difference<sup>1</sup> between two successive subgradients, and is used to calculate space dilation along this direction.

The algorithm requires that  $f$  is an “almost” differentiable, convex function on  $R^n$ , i.e.  $f$  is differentiable on its domain except on a set of measure zero. Despite this requirement the algorithm has performed well when executed on nonconvex problems of dimensionality less than 20 [41].

An arbitrary subgradient of  $f$  at  $x_k$  is denoted by  $g_f(x_k)$ .  $B_k$  denotes the space transformation matrix in iteration  $k$ . The space dilation coefficient is denoted by  $\alpha$ . The algorithm also uses a transformed function  $\phi_k(y) = f(B_k y)$  and the subgradient of that function at the point  $\tilde{y}_k = B_k^{-1}x_{k-1}$  is denoted by  $\tilde{g}_k$ . The step in iteration  $k$  is denoted by  $h_k$  and the ideal step in iteration  $k$  is denoted by  $h_k^*$ .

Let  $\beta = 1/\alpha$ . Then, when  $\beta = 0$  and  $h_k = h_k^*$ , the algorithm is a variant of the conjugate gradient method and when  $\beta = 1$ , the algorithm is equal to the steepest descent method. Thus, by changing the value of  $\beta$ , it is possible to achieve a method that is somewhere between the steepest descent and conjugated gradient method.

In the description provided by Shor [73], the algorithm also includes changing the value of  $\beta$ , although no discussion is provided of how or when this should be performed. Similarly, it is mentioned that the algorithm benefits from periodical restarting, through resetting of the space transformation matrix. Restarting is discussed further by Kappel and Kantsevich [40].

### 11.4.1 r-Algorithm Pseudocode

The pseudocode in Figure 11.3 is a description of the r-algorithm and the line numbers refer to that figure. Before the algorithm starts, the following must be set: the starting point  $x_0$ , the first step  $h_1$ ,  $k \leftarrow 1$ ,  $\alpha > 1$ , and  $B_1 \leftarrow I$  (the unit matrix of size  $n \times n$ ). Then set  $\tilde{g}_1 \leftarrow g_f(x_0)$ . From the subgradient and the starting point, the first point  $x_1$  is calculated as  $x_1 \leftarrow x_0 - h_0 g_f(x_0)$ .

Lines 1–11 are performed as long as the termination criterion is not met. The termination criterion (line 1) is discussed further in Section 11.4.2. On line 2, the subgradient of the current point is calculated. Line 3 calculates the subgradient of the function  $\phi$  at the point  $y_k = B_k^{-1}x_k$ , where  $\phi(y) = f(B_k y)$ . The difference,  $r_k$ , between the subgradient calculated on line 3 and the previous subgradient  $\tilde{g}_k$  is then calculated on the next line. Then, line 5 normalizes this difference.

On line 6, the matrix  $R_\alpha$  is updated using the normalized difference between the subgradients calculated on the previous line. The inverse of the updated matrix is then

<sup>1</sup>The original algorithm used the subgradient, not the *difference* between subgradients, to determine the space dilation. The latter version replaces the former and is the one described here.

---

```

1 while Termination-criteria not met do
2   Calculate  $g_f(x_k)$ 
3   Calculate  $g_k^* \leftarrow B_k^T g_f(x_k)$ 
4    $r_k \leftarrow g_k^* - \tilde{g}_k$ 
5    $\xi_{k+1} \leftarrow r_k / \|r_k\|$ 
6    $R_\alpha \leftarrow I + (\alpha - 1)\xi_{k+1}\xi_{k+1}^T$ 
7    $B_{k+1} \leftarrow B_k^T R_\alpha^{-1}$ 
8    $\tilde{g}_{k+1} \leftarrow B_{k+1}^T g_f(x_k)$ 
9   Calculate  $h_{k+1}$ 
10   $x_{k+1} \leftarrow x_k - h_{k+1} B_{k+1} \tilde{g}_{k+1}$ 
11   $k \leftarrow k + 1$ 

```

**Figure 11.3:** The  $r$ -algorithm by Shor.

used on line 7 where the space transformation matrix  $B_k$  is updated.

Line 8 calculates a new subgradient of the function  $\phi_{k+1}(y) = f(B_{k+1}y)$  at the point  $\tilde{y}_{k+1} = B_{k+1}^{-1}x_k$ . On line 9, the next step  $h_{k+1}$  is calculated. This step must be at least as long as the ideal step, i.e.  $\|h_{k+1}\| \geq \|h_{k+1}^*\|$ .

To get the next position  $x_{k+1}$ , the new step is used together with the updated transformation matrix and the new subgradient (line 10). The last thing to do in this iteration is to update the iteration counter.

From the pseudocode we can see that the difference between the subgradients,  $r_k$ , is the difference between the subgradients of the function  $\phi_k$  and that it is used to update the transformation matrix  $B$ . The new position is calculated using the subgradient of  $f$ , the updated transformation matrix and the step  $h_{k+1}$ .

### 11.4.2 Termination Criteria

The termination criteria  $g_f(x_k) = 0$  is suggested by Shor [72], while Kappel and Kantsevich [40] suggest that the algorithm is terminated when both  $|x_{k+1}^i - x_k^i| \leq \delta_x |x_{k+1}^i|$ ,  $i = 1, \dots, n$  and  $|f(x_{k+1}) - f(x_k)| \leq \delta_f |f(x_{k+1})|$  hold, for some parameters  $\delta_x$  and  $\delta_f$ .

## 11.5 Summary

Direct search algorithms evaluate the objective function directly, and attempts to find points with better objective function values using only the current points and their values. The Nelder-Mead Simplex method is probably the most well-known method. It uses a simplex that moves on the objective function surface, and the shape of the simplex changes during the execution, depending on the improvement in the previous iteration.

The Complex algorithm is an extension of the Nelder-Mead Simplex algorithm, and is able to handle constraints.

Subplex is another extension of Nelder-Mead's algorithm and is intended for use in high-dimensional spaces. The main idea is to divide the search space into several lower-dimensional spaces, and apply the Nelder-Mead algorithm in each one, in order to overcome the fact that the Nelder-Mead Simplex method performs poorly in such spaces.

The r-algorithm uses subgradients and space transformations in order to find points with better objective function values. The space transformation changes in each iteration, depending on the difference between the subgradient in the new position and subgradient in the previous position. Despite the requirement that  $f$  must be convex, the r-algorithm has performed well on nonconvex problems.

In this chapter we have presented some of the many direct search algorithms, and there are many other alternatives, such as pattern search [34], the DIRECT algorithm [38] as well as meta-heuristics such as simulated annealing [42], tabu search [29, 30] and genetic algorithms [45]. Such algorithms are other options for the kind of problems that we are interested in solving, but we will not treat them here.



# 12

---

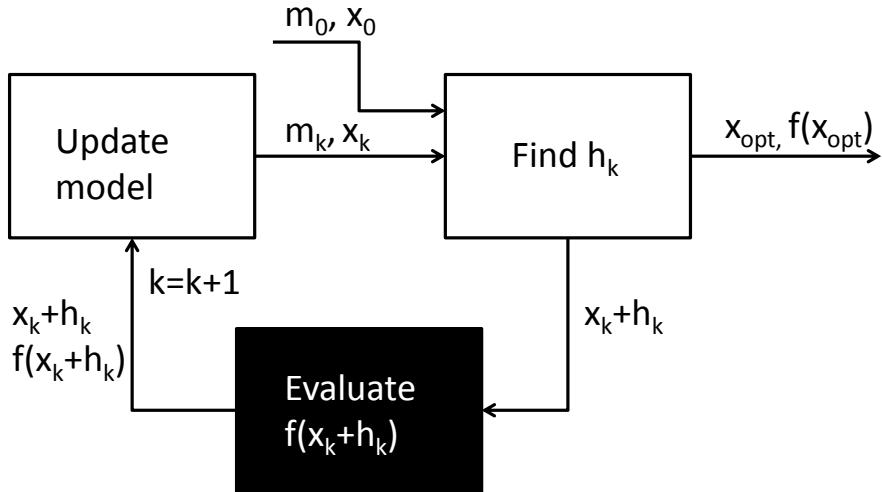
## Model Building Algorithms

Instead of evaluating  $f$  directly, a commonly used method is to build a model, i.e. a function, denoted by  $m$ , which approximates  $f$  in the region of interest. The idea is to let  $m$  approximate  $f$ , while still being a mathematically “simple” function that is constructed to be considerably less computationally demanding to evaluate than  $f$ . It is also easier to find a minimum of  $m$  since the gradient and Hessian are readily available. Thus it should be possible to perform a greater number of evaluations of  $m$  in order to find a point in which  $m$  has a low function value. Once that point is determined,  $f$  is evaluated at that point. Since most of the work is performed on  $m$ , this should lead to fewer evaluations of  $f$  and thus to a total performance improvement.

Model building algorithms have two very important components: one is the model itself and how it is updated and the other is the optimization algorithm used to find the best value of the model. The model typically changes during execution of the algorithm, therefore it is often subscripted with the iteration number, i.e. the model in iteration  $k$  is denoted by  $m_k$ .

Figure 12.1 shows a high-level overview of the two main components, as well as the black-box objective function. The part of the algorithm that minimizes the model takes as input a model  $m_k$  and a point  $x_k$ , from which the optimization begins. The output is a step  $h_k$ , relative to  $x_k$  and the function is then evaluated in the point  $x_k + h_k$ . The iteration variable  $k$  is increased, and the objective function value and the point are given as input to a function that updates the model. The output is an updated model, which together with the new point is used as input to the optimization algorithm. This cycle continues until the algorithm eventually terminates, and the best point  $x_{opt}$  and objective function value  $f(x_{opt})$  are returned to the user.

The minimization of the model is typically performed within a small region centered in  $x_k$ . The model is a local model that interpolates a certain number of points and it can



**Figure 12.1:** High-level overview of the parts in a model building algorithm for derivative-free optimization.

only be trusted within a small region, hence the name *trust region*. Within this region  $m_k$  is minimized, with the goal of finding a point to evaluate  $f$  in. The model, and the algorithm that minimizes  $m_k$  within the trust region are the subjects of interest in the next few sections. We will begin with some theory behind the interpolation before describing the commonly used Lagrange interpolating polynomials in more detail.

## 12.1 Building the Interpolation Model

The model is built using a set of points  $Y = \{y_0, \dots, y_p\}$ . In each of these points,  $f$  has been evaluated and  $m$  is fitted to be equal to  $f$  in all points  $y_i \in Y$ :

$$m(y_i) = f(y_i), \forall y_i \in Y \quad (12.1)$$

We say that  $m$  *interpolates*  $f$  if Equation 12.1 holds and that  $Y$  is the *interpolation set*.

Here, it is assumed that it is desirable to have  $m$  interpolate  $f$  exactly, which may not be the case if the objective function is e.g. non-deterministic. For a short discussion of how this situation can be handled, the reader is referred to Section 12.1.6.

### 12.1.1 Polynomial Bases

Models of different fidelity can be used to approximate  $f$ . A simple method is to use a first degree (linear) polynomial and more advanced modeling option is to use a second or higher degree polynomial. A higher order polynomial should provide a better estimate of

$f$  but requires more interpolation points and thus requires a greater number of function evaluations to create.

The most common choices for interpolating functions are first and second order polynomials. A linear model requires  $n+1$  interpolation points and a quadratic model requires  $\frac{1}{2}(n+1)(n+2)$  points. If  $n$  is high, this is a potential disadvantage of quadratic models, as a greater number of objective function evaluations must be performed before any progress can be made towards finding a better objective function value. However, there are algorithms that attempt to mitigate this problem, see Sections 13.1 and 13.3.

The interpolating polynomial is typically based on a Taylor approximation around the point  $x$ . The most common polynomial is the second-degree Taylor polynomial:

$$m(x+h) = f(x) + h^T g + \frac{1}{2} h^T H h \quad (12.2)$$

where  $g$  is the model's gradient,  $H$  is the model's Hessian and  $x, h, g \in R^n$  and  $H \in R^{n \times n}$ . The coefficients of  $g$  and  $H$  are determined by the interpolation conditions to be described in the following sections.

Consider  $P_n^d$ , the space of polynomials of degree less than or equal to  $d$  in  $R^n$ . Let  $p_1 = n+1$  be the dimension of this space. For  $d=1, p_1=2$  and for  $d=2, p_1=(n+1)(n+2)/2$ . Then, a basis  $\phi = \{\phi_0(x), \phi_1(x), \dots, \phi_p(x)\}$  of  $P_n^d$  is set of  $p_1$  polynomials of degree less than or equal to  $d$ , that spans  $P_n^d$ .

Since  $\phi$  is a basis in  $P_n^d$ , any polynomial  $m(x) \in P_n^d$  can be written as  $m(x) = \sum_{j=0}^p \alpha_j \phi_j(x)$ , for some real-valued coefficients  $\alpha_j$ .

### 12.1.2 Natural Basis

One of many possible bases is the natural basis  $\bar{\phi}$ . Let the vector  $\alpha^i = (\alpha_1^i, \dots, \alpha_n^i) \in N_0^n$ . Then, the elements of the natural basis are defined as

$$\bar{\phi} = \frac{x^{\alpha^i}}{(\alpha^i)!}, i = 0, \dots, p, |\alpha^i| \leq d$$

with

$$x^{\alpha^i} = \prod_{j=1}^n x_j^{\alpha_j^i}, (\alpha^i)! = \prod_{j=1}^n (\alpha_j^i!), |\alpha^i| = \sum_{j=1}^n \alpha_j^i$$

for any  $x \in R^n$ .

The natural basis can be written as

$\bar{\phi} = \{1, x_1, x_2, \dots, x_n, x_1^2/2, x_1 x_2, \dots, x_{n-1}^{d-1} x_n / (d-1)!, x_n^d / d!\}$ . As an example,  $n=3$  and  $d=2$  yields  $\bar{\phi} = \{1, x_1, x_2, x_3, x_1^2/2, x_1 x_2, x_2^2/2, x_1 x_3, x_2 x_3, x_3^2/2, \}$ .

### 12.1.3 Polynomial Interpolation

Assume that we are given a set of interpolation points  $Y$  and that  $m$  denotes a polynomial of degree less than or equal to  $d$ . Also assume that  $m$  interpolates  $f$  at the points in

$Y$ . Then, by determining the coefficients  $\alpha_0, \alpha_1, \dots, \alpha_p$ , we determine the interpolation polynomials that make up  $m(x)$ . The values of the coefficients can be determined from the interpolation conditions

$$f(y_i) = m(y_i) = \sum_{j=0}^p \alpha_j \phi_j(y_i) \quad i = 0, \dots, p. \quad (12.3)$$

Together, the conditions in Equation 12.3 form a linear system in terms of the interpolation coefficients, which can be written in matrix form as

$$M(\phi, Y) \alpha_\phi = f(Y) \quad (12.4)$$

where

$$M(\phi, Y) = \begin{pmatrix} \phi_0(y_0) & \phi_1(y_0) & \cdots & \phi_p(y_0) \\ \phi_0(y_1) & \phi_1(y_1) & \cdots & \phi_p(y_1) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_0(y_p) & \phi_1(y_p) & \cdots & \phi_p(y_p) \end{pmatrix},$$

$$\alpha_\phi = \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \vdots \\ \alpha_p \end{pmatrix}, \quad f(Y) = \begin{pmatrix} f(y_0) \\ f(y_1) \\ \vdots \\ f(y_p) \end{pmatrix}$$

#### 12.1.4 Lagrange Polynomials

For derivative-free optimization, the most common choice used to calculate the interpolation polynomial is the Lagrange polynomial. Assuming a set of interpolation points  $Y = \{y_0, \dots, y_p\}$ , a basis of  $p_1 = p + 1$  polynomials  $L_j(x), j = 0, \dots, p$  in  $P_n^d$  is a Lagrange basis if

$$L_j(y_i) = \delta_{ij} = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{if } i \neq j. \end{cases} \quad (12.5)$$

where  $\delta$  is the Kronecker delta.

If  $Y$  is poised (see Section 12.2), then the basis of Lagrange polynomial exists and is unique [17].

Lagrange polynomials can also be written in a manner similar to Equation 12.3. For any function  $f : R^n \rightarrow R$  and any poised set  $Y = \{y_0, \dots, y_p\} \subset R^n$ , the unique polynomial  $m(x)$  that interpolates  $f(x)$  on  $Y$  can be expressed as

$$m(x) = \sum_{i=0}^p f(y_i) L_i(x) \quad (12.6)$$

with  $\{L_i(x), i = 0, \dots, p\}$  being the basis of Lagrange polynomials for  $Y$ .

**Alternative Formulations.** There are several other definitions of Lagrange polynomial, which are more useful when discussing measures of poisedness (see Section 12.2.1). Therefore, we present these definitions here.

Assume as given a poised set  $Y = \{y_0, \dots, y_p\} \subset R^n$ , a point  $x \in R^n$  and a non-singular matrix  $M(\phi, Y)$ . As  $M(\phi, Y)$  is non-singular, it is possible to express the vector  $\phi(x)$  uniquely in terms of the vector  $\phi(y_i)$ ,  $i = 0, \dots, p$  as

$$\sum_{i=0}^p \lambda_i(x) \phi(y_i) = \phi(x). \quad (12.7)$$

This can be written in matrix form as

$$M(\phi, Y)^T \lambda(x) = \phi(x) \quad (12.8)$$

where  $\lambda(x) = [\lambda_0(x), \dots, \lambda_p(x)]^T$ .

From Equation 12.8 we can see that  $\lambda(x)$  is a vector of polynomials in  $P_n^d$  and from Equation 12.7 we can see that  $\lambda_i$  is the  $i$ -th Lagrange polynomial for  $Y$ .

Another alternative definition is as follows. Given the set  $Y$  and a point  $x \in R^n$ , the set  $Y_i(x)$  is defined as  $Y_i(x) \setminus \{y_i\} \cup \{x\}$ ,  $i = 0, \dots, p$ . From applying Cramer's rule on Equation 12.8, the  $\lambda_i$ 's can be written as

$$\lambda_i(x) = \frac{\det(M(\phi, Y_i(x)))}{\det(M(\phi, Y))}. \quad (12.9)$$

From Equation 12.9 we can see that  $\lambda_i(x)$  is a polynomial in  $P_n^d$  and that it satisfies Equation 12.7. Thus  $\{\lambda_i(x), i = 0, \dots, p\}$  is the set of Lagrange polynomials.

Furthermore, consider a set  $\phi(Y) = \{\phi(y_0), \dots, \phi(y_p)\}$  in  $R^{p_1}$ . Let  $\text{vol}(\phi(Y))$  be the volume of the simplex that is created by the vertices in  $\phi(Y)$  and be defined as

$$\text{vol}(\phi(Y)) = \frac{|\det(M(\phi, Y))|}{p_1!}. \quad (12.10)$$

The aforementioned simplex is the  $p_1$ -dimensional convex hull of  $\phi(Y)$ . Then the absolute value of the  $i$ -th Lagrange polynomial at the point  $x$  is the change of volume of this hull when  $y_i$  is replaced by  $x$ , i.e.

$$|\lambda_i(x)| = \frac{\text{vol}(\phi(Y_i(x)))}{\text{vol}(\phi(Y))}. \quad (12.11)$$

**Time Complexity.** To calculate a Lagrange polynomial basis takes  $O(p^3)$  time. Updating a polynomial when the set  $Y$  is updated with one point takes  $O(p^2)$  time.

**Error Bounds.** The error bound on the  $r$ -th derivative is

$$\|D^r f(x) - D^r m(x)\| \leq \frac{1}{(d+1)!} v_d \sum_{i=0}^p \|y_i - x\|^{d+1} \|D^r L_i(x)\|, \quad (12.12)$$

with  $D^r$  denoting the  $r$ -th derivative of  $f(x)$  and  $m(x)$  respectively, and  $v_d$  being an upper bound on  $D^{r+1}$ . Similarly, the difference between  $f(x)$  and the model  $m(x)$ , constructed using Lagrange polynomials, is

$$|f(x) - m(x)| \leq \frac{1}{(d+1)!} p_1 v_d \Lambda_l \Delta^{d+1}, \quad (12.13)$$

where

$$\Lambda_l = \max_{0 \leq i \leq p} \max_{x \in B(Y)} |L_i(x)|, \quad (12.14)$$

and  $\Delta$  is the radius of the smallest ball  $B(Y)$  containing  $Y$ .

### 12.1.5 Newton Polynomials

Another option for calculating the interpolation polynomial is to use Newton polynomials. They are more complicated to calculate as compared to the Lagrange polynomials, but updating and evaluation of the polynomials is less computationally demanding. Newton polynomials were used in the early versions of the DFO algorithm, see Section 13.1.

Since a polynomial of degree  $d$  is unique regardless of whether it is calculated using Lagrange or Newton polynomials and as the commonly used measures of poisedness (see Section 12.2.1) are typically derived in terms from Lagrange polynomials, we will not treat Newton fundamental polynomials here. For more information the reader is referred to the following references: Sauer and Xu [66] described Newton fundamental polynomials for interpolation. For practical implementation details as well as some numerical experiments, we refer to Sauer [65]. The books by Conn et al. [12, 17] provide more general information.

### 12.1.6 Measurement Errors or Noise in the Objective Function

In the presence of noise or measurement errors in  $f$ , it might not be desirable to let the model to through the points  $y_i$  exactly. This can for example happen if there is measurement uncertainty or  $f$  models some naturally occurring phenomena, where  $f(x)$  may return different values despite the same  $x$ .

In such cases, Equation 12.1 can be defined as

$$\|f(y_i) - m_k(y_i)\| \leq v, \forall y_i \in Y \quad (12.15)$$

for some small interpolation tolerance  $v$ . Equation 12.15 is typically solved as a least-squares problem, i.e. the interpolation coefficients  $\alpha_\phi$  (see Equation 12.4) are found, such that  $\|M(\phi, Y)\alpha_\phi - f(Y)\|^2$  is minimized. The reader is referred to the book by Conn et al. [12] for more information about this.

## 12.2 Poisedness

It is not sufficient that  $Y$  has the correct cardinality. Equation 12.4 must also have a unique solution. The term used for this is that  $Y$  should be *poised*. The set  $Y$  is poised if

$$\det(M(\phi, Y)) = \det \begin{pmatrix} \phi_0(y_0) & \phi_1(y_0) & \cdots & \phi_p(y_0) \\ \phi_0(y_1) & \phi_1(y_1) & \cdots & \phi_p(y_1) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_0(y_p) & \phi_1(y_p) & \cdots & \phi_p(y_p) \end{pmatrix} \quad (12.16)$$

is non-zero, i.e. the matrix  $M(\phi, Y)$  is non-singular. If so, then there exists a linear or quadratic polynomial interpolation of function values [15, 17]. We require a unique solution to the system of equations, therefore Equation 12.4 must be non-singular.

A non-zero determinant is the primary requirement but a set of points that yields an almost singular matrix should also be avoided. This is because such a set can cause numerical problems when solving Equation 12.4.

Naturally, the poisedness is not only affected by which points are used to create the initial polynomial but also by the points that are included in  $Y$  as execution progresses. Typically, the number of points in  $Y$  is fixed. Thus, if one point enters the set, another point must leave it. Poisedness measures can be used, directly or indirectly, to determine which point that should leave  $Y$ . In the next section, we will discuss measures of poisedness. In Section 12.3.3 we discuss how this can be taken into consideration when deciding which points that should enter and leave  $Y$ .

### 12.2.1 Measures of Poisedness

There are several measures of poisedness and in this section we will briefly discuss these measures. For a more thorough discussion, the reader is referred to the book by Conn et al. [17].

If the matrix  $M(\phi, Y)$  is poised, then the absolute value of  $\det(M(\phi, Y))$  is a measure of the linear independence of the columns [15]. Thus, a greater value of  $|\det(M(\phi, Y))|$  is desirable as this means that the distance to linear dependence between the columns in  $M(\phi, Y)$  is greater.

Another, more commonly used measure of poisedness is  $\Lambda$ -poisedness, which is defined as follows. Let a scalar  $\Lambda > 0$ , a set  $B \in R^n$  and  $\phi = \{\phi_0, \dots, \phi_p\}$ , a basis of  $P_n^d$ , be given. A poised set  $Y = \{y_0, \dots, y_p\}$  is said to be  $\Lambda$ -poised in  $B$  if and only if any of the following equivalent requirements hold:

1. for the basis of Lagrange polynomials associated with  $Y$

$$\Lambda \geq \max_{0 \leq i \leq p} \max_{x \in B} |L_i(x)|, \quad (12.17)$$

2. for any  $x \in B$  there exists  $\lambda(x) \in R^{p_1}$  such that

$$\sum_{i=0}^p \lambda_i(x)\phi(y_i) = \phi(x) \text{ with } \|\lambda(x)\|_\infty \leq \Lambda \quad (12.18)$$

3. replacing any point in  $Y$  by any  $x \in B$  can increase the volume of the set  $\{\phi(y_0), \dots, \phi(y_p)\}$  by at most a factor  $\Lambda$ .

It should be noted that  $\Lambda$ -poisedness is inversely related to poisedness as measured by the absolute value of the determinant: it is desirable to have a large value of  $|\det(M(\phi, Y))|$  while it is desirable to have a small value of  $\Lambda$ . It is also possible to use  $\Lambda$ -poisedness as a measure to linear dependence. We will now derive such an expression, adapted from Conn et al. [17].

Assume as given an interpolation set  $Y$ , a value of  $\Lambda$  and let  $B(y_0, \Delta(Y))$  be the smallest closed ball containing  $Y$ , centered in  $y_0$ . Furthermore, assume that for the given  $\Lambda$ ,  $Y$  is not  $\Lambda$ -poised in  $B(y_0, \Delta(Y))$ . Then, Equation 12.18 does not hold and there exists a  $z \in B(y_0, \Delta(Y))$  such that

$$\sum_{i=0}^p \lambda_i(z)\phi(y_i) = \phi(z) \text{ with } \|\lambda(z)\|_\infty > \Lambda. \quad (12.19)$$

Thus, we say without loss of generality, that  $\lambda_1(z) > \Lambda$ . If we divide Equation 12.19 by  $\Lambda$  we get

$$\sum_{i=0}^p \frac{\lambda_i(z)\phi(y_i)}{\Lambda} = \sum_{i=0}^p \gamma_i(z)\phi(y_i) = \frac{\phi(z)}{\Lambda} \text{ with } \gamma_1(z) > 1. \quad (12.20)$$

From this we get

$$\left\| \sum_{i=0}^p \gamma_i(z)\phi(y_i) \right\|_\infty \leq \frac{\max_{x \in B(y_0, \Delta(Y))} \|\phi(x)\|_\infty}{\Lambda} \quad (12.21)$$

As an example, let  $\bar{\phi}$  be the natural basis,  $y_0 = 0$  and the radius  $\Delta(Y)$  of  $B(y_0, \Delta(Y))$  be 1. Then  $\max_{x \in B(0,1)} \|\bar{\phi}(x)\|_\infty \leq 1$  and  $\left\| \sum_{i=0}^p \gamma_i(z)\bar{\phi}(y_i) \right\|_\infty \leq \frac{1}{\Lambda}$  with  $\gamma_1 > 1$ . In some sense,  $1/\Lambda$  bounds the distance to linear dependence, but the actual distance to linear dependency depends on the choice of basis.

It can be shown that  $\Lambda$  poisedness is not affected by scaling or shifting of the coordinates, for more information we refer the reader to Conn et al. [17].

## 12.3 The Trust Region Framework

Once the model is built and the starting point is determined, the actual optimization starts. Here the algorithm has two contradictory objectives: one is to find points in which the objective function value improves and the other is to explore the parts of the search space

where the objective function have been evaluated a few times or not at all. This is sometimes called the *exploitation/exploration* dilemma as the algorithm must decide whether to try to improve the objective function value (exploitation) or improve the model (exploration). For some algorithms this difference is very explicit while for others it is more implicit.

The algorithms that are discussed here use either a linear or quadratic interpolation model, and use the concept of a trust region during the optimization. The trust region is a region in which the model is assumed to provide a good approximation of  $f$ , thus its name. Within the trust region, the next point in which to evaluate  $f$  is to be found.

Generally, the trust region  $B$  is described using a midpoint  $x_m$  and a radius  $\Delta$ . The radius often changes during the execution, therefore both the  $B$  and  $\Delta$  are subscripted with the iteration number:  $B_k(x_m, \Delta_k)$ . Since the midpoint typically is the current iterate  $x_k$ , we will use that notation here.

Each optimization of the model  $m_k$  starts in  $x_k$  and the step  $h_k$  cannot be longer than the trust region radius, thus  $\|h_k\| \leq \Delta_k$ . The goal is to find a *candidate point*  $x_k^+$  with the minimal value of  $m_k$  subject to  $\|h_k\| \leq \Delta_k$ . That is:

$$x_k^+ \leftarrow \operatorname{argmin}\{m_k(x_k + h_k), \text{s.t.} \|h_k\| \leq \Delta_k\}. \quad (12.22)$$

We call Equation 12.22 the *trust region subproblem*.

Once  $x_k^+$  is found,  $f(x_k^+)$  is calculated. By minimizing  $m_k$  within  $B_k$ , an attempt is made to find a point where  $f$  has a lower (better) value. Naturally, the better  $m_k$  approximates  $f$ , the more likely it is that the lower value in  $m_k$  corresponds to a lower value in  $f$ .

The exact method for minimizing  $m_k$  varies between different algorithms, but the Moré-Sorensen algorithm [47] and conjugate gradient methods are popular choices, see e.g. [14, 16, 60, 61].

### 12.3.1 Updating the Trust Region Radius

As earlier discussed, first the model  $m$  is minimized inside the trust region and from this it is possible to calculate the expected decrease in objective function value as  $m_k(x_k) - m_k(x_k^+)$ . Then  $f(x_k^+)$  is evaluated. Given this, the achieved decrease in objective function value is calculated as  $f(x_k) - f(x_k^+)$ , where  $f(x_k)$  was calculated in a previous iteration. From these two values, the ratio

$$r_k \leftarrow \frac{f(x_k) - f(x_k^+)}{m_k(x_k) - m_k(x_k^+)} \quad (12.23)$$

is calculated. It measures the ratio between the achieved improvement and the expected improvement. Note that the numerator is interpreted as the expected decrease in objective function value even though it is known that there might be a big difference between  $m_k$  and  $f$ .

Depending on  $r_k$ , the trust region radius will either increase or decrease. The constants  $\eta_0, \eta_1, \gamma_1$ , and  $\gamma_2$  with  $\gamma_1 \in (0, 1), \gamma_2 > 1, 0 \leq \eta_0 \leq \eta_1 < 1$  and  $\eta_0 \neq \eta_1$ , are supplied by the user. The constant  $\eta_0$  is not used here but will be used in Section 12.3.2. They are used when determining how the radius will change. If  $r_k \geq \eta_1$  then the iteration is judged to be successful and the trust region radius is increased through multiplication by  $\gamma_2$ . Otherwise, the radius is decreased through multiplication with  $\gamma_1$ . The next iteration's trust region radius  $\Delta_{k+1}$  is thus modified according to

$$\Delta_{k+1} \leftarrow \begin{cases} \gamma_1 \Delta_k & \text{if } r_k < \eta_1, \\ \gamma_2 \Delta_k & \text{if } r_k \geq \eta_1. \end{cases} \quad (12.24)$$

### 12.3.2 Adding a Point to the Interpolation Set

To determine whether the candidate point  $x_k^+$  is accepted into  $Y$ , different algorithms have different methods. Here we first describe how the DFO method (see Section 13.1) by Conn et al. [16] handles this.

The DFO method uses the ratio  $r_k$  to determine whether the point shall be included into  $Y$ : if  $r_k \geq \eta_1$  then  $x_k^+$  is added to  $Y$ , as the decrease in  $f$  was sufficient to motivate the inclusion of the point. This can be seen as an exploitation step.

Otherwise, if  $r_k \geq \eta_0$ , the point is included if certain conditions are fulfilled. The rationale is that even though the decrease was not as good as predicted, the information gained from evaluating  $f$  shall be used. Note that this means that although  $x_k^+$  is included in  $Y$ ,  $\Delta_{k+1}$  may decrease, i.e. when  $\eta_0 \leq r_k < \eta_1$ .

In all other cases,  $x_k^+$  is rejected.

Some algorithms, e.g. UOBYQA and NEWUOA (see Sections 13.2 – 13.3), always include  $x_k^+$  into  $Y$ , regardless of the value of  $r_k$ . The motivation behind this is that even though the objective function value might have been poor, the new point provides information about  $f$ .

If  $x_k^+$  is included into the interpolation set  $Y$ , an existing point must be removed. Which point this is can be determined in different ways. Several procedures for determining which point to remove, and the reasoning behind them are discussed in Section 12.3.3.

### 12.3.3 Updating the Interpolation Set

The interpolation set can be updated for two different reasons. The first case is that a new point  $x_k^+$  is to be added to  $Y$ , e.g. since  $f(x_k^+) < f(x_k)$ . The other case is when it is believed that the current model prevents progress and the model needs to be updated with a new point in order to facilitate further progress. This is called a *model iteration*, since the iteration is spent trying to find a point that will improve the model. Regardless of the reason, a point  $y_i \in Y$  must be removed. Both these cases are discussed below.

When  $Y$  is to be updated because  $x_k^+$  shall be added, the objective is often to find a point in  $Y$ , to be removed, that is far away from  $x_k^+$ . However, it is also desirable that the

objective function has a large value in the point, since removing such a point is believed to improve the conditioning of the model. The UOBYQA algorithm (see Section 13.2) finds a point  $y_i$  that maximizes  $\max(1, (d/\rho_k)^3) \cdot m_k(y_i)$ , where  $d$  is the Euclidian distance between  $x_k^+$  and  $y_i$ . The variable  $\rho_k$  is described in Table 13.1 on page 172.

Scheinberg [69] described an algorithm that uses different criteria for choosing which point to replace, depending on the value of  $r_k$  (see Equation 12.23) as well as the poisedness value, as measured by  $\Lambda$ . If  $r_k \geq \eta_1$ , then a point  $y_i$  is found that either is far from  $x_k^+$ , or for which  $|l_i(x_k^+)|$  is the largest. If the condition is not fulfilled, then the algorithm has several options. First, control whether there is a point that is far away. If so, replace it with  $x_k^+$ . If there is no such point, control whether there is a point  $y_i$  for which  $|l_i(x_k^+)| > \Lambda$  for some  $\Lambda > 1$ . If so, replace it by  $x_k^+$ . Otherwise, the model is kept unchanged and the trust region radius is decreased.

The algorithms described above explicitly care about the geometry of the interpolation set, whereas Fasano et al. [21] presented an algorithm that did not regard the geometry of  $Y$ . Despite this, the algorithm achieved good performance. However, Scheinberg and Toint [68] showed that Fasano et al.'s algorithm could converge to non-stationary points. Furthermore, they presented an algorithm of their own, with what they called self-correcting geometry. Through an improved choice of which interpolation point to remove, they do not have to perform the model iterations mentioned above. The concept of self-correcting geometry is also used, albeit in a slightly modified form, in the extended version of DFO (see Section 13.1.2).

Algorithms that use an explicit model iteration to improve  $m$ , e.g. UOBYQA and NEWUOA, typically find a point  $y_i$  far from  $x_k$ . Then the Lagrange polynomial corresponding to  $y_i$  is maximized within a trust region centered in  $x_k$ . This yields a new point that replaces  $y_i$  in  $Y$ .

## 12.4 Summary

Model building algorithms differ significantly from direct-search algorithms in that they build a model of the objective function instead of evaluating the objective function directly. This kind of algorithm consists of two major parts: one part is building and updating the model and the other is the optimization algorithm that solves the trust region subproblem.

The model is built using a set of points and their values, and the model is then used to find a point in which to evaluate the objective function. The trust region subproblem consists of finding the minimum of the model within a trust region. The reason for the trust region is that the model is local, and is typically only trusted within a small distance from its midpoint.

The trust region subproblem is solved using some optimization algorithm, often a conjugated gradient algorithm. The algorithm returns a candidate point  $x_k^+$  and the objective function is evaluated in that point.

Once the objective function has been evaluated in  $x_k^+$ , the ratio between the achieved decrease in the objective function value and the predicted decrease is calculated. The ratio is typically used to decide whether to decrease, keep or increase the trust region radius. Some algorithms also use it to determine whether the new point is included in the set of interpolation points, whereas other algorithms always include the new point.

# 13

---

## Existing Model Building Algorithms

In this section we will present some of the most well-known algorithms for model building derivative-free optimization in more detail. The algorithms here use linear or quadratic interpolation models.

### 13.1 The DFO Algorithm

The DFO (Derivative Free Optimization) method was developed by Conn et al. The theory behind this type of algorithm is described by Conn et al. [15] and the algorithm itself by the same authors [16]. The source code is available for download [13].

The DFO algorithm follows the general description of a model building algorithm from Chapter 12 quite closely, and in many aspects, it can be seen as a generic blue-print for this type of algorithms.

The model in the DFO algorithm is different from other optimization algorithms in that the number of interpolation points may vary during the execution. At the beginning of execution, at least  $p_1 = n + 1$  points for a linear model are required. If points of sufficiently high quality are found, the model can expand up to a full quadratic model during execution. The original DFO presented here, uses Newton fundamental polynomials (see Section 12.1.5) to create the model.

#### 13.1.1 DFO Pseudocode

The pseudocode for the DFO algorithm is available in Figure 13.1 and is further described below.

Assume as input a starting point  $x_s$  with the objective function value  $f(x_s)$ , and an initial interpolation set  $Y$  containing  $x_s$ . Assign  $k \leftarrow 0$ ,  $\Delta_k$  and  $x_k \leftarrow \operatorname{argmin}\{f(y_i)\}$ ,  $y_i \in$

$Y\}$ . At this point all necessary values have been set and the algorithm can start.

On line 2, the model  $m_k$  is built using the interpolation set  $Y$ . This requires building a model from parts of, or the complete set,  $Y$ . Lines 3–9 are performed as long as the algorithm executes.

The candidate point  $x_k^+$  is calculated on line 4. It is the result of finding the step  $h_k$  that minimizes  $m_k(x_k + h_k)$  with the trust region. Naturally, as the model  $m_k$  is trusted within the trust region with radius  $\Delta_k$ ,  $\|h_k\|$  may not be greater than  $\Delta_k$ .

On line 5 it is checked if the candidate point is the same as the starting point in this iteration. If so, then line 6 is skipped. Otherwise, the ratio  $r_k$  is calculated on line 6, according to Equation 12.23. The next thing to do is to update the model (line 7). This involves looking at the cardinality of  $Y$  as well as the ratio  $r_k$ .

- If  $|Y| < \frac{1}{2}(n+1)(n+2)$ , then the point  $x_k^+$  is added to  $Y$ .
- If  $r_k$  is good enough, then  $x_k^+$  is added to  $Y$ , even if this requires removing some other point from  $Y$ .
- If  $r_k$  is too low, then an attempt to find a *new* point is performed and possibly added to  $Y$ .
- If some points in  $Y$  are too far from  $x_k$ , they are removed from  $Y$ .

The function `Update-radius` updates the region radius  $\Delta$  (line 8) according to the following rules: if  $r_k$  is good, increase  $\Delta$ . If  $r_k$  is bad and the model is at least linear, decrease  $\Delta$ . If  $r_k$  is too small, terminate the algorithm.

The last thing to do in this iteration is to increment  $k$  by one and set  $x_k$  to be the point in  $Y$  with the smallest function value. The execution then continues with another iteration on line 3.

In the original description [16], there was no details on how to update the trust region radius or how to determine whether a point to good enough to be included in  $Y$ . Such a description was later added [17], and is here described in Section 12.3.1 and Section 12.3.2, for updating of the trust region radius and the interpolation set, respectively.

### 13.1.2 Extensions

The DFO algorithm has been developed further and a newer version [78] uses Lagrange polynomials to create the model  $m_k$ . The paper also presents extensions to the Moré-Sorensen algorithm [47] that allows it to handle upper and lower bounds. The extended algorithm also has a significantly more advanced procedure for updating the interpolation set. Very briefly, if the point that is furthest away from  $x_k^+$  is judged to be close enough, then it will remain in  $Y$ . Instead a point close to  $x_k^+$  will be replaced, if there is a point that fulfills certain requirements. If there is no such point,  $Y$  remains unchanged and the trust region radius is decreased instead.

---

```

1  $k \leftarrow 0, x_k \leftarrow \operatorname{argmin}\{f(y_i), y_i \in Y\}$ 
2 Build-model
3 loop
4    $x_k^+ \leftarrow \operatorname{argmin}\{m_k(x_k + h_k), s.t. \|h_k\| < \Delta\}$ 
5   if  $x_k^+ \neq x_k$  then
6      $r_k \leftarrow \frac{f(x_k) - f(x_k^+)}{m_k(x_k) - m_k(x_k^+)}$ 
7      $m_{k+1} \leftarrow \text{Update-model}(r_k)$ 
8      $\Delta_{k+1} \leftarrow \text{Update-radius}(r_k)$ 
9    $k \leftarrow k + 1, x_k \leftarrow \operatorname{argmin}\{f(y_i), y_i \in Y\}$ 

```

**Figure 13.1:** Pseudocode for the DFO algorithm.

## 13.2 The UOBYQA Algorithm

M.J.D. Powell has developed several methods for optimization without derivatives. One of them is COBYLA (Constrained Optimization BY Linear Approximations) [56], which uses a linear model and linear approximations of non-linear constraints. Then, the algorithm UOBYQA (Unconstrained Optimization BY Quadratic Optimization) [58, 59] was developed. UOBYQA used a full quadratic model as  $m_k$  but since the updating procedure of the model can be time-consuming when  $n$  is large, it is recommended to use UOBYQA when  $n$  is small.

To minimize  $m_k(x_k + h_k)$  within the trust region, UOBYQA uses a version of the algorithm by Moré and Sorensen [47].

UOBYQA was the first algorithm to have an analytical measure of the difference between the real objective function and the model. If  $f$  has third derivatives that are bounded by the variable  $M$ , then the difference between the model  $m$  and  $f$  in the point  $x$  satisfies

$$|m(x) - f(x)| \leq \frac{1}{6} M \sum_{i=1}^p |L_i(x)| \|x - x_i\|^3. \quad (13.1)$$

This measure of the difference between the model and the real objective function was published before the one in Equation 12.13, which may be why they are different. The error measure  $M$  is updated during the execution of the algorithm and while the value itself is not used, the number of updates of the value is used.

### 13.2.1 UOBYQA Pseudocode

Here we give the pseudocode of the UOBYQA algorithm, based on the description by Han and Liu [33]. For the full details the reader is referred to the original description by Powell [59].

## 13. Existing Model Building Algorithms

---

In addition to the terms in Table A on page 311, UOBYQA and NEWUOA (see Section 13.3) use several additional terms. These are described in Table 13.1. The variables  $\rho_{beg}$ ,  $\rho_{end}$ ,  $\Delta$  and the starting point  $x_0$  are assumed to be given by the user as input to the algorithm.

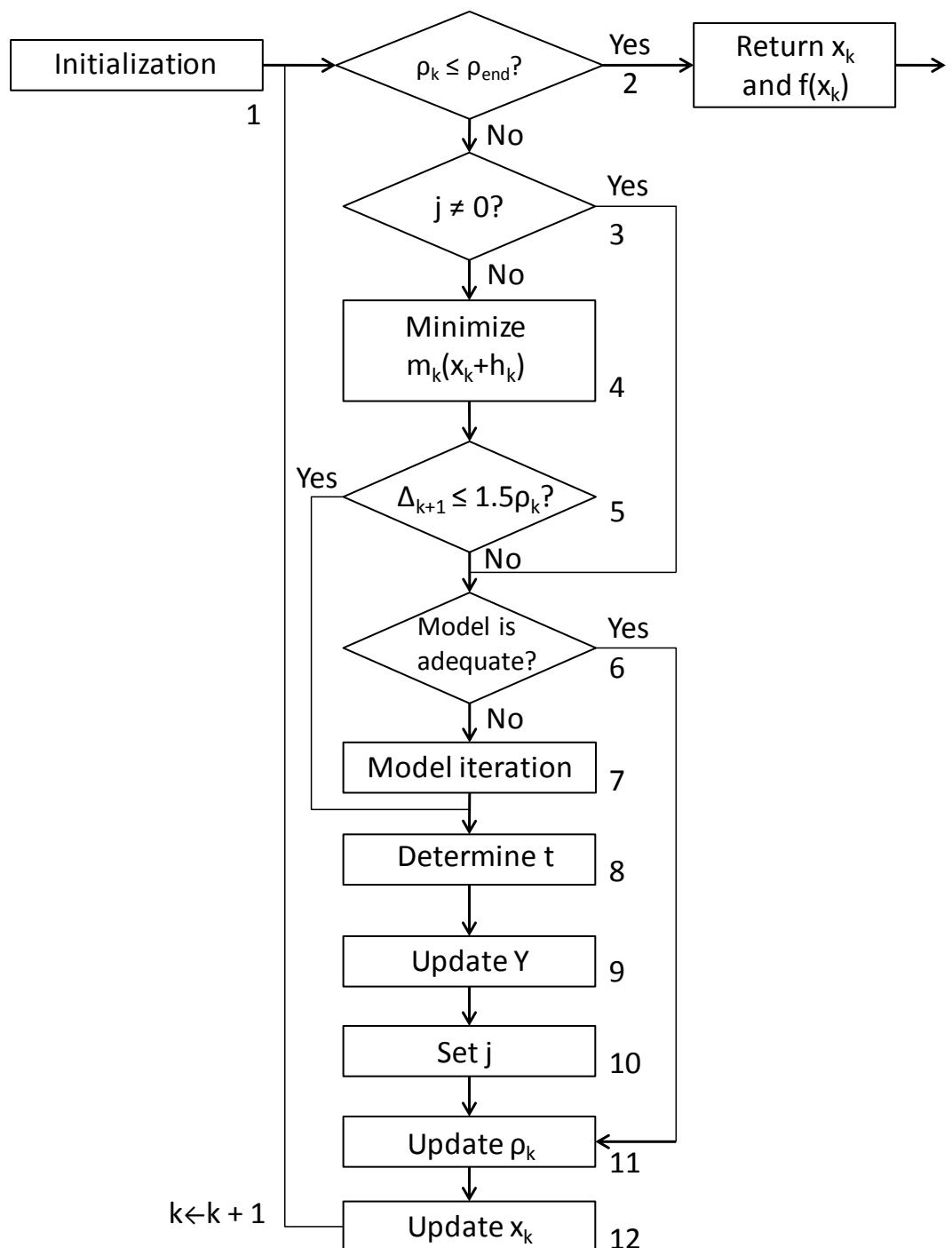
Term	Description
$M$	Bound on $f$ 's third derivative.
$\rho_{beg}$	Initial value of trust region radius.
$\rho_{end}$	Final value of trust region radius, $\rho_{end} \leq \rho_{beg}$ . The value of $\rho_{end}$ should have the magnitude of the required accuracy in the final value of the solution.
$\rho_k$	Current value of the trust region radius, with $\rho_{end} \leq \rho_k \leq \rho_{beg}$ .
$j$	Determines whether a trust region step ( $j = 0$ ) or a model-step ( $j \neq 0$ ) shall be performed. When ( $j \neq 0$ ), then it stores the index of the point to be replaced.
$t$	The index of a point in $m$ to be replaced.
$h_{norm}$	Length of the step $h$ .
$h_{move}$	Distance that a point is moved.
$reduce$	Whether $\rho$ shall be reduced or not. True/false.
$c_{min}$	The value of the least positive curvature of $f$ .
$d$	The distance between two points.

**Table 13.1:** Terms used in the UOBYQA and NEWUOA algorithms.

The fact that both  $\Delta$  and  $\rho$  are described as trust region radii deserves some discussion. The variable  $\Delta$  is used in trust region iterations, when a function is to be minimized within a certain distance from a given starting point, which is the typical use of the trust region radius. The variable  $\rho_{beg}$  is used when placing the points used to build the initial model. The variable  $\rho_k$  is initialized to  $\rho_{beg}$ . It is used to check whether the points to be included in the model are judged to be sufficiently far away from each other, and also in model iterations, when the model is improved. The value of  $\rho_k$  is decreased throughout the execution of the algorithm, and when it is less than or equal to  $\rho_{end}$ , the algorithm terminates. During execution,  $\rho_k$  and  $\Delta_k$  also affect each other.

The numbers below refer to the numbered boxes in Figure 13.2.

1. Initialization. An initial model is constructed from  $\frac{1}{2}(n+1)(n+2)$  points. In each point,  $f$  is evaluated and the point is associated with a Lagrange polynomial. Also,  $M \leftarrow 0$ ,  $\rho_k \leftarrow \Delta_{beg}$ ,  $j \leftarrow 0$ ,  $k \leftarrow 0$ ,  $reduce \leftarrow false$  are set. Set  $x_k$  to be the point with the lowest objective function value.
2. Termination test. If  $\rho_k \leq \rho_{end}$ , then the algorithm terminates.

**Figure 13.2:** Flowchart for the UOBYQA algorithm.

### 13. Existing Model Building Algorithms

---

3. If  $j = 0$ , then execution continues at step 4. If  $j \neq 0$  then execution continues at step 6.
4. Trust region iteration. Let  $x_k^+ \leftarrow \operatorname{argmin}\{m_k(x_k + h_k), \text{ s.t. } \|h_k\| \leq \Delta_k\}$ .  $h_{norm} \leftarrow \|h_k\|$ . If  $h_{norm} < \frac{\rho_k}{2}$ , then execution continues directly in step 4. Otherwise, calculate  $f(x_k^+)$ ,  $m_k(x_k^+)$  and  $L_i(x_k^+)$  for  $i = 1, 2, \dots, p_1$ . Update

$$\frac{M}{6} \leftarrow \max \left\{ \frac{M}{6}, \frac{|m_k(x_k^+) - f(x_k^+)|}{\sum_{i=1}^p |L_i(x_k^+)| \|x_k^+ - x_i\|^3} \right\}. \quad (13.2)$$

Calculate  $r_k$  according to Equation 12.23. Update  $\Delta$  according to the following:

$$\Delta_{k+1} \leftarrow \begin{cases} \max\{\Delta_k, \frac{5}{4}\|h_k\|, \rho_k + \|h_k\|\} & \text{if } r_k \geq 0.7, \\ \max\{\frac{1}{2}\Delta_k, \|h_k\|\} & \text{if } 0.1 < r_k < 0.7, \\ \frac{1}{2}\|h_k\| & \text{if } r_k \leq 0.1. \end{cases} \quad (13.3)$$

Then continue in step 4.

5. If  $\Delta_{k+1} \leq \frac{3}{2}\rho_k$ , then assign  $\Delta_{k+1} \leftarrow \rho_k$  and continue execution at step 8. Else continue at step 6.
6. Determine whether the model is adequate. Let  $\lambda_m$  be the least eigenvalue of  $m$ . Assign

$$\epsilon \leftarrow \begin{cases} \frac{1}{2}\rho_k^2 \cdot \max\{0, \lambda_m\} & \text{if } \|h_k\| < \frac{1}{2}\rho_k \\ 0 & \text{else.} \end{cases} \quad (13.4)$$

The model is adequate if either of the conditions below is satisfied.

- (a) If  $h_{norm} < \frac{1}{2}\rho_k$  and for every integer  $j \in [1, p_1]$ , at least one of the following conditions holds:

$$\frac{M}{6}\theta_j\|x_j - x_k\|^3 \leq \epsilon \text{ and } \|x_j - x_k\| \leq 2\rho_k \quad (13.5)$$

with  $\theta_j \leftarrow \max\{|L_j(x_k^+)| : \|h_k\| \leq \rho_k\}$ , or if  $\frac{1}{2}\rho_k \leq h_{norm} \leq \rho_k$ ,  $f(x_k^+) \geq f(x_k)$  and for every integer  $j \in [1, p_1]$  at least one of the conditions in Equation 13.5 is satisfied, then  $j \leftarrow 0$ , and  $reduce \leftarrow true$ . Execution continues in step 11.

- (b) If  $h_{norm} > \rho_k$ ,  $f(x_k^+) \geq f(x_k)$  and for every integer  $j \in [1, p_1]$  at least one of the conditions in Equation 13.5 holds, then  $j \leftarrow 0$  and  $reduce \leftarrow false$ . Execution continues in step 11.

If neither of the above conditions hold, then the model is inadequate. In that case, the integer  $j > 0$  that causes the conditions in Equation 13.5 to not hold, is stored.

7. Model iteration. The model step  $h_k$  is calculated by solving the subproblem  $x_k^+ \leftarrow \operatorname{argmax}\{|L_j(x_k + h_k)| \text{ s.t. } \|h_k\| \leq \rho_k\}$ , for a new value of  $h_k$  that is determined here. Calculate  $f(x_k^+)$ ,  $m_k(x_k^+)$  and  $L_i(x_k^+)$ ,  $i = 1, 2, \dots, p_1$ . Update  $M$  according to Equation 13.2.
8. Select  $t$ . Determine the interpolation point indexed by  $t$  that will be replaced by the new point  $x_k^+$ . If  $j > 0$ , then  $t \leftarrow j$ . Else select  $t$  to keep or improve the geometric condition of  $Y$ . In that case,  $t$  is typically chosen so that it is far from the point with the best current value. If no suitable point is found,  $t \leftarrow 0$ .
9. Update the interpolation set  $Y$ . If  $t > 0$ , let  $x_t$  be the interpolation point indexed by  $t$ . Move  $x_t$  to the position of  $x_k^+$ :  $\tilde{x}_t \leftarrow x_k^+, x_t \leftarrow \tilde{x}_t$ .  
Update the coefficients of the Lagrange polynomials and  $m_k$ . If  $f(\tilde{x}_t) < f(x_k)$  then  $h_{move} \leftarrow \|\tilde{x}_t - x_t\|$ , else  $h_{move} \leftarrow \|x_k - x_t\|$  for the old position of  $x_t$ .
10. Choose between trust region and model step iteration. If  $t > 0$  and at least one of the following conditions hold:  $j > 0$ ,  $f(\tilde{x}_t) < f(x_k)$ ,  $h_{norm} > 2\rho_k$  or  $h_{move} > 2\rho_k$ , then  $j \leftarrow 0$ . Otherwise  $j \leftarrow -1$ .
11. Update  $\rho$ . If  $reduce = \text{true}$  then  $\Delta_{k+1} \leftarrow 0.5\rho_k$ ,  $\rho_{k+1} \leftarrow 0.1\rho_k$  and  $reduce \leftarrow \text{false}$ , else  $\rho_{k+1} \leftarrow \rho_k$ .
12. Update  $x_k$ . If  $f(\tilde{x}_t) < f(x_k)$ , then  $x_{k+1} \leftarrow \tilde{x}_t$ , else  $x_{k+1} \leftarrow x_k$ . Assign  $k \leftarrow k + 1$ . Execution continues in step 2.

### 13.2.2 Extensions

UOBYQA has been extended with constraint handling for both bounds, linear and non-linear constraints in CONDOR [5].

### 13.2.3 Summary

The UOBYQA algorithm either minimizes the model  $m_k$  within the trust region (trust region step) or finds a new point to improve the model (model step). Regardless of which, a new point is found and the objective function  $f$  is evaluated in that point. Then the new point is added to the model and another point is removed. If the value of the objective function in the new point  $x_k^+$  is lower than the previous starting point, then the next iteration will start in the new point. This process continues until the condition  $\rho_k \leq \rho_{end}$  applies, in which case the algorithm terminates and returns the best point  $x_k$  and its value.

## 13.3 The NEWOUA Algorithm

The previously described UOBYQA algorithm was replaced by NEWOUA (NEW Unconstrained Optimization Algorithm) [60], which is also developed by M.J.D. Powell.

The motivation to develop a new algorithm was to solve problems involving a greater number of variables. To allow for this, some of the parts of UOBYQA that were deemed to be too time-consuming were changed.

NEWUOA is different from other model building algorithms in that it allows the use of fewer interpolation points than  $\frac{1}{2}(n+1)(n+2)$ : instead it uses  $p_1$  points, where  $n+2 \leq p_1 \leq \frac{1}{2}(n+1)(n+2)$ . The recommended value of  $p_1$  is  $2n+1$ , as using more interpolation points is considered to make the updating too time-consuming. If less than  $\frac{1}{2}(n+1)(n+2)$  points are used, then the remaining degrees of freedom are determined through calculating the difference between two consecutive model Hessians and then minimizing the Frobenius norm of this difference.

While there is a choice in the number of interpolation points, once the choice is made, the number is fixed for the remainder of the execution. If a new point that has a lower  $f$ -value than the previously lowest value is found, then this point is added to  $Y$  and another point must be removed. Typically, the point farthest from the new point is removed, but as the poisedness value of the model is affected by the choice of which point to remove, this is also taken into consideration.

Another difference is that NEWUOA uses a truncated conjugated gradient algorithm to find the minimum of  $m_k$  within the trust region. This algorithm requires time proportional to  $O(n^2)$  instead of the algorithm in UOBYQA that requires time in order of  $O(n^3)$ .

### 13.3.1 NEWUOA Overview

NEWUOA differs from other model building algorithms in that it has intermingled trust region iterations and model iterations for efficiency reasons. The objective of a trust region step is to find a better objective function value, while the purpose of the model iteration is to improve the model. Compare with the discussion about exploration/exploitation in Section 12.3.

All iterations start with solving the trust region subproblem from Equation 12.22. In each iteration, the starting point  $x_k$  is the point with the lowest objective function value. The result of this optimization is the step  $h_k$ , which is added to  $x_k$  to give the new point;  $x_k^+ \leftarrow x_k + h_k$ . Depending on the length of  $h_k$ , the iteration is either a trust region iteration or a model iteration. If  $h_k$  is “too short”, then this is taken as an indication that the model must be improved, and the iteration becomes a model iteration. Otherwise, the iteration is a trust region iteration.

In a typical trust region step, where  $f(x_k^+) < f(x_k)$ , a point in  $Y$  is replaced by the new point  $x_k^+$ . This essentially moves the trust region towards  $x_k^+$ , and away from the removed point. As  $x_k^+$  is the point with the currently lowest  $f$ -value, it makes sense to do so, as the next minimization of  $m_k$  will start there.

In a model iteration, the point  $x_k^+$  will not be added to the model. Instead a new point will be calculated and replace the point furthest from  $x_{opt}$ . The new point is chosen in such a way that it improves  $m_k$ , see Sections 12.2 and 12.3.3.

The algorithms have several parameters related to the trust region radius  $\Delta$ . The value  $\rho_{beg}$  is used to initialize the current radius  $\rho_k$  and also provides an upper limit. As the value of  $\rho_k$  shrinks during execution,  $\rho_{end}$  is a lower limit. The purpose of  $\rho_{beg}$  is also to keep enough distance between the interpolation points in the initial model so that it is still accurate in cases of errors in the evaluation of  $f$ . There is also the normal trust region radius  $\Delta_k$  that is used to limit the step length, in the same way as in other trust region algorithms. The value of  $\Delta_k$  is affected by the value of  $\rho_k$  during the execution.

### 13.3.2 NEWUOA Pseudocode

In this section we provide a more detailed explanation of the NEWUOA algorithm, but for a complete description with all details of the algorithm, the reader is referred to [60]. NEWUOA uses the terms specified in Table A on page 311, and in Table 13.1 on page 172.

The numbers below refer to the numbered boxes in Figure 13.3. Boxes 2–6 are performed in order and together form a trust region step, while most of the other boxes together form a model-improvement step.

1. Using the input, the set  $Y$  is determined and used to construct the initial model  $m_k$  using Lagrange polynomials. The starting point is set to the point in  $Y$  with the lowest objective function value:  $x_k \leftarrow \operatorname{argmin}\{f(y_i), y_i \in Y\}$ . Also,  $k \leftarrow 0$ .
2. Solve the trust region subproblem  $\min m_k(x_k + h_k)$  subject to  $\|h_k\| \leq \Delta$ , for some value of  $h_k$  that is found here. The minimization is done through the use of a truncated conjugate gradient method.

Let  $x_k^+ \leftarrow x_k + h_k$ . If  $\|h_k\| < \Delta_k$ , i.e.  $x_k^+$  is not on the trust region border, then  $f$  has a positive curvature in all search directions. In such a case, the variable  $c_{min}$  is set to the least curvature of  $m_k$ . The variable  $c_{min}$  is not used here, but is set here for efficiency reasons. It is possibly used in box 14 later during the execution.

3. Here, only the condition  $\|h_k\| \geq \frac{\rho_k}{2}$  is checked. If the condition is satisfied, then execution continues in box 4, otherwise execution continues in box 14. The reason is that for such a short step, the model must most likely be updated.
4. First  $f(x_k^+)$  is evaluated and then the ratio  $r_k$  is calculated according to Equation 12.23.

Also, the interpolation points of  $m_{k+1}$  are determined here. Typically, the point  $x_k^+$  is added to  $m_{k+1}$ . The point  $x_r$ , to be removed from  $Y$ , is determined.

However, if no improvement of the objective function value is achieved and certain other conditions are fulfilled, then the model remains unchanged at this point. In that case,  $x_r$  is nil. See [60] for more details of when this happens.

5. Update the model. If  $x_r$  is not nil, then  $x_r$  is removed from  $Y$  and the model is updated such that  $m_k$  interpolates  $f$  in the point  $x_k^+$  instead of in  $x_r$ . If  $f(x_k^+) <$

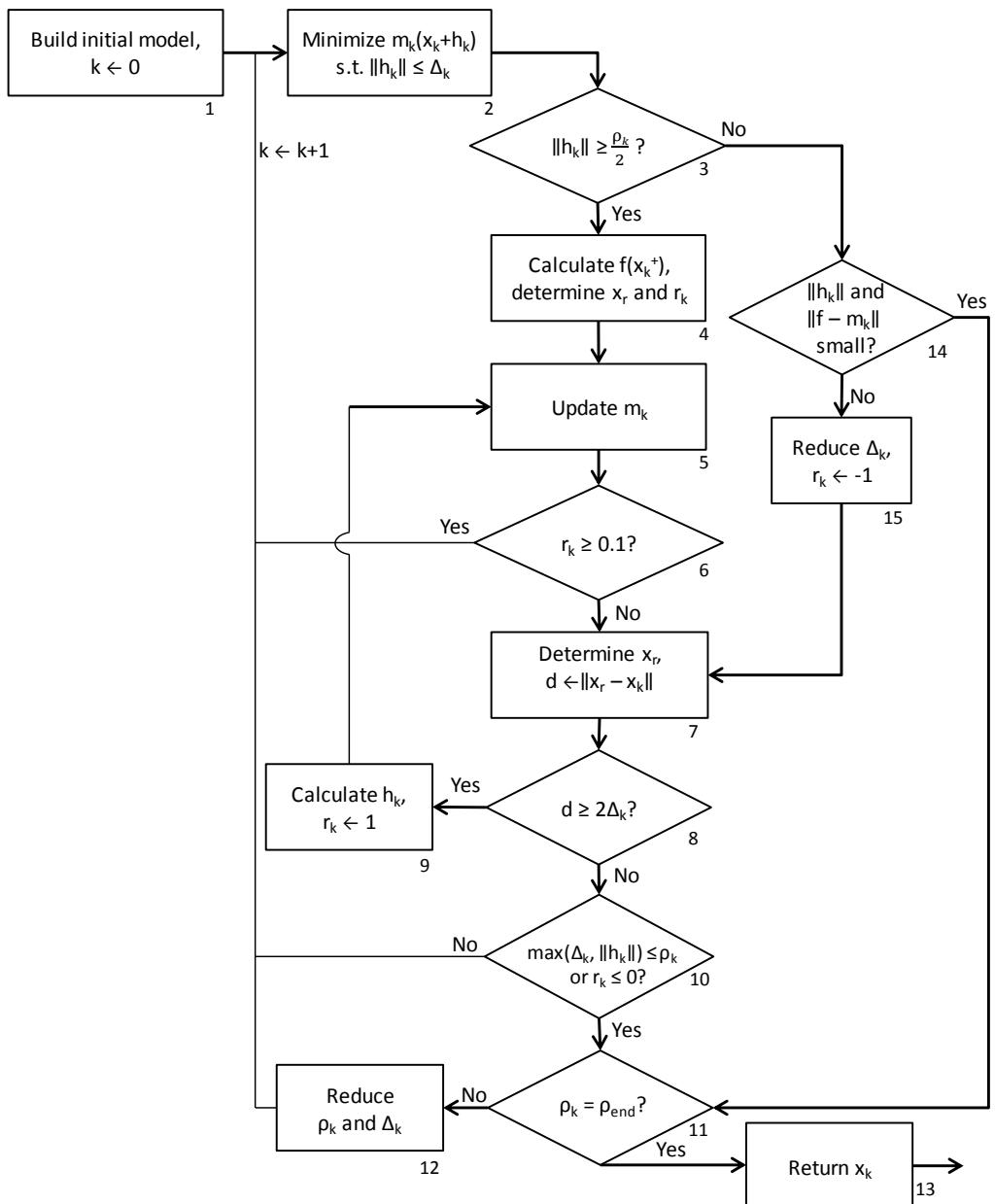


Figure 13.3: Flowchart for the NEWUOA algorithm.

$f(x_k)$ , then  $x_k \leftarrow x_k^+$ , else  $x_k$  remains unchanged. Thus the next iteration will start in the point with the lowest objective function value.

6. If  $r_k$  is sufficiently large, execution continues with another trust region iteration in box 2. Otherwise, execution continues to box 7.
7. Execution is transferred here if the actual improvement in objective function value is considerably less than the predicted improvement or if the requirements in boxes 3 and 14 were not fulfilled. Typically this occurs if the positions of the interpolation points in  $Y$  are unsuitable for maintaining a quadratic model. This can happen when some of the distances  $\|x_k - y_i\| \gg \Delta_k$ , for some  $y_i \in Y$ .

The algorithm here finds the point  $x_r$  that maximizes the distance to  $x_k$ . If  $\|x_r - x_k\| \geq 2\Delta_k$ , then it is believed that the quality of  $m_k$  can be significantly improved by the removal of  $x_r$ . Set  $d \leftarrow \|x_r - x_k\|$  for use in box 8.

Note that no replacement is performed here, only the point to possibly be replaced is determined here.

8. If  $d \geq 2\Delta_k$ , then execution continues in box 9, otherwise it continues in box 10.
9. The point  $x_r$ , which was determined previously in box 7, is going to be replaced by a new point  $x_k + h_k$ , for a new value of  $h_k$ , that is calculated here, subject to  $\|h_k\| \leq \Delta_k$ . The value of  $h_k$  is determined in such a way that it helps the conditioning of  $m_k$ . The actual replacement takes place in box 5, in which execution continues after it is finished here.

To assure that the new model is used directly after the update, the assignment  $r_k \leftarrow 1$  is done here, for use in box 6.

10. If either of the conditions  $\max(\|h_k\|, \Delta_k) \leq \rho_k$  or  $r_k \leq 0$  hold, then execution continues in box 11. Otherwise execution continues in box 2.
11. Is  $\rho_k = \rho_{end}$ ? If so, that means that we are basically finished and in that case, execution continues in box 13, otherwise it continues in box 12.
12. Reduce the trust region radii  $\rho_k$  and  $\Delta_k$ , but first set  $\rho_{old} \leftarrow \rho_k$ . The radius  $\rho_k$  is reduced by a factor of approximately 10. Then  $\Delta_k \leftarrow \max(\frac{1}{2}\rho_{old}, \rho_k)$ .
13. When execution comes here, the execution will soon end. However, if execution came here from box 14, via box 11, then  $f(x_k + h_k)$  has not been calculated and this must be done at this point. If  $f(x_k + h_k) < f(x_k)$ , then  $x_k \leftarrow x_k + h_k$  is assigned. Finally  $x_k$  is returned and the execution terminates.
14. Execution comes here only from box 3, i.e. the condition  $\|h_k\| < \frac{\rho_k}{2}$  was fulfilled. In that case, the model  $m_k$  is convex. Under the assumption that  $c_{min}$  is a useful

estimate of the least eigenvalue of  $H$ , then no calculation of  $f(x_k + h_k)$  is performed. The reason for this is that the predicted reduction in function value of the model:  $m_k(x_k) - m_k(x_k + h_k)$ , is believed to be less than  $\frac{\rho_k^2 c_{min}}{8}$ .

If the conditions  $\|h_k \leq \rho_k\|$  and  $\|m_k(x_k + h_k) - f(x_k + h_k)\| < \frac{\rho_k^2 c_{min}}{8}$  both hold for the current value of  $\rho_k$  and the last three pairwise values of  $h_k$  and  $x_k$  then it is believed that the model is accurate. Furthermore, it is believed that any attempt to try to improve the accuracy of the model would be a waste of effort and in that case, execution continues in box 11. If the above conditions are not fulfilled, execution continues in box 15.

15. The radius  $\Delta_k$  is decreased by a factor of about 10 if possible. If this is not possible, this is because this decrease would make the value to be less than the lower limit  $\rho$ . In such a case,  $\Delta_k \leftarrow \rho_k$ . Finally,  $r_k \leftarrow -1$  and execution continues in box 7.

### 13.3.3 The BOBYQA Algorithm

BOBYQA (Bounded Optimization BY Quadratic Approximation) is an extension of NEWUOA, that also can handle bounds [61]. It is in many ways similar to NEWUOA, with the exception that when NEWUOA performs the optimization within the trust region, BOBYQA performs this optimization in the intersection between the trust region and the hyper rectangle formed by the bounds. The user may set the size of the interpolation set, just like in NEWUOA.

### 13.3.4 The LINCOA Algorithm

The latest development by Powell is the LINCOA algorithm, where the name stands for LINearly Constrained Optimization Algorithm. It can be seen as a further extension of BOBYQA in the sense that it handles both bound and linear constraints. Like in NEWUOA and BOBYQA, the size of the interpolation set can be set by the user. At the time of writing, there is no paper describing the algorithm, only the code and release notes are available [62].

### 13.3.5 Extensions

NEWUOA has also been subject to further research and extensions. Aroux t et al. [2] modified the algorithm to use the max norm when solving the trust region subproblem and an active set strategy to handle bound constraints. In a majority of the reported test cases, the modified algorithm performed better than BOBYQA.

## 13.4 Summary

The existing model building algorithms presented here: DFO, UOBYQA and NEWUOA have several things in common but also several differences.

Where UOBYQA always uses a quadratic model with  $\frac{1}{2}(n+1)(n+2)$  points, DFO can begin with a linear model using  $n+1$  points and then additional points can be added during execution until a full quadratic model is created. The user can set the number of interpolation points in NEWUOA within an interval, but once it is set, the number is fixed for the duration of the execution. If fewer points than a full quadratic model are used, the remaining degrees of freedom are taken up through calculating the difference between two consecutive model Hessians and then minimizing the Frobenius norm of this difference.

All algorithms minimize the model within a trust region; UOBYQA uses a version of the Moré-Sorensen algorithm, and NEWUOA uses a truncated conjugate gradient method in order to save time for high-dimensional problems.

DFO decides whether to include the candidate point  $x_k^+$  into the interpolation set depending on the ratio between achieved decrease in the objective function value and the predicted decrease, whereas the other algorithms always include  $x_k^+$ , regardless of whether the point yielded a low objective function value or not. All algorithms determine the size of the trust region radius using the aforementioned ratio. UOBYQA and NEWUOA also employ certain model iterations, with the sole purpose of finding points that are added to the model in order to improve it.

All three algorithms have been significantly extended, both by the original authors themselves as well as by others, and represent the state of the art in model building algorithms for derivative-free optimization. The original DFO algorithm and Powell's algorithms are available for download online.



# 14

---

## Radial Basis Functions

Another type of model building algorithm that is often referred to by the name from the basis functions that are used to build the interpolation models, is Radial Basis Functions (RBFs). In themselves, RBFs are a way to build a model of a function, and are used in many other areas than optimization.

When used in an optimization context, RBFs typically consist of two closely connected parts: one part for model building, and an optimization algorithm, similar to the other types of model building algorithms presented in the previous chapters. Here we first discuss the model building part and later we discuss the optimization algorithm.

### 14.1 Model Building for Radial Basis Functions

The main differences between model building in the RBF context compared to the model building algorithms in previous chapters are that RBFs build a single global model of the function and that the set of interpolation points  $Y$  may contain an arbitrary number of points.

A set  $Y$ , consisting of positions  $x_1, \dots, x_{p_1}$  with the corresponding objective function values  $f(x_1), \dots, f(x_{p_1})$ , is used in the model building part. The  $\lambda$ -variables  $\lambda_1, \dots, \lambda_{p_1} \in R$  and  $p \in \prod_m^n$ , which is the space of polynomials in  $n$  variables, of degree less than or equal to  $m$ .  $\phi(r)$  is one of the different types of functions from Table 14.1.

The radial basis function interpolation  $m$ , has the form

$$m(x) = \sum_{i=1}^{p_1} \lambda_i \cdot \phi(\|x_i - x\|_2) + p(x). \quad (14.1)$$

The parameters  $\lambda, a$  and, when occurring,  $b$ , are obtained through solving the system of linear equations specified in Equation 14.2. In Equation 14.2,  $\Phi$  is an  $p_1 \times p_1$

RBF	$\phi(r) > 0$	$p(x)$
linear	$r$	$a$
cubic	$r^3$	$b^T x + a$
multiquadric	$(r^2 + \gamma^2)^{1/2}, \gamma > 0$	$a$
inverse multiquadric	$(r^2 + \gamma^2)^{-1/2}, \gamma > 0$	$a$
Gaussian	$e^{-(\gamma r)^2}, \gamma > 0$	$a$
thin plate spline	$r^2 \log r$	$b^T x + a$

**Table 14.1:** Different radial basis functions.

matrix containing  $\Phi_{ij} = \phi(\|x_i - x_j\|_2)$  while  $P, \lambda, c$  and  $y$  are specified according to Equation 14.3. Each  $x_i^T$  is a vector of dimensionality  $n$ .

$$\begin{pmatrix} \Phi & P \\ P^T & 0 \end{pmatrix} \begin{pmatrix} \lambda \\ c \end{pmatrix} = \begin{pmatrix} y \\ 0 \end{pmatrix} \quad (14.2)$$

$$P = \begin{pmatrix} x_1^T & 1 \\ x_2^T & 1 \\ \vdots & \vdots \\ x_{p_1}^T & 1 \end{pmatrix}, \lambda = \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_{p_1} \end{pmatrix}, c = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \\ a \end{pmatrix}, y = \begin{pmatrix} f(x_1) \\ f(x_2) \\ \vdots \\ f(x_{p_1}) \end{pmatrix}. \quad (14.3)$$

If the rank of  $P$  is  $n + 1$ , the leftmost matrix in Equation 14.2 is non-singular and Equation 14.2 has a unique solution [55]. Thus there exists a unique RBF interpolation function  $m$  to  $f$ , defined by the points in  $Y$  [32, 57].

When used in RBFs, there is no fixed size of  $Y$  but Equation 14.2 becomes more computationally demanding to solve as  $Y$  grows. Also, if there are several points very close to each other, then the leftmost matrix in Equation 14.2 approaches linear dependence. For these reasons, points may be removed from  $Y$  in order to decrease the computational burden and improve the conditioning of the system. This has led to research aimed towards improving the conditioning of the system, see e.g. [4, 39].

There is a wealth of information about RBFs, for more extensive coverage the reader is referred to Buhmann's book [9] and the references therein.

## 14.2 Optimization Algorithms for Radial Basis Functions

Once the model is created, an optimization algorithm is applied in order to find the best value. As a global model is built, it is common to apply a global optimization algorithm in order to find the globally best value of the model. This is a very difficult problem since the

model may be nonconvex and there might be non-linear constraints involved. Examples of methods that can be used for solving the optimization problem are QP-methods and SQP-methods. See Fletcher [23] for more information about such methods. Another option is e.g. augmented Lagrangian methods [52, 3]. As there is no requirement to use a specific solver in the optimization of an RBF, any optimization algorithm capable of solving the problem can be used. Since the problem optimization is very difficult, commercial solvers are often used. Commonly used software packages in this context are SNOPT [27] and KNITRO [10].

## 14.3 Comparison With Other Model Building Algorithms

In this section, we compare and contrast RBF with the model building algorithms discussed in previous sections, which use quadratic interpolation models. There are several similarities between such algorithms and RBFs. Both types of algorithms build the model  $m$  as a sum of simpler functions. But where the algorithms described in Chapter 13 are limited to using first or second-degree polynomials, RBFs have a greater choice of functions to use to build the model from.

Both types of algorithms separate the model building part and the optimization part and can, at least in theory, apply any optimization algorithm that is capable to solve the problem under consideration.

While there are similarities between RBFs and the other kind of model building algorithms, there are also a number of differences. In RBFs, there is no upper limit on the number of points used to build the model. However, this increases the size of the system specified in Equation 14.2. As the effort required solving the system increases with size, it can be beneficial to remove some points from  $Y$  for efficiency reasons. However, the efficiency concern should be contrasted with the other model building algorithms, which have a strict upper limit on the cardinality of  $Y$  in order to create a unique interpolation polynomial.

RBFs build and update a single global model of  $f$ , while the other algorithms build a local model that is sequentially updated as the search progresses through the search space. The local model only cover a part of the search space at any one time, which is a distinct difference from RBFs. The fact that the model used by the RBFs is global also affects the choice of optimization algorithm. Other algorithms typically consider the model to be unreliable outside the trust region, while RBFs impose no such restriction.

RBF lends themselves naturally to parallelization, as there is no upper bound on the cardinality of  $Y$ . Thus one can imagine an RBF implementation that evaluates  $f$  in order to get better objective function values, but at the same time evaluates  $f$  at points where it believes that  $m$  is a poor model of  $f$ . To determine the latter points, one could use e.g. the methods from Section 15.6.1. In this case, both exploration and exploitation are performed at the same time.

Due to the more flexible model building, the additional exploration points do not require that any points are removed from  $Y$ . This process with evaluating points in order to improve the model can continue until it is believed that  $m$  is a good global model of  $f$ .

## 14.4 Summary

Using Radial Basis Functions is another way of constructing an interpolating model given a set of points. It differs significantly from the other model building algorithms in that an arbitrary number of points can be included in the interpolation set. From this, a single global model is created and then updated as execution progresses. The increasing size of the interpolation set as well as the fact that many points may be close to each other mean that solving the equation system to determine the interpolation model may be increasingly difficult.

Since RBFs interpolate all points, the model is often nonconvex, which makes finding an optima much more difficult compared to the model building algorithms presented in previous sections. Since few organizations have the software necessary for solving this kind of problems, third-party commercial solvers are often used in the optimization part of the algorithm.

# 15

---

## Parallelism in Algorithms for Derivative-Free Optimization

Just like many other algorithms, the algorithms for derivative-free optimization are inherently sequential. In each iteration, they use the best point as starting point for the optimization, then a new point is generated, that hopefully has a better value. In the next iteration, the optimization continues from that point and so on. Even from this short algorithm description, it is evident that such algorithms do not take advantage of several computers for parallel execution and evaluation. In this chapter we present some ways in which existing algorithms for derivative-free optimization such as the ones described in Sections 13.1–13.3 can be parallelized and extended to be more suitable for the case where there are several computers available for evaluating the objective function.

Considering the characteristics of the problems that we are focusing on, that were presented in Section 10.3, we will present extensions that are especially appropriate for multimodal problems, where we are interested in finding several, possibly all, of the local optima. However, the ideas presented here are applicable in other cases as well. We assume that evaluation of the objective function takes a long time and at that we want to minimize the *clock time* until the algorithm converges. If there is a limit on the number of function evaluations, we could perform all of them sequentially. However, since this would take a long time, we prefer to perform several evaluations in parallel, in order to decrease the clock time until the algorithm is finished.

Parallelism exists in contemporary computers on several levels, ranging from instruction-level parallelism to task-level parallelism. However, a user has only access to some of these levels of parallelization. In general, there are three things to parallelize in the kind of optimization algorithms that we consider here: the evaluation of the objective function  $f$ , the optimization process itself at a higher level, and the linear algebra calculations in each iteration [70]. Here we will discuss all of these in turn.

If we decide to use an existing linear algebra package, then we have limited control of

how the calculations are performed, and we can do very little to influence this part of the algorithm. However, many linear algebra packages are explicitly written with parallelization in mind, and often take advantage of the CPU's inherent parallelization capabilities without any involvement by the user.

Also, for the kinds of problems that we are interested in, the time spent performing linear algebra calculations is very small in comparison to the time spent evaluating the objective function. Therefore, decreasing the calculation time for linear algebra calculations will only give a minimal performance improvement overall.

Control of the optimization process itself on a higher level is important, and can potentially yield large performance improvements if done correctly. This is the topic of Section 17.

The parallelization that is most likely to give a large performance improvement is parallelization of the evaluations of the objective function. Here we must consider parallelization on several levels. On a low level we consider how each evaluation is performed. Is each evaluation itself distributed over several computers or just one? On each computer, does it use one, a few or all available CPU cores? Regardless of the answer, it is unlikely that we can influence this since many simulators are standalone products that do not allow the user to affect the use of computer resources. For now we assume that each function evaluation uses all available computational power on a single computer.

What we have control over is how we use the available computers. In the ideal case, we would like to use all available computers for the complete allotted time or until the available function evaluations are exhausted, and at that time, we release all used computers. To perform evaluation of several points in parallel of course requires that several points are generated, and different ways that this can be performed will be discussed later in this chapter.

Since we discuss general parallel extensions here, we place no requirements about the implementation of parallelization. For now, we assume synchronized evaluation of the points. This means that all points that are sent for evaluation at any one time wait for all other points sent for evaluation at the same time, to be finished, before execution continues. This is suitable when function evaluations of different points take approximately the same time. For information about an example of how parallelization can be implemented when this is not the case, we refer the reader to Section 18.7.1.

## 15.1 Parallelization Terms

We assume that  $C$  computers are available to perform evaluations of the objective function in parallel. Since several function evaluations can be performed in parallel, we are not interested in minimizing the *number* of function evaluations ( $NE$ ), but rather in minimizing the number of *times* one or more points are evaluated. We call this the number of parallel evaluations ( $NPE$ ). If we evaluate two batches, with 10 points in each, then  $NE = 20$  and  $NPE = 2$ . Since these are performed in parallel, this is faster than evaluating 4 points

sequentially. Not only have we received more information about the objective function, but we have used only half of the time.

Assume as given  $O$  different starting points. How  $O$  and  $C$  relate to each other may vary in different situations. At first sight, it might seem obvious at  $O \geq C$  should always apply since some resources would otherwise be unused. This makes sense if each starting point generates a single point, but if several points are generated by some starting point, then it might be desirable to keep  $O \leq C$ , to make sure that all points can be evaluated in parallel.

From now on, each starting point will be implemented as an *optimization run*, which will be described next.

## 15.2 Optimization Runs

In order to keep the basic structure of the algorithms unchanged, we introduce the concept of an optimization run. Each instance of an optimization run contains everything that may change during the execution such as starting point, trust region radius, interpolation model, etc. In essence, each optimization run is one start in a multistart algorithm, and attempts to solve the original optimization problem, but with different parameters. However, an optimization run is not limited to just representing a start in a multistart algorithm; all optimization runs may have the same starting point, but should differ in at least one parameter value.

By using several simultaneous optimization runs, that each generates a point for optimization, it is possible to utilize the computational resources more efficiently as this allows us to evaluate several points in parallel.

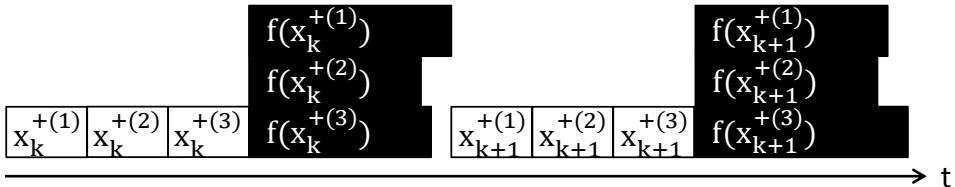
While using several optimization runs and not doing any further changes can be an improvement, it is possible to let the optimization runs take advantage of each other's information during execution and hopefully further improve the total performance. Some ways to do this will be examined in Chapter 16.

To distinguish between the optimization runs, we assume that there are  $O$  optimization runs currently executing, each with their own model  $m^i$ , starting point  $x_0^i$ , trust region radius  $\Delta^i$ , etc.,  $1 \leq i \leq O$ . In this thesis, such parameters will only be superscripted when several are discussed, and it is necessary to separate them from each other.

The concept of optimization runs also helps with implementing the algorithms according to our implementation principles, described in Section 18.3. In our implementation, each starting point is represented by an optimization run, and where the meaning is clear, we will use the terms interchangeably.

Figure 15.1 displays an example of three optimization runs running in synchronous parallelization mode. The optimization runs are indicated by their respective candidate points  $x_k^{+(1)}, x_k^{+(2)}$  and  $x_k^{+(3)}$ . They execute in sequence as indicated by the boxes along the timeline.

In iteration  $k$ , the optimization runs execute, and each yield a single point for eval-



**Figure 15.1:** Overview of synchronous parallelization.

uation. The objective function is then evaluated in each one of these points. The time required for each evaluation may take slightly different time, but the algorithm must wait until all points have finished evaluating until it can continue.

When all evaluations are finished, the optimization runs continue to execute, in iteration  $k + 1$ . Each one yields a new point, which is evaluated in parallel. The algorithm continues like this, alternating between sequential calculations of points, and parallel evaluations of points, until it is finished. Of course, it might happen that some optimization run converges before the others, and in that case the other continue until they are finished.

### 15.3 Building the Initial Model in one Step

In this and the following sections we will discuss different ways to use parallelization in order to improve performance. We will begin from the beginning of the algorithm and continue throughout the execution.

When the initial model in model building algorithms is created, it is common to first place a set of interpolation points, evaluate them and use their values to determine where to place the remaining points. This is typically done to try to increase the chances of getting low objective function values for points in the initial model. As an example, assume that two points differ only in their  $x_1$  and  $x_2$ -coordinates, where increasing any of the coordinates provides a better objective function value. Then, the algorithm might place another point by increasing both the  $x_1$  and the  $x_2$ -values.

This procedure is reasonable since it can yield an initial model consisting of points with lower values, but it is not necessary in order to achieve good total performance for the algorithm. The disadvantage of this procedure is that it requires that points are evaluated in several (at least two) batches, which increases the time before the model is ready for use.

One option to alleviate this problem is to first place all points and then evaluate them in parallel. This makes it possible to evaluate all points in one batch, assuming that a sufficient number of computers are available.

However, even if sufficiently many computers are available, it is not certain that parallel model building is better than sequential model building. A counter argument against parallel model building could be that sequential model building could possibly yield an

interpolation set  $Y$  that is better poised for interpolation. Looking at Equation 12.4 on page 160, we see that parallel model building replaces some of the interpolation points in  $Y$  and the corresponding  $\phi$ -polynomials will have different values, as compared to sequential model building. While this will affect the value of the determinant (see the discussion of poisedness in Section 12.2 for the significance of this), it is difficult to determine in advance how the determinant will be affected. Naturally we presume that we do not allow the points in  $Y$  to become linearly dependent, i.e.  $Y$  will still be poised.

Since we cannot easily a priori determine the effect of adding a certain point to  $Y$ , it can be worthwhile to place all initial points before any of them are evaluated, in order to decrease the time before the model can be used and the optimization can start.

## 15.4 Non-spherical Trust Region Shapes

Once the initial model has been created, the actual optimization can start. Model building algorithms for derivative-free optimization typically use a hyper spherical trust region. They allow for steps of equal length in all directions. However, one can easily imagine using trust regions shaped in other ways, where an obvious choice is the more general ellipse. If the optimization part of the algorithm has calculated steps in a certain direction in several iterations, then a rational choice would be to allow longer steps in that direction in the future.

Another situation where one would like to allow the algorithm to take longer steps is when the ratio between achieved and predicted improvement (see Equation 12.23) differs significantly in different directions, i.e. there is a significantly smaller difference between  $m_k$  and  $f$  in one direction than in another.

In this section we present some possible ways to create non-spherical trust regions. Recall from Section 12.3 that the algorithms for solving the trust region problem work within a spherical trust region. Thus, to be able to use the existing algorithms, we must determine a variable transformation from the original variable space  $X$  with a (possibly elliptical) trust region to a variable space  $X'$  with a circular trust region, where the trust region subproblem will be solved.

Assume as given a transformation matrix  $T \in R^{n \times n}$ . Then a transformation of point  $x \in X$  to  $x' \in X'$  is performed as  $x = Tx'$ . Consequently  $x' = T^{-1}x$  for the inverse transformation. Similarly for a step  $h \in X$ . Naturally, the model must also be transformed: with  $m$  as the model in  $X$ , let  $m'$  denote the transformed model in  $X'$ . The quadratic model from Equation 12.2 then becomes

$$m'(x + h) = f(x) + (Th')^T g + \frac{1}{2}(Th')^T H(Th'). \quad (15.1)$$

Even though we are concerned with unconstrained optimization, we present transformations of constraints for completeness. Any linear constraints are transformed as  $A'x' \leq b$  with  $A' = AT$ . Bounds like  $x \geq b_l \in X$  shall correspond to  $x' \geq b'_l \in X'$ . Since  $x' = T^{-1}x$ , we get  $b'_l = T^{-1}b_l$  and analogously for upper bounds.

Non-linear constraints are more difficult to handle; either a symbolic representation is required, that allows transformations similar to the above, or a point  $x' \in X'$  must be inverse transformed to  $X$  before evaluation in any non-linear constraints can be performed.

Having showed that we can perform the transformation, the obvious question is then how to calculate the transformation matrix  $T$  and this will be the topic of the next few sections. Since these modifications are intended as extensions to existing algorithms, we keep the existing mechanism for changing the trust region radius and, in each iteration, the shortest semi-axis,  $l_{\min}$ , will be equal to the trust region radius. Thus, using a non-spherical trust region shall be at least as permissive as a spherical trust region.

### 15.4.1 Static Trust Region Shapes

The simplest choice of an elliptical trust region is to decide the directions in which the ellipse will extend and use a static shape throughout the execution of the algorithm. Here there are two choices that must be made: in which directions should the ellipse extend and how oblong it should be. Neither of these questions have any obvious answers.

Regarding the directions of the ellipse, unless one has some idea about in which direction the objective function value improves, one cannot choose a single direction that guarantees improvement. In that case, one can exchange the single optimization run with a spherical trust region for several optimization runs with elliptical trust regions, all with the same starting point but with the ellipse extended in different directions. Since we assume that several points can be evaluated in parallel, this exchange does not necessarily increase the execution time. With  $C$  as the number of available computers, we can use  $\min(C, n)$  different optimization runs and still be able to evaluate them in parallel.

The shape of the ellipsis is also difficult to determine in advance. A simple option is to set the relative lengths of the semi-axes, such that the shortest distance,  $l_{\min}$  is one, and the longest is  $l_{\max} \geq l_{\min}$ , with all other  $l_{\min} \leq l_i \leq l_{\max}, 1 \leq i \leq n$ . Together with  $\Delta$  this determines the actual size of the ellipse. E.g. with  $n = 2$ ,  $\Delta = 3$ ,  $l_{\min} = 1$  and  $l_{\max} = 2$ , then the distance from the trust region midpoint, along the semi-axes, to the closest and the farthest points on the ellipsis, are 3 and 6 respectively.

By setting the relative size of the ellipse, instead of the absolute size, we do not have to change anything related how  $\Delta$  changes during execution, since it is affected by  $\rho$  as before.

### 15.4.2 Dynamic Trust Region Shapes

The static trust region shapes have the obvious disadvantage of being unable to adjust to changing circumstances, e.g. a distinct change in step direction. Using a dynamic trust region shape offers a way around this, but instead one must solve the problem of how to calculate the transformation matrix  $T$ . There is a plethora of options, and exploring them all is well beyond the scope of this thesis. Here we will present a few of the available options.

**Switching Axis-Aligned.** As mentioned earlier, a big disadvantage of the static axis-aligned elliptical trust regions is that we do not know which alignment is beneficial. However, one way to use the axis-aligned elliptical trust regions even though we do not know which one to prioritize is to switch axis-alignment regularly, e.g. in every iteration. This way, if the problem benefits from a certain alignment, this alignment will occur every  $n$ th iteration, and in the other iterations, there should hopefully be no big disadvantage from using the other alignments.

**Adaptive To Steps Using Principal Component Analysis.** The idea of this transformation is to adapt the shape of the trust region to e.g. the previous steps that the algorithm has taken. Here we use the term step loosely, and the algorithm can be used when step represents something other than  $h_k$ , i.e. something else than the solution of the trust region subproblem. An obvious example is to use the movement of the currently best point as the step, so that the shape of the trust region only changes when a better point is found.

We can achieve this by creating a matrix consisting of previous steps, and then calculate the eigenvalues and eigenvectors of this matrix. The eigenvectors will be the basis vectors in a new coordinate system and the eigenvalues will be the variation of the steps along each such new axis.

Since we have defined the transformation matrix  $T$  to transform the function from  $X$  with a possibly elliptical trust region to  $X'$  with a circular trust region, we here calculate and use its inverse,  $T^{-1}$ , to perform the inverse transformation.

Through eigendecomposition, any square matrix with linearly independent columns, e.g.  $T^{-1}$ , can be decomposed as  $T^{-1} = V L V^T$ , where  $L$  is a diagonal matrix with the eigenvalues of  $T^{-1}$  as diagonal elements, and  $V$  is an orthonormal matrix with the eigenvectors of  $T^{-1}$  as columns.

Assume that the eigenvalues are sorted such as  $\lambda_1 > \lambda_2 > \dots > \lambda_n$  and that column  $i$  in  $V$  is the eigenvector that corresponds to  $\lambda_i$ . With the eigenvalues and vectors in that order,  $T^{-1}$  corresponds to a rotation of the coordinate system in such a way that the rotated coordinate system's first axis is aligned in the direction that the steps have the greatest variance. The second axis is orthogonal to the first axis and is aligned in the direction that the steps have the second highest variance in, and so on. The new axes define the basis vectors of the transformed space.

This transformation that is presented below is known under different names in different research areas, such as Karhunen-Loëve Transform, Hotelling transform and Principal Component Analysis (PCA). Such transforms are often used in image analysis and image compression and we refer the reader to the books by Bishop [6], Fukunaga [26], Sayood [67] and Dony [19] for more information.

As can be seen in the pseudocode in Figure 15.2, the transformation matrix is built iteratively. The main reason for this is that the kind of transformation used here is typically used when a large number of steps has been calculated and saved. In our case, this means that we would use some static shape until a sufficient number of steps had been taken

before we could calculate the desired transformation matrix and make the change to an elliptical trust region.

Instead we propose to use a convex combination of the existing  $T^{-1}$  and a rank one matrix created from the step  $d_k$ , i.e.  $d_k d_k^T$ . To determine the relative weights of the existing  $T^{-1}$  and the addition  $d_k d_k^T$ , a forgetting factor  $\gamma \in [0, 1]$  is used. A higher value of  $\gamma$  prioritizes the current step where  $\gamma = 1$  completely ignores the existing  $T^{-1}$ . Similarly  $\gamma = 0$  keeps the old matrix as  $T^{-1}$ . A value of  $\gamma < 1$  will also mean that  $T^{-1}$  will be combination of several steps, where older steps will make a smaller impact on the transformation matrix.

```

1   $d_k \leftarrow \frac{d_k}{\|d_k\|}$ 
2   $T_k^{-1} \leftarrow (1 - \gamma)T_k^{-1} + \gamma d_k d_k^T$ 
3   $(L, V) \leftarrow \text{Calculate-eigensystem}(T_k^{-1})$ 
4   $(L, V) \leftarrow \text{Regularize}(L, V)$ 
5   $T_{k+1}^{-1} \leftarrow VLV^T$ 
6   $T_{k+1} \leftarrow \text{Invert}(T_{k+1}^{-1})$ 
7   $k \leftarrow k + 1$ 
```

**Figure 15.2:** Updating  $T$  based on the step  $d_k$  and the eigensystem of  $T^{-1}$ .

The pseudocode in Figure 15.2 updates  $T$  based on the step  $d_k$  and the eigensystem. Assume as input the step  $d_k$ . Let  $T_k^{-1}$  denote the inverse transformation matrix in iteration  $k$  and let  $T_0^{-1}$  be initialized to the identity matrix. On line 1, the step  $d_k$  is normalized. The normalized step is then used to create a symmetric matrix through  $d_k d_k^T$ . This matrix is used to create a convex combination of it and the existing  $T_k^{-1}$ , using the forgetting factor  $\gamma$  (line 2).  $T_k^{-1}$  will always be a symmetric matrix since a convex combination of two symmetric matrices must always be symmetric. Line 3 calculates the eigensystem of  $T_k^{-1}$ , yielding the eigenvalues and the eigenvectors. Since  $T_k^{-1}$  is symmetric and real, the calculation of the eigensystem will always succeed and the output will be real valued. Then regularization will take place, meaning that the matrices of eigenvalues and eigenvectors might be modified to avoid an ellipse that is too elongated in any direction. After regularization,  $T_{k+1}^{-1}$  is set, using the new matrices (line 5). The next thing to do is to determine the inverse transformation matrix for the next iteration using the eigensystem. The second to last thing to do is to calculate the inverse of  $T_{k+1}^{-1}$  and assign it to  $T_{k+1}$ . Finally the iteration variable is incremented.

We mentioned earlier that the term step was used loosely here and the reason for this is that it is probably not suitable to use the normal step  $h_k$  as  $d_k$ . The reason for this is that  $h_k$  will be the result of an optimization of  $m_k$ , and we know that there will sometimes be a large difference between  $f$  and  $m_k$ . Thus will lead to steps where  $f(x_k^+) > f(x_k)$ , and using such steps would counteract what we want to achieve. Therefore, we suggest to only use  $h_k$  as  $d_k$  in the cases where  $f(x_k^+) < f(x_k)$ , as such steps have led to an

improvement in objective function value and allowing longer steps in that direction may be beneficial.

Another option for determining the step may be to keep track of for which  $h_k$  the ratio  $r$  (see Equation 12.23) has a high value, and use such steps as  $d_k$ . Not only would such a calculation implicitly keep track of improvement of the objective function value, but also measure the difference between  $f$  and  $m_k$ , leading to longer allowed steps in direction in which there is a small difference between  $f$  and  $m_k$ .

We can see that the update of the transformation that we use (line 2 in Figure 15.2) is somewhat similar to the calculation of the matrix  $R_\alpha$ , that is used to update the transformation matrix in the r-algorithm (see Section 11.4.1). The matrix  $R_\alpha$  is created through addition of a rank one matrix to the unit matrix. Where we create a rank one matrix and create a convex combination of it and the previous transformation matrix to get the new transformation matrix, the r-algorithm creates the new transformation matrix as a matrix multiplication between the old transformation matrix and the inverse of  $R_\alpha$ . Both methods include some of the old existing transformation matrix and add some influence of the new matrix which is created through the use of a new direction that is promising. The amount of influence of the new matrix is adjusted through the value of  $\alpha$  in the r-algorithm and the value of  $\gamma$  on our algorithm. Thus, the ways in which we create the new transformation matrix and how it is created in the r-algorithm are very similar.

**Circular Level Curves.** Another option is to attempt to transform the level curves of the model  $m_k$  with the intention to make them as circular as possible in the transformed space. The reason for this is to try to make the trust region subproblem easier to solve. This is not a new thought, cf. conjugated gradient methods [52]. This can be achieved through the transformation  $h'^T T^T H T h' = \psi I$ , where  $\psi$  is a scale factor  $> 0$  and  $I$  is the identity matrix. For simplicity, we choose  $\psi = 1$ .

However, transforming to get an identity matrix is only achievable if  $H$  is positive definite, and since we want to preserve the definiteness of the Hessian after transformation, we must allow for other options: if  $H$  is negative definite, then we want  $-I$  and if  $H$  is indefinite, then we want  $-1$  instead of  $1$  for each negative eigenvalue. Thus, instead of the identity matrix  $I$ , we allow a diagonal matrix  $E$  with elements  $\pm 1$ , with  $-1$  if  $H$ 's corresponding eigenvalue is negative, and  $1$  if is positive. When an eigenvalue is zero, then it will not be possible to get the desired transformation exactly. In that case, we want the corresponding element in  $E$  to be  $1$ .

To achieve the desired transformation we again use eigendecomposition of a matrix. E.g.  $H$  can be decomposed as  $H = V L V^T$  where  $V$  is a matrix with the eigenvectors of  $H$  as columns, and  $L$  is a diagonal matrix consisting of  $H$ 's eigenvalues. Then,  $h'^T T^T H T h' = h'^T T^T V^T L V T h'$  through the use of eigenvalue decomposition. We recognize that  $T^T V^T L V T$  should be equal to  $E$  and by choosing  $T = V |L|^{-1/2}$ , we get  $h'^T |L|^{-1/2} V^T V L V^T V |L|^{-1/2} h' = h'^T (|L|^{-1/2} L |L|^{-1/2}) h' = h'^T E h'$ . Here we have exploited that  $V^T V = I$  since  $V$  is orthonormal.

To achieve the requirement above that any zero eigenvalue should give a  $1$  in the  $E$

matrix, we do not use the normal matrix inverse but rather the Moore-Penrose generalized inverse of  $L$ , which achieves exactly this.

Thus, we choose the transformation matrix  $T = V|L|^{-1/2}$ . This choice of transformation matrix will maintain the definiteness of the Hessian and provide circular level curves if possible. s

### 15.4.3 Other Trust Region Shapes

So far we have only discussed different ways to create elliptical trust region shapes. However, there is nothing that prevents arbitrarily shaped trust regions, as long as it is possible to determine the required transformation and its inverse. Here one could use kernel principal component analysis (KPCA) to determine nonconvex trust regions. Since KPCA is a non-linear transformation, it would not be sufficient to use a single transformation matrix to perform the transformation. See Bishop [6] for more information about KPCA.

## 15.5 Generating Several Points in Each Iteration

Having several optimization runs will most likely improve the total performance of the optimization algorithm, but this can probably be further improved by letting each optimization run generate several points for evaluation in each iteration. This is especially evident when  $C < O$  since not all available computers are used in that case. Ideally, we would like to use all computers all the time.

In the following sections we discuss some ways in which we can let the optimization runs generate several points in each iteration. This extends and generalizes the algorithms discussed previously. From now on, we assume that each optimization run  $i, 1 \leq i \leq O$ , generates  $G_k^i$  points in each iteration  $k$ . So far in this thesis,  $G_k^i$  has been one, but here we explore options for generating several points in each iteration. How to generate these additional points is the subject of the below sections. However, the generation of several points will also create some additional things to consider in the algorithm and these consequences will be discussed in a later section.

To use the available computers as efficiently as possible, we recommend  $\sum_{i=1}^O G_k^i = C$  if possible. In that way, we always use all available computers in every iteration. We designate the candidate point generated by optimization run  $i$  in iteration  $k$  by  $x_k^{+(i)}$  and the additional points by  $x_k^{+(i,j)}$ , where  $1 \leq j \leq G_k^i - 1$ .

### 15.5.1 Solving the Trust Region Subproblem with Different Radii

One way to generate several points in each iteration is to solve the trust region subproblem with different values of  $\Delta$ . In this section we will discuss why this can be relevant.

When solving the trust region subproblem, there are two principally different cases that can occur. Either  $\|h_k\| = \Delta_k$  (a descent direction is found) or  $\|h_k\| < \Delta_k$  (a minimum is found inside the trust region).

In the first case, the optimization algorithm would like to take a long step since it has found a descent direction. However, the length of the step is limited by the trust region radius. In that case, it can be beneficial to also solve the trust region subproblem with radius  $\gamma_2 \Delta_k$  in the same iteration (see Section 12.3.1 for more information about  $\gamma_2$ ).

In the latter case, we have no indication that we will improve anything by using a shorter radius. Clearly the trust region subproblem indicates that the desired step length is sufficient, since it has found an optimum of  $m_k$ . Setting a shorter step length, i.e. solving the trust region subproblem with radius  $\gamma_1 \Delta_k$ , it would yield a point that the optimization algorithm believes is worse. The only reason for solving with  $\gamma_1 \Delta_k$  would be to generate an additional point for evaluation on a computer that would otherwise be unused.

While we could solve the trust region subproblem with the original radius  $\Delta_k$  and with the additional radii  $\gamma_1 \Delta_k$  and  $\gamma_2 \Delta_k$  in the same iteration, we choose to solve it with the original radius and  $\gamma_2 \Delta_k$ , for the reason discussed above.

A possible misconception about solving the trust region subproblem with different radii in the same iteration is that we somehow would “save” iterations. In reality, the models in the parallel and sequential cases will be different. If the problems are solved in different iterations, the resulting point  $x_k^+$  (from iteration  $k$ ) is included in the model in iteration  $k + 1$ . If we solve several problems in iteration  $k$ , then  $x_k^+$  (from iteration  $k$ ) is naturally not included when solving the trust region subproblem with  $\gamma_2 \Delta_k$  in the same iteration. The effect is that the models are different in the two cases, which may cause the optimization runs to take different paths through the search space in future iterations. Not only will the models be different, the optimization may also start in different points, since it always starts in the point with the lowest value.

Even if we do not know the long-term effect of solving several trust region subproblems in the same iteration, we have gained some additional information about  $f$  when solving the trust region problem with  $\gamma_2$ . This can be beneficial for several reasons. Since we are interested in finding a robust optimum (see Section 10.3), we have gained more information about the objective function close to other points. Also, if we attempt to move towards global optimization (see Section 15.6), then we have more information about the objective function since we have evaluated a greater number of points.

There are several options regarding how the information from the new points can be used, and this is the topic of Section 15.5.3.

## 15.5.2 Optimistic Polling

Once the candidate point  $x_k^+$  has been determined, one option is to place additional points around it, to get more information about the objective function in the vicinity of  $x_k^+$  and thereby use any computers that would otherwise be idle.

There are numerous options how to place the additional points. For example, we can

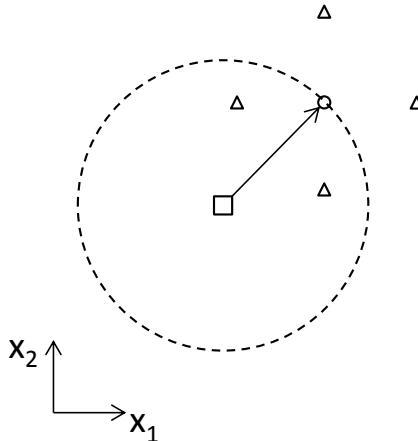
randomize a certain number of points in the vicinity of  $x_k^+$ . Another, deterministic option, is to draw inspiration from pattern search methods [34], which place the points in a certain pattern. A commonly used pattern is a plus sign (actually, it is a hyper plus sign since the number of dimensions is arbitrary). In this case the plus sign would be centered in  $x_k^+$ . This would place a total of  $2n$  additional points, i.e.  $G_k^i = 2n + 1$ . However, see below for situations where we would like to generate fewer points.

In pattern search, it is common to decrease the distance between the middle points and the other points as progress is made. In the context of the algorithms under discussion here, a logical option is to tie the distance between  $x_k^+$  and the additional points to  $\|h_k\|$ .

If  $h$  is short, i.e.  $\|h_k\| < \Delta_k$ , then the model is convex and it is believed that  $f$  is convex. This motivates a shorter distance to the additional points, since we want to get additional information about the objective function close to  $x_k^+$ . On the other hand, if  $\|h_k\| = \Delta_k$ , i.e. the point is on the trust region border, we would like to take longer steps so it is logical to place the additional points farther from  $x_k^+$ .

Depending on how many points we would like to generate for each optimization run, there are different cases to consider.

The simplest case is when we want to generate all  $2n$  additional points for all optimization runs. In that case, we place the points  $x_k^{+(i,j)}$ ,  $1 \leq j \leq 2n$ , a distance  $\nu\|h_k\|$  from  $x_k^{+(i)}$ , where  $\nu \in R^+$ . We use  $\nu$  as scaling factor, to be able to place the new points closer to  $x_k^{+(i)}$  in case we do not want to move them the complete distance  $\|h_k\|$ . While it is possible to rotate the plus sign in any direction, we choose to keep it axis-aligned for simplicity. Therefore, the points are placed according to  $x_k^{+(i,j)} = x_k^{+(i)} \pm \nu\|h_k\|e_l$ ,  $1 \leq i \leq O$ ,  $1 \leq j \leq 2n$ ,  $1 \leq l \leq n$ , where  $e_l$  is the unit vector along the  $l$ -th coordinate-axis.



**Figure 15.3:** Generating additional points in a plus sign pattern centered in the candidate point  $x_k^+$ .

Figure 15.3 displays an example of generating additional points in a plus pattern. The starting point of the optimization is the point  $x_k$ , which is the square in the middle of the trust region (dashed circle). The candidate point  $x_k^+$  is marked by a circle on the trust region and the step  $h_k$  is marked by the arrow from  $x_k$  to  $x_k^+$ . The four additional points (marked by triangles) are spaced at equal distance along the coordinate axes from  $x_k^+$ .

If we want to generate fewer points than  $2n + 1$ , then we use a slightly more advanced scheme to determine the positions of new the points. The problems here are to determine which optimization runs that shall be allowed to create extra points and also where the new points should be placed. Our suggestion is to look at each component of  $h_k^i$ , for all the optimization runs. Consider the components of step  $h_k$  for optimization run  $i$ :

$$h_k^i = \begin{pmatrix} \delta_1^i \\ \delta_2^i \\ \vdots \\ \delta_n^i \end{pmatrix}. \quad (15.2)$$

We look at each component's absolute value and “normalize” the length of the step using the owning optimization run's current trust region radius  $\Delta_k^i$ . The reason for the normalization is that since  $\Delta_k^i$  may differ between the optimization runs, a straight comparison of the components' lengths would typically prioritize optimization runs that are further from convergence, since they have larger values of the trust region radius, thus allowing longer steps.

Then, a greater normalized value is given priority over a lower value, when deciding upon the creation of the next point.

When we have determined which component of which  $h_k^i$  that shall be used to create the additional point, we place the new point farther away from  $x_k^{+(i)}$  along the chosen component's coordinate axis. Just like in the above case where all points are generated, we use  $\|h_k^i\|$  as well as the factor  $\nu$  to determine the placement of the additional point.

```

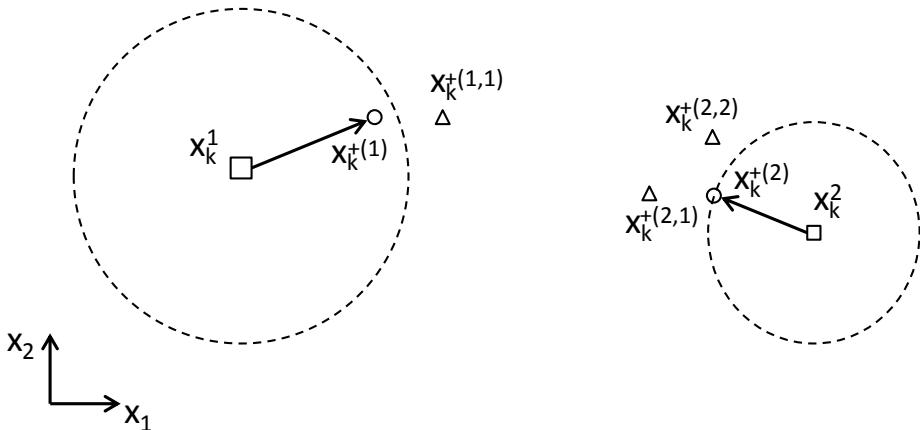
1    $P \leftarrow C - O$ 
2   for  $p = 1$  to  $P$  do
3        $l \leftarrow \text{argmax} \left\{ \frac{|\delta_j^i|}{\Delta_k^i} \right\}$ , s.t.  $\delta_j^i > 0, \forall i, \forall j$ 
4        $x_k^{+(i,j)} \leftarrow x_k^{+(i)} + \text{sign}(\delta_j^i) e_l \nu \|h_k^i\|$ 
5        $\delta_j^i \leftarrow 0$ 
```

**Figure 15.4:** Determining which extra points to generate from all optimization runs' steps and candidate points.

Pseudocode for the algorithm to find the locations of the additional points is available in Figure 15.4. In line 1, the number of extra points to be generated is determined. We here assume that we want to use all  $C$  computers and that there are  $O$  optimization runs

that each has generated one point. Lines 2-5 are repeated  $P$  times to create this number of points. The next thing to do is to find the longest step along a coordinate axis, relative to the maximum allowed step length (line 3). We also store some way to access the component so that we can set it to zero in line 5. The coordinates for the extra point are then determined and assigned in line 4. As before,  $e_l$  is the unit vector along the  $l$ -th coordinate axis. To move  $x_k^{+(i,j)}$  in the correct direction, we use the `sign()` function, which returns 1 if the argument is positive and -1 if the argument is negative. Setting  $\delta_j^i$  to zero in line 5 prevents the same component from being chosen again.

As presented above, we have excluded directions where  $\delta_j^i = 0$  from being candidates for the creation of additional points, even if the value of  $P$  is large enough. The reason for this is that when  $\delta_j^i = 0$ , then the model does not want to go in direction  $j$  at all, so we believe that generating such a point would be of limited value and can be better used elsewhere. The extreme case of this is when a null step is generated, which happens when  $x_k$  is a local minimum of  $m_k$ . In that case there is a lack of information about in which direction any additional points should be placed and we generate no additional points for that optimization run.



**Figure 15.5:** Generating a fixed number of additional points around the candidate points  $x_k^{+(1)}$  and  $x_k^{+(2)}$ .

Figure 15.5 displays an example of a two-dimensional problem with two optimization runs with different radii. The starting point of each optimization is marked by a square, the candidate points are marked by circles and the additional points are marked by triangles. Here we assume that at most three additional points can be created.

For the left optimization run, there is the candidate point  $x_k^{+(1)}$  as well as an additional point  $x_k^{+(1,1)}$ . For the optimization run on the right, there are two additional points  $x_k^{+(2,1)}$  and  $x_k^{+(2,2)}$  in addition to the candidate point  $x_k^{+(2)}$ .

The additional points are created in order of  $x_k^{+(2,1)}$ ,  $x_k^{+(1,1)}$  and  $x_k^{+(2,2)}$ . Even though

the left optimization run's step,  $h_k^1$ , is longer in absolute terms than  $h_k^2$ , the latter is longer when both are normalized with the corresponding optimization run's  $\Delta_k$ .

The value of  $\delta_1^2$ , i.e. the  $x_1$ -component in  $h_k^2$  has the largest normalized absolute value of any component in any  $h_k$ . Therefore, the first additional point becomes  $x_k^{+(2,1)}$ . It is placed the additional distance  $\nu\|h_k^2\|$  in the negative  $x_1$ -direction from  $x_k^{+(2)}$ . Then  $x_k^{+(1,1)}$  is created since  $\delta_1^1$  is the second largest normalized absolute value in any  $h_k$ . It is placed  $\nu\|h_k^1\|$  along the positive  $x_1$ -axis from  $x_k^{+(1)}$ . Finally  $x_k^{+(2,2)}$  is created,  $\nu\|h_k^2\|$  further along the positive  $x_2$ -axis from  $x_k^{+(2)}$ . That concludes the creation of additional points since only three points can be created.

Here we have assumed that it does not matter if certain optimization runs create several additional points while some others create no additional points. If this is undesired, then we can impose a simple constraint on the number of additional points created from each optimization run. This would distribute the points more evenly between the optimization runs.

### 15.5.3 Algorithm Consequences

In purely sequential model building algorithms, such as the ones described in Chapter 12, Equation 12.23 on page 165, which measures the ratio between achieved improvement and expected improvement, is used to determine the trust region radius in the next iteration. Some algorithms also use it to determine whether  $x_k^+$  shall be included in  $Y$  (see Section 12.3.3). Since we generate and evaluate several points in each iteration, there are several values of  $r$ . Thus there are two principally different questions that must be considered: how do we determine which point(s) to include into  $Y$  and how to update  $\Delta$ . For both of these questions, there are many options and we will only explore some of them here.

Regarding the inclusion of a point into  $Y$ , this is somewhat of a dividing line in existing algorithms. The algorithms developed by M.J.D. Powell always include  $x_k^+$ , even if  $f(x_k^+) > f(x_k)$ . The reason for this is that  $x_k^+$  brings information about the objective function that should be included into  $m_k$  in order to improve it. DFO and its successors (see Section 13.1) do not always include  $x_k^+$ . Instead they use a more elaborate procedure for updating  $m_k$ . Due to how their updating procedure works (see Section 13.1.2 for a very brief description), we think that it would be more difficult to extend their updating procedure to handle the case where several points are generated in each iteration.

In principle, we agree with Powell's view, and always want to include at least one of the generated points into  $Y$ . We propose the very simple solution of including the point that yielded the lowest objective function value, regardless of  $r$ . The rationale for this is that even if  $r$  is low, we have achieved a good objective function value, which is considered more important than a high ratio.

A more advanced option is to select  $p_1$  points from all evaluated points and create a completely new model. This can be done in every iteration or more rarely. In some cases, the creation of a new model in each iteration, possibly for several optimization runs, may

be time-consuming. However, in the case under consideration here, the objective function is more time-consuming, thus the time for creating new models would be very small in comparison. The advantage of creating a new model in this matter is that we can always get a model that contains points with low objective function values. Here it is important to not only look at the objective function value, but also consider the linear independence of the interpolation set. Ignoring the linear independence requirement could cause the matrix in Equation 12.16 on page 163 to approach linear dependence, which could lead to a model with numerical problems.

If we are to use Equation 12.23 to calculate  $r$  and then use the value in Equation 12.24 to update  $\Delta^i$ , we need to either use a single one of these values or create a surrogate value that uses several of the  $r$ -values. For simplicity we suggest to use the same point that was included into  $Y$ , i.e. the point with the lowest objective function value of the evaluated points, to determine the new value of  $\Delta$ , but with an additional restriction. That is that we do not always use Equation 12.24 to update  $\Delta^i$ . Instead we check what kind of point the included point is. If the included point is  $x_k^+$ , i.e. the original point, we adjust  $\Delta^i$  according to Equation 12.24. If the point is one of the additional points, then we leave  $\Delta^i$  unchanged, regardless of the value of  $r$ . The reason for this is that we know that the model is less reliable farther away from  $x_k$ , and using Equation 12.24 unchanged could lead to the trust region radius being changed more or less randomly, since the denominator of Equation 12.23 could change considerably.

## 15.6 Towards Global Optimization

Considering that many of the functions that we are interested in are multimodal, and that we want to find the lowest objective function value within the feasible region, which may be all of  $R^n$ , global optimization is a logical extension.

In the research area of global optimization, algorithms are developed that are guaranteed to find the global optimum of the objective function, regardless of where it is located, under certain circumstances. Also, recall from Section 10.3 that it is beneficial if an optimum is robust, i.e. a small change in variable value shall yield a small change in objective function value. While this is not a goal per se in global optimization, it may nonetheless require exploration of large parts of the search space.

Global optimization is a huge area and we cannot hope to make it justice here. Instead the interested reader is referred to e.g. the books by Horst and Pardalos [35], and Pardalos and Romeijn [54] and the references therein. Here we will focus on some areas of global optimization that we think can be useful when  $f$  is a black-box function.

Although combining global optimization and derivative-free optimization is a very difficult proposition, we will here give some suggestions on how one can move towards global optimization in the context of derivative-free optimization. The suggestions in this section have not been implemented completely and are to some extent intended as directions for future research.

In the problem under consideration here,  $f$  is unknown and we cannot guarantee that we will find the global optimum. To increase the likelihood that we find the global optimum of  $f$ , we may first need to explore large parts of the search space, in order to identify regions that we consider promising. Naturally, even coming close of the global optimum of  $f$  may require a large number of function evaluations. To decrease the clock time required, we will employ parallelization as a tool to increase the likelihood that we find the global optimum. This is something that ties in very well with our use of several concurrent optimization runs, as described in Section 15.2.

When we have found one or more regions in which we believe that the global optimum lies, then we must be able to focus our efforts there and attempt to find an optimum in a systematic manner. Thus, we need to do both exploration and exploitation (see Section 12.3). Here we discuss both of these stages in turn.

In the sections treating global optimization, we assume that the feasible region is bounded. If the region was unbounded, then an optimum would in many cases lie infinitely far away, consider e.g. a concave function. However, we make no assumption of how the region is bounded, i.e. whether there are bound constraints, linear and non-linear constraints.

### 15.6.1 Exploration

During the exploration stage, the goal is to identify regions that may contain the global optimum. To attempt to identify where such regions are located, we must perform some function evaluations in order to get objective function values. If no function evaluations have been performed, then the points in which the function is evaluated should be spread evenly throughout the feasible region. Placing the points is a problem that has been subject to extensive research in the area of Design of Experiments [22, 46, 79]. See also Section 10.2.2.

In the case where one or more points have already been evaluated, we want to take these points into consideration when calculating the placement of new points. Since we are doing exploration, we believe that it is suitable to place the new points far from the existing points. Thus the new points shall be placed such that the minimum distance between each pair of points, both new and old, is maximized. Stinstra et al. [77] have discussed methods for solving the case where no points have been placed, and their method can be modified to solve the case with existing points.

A difference between DoE and the optimization here is that we might not be satisfied by evaluating a single point, instead we might want to create a complete optimization run at each location. Naturally, this requires a large number of function evaluations as each optimization run's model requires  $\frac{1}{2}(n + 1)(n + 2)$  function evaluations.

Once all of the optimization runs have been created and their points have been evaluated, we will use the values of the points to attempt to identify interesting regions. For example, it might be that all points in an optimization run have considerably higher values than all other points. We would then consider it less likely that the region in which that

optimization run is located will yield good values, i.e. that region is less interesting.

Once the interesting regions have been identified, some algorithms may concentrate completely on these regions, in order to increase the chance of finding the global optimum. Here however, we do not want such a distinct shift of behavior, since there is always a risk that we have missed the region in which the global optimum is located. Instead we might terminate some of the least promising optimization runs that are in regions where we believe it is unlikely that the global optimum is located (see Section 17.4.1). However, when doing so, we must be aware of the fact that we risk losing an optimum. Naturally, we will let some optimization runs continue to live. Some of them will be located in areas that we believe to be interesting, and some other will also be allowed to live, even if they are in less interesting regions. By doing this, we still have some chance of finding better objective function values, even if they are in the regions that we consider less promising.

### 15.6.2 Exploitation

In the exploitation stage, we are interested in finding the global optimum of  $f$ , given one or more regions that we believe to be of interest. Since  $f$  is unknown, we are limited to using what information we get during execution in order to steer the algorithm in the right direction. Most obvious, we will use the objective function values in the different points and we can use these to calculate an estimate of the *Lipschitz constant*. The Lipschitz constant  $L$  of a continuous function  $f$  is a value such that  $|f(x') - f(x'')| \leq L\|x' - x''\|$  for all  $x', x'' \in R^n$ . That is, the Lipschitz value is an upper bound on the slope of the function in any point.

The reader should note that not all functions are Lipschitzian. As an example, consider the simple function  $f(x) = x^2$  as  $x$  tends to infinity.

The Lipschitz constant has been used in optimization earlier. One of the most known examples is by Jones et al. [38]. For more information, the reader is referred to the survey by Sergeyev and Kvasov and the references therein [71]. Here we will give a short description of some of the relevant work and then describe how it can be adapted to the current context.

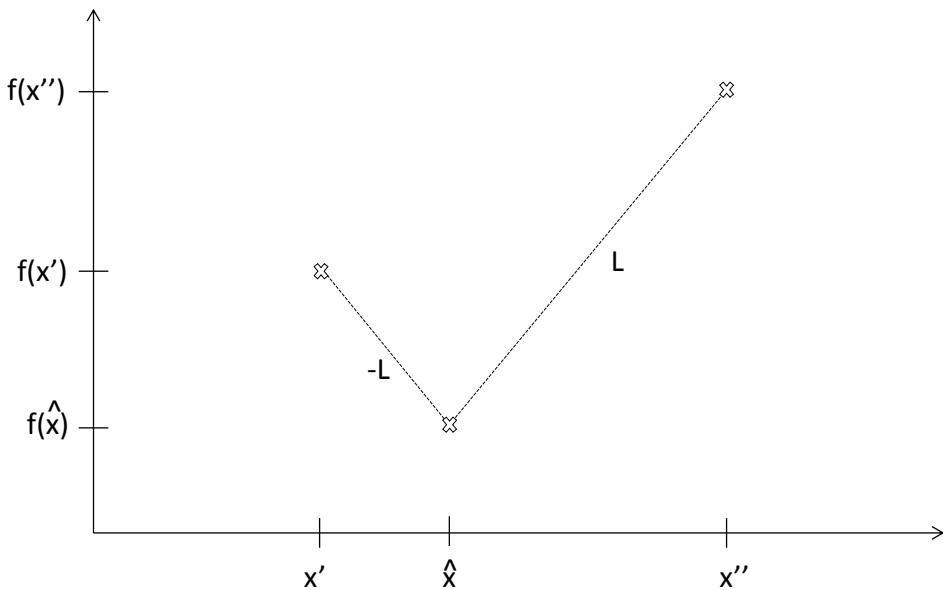
Figure 15.6 displays some points  $x'$ ,  $x''$ , which are functions of one variable, and their values. Given this and  $L$ , it is possible to calculate the lowest possible value  $f(\hat{x})$  as well as the point  $\hat{x}$  where this value will be found. The slope of the lines is  $\pm L$ . The lines intersect in  $f(\hat{x})$ , which has the value

$$f(\hat{x}) = \frac{f(x') + f(x'')}{2} - L \frac{x'' - x'}{2} \quad (15.3)$$

and the point  $\hat{x}$  itself is calculated as

$$\hat{x} = \frac{x' + x''}{2} - \frac{f(x') - f(x'')}{2L}. \quad (15.4)$$

As shown above, if  $L$  is known, then it can be used to calculate the minimum value  $f(\hat{x})$  for some point  $\hat{x}$ , located between  $x'$  and  $x''$ . This way of calculating a minima



**Figure 15.6:** Calculating the lowest possible value  $f(\hat{x})$  using the points  $x'$ ,  $x''$ , their values and  $L$ .

can be generalized to an arbitrary number of dimensions and forms the basis for the algorithm by Schubert [75] as well as the DIRECT algorithm [38]. The DIRECT (DIviding RECTangles) algorithm has been successfully used for solving problems in derivative-free optimization, especially low-dimensional problems. It works by evaluating  $f$  repeatedly in a hyper rectangle and then dividing that hyper rectangle into three smaller rectangles. The three smaller rectangles together cover the same volume as the larger one. The objective function is always evaluated in the middle of each rectangle and by splitting the rectangle into three smaller, only two new evaluations are performed. In the next step,  $f$  is evaluated in the middle of each smaller rectangle. This process continues until the algorithm is terminated, usually after a certain number of iterations or when the minimum rectangle size reaches a lower limit.

The problem with using the Lipschitz constant in the problems under discussion here, is of course that its value is unknown. However, each pair of points and their values give an underestimate of  $L$ , and we can use the highest of these to get a lower bound. A problem with this is that in some cases even the lower bound may be so large that it is practically useless. That is,  $f$  may vary so much that a very large number of points must be evaluated in order to rule out the global optima in a region. There are attempts to mitigate this, e.g. to use “local” values of  $L$  [71]. Instead of using a single global value of  $L$ , there are several values of  $L$ , each which is used only in the vicinity of the points that were used to calculate it.

If we assume that we have at least some approximation of  $L$ , then we can use this to calculate points like  $\hat{x}$ , i.e. points where we believe that  $f$  may take on a good value. While DIRECT evaluates a single point, we instead create a complete optimization run and let it run. Of course, we are not limited to doing this in a single point  $\hat{x}$ , but rather we can do this until a sufficient number of optimization runs have been created, depending on the number of available computers.

Determining a termination criterion for this type of algorithm may be very difficult in the general case, if we want to guarantee that we find an optimum, since we have so little knowledge of  $f$ . Even if we settle for giving a probability that we have found an optimum, this may be almost as difficult. Instead we propose the simple termination criteria of setting a limit on the total number of function evaluations, for all optimization runs. We believe that this is a sound termination criterion since each optimization run may continue to execute as normal, until either it converges, or it is terminated. Reason for termination of an optimization run can e.g. be that it is judged that it will not yield a good objective function value (see Section 17.4.1), or that the number of function evaluations has been reached. If an optimization run converges, then the remaining optimization runs may create additional points in the following iterations (see Section 15.5), so that all available computers are used. Another option is to create additional optimization runs.

### 15.6.3 Connections with Meta-Heuristics

The material in this section, together with the material in Section 17.4, has several connections with meta-heuristics, especially with genetic algorithms [45].

The exploration stage is conceptually very similar to diversification in genetic algorithms, were the goal is to ensure diversity of the gene pool, even if they do not currently yield very good objective function values. Since we know very little about the objective function, we want the optimization runs to together cover a large part of the search space. The reason for this is that we want to evaluate  $f$  in different regions, to increase the chance of finding low objective function values, thereby identifying promising regions.

The exploitation stage that comes next is very similar to concentration in genetic algorithms, where both algorithms concentrate on finding better objective function values, given one or more smaller regions, or a relatively homogenous gene pool, respectively. At this point the focus is on finding the global optimum.

Just like in any algorithm that attempts global optimization, the question is how to change from one stage to another, i.e. when shall we change from exploration to exploitation? We believe that the transition should be fairly gradual, although we have no formal proof of this.

## 15.7 Summary

In this chapter we have presented some ways in which we can parallelize algorithms for derivative-free optimization. This is highly relevant for the kinds of problems that we

are interested in solving (see Section 10.3 for distinguishing problem characteristics), since each function evaluation takes a long time and we employ parallelization in order to decrease the clock time required for finding a high quality solution. As the majority of the algorithms execution time will be spent evaluating the objective function, we believe that the greatest potential for improving the performance is to generate several points in each iteration, where each point can be evaluated in parallel.

We introduced the term *NPE*, representing the number of times one or more points are evaluated, in parallel. This is contrasted by *NE*, which is the total number of function evaluations. We wish to minimize *NPE*, while *NE* is unimportant.

The concept of an optimization run was introduced in this chapter. Each optimization run contains all variables that are required to solve the optimization problem, such as starting point, trust region radius, etc. In practice, each starting point in a multistart algorithm is an optimization run.

The fact that the initial model can be built in parallel, as opposed to sequentially as is commonly done, was discussed in this chapter.

Trust regions of different shapes were introduced here, and non-circular, e.g. elliptical, trust regions were of particular interest. If one has a priori information about the objective function, the static axis-aligned elliptical trust regions can be very useful as it allows longer steps in the directions that are known to be of interest. If this knowledge is missing, then the axis-aligned elliptical trust regions are probably of limited interest since they are limited to allowing longer steps in just a few directions, and is unable to adjust to changing circumstances. To improve the flexibility of elliptical trust regions, we presented an algorithm that iteratively creates a transformation matrix that is used to create an elliptical trust region. In addition to this, we have also derived an expression for a transformation matrix that creates perfectly circular level curves if possible.

We have also discussed several different ways in which each optimization run can generate several points in each iteration. One option that we discussed was solving the trust region subproblem with different radii in the same iteration. A simple method for generating a total of  $2n + 1$  points in each iteration, for each optimization run for also presented. This is achieved by placing the points in a plus sign pattern centered in the original candidate point. If this number of additional points is too high, then we can instead generate a fixed number of points by another method. That method uses vector decomposition of the step  $h$  in order to find directions that are believed to be promising. The component of the step with the largest absolute value is judged to be more promising, and is used to create an additional point farther away from  $x_k^+$ . Then the second longest distance is used, and so on, until the number of desired additional points has been created.

Since the problems under consideration here are typically multimodal and it is desirable to find a robust optimum, we discussed how we can go in the direction of global optimization in the context of derivative-free optimization. We presented some venues of approach that we consider promising to find the global optimum. However, one should be aware of the fact that such attempts are very time-consuming since a large number of points must be evaluated, even for relatively simple problems.



# 16

---

## Information Sharing

Assuming that there are several optimization runs, an obvious question is whether it is possible to achieve synergy between them. That is the topic in this chapter different ways, where we discuss how we can let the different optimization runs take advantage of each other. None of these are required by the parallel extensions presented in Chapter 15, but are different ways to attempt to decrease the number of function evaluations.

### 16.1 Evaluating a Point in Several Models

When there are several optimization runs, one way to take advantage of them is to evaluate each point not just in the optimization run that the point belongs to, but instead use several optimization runs' models to hopefully get a better estimate of the true value in the point.

There are many different ways to estimate the value of a point, and we would like to avoid abrupt changes in the predicted value when several models are used and one way to achieve this is to use a convex combination of the values from all optimization runs' models. To do this, for each model we use both the model's predicted value  $m^i(x_k^+)$  as well as a weight  $w^i \in R^+$ .

To find the weight we draw inspiration from how the trust region subproblem is solved (see Section 12.3). We see that the trust region radius  $\Delta$  is used to limit the step length and the reason for this is that it is believed that the model can be trusted within a distance of  $\Delta$  from the trust region  $x_m$ . The value of  $\Delta$  is increased when the optimization is successful, see Equation 12.23 and Equation 12.24, and decreased when the opposite applies.

Since the model is trusted less farther away from  $x_m^i$  and we want the influence of a model to decrease with increasing distance between  $x_m^i$  and  $x_k^+$ , we will use  $\Delta$  and the distance between  $x_k^+$  and  $x_m^i$ , to determine the weight of the model. The greater distance, measured in the number of  $\Delta$ , the lower the weight should be. However, if the distance

between the candidate point  $x_k^+$  and  $x_m^i$  is less than or equal to  $\Delta$ , the weight should be one, so that it corresponds to when only a single model is used, such as in previous sections. Given these requirements, one possible formulation of the weight  $w^i$  is

$$w^i(x_k^+) = \begin{cases} 1 & \text{if } \|x_k^+ - x_m^i\| \leq \Delta^i \\ \left(\frac{\Delta^i}{\|x_k^+ - x_m^i\|}\right)^\alpha & \text{else} \end{cases} \quad (16.1)$$

where  $\alpha \geq 1$ . A higher value of  $\alpha$  will cause the weight of model  $i$  to decrease faster with increasing distance.

When we have evaluated  $x_k^+$  in all  $O$  models, the *expected* value  $e(x_k^+)$  is created as a convex combination of all models' values and their weights:

$$e(x_k^+) = \frac{1}{W} \sum_{i=1}^O w^i(x_k^+) m^i(x_k^+) \quad (16.2)$$

where

$$W = \sum_{i=1}^O w^i(x_k^+) \quad (16.3)$$

i.e. the sum of all models' weights. Note that there is no iteration counter for the models, since all models may not be in the same iteration, and some may have even finished executing. This is fine as long as we know that the models are valid for evaluating  $x_k^+$ . See Section 17.2 for situations when this might not apply.

Once  $e(x_k^+)$  has been determined, the value can be used just as if a single model had been used, e.g. to determine the denominator in Equation 12.23. Thus, the additional models only influence the optimization run indirectly, through the calculation of  $e(x_k^+)$ .

Equation 16.2 is a generalization of how the model value is determined when using a single model, as done in existing algorithms, and will yield the same result as before when there is only model, i.e.  $m(x_k^+) = e(x_k^+)$ .

## 16.2 Cache of Points

Since the evaluation of a point is time consuming in the kind of problems that we are interested in, an obvious way to decrease the number of evaluations is to store all evaluated points and their values. We call the stored set of points and their values a *cache*.

When a point is to be evaluated, it is first checked whether it has already been evaluated, i.e. is in the cache. If the point is in the cache, the objective function value in that point and any other stored information related to it can be returned immediately and the optimization run that wanted to evaluate the point can continue execution immediately. Immediately returning the stored value can always be done if the simulator is deterministic and reevaluating the point would thus return the same value as already stored. If the simulator is not deterministic or if some natural phenomenon is measured, a reevaluation may be in order. However, in such cases, it must be determined how to handle both the

old value(s) as well as the new. Here there is a plethora of options, i.e. using the min, max or average values. See also Section 11.3.4.

Even if the simulator is deterministic and the new point is different from all points in the cache, one might wish to return the value from another point. Ideally, we would like to return the value of the point whose value is the closest to the new point's value, but since the latter is unavailable, we instead check if the closest point in the cache is close enough. If there is a point in the cache that is judged to be sufficiently close to the new point, then that point's value can be returned. The greatest benefit from this would be of a function evaluation is very time-consuming and it is believed that the objective function is fairly smooth around the new point. Then, returning another point's value would probably not make a big difference in objective function value.

Due to the finite accuracy of computer arithmetic, we must make complicated calculations to determine whether two positions are considered equal. Such comparisons typically require some arbitrary threshold, and if the difference between the numbers is less than this, then the positions are considered equal. Therefore we can allow the user to set a maximum allowed difference for two positions to be considered equal. Another option is to let the threshold be a fraction of the current value of the trust region radius. This would tighten the threshold during the execution, and would make it less likely that another point's value would be returned. While setting a higher threshold, i.e. allowing a longer distance between points that are considered equal, risks that the objective function values are worse, it can potentially save valuable time.

If no point in the cache is considered equal to the new point, it is added to a *priority queue*, which is a prioritized queue containing all points waiting to be evaluated. See Section 16.3 for more information about the priority queue.

Once the point has been evaluated, the value is returned to all optimization runs that awaits the point's value.

## 16.3 Determining Order of Evaluation

Once the predicted value of a point has been calculated, and it has been determined that the point is not in the cache, it is added to the priority queue. When a computer is finished with evaluating a point, the most promising point is removed from the priority queue and evaluated. The question is then how to determine what point is the most promising. Ideally, we would like to use  $f(x_k^+)$ , but since we are queuing points in order to determine just this value, it cannot be used. There are many different alternatives available, and here we consider two alternatives.

### 16.3.1 Lowest Expected Value

The first and most obvious way to prioritize the points is to use  $e(x_k^+)$ , where a lower value means that  $x_k^+$  is more prioritized for evaluation.

While it might seem obvious that  $e(x_k^+)$  should be used to determine the order of evaluation, we know that the model values are often far from the true objective function values. Thus, using alternatives to  $e(x_k^+)$  might also be interesting.

### 16.3.2 Lowest Parent Value

We say that  $x_k$  is the parent point of  $x_k^+$ , if  $x_k^+$  was generated as a solution to the trust region subproblem starting in  $x_k$ . We can use the parent point's real objective function value as another way to determine the order of evaluation, since  $f(x_k)$  must already have been evaluated in order for a trust region subproblem to start in  $x_k$ . We first use  $f(x_k)$ , and then  $e(x_k^+)$  as a tie-breaker. Since  $f(x_k)$  is the true value, this can hopefully offer some guidance as to the value of  $f(x_k^+)$ , at least in when  $f$  is smooth in a region around  $x_k$ .

### 16.3.3 Fewest Points on the Queue

If the number of generated points in each iteration is greater than  $C$  or when several points are generated by each optimization run (see Section 15.5 for examples of when this can happen), then using any of the prioritization criteria suggested so far may introduce unwanted delays. The reason for this is that we assume that each optimization run must wait until all its points on the queue have been evaluated until it can make an informed decision about which point(s) to add to  $Y$ . If this assumption is not made, then the delay will never occur.

As an illustrative example, consider a situation with two available computers, and two optimization runs which have generated two points each. Further assume that the points' expected values are 1 and 3 for the first optimization run, and 2 and 4 for the second optimization run. Thus, if lowest expected values are used to prioritize the points (see Section 16.3.1), then both optimization runs would get one point evaluated and have to wait for the second point to be evaluated. Thus neither of the optimization runs would make any progress with the optimization until all points have been evaluated. Cases with the same effect can be constructed when using the lowest parent value prioritization criterion (see Section 16.3.2).

A possible prioritization criterion developed to handle such cases could be to prioritize optimization runs with fewer points on the priority queue, regardless of their expected values. Thus an optimization run with one point on the queue would get that point evaluated before an optimization run with two points on the queue would get its points evaluated. This prioritization criterion decreases delays, but the overall effect in achieving low objective function values is difficult to determine in advance.

### 16.3.4 Choosing Prioritization Criterion

The prioritization options described earlier can be considered as less risky (use  $f(x_k)$  primarily and  $m(x_k^+)$  as a tie-breaker) and more risky (use  $m(x_k^+)$ ), where the latter

might potentially yield better objective function values faster.

The number of optimization runs spread throughout the search space can also affect the choice of prioritization criterion. If  $O \gg C$ , then it can be worthwhile to use the more conservative lower parent value in order to hopefully explore larger parts of the search space in parallel. Another option is to use the prioritization criteria of fewest points on the queue to decrease the time that the optimization runs have to wait.

### 16.3.5 Adjusting a Point's Queue Position

So far we have assumed that all points to be added on the priority queue are unique. However, it may happen that the point is already on the queue, especially if an optimization run has spawned numerous children (see Section 17.4.2), all with the same model. In that case, several things are relevant to consider:

- The point's expected value  $e(x_k^+)$  might be different this time. If  $e(x_k^+)$  is used to determine the priority (see Section 16.3.1), the lowest of the predicted values is used when determining the position on the queue. The principle is that no optimization run shall suffer from the fact that another optimization run have predicted a poor objective function value in the point, since the models may have improved or become worse with regards to predicting  $f(x_k^+)$ .
- If  $x_k^+$ 's parent's predicted value is used to determine  $x_k^+$ 's position on the queue (see Section 16.3.2), the lowest of the parents' values is used. The principle behind this is the same as above. Also, the optimization run with the lowest achieved value believes that the point will yield an even lower value.

### 16.3.6 Removing a Point from the Queue

If an optimization run is terminated (see Section 17.4.1) then its points can be removed, provided that no other optimization run awaits evaluation of the same points. However, the termination of an optimization run is not the only reason that a point may be removed from the priority queue. If it is predicted that a point's value will be so high that it is not interesting to evaluate at all, then it can be removed so that some more promising point can be evaluated instead.

## 16.4 Summary

In this chapter we have presented several ways in which it is possible to take advantage of the fact that there are several concurrent optimization runs. As each optimization run can be used to predict the value of a point, we can create a convex combination of the different models' values and use the expected value instead of the value calculated when evaluating a point in a single model.

Once the point's value has been calculated in the models, it checked whether the point is in the cache. The cache stores all evaluated points and their values, and if a point has already been evaluated, its value can be retrieved and returned immediately. This can significantly decrease the number of function evaluations required to solve a problem, since evaluation of the same point may be requested repeatedly.

If the point was not in the cache, it is added to a priority queue, which stores all the points scheduled to be evaluated. The queue can be prioritized in different ways, for example using each point's predicted value. Another option is to use  $f(x_k)$ , i.e. the value of the point that was used as a starting point in the trust region subproblem that yielded the candidate point. The motivation behind this is that there might be a large difference between the model and the real objective function, which may lead to faulty prioritization of the points on the queue. At least when  $f$  is smooth,  $f(x_k)$  should offer some guidance about the value of  $f(x_k^+)$ .

If several points are generated by each optimization run and the number of points on the queue is greater than the number of computers, then another option is to prioritize points whose owning optimization runs have few points on the queue. The motivation for this is to decrease delays, i.e. decrease the time until an optimization run can continue execution, since we assume that the optimization runs wait until all their points have been evaluated before they continue.

# Control of Optimization Algorithms

Previously we have discussed parallelization and information sharing, and here we will discuss different control measures of the algorithms that can be used to improve the performance when solving difficult optimization problems.

## 17.1 Separability

The number of points required to build the initial model is proportional to the square of the number of variables, so one can attempt to break the problem into several smaller problems that each involve fewer variables. Since a quadratic model requires  $\frac{1}{2}(n + 1)(n + 2)$  points, separating  $m$  into several models of lower dimensionality will decrease the number of function evaluations required for the models. Consider an objective function with  $n = 5$ , which can be separated into two smaller functions, with  $n_1 = 2$  and  $n_2 = 3$ . This decreases the number of function evaluations for the initial models from 21 to 16 ( $10 + 6$ ). Once the initial models are created, the two optimization problems can be solved as normal.

As can be seen from the above example, the benefit of separability increases with  $n$ , but the difficulty in determining whether a function is separable also increases. Especially since the objective function is unknown, it is unlikely that one can infer that  $f$  is separable. However, if the user knows from experience that  $f$  is separable or that some variables affect each other very little, he may consider the problem separable and split it into several smaller problems.

Naturally, the gain from separation is the greatest when it is discovered that the problem is separable by the user, before the optimization begins. If this is not the case, then one option is to algorithmically determine whether a function is separable. One way to try to do this is to check whether there are any products of two variables in the model. If

no such terms exist, then the function can be considered to be separable. However, even if there are product terms of two variables, the function may be separable. As an example, if the terms  $\{x_1x_3, x_2x_4, x_3x_5\}$  exists in a particular model, one could separate this into two models with  $\{x_1x_3, x_3x_5\}$  and  $\{x_2x_4\}$ . Naturally,  $x_1$  and  $x_5$  cannot be separated since they both have a product term with  $x_3$ . However, one should be aware of the fact that  $f$  might still have product terms of two variables even if  $m$  does not, since the model is not accurate if  $f$  is sufficiently complicated. If there are constraints, then they must also be taken into consideration. Each constraint may only variables in a single model above. If there are constraints involving variables from both groups above, then the separation cannot be done.

We would like to connect the discussion to the Subplex method (see Section 11.3). That algorithm performs such as separation by first sorting and then scanning the “progress vector” for gaps in the values. The method described above attempts to be more intelligent than Subplex, but the same method for determining the subspaces can be used here also.

If it is discovered that the function is separable during execution, then the greatest benefit from this it probably if new optimization runs are created, since it that case, we can split the problem into several lower-dimensional problems and create new models for them. The fact that the problem is separable is of limited interest to the existing optimization runs since their models have already been created.

## 17.2 Guarding Against Ill-Conditioned Models

During the course of execution, it can happen that some values, e.g. the coefficients for the polynomials that in turn make up the gradient and the Hessian, become very large or very small. Since this can cause numerical inaccuracy and potentially also numerical instability, it should be avoided if possible. The reason for the extreme coefficient values can be that the values of  $f$  in the different interpolation points vary greatly and the polynomials must thus change drastically to interpolate the points. This is especially problematic when the distance between the points is very small.

Therefore, to guard against this, one can perform a check to see whether e.g. the values in the gradient and Hessian are within acceptable limits. If they are not, then this is considered as an indication that the model is ill-conditioned. To decrease the computational load and allow the available computers to evaluate points from other optimization runs, the easiest solution is to terminate the optimization run with the ill-conditioned model, and alert the user about this. The user may then decide about further action.

A slightly more advanced option is to discard the current model and create a new model automatically. The new model gets as starting point the currently best point in the model and the rest of the necessary parameters, e.g. trust region radius, are copied from the discarded model. Using these parameters, a new model is created, preferably using one or more of the already stored points, as discussed in Section 17.3. The reason for

using the existing model parameters is to attempt to continue from the current progress level and therefore avoid unnecessarily slow convergence.

## 17.3 Model Building from Existing Points

There are several situations where it is desirable to create a model from existing points. Recall from Section 10.2.2 that when optimization is used in product development, it is often the last step in a chain of steps. Several of the steps involve evaluating the objective function in order to perform e.g. Design of Experiments. Naturally, these points and their values should be stored during the process and, if possible, used when building the initial model. Also, an optimization run may have been terminated due to an ill-conditioned model and a new model is required to continue execution. Another situation is when one or more optimization runs have terminated and a new one is created, e.g. if the user is not satisfied with the results and want to perform further investigation, or as a result of some of the control strategies discussed in Section 17.4.

When taking existing points into consideration when building a model, there are two extremes: one is to take the set of existing points and try to build a model from them. The other is to first determine the desired positions of all points in the new model and then check if there are previously evaluated points at any of the positions.

The first case will create a model, provided that sufficiently many points are available, but the points may be very far and/or close to each other, and thus create a very ill-conditioned model. The other case will not use the stored points at all, unless they are located at the exact right position. The disadvantage is that it will waste valuable time due since it will evaluate new points, but the advantage is that it will yield a model where the points are placed exactly as desired.

While both cases certainly are possible, we suggest a middle ground where we first determine the ideal positions of points and then check for the closest stored point to each one of these. If that point is judged to be sufficiently close, it is used to build the model. After all such points have been retrieved, any remaining points are evaluated. This also allows the user to determine what is “sufficiently close”, which may be useful when objective function evaluations are very expensive.

Naturally, using existing points is not limited to only using points that are stored from e.g. Design of Experiments. If there are several optimization runs and a new one is to be added, some points can possibly be shared between the different optimization runs. See Section 16.2 for more discussion about stored points.

## 17.4 Controlling the Number of Optimization Runs

As earlier stated, an optimization run contains everything that can change when solving a problem (see Section 15.2). We can create several optimization runs which individually attempt to solve the problem, to gather more information about the objective function and

possibly find different optima. However, some of the optimization runs may not yield any useful information and only consume computational resources that could be used by the more successful optimization runs. Therefore it is reasonable to consider when optimization runs shall be terminated and also whether new shall be created. We again assume that we allow at most  $O$  optimization runs to exist concurrently.

### 17.4.1 Strategies for Terminating Optimization Runs

The first and most obvious strategy is to not delete any optimization runs at all, no matter what happens during execution. Since all optimization runs should have at least one different variable value, there is always a chance that they will take different paths through the search space and thus provide more information about the objective function. With this strategy all optimization runs are allowed to execute without interference until they terminate.

Even though the above is possible, we believe that it is unlikely and that it can be beneficial to terminate some of the optimization runs, to allow the more promising optimization runs to use the computational resources. The question is to determine which optimization runs that should be terminated. The most obvious reason for termination is that the optimization run makes little or no progress. Here we will first discuss why it is difficult to use a single criterion to algorithmically determine whether any progress is made, and then what other options that might be used instead.

An obvious option is use the ratio in Equation 12.23 on page 165, which measures the ratio between the achieved decrease in the objective function and the predicted decrease. However, it is known that the ratio will in some cases be very unreliable since there is a large difference between  $f$  and  $m$ , and the ratio will also vary considerably between iterations. Thus, using Equation 12.23 might not be a good idea.

A second option is to use the values of the objective function in the optimization runs' points, especially  $x_k$ . However, strictly prioritizing lower objective function values should be avoided since we are interested in gathering information about several local optima if such exist.

If the objective function value in the global optimum was known, we could measure the relative difference between  $x_k$  and that value. Then we could use a rule that terminated the optimization runs that were more than a certain percentage from the optimal value after a certain number of iterations. However, since we do not know the optimal value, then this criterion for terminating optimization runs cannot be used.

Similarly, it would most likely be a bad idea to use the relative decrease in objective function value, i.e. the numerator of Equation 12.23. There are several reasons for this. As earlier stated,  $f$  may be noisy where points close to each other have very different objective function values. Also, the optimization runs might converge to different points, and during different iterations. Thus, one optimization run may get a large decrease while another is achieving very small increases since it is taking minimal steps close to an optimum.

Yet another option is to consider the movement of the point with the best objective function value. Ideally, every new point should improve the objective function value and thus the point with the best objective function value should be different in every iteration. However, there are several legitimate situations where the best point of an optimization run may not have changed for several iterations.

One possibility is that the optimization run is converging to a local minimum and will soon terminate. If such an optimization run is terminated prematurely, the information that the point is a minimum will be lost. Another possibility is that there might be a big difference between  $f$  and  $m$ . This leads to two separate cases. We first consider the case when a trust region iteration has been performed. Since the best point is unchanged, this implies that the new point did not yield a lower value of  $f$ . However, the point will still be included in  $m$ , which will improve it in future iterations, and will hopefully allow it to make progress. The other case is that the model is being improved by performing a model iteration, which do not attempt to yield a point with a better objective function value, but rather to provide a point to in order to improve the model.

On the basis of the above discussion, we find it difficult to give a solid motivation for a termination criterion based solely on a single snapshot value obtained during the execution. Instead, to determine which optimization runs that shall be terminated, we propose to use a customizable function that takes some of the above factors, and possibly also other factors, into consideration.

We suggest to use the objective function value and the minimum distance between optimization runs, that shall be maximized. That is, we use

$$\max \min \|x_k^i - x_k^j\|, \forall j \neq i \quad (17.1)$$

as criterion. The use of objective function value is fairly self-explanatory and regarding the distance between the optimization runs' best points, we would like to connect this to the discussion about the exploitation/exploration dilemma in Section 12.3 as well as to the discussion about global optimization in Section 15.6.

If we prioritize exploration, then then we would like to take the distance into consideration, while if we concentrate on finding the lowest objective function value (exploitation), then we might ignore the distance factor. In exploration, we want to keep the optimization runs spread out from each other, and to achieve this, we use a prioritization function where each optimization run is assigned a merit value for each of the different parameters.

Here we give a small example of how this prioritization works. To the left in Table 17.1 we can see the comparison criteria: lowest value of  $f$  and the minimum distance to other optimization runs, and their values for the three optimization runs A, B and C. To the right in Table 17.1, we see the optimization runs' merit values in the two criteria.

The merit value for each category is decided through comparing all optimization runs with regards to a certain criterion. If a certain optimization run has the best value, it is assigned the merit value  $O$  (i.e. a value equal to the number of optimization runs), the second best is assigned the merit value  $O - 1$  and so on, until the optimization run with

Optimization run	A	B	C	Optimization run	A	B	C
Lowest $f$	10	40	21	$M_f$	3	1	2
Minimum distance	4	4	12	$M_d$	2	2	3

**Table 17.1:** Example of optimization run data and merit values.  $M_f$  is the merit value for low objective function value and  $M_d$  is the merit value for distance from other optimization runs.

the worst value gets the value 1. Thus, a higher merit value is better. By ranking the optimization runs' against each other, we avoid the use the explicit values directly and thus we do not need to decide whether a certain value is good or bad.

Since optimization run A has the lowest objective function value, it receives the best merit value,  $M_f = 3$ . The minimum distance between optimization run C and the other two is 12, so it is assigned the merit value 3 in that category. Since optimization runs A and B are the closest to each other, with the minimum distance 4, they are both assigned the merit value 2 in the category for minimum distance,  $M_d$ .

When all merit values have been assigned, we determine the total merit value as a convex combination of the two merit values:

$$M_{tot} = w_e M_d^i + (1 - w_e) M_f^i, \forall i \quad (17.2)$$

where  $w_e \in [0, 1]$  is the *exploration weight*.

Depending on the value of  $w_e$ , we either prioritize exploration (high value of  $w_e$ ) or exploitation (low value of  $w_e$ ). When the total merit value has been assigned to all optimization runs, we know which optimization runs shall be prioritized for keeping. However, we must decide how many that shall be terminated. Typically, we decide to keep a certain fraction of all available optimization runs and terminate the rest. When looking at Table 17.1, we can see that optimization run B is dominated by the other two since none of its merit values is higher than the corresponding merit value of the other two optimization runs. Thus it will always be terminated.

Naturally, we can change the exploration weight during execution in order to focus more on exploration or exploitation depending on which stage the optimization is in or what the user wishes.

Here we have just scratched the surface of possible factors to consider when determining factors to use to terminate optimization runs. Other factors that we consider are the decrease in objective function value and the ratio  $r$ . For these two factors, we plan to use a history of a certain number of values and attempt to detect trends in the values. For example, if both the decrease in objective function value and  $r$  are poor, then the model is poor and/or the objective function is, at least locally, very noisy, so that particular optimization run makes very little progress. For this reason, it can be terminated. For the cases where there are several factors, it is not as easy to use a convex combination of

values. To prioritize the factors in that case we instead propose to use a weighted sum of the factors.

## 17.4.2 Strategies for Creating New Optimization Runs

Once some optimization runs have been terminated, new ones should be created to utilize the available computers effectively. Here we distinguish between two major cases, that both will be discussed. The first case is when we create completely new optimization runs and the other is when we create new optimization runs from existing ones.

We will typically create completely new optimization runs when we want to spread the optimization runs throughout the search space. An example of time when this might be done is in the exploration stage of global optimization, see Section 15.6.1. To find locations far from other optimization runs where they may be placed, we can use e.g. the algorithm by Stinstra et al. [77]. Other options are to use methods from DoE or just simple randomization of points. When a new starting point for an optimization run has been determined, it is beneficial to try to use existing points when building the initial model (see Section 17.3) in order to decrease the time until the optimization can start.

The other case is when we have existing optimization runs that we believe are located in interesting areas and we want to explore these areas further. By creating more optimization runs with different variable values, they will hopefully take different paths through the search space and thus explore it more.

We say that an optimization run is a parent if at least one optimization run is created from it. The optimization runs with the same parents are called siblings. All siblings have at least one variable that is different from each other as well as the parent. Although we will typically vary the shape of the trust region, all variables can be varied.

When the new optimization runs are created from existing ones, there are two basic questions: which of the remaining optimization runs should be allowed to be parents, and what children should be created from the parents?

For the first question, we think that all remaining optimization runs shall have the possibility to have children. They were kept since it is believed that they will provide useful information in the future, even though they may not currently have the lowest objective function values.

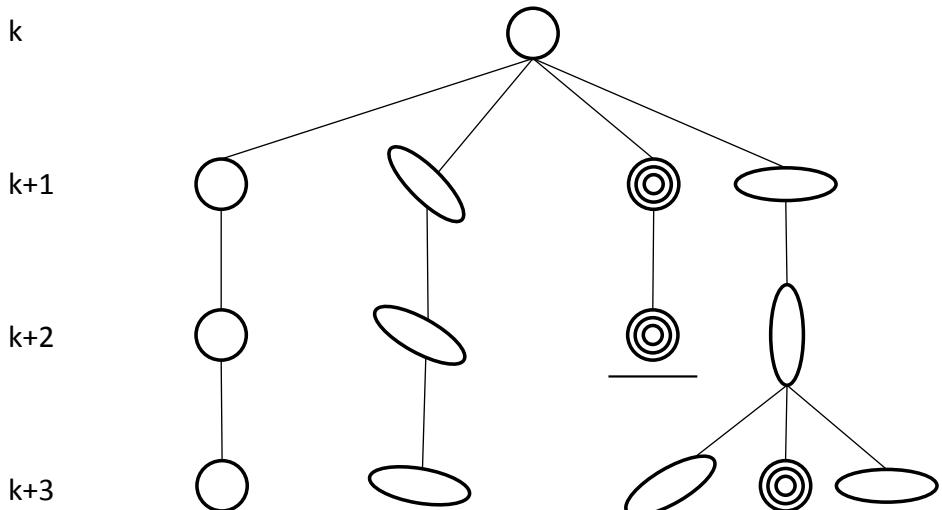
The second question is more interesting: how we determine which of the trust region shapes that shall be created? For an  $n$ -dimensional problem, at least the  $n + 3$  different trust region shapes presented in Section 15.4 can be created, in addition to the standard circular. Since the parent is kept and has one of the trust region shapes,  $n + 3$  children with unique trust region shapes can be created from each parent. Creating all optimization runs from all parents might not be the best option since it might exceed the number of available computers. The question is then which trust region shapes to prioritize.

As discussed in Section 15.4.1, unless one has information about the objective function, it is difficult to prioritize one axis-aligned elliptical trust region over another. If the number of available computers is sufficient, i.e.  $C \geq n + 4$ , we create one each of spher-

ical, switching axis-aligned elliptical, adaptive elliptical and a trust region with circular level curves (see Section 15.4.2 for information about the latter two) and an axis-aligned trust region for each variable in the objective function.

If the number of computers is not sufficient to create  $n + 3$  new optimization runs, we prioritize creating one each of spherical, switching axis-aligned elliptical, adaptive elliptical and a trust region with circular level curves, assuming that these are different from the parent's trust region shape. After doing this for all original trust regions, we can add axis-aligned elliptical trust regions if there are unused computers available.

## Iteration



**Figure 17.1:** Example of termination and creation of optimization runs in different iterations.

Figure 17.1 shows an example of termination and creation of optimization runs in different iterations. Assume  $n = 2$  and  $C = 5$ . In iteration  $k$ , there is a single optimization run with a circular trust region. In the end of that iteration, three more optimization runs are created: one with an adaptive elliptical trust region, one that transforms the problem to have circular level curves and one that has an axis-aligned switching trust region shape. All four optimization runs execute in iterations  $k + 1$  and  $k + 2$ . At the end of iteration  $k + 2$ , the optimization run that creates circular level curves is terminated. To fill up the number of available computers, the optimization run with axis-aligned switching trust region shape has children and creates two more optimization runs, one with an adaptive elliptical trust region and one that creates circular level curves. Thus, in iteration  $k + 3$ , there is a total of five optimization runs.

We note that using the control strategies suggested here may delay termination of the

algorithm in some cases. As an example, consider a case with three optimization runs. Two of them are close to termination as the trust region radius is close to  $\rho_{end}$ , while the third is far from termination since its trust region radius is far from  $\rho_{end}$ . However, the objective function value that the third optimization run has achieved is the lowest. If we were to use the strategy of keeping the optimization run with the lowest value, then it would be kept and the other two terminated. Since the new optimization runs inherit the parent's trust region radius, they are further from termination than the optimization runs they replaced.

A potential counter argument against the deletion and subsequent creation of new optimization runs is that this can be time-consuming, especially if the number of optimization runs terminated/created in each iteration is large. However, since the objective function itself is very time-consuming to evaluate for the problems considered here, this should not be a major concern. For problems where it is quick to evaluate  $f$ , the time required for terminating/creating optimization runs may be a greater fraction of the total execution time, and thus more important to consider.

## 17.5 Summary

In this chapter we discussed different aspects of derivative-free optimization that might not be obvious at first, but that can be very important in practice.

Since we are interested in solving real world problems, we discussed how we can guard against ill-conditioned models, e.g. models where the coefficients for the gradient and/or the Hessian have very small/large values, which can cause numerical instability. Our first step in this direction is to attempt to detect such values and alert to user of the situation. However, since this requires input from the user, we might automatically terminate such optimization runs and create new optimization runs with new models, in the future.

We have also discussed how to control optimization algorithms on a higher level. In this first step towards a better control of the algorithm, we terminate and create new optimization runs during the course of execution.

We believe that it is difficult to give a termination criterion based on a single variable value. Instead we propose to use a configurable function that uses at least two different factors. In that function, all optimization runs are ranked relative all other, to get a merit value for each factor. Each merit value has a weight and the total merit value is calculated as a weighted sum of all merit values. Once the total merit value is calculated, a certain fraction of the optimization runs is kept and the rest are terminated.

When it comes to creating new optimization runs, there are many different options. The most important question is whether the new optimization runs shall be created from existing optimization runs or be spread throughout the search space. Here we have mostly been concerned with creating children from the surviving optimization runs.

Creating children from existing optimization runs is of course only relevant if it will

yield points that are different from the parent's points. To achieve this and to separate the different child optimization runs from each other, we vary at least one variable value between the optimization runs. This is typically the trust region shape, although other options are possible.

For simplicity, we create children from all surviving optimization runs, but we might not create all possible children from it. Since we have defined a total of  $n + 4$  different trust region shapes and the parent has one of them, each optimization run can generate  $n + 3$  children with unique trust region shapes. We believe that the axis-aligned trust region shapes are the least promising, so we only create these if we know that the number of computers is sufficient, i.e.  $C \geq (n + 4)O$ . Otherwise we prioritize the creation of the other trust region shapes.

# 18

---

## Implementation Details

In this chapter we describe our implementation and the principles behind it. We also describe the choice of third party software packages that were used in the implementation. We will describe the general standalone implementation, including some information about the synchronous parallelization. Then, in Section 18.7, we describe the specific modifications made during integration with the BEAST framework.

### 18.1 Choice of Algorithm

We choose to implement the UOBYQA algorithm (see Section 13.2) to test the parallelization extensions and modifications presented in Chapters 15–17. The UOBYQA algorithm was chosen for several reasons: the problems that we are interested in solving involve a small number of variables, so we have no great need for the improved initial model building of NEWUOA (see Section 13.3). Also the control flow of the algorithm is easier to understand than e.g. NEWUOA and the actual Fortran implementation is shorter, better documented and thus easier to understand. This makes it easier to expand, modify and to maintain our implementation.

The DFO algorithm (see Section 13.1) was another candidate, but in the end we choose to not use it since it is quite outdated and difficult to understand. The extended version of DFO (see Section 13.1.2) would have been another option. However, the source code is not publicly available and it is not certain that the description by Tröltzsch [78] is sufficient to get a correct implementation, especially considering the discussion about the implementation in that thesis.

## 18.2 Choice of Third-Party Software

Since the implementation of the optimization algorithm described in this thesis is to be used in an industrial setting, it is imperative that it is implemented with this in mind. Usage in an industrial setting places high requirements of maintainability, expandability, documentation, testing and performance. Since a lot of modern software, including simulators, is implemented in C++, this programming language was used in our implementation to make integration with external software easier (an example is available in Section 18.7).

Before the implementation was started, an investigation was performed in order to see whether any existing third-party software packages could be used, in order to avoid rewriting of algorithms and data structures. The main requirements were that the software packages must be cross-platform to allow the software to be used both on Windows and Linux, open source and free to use in commercial products. For linear algebra, it was important that LAPACK [43] was used, since it is the de facto standard software for numerical linear algebra.

With these requirements in mind, we chose to use the Armadillo linear algebra library [64]. Armadillo also supplies vector and matrix functionality. In addition to fulfilling all the above requirements, Armadillo is continuously developed, and provides an easy to use interface to LAPACK. The Google test framework [31] was chosen for testing, to allow different forms of testing: from low-level unit testing to higher-level functional testing.

Some parallelization primitives such as threads and locks were required for implementing the parallelization of the algorithms. Here the Boost [7] framework was used since the parallelization extensions of the new C++-standard C++11 [11] might not be completely supported by the used compilers, at the time of implementation.

## 18.3 Implementation Principles

As earlier mentioned, the kinds of optimization algorithms that have been presented in this thesis are not inherently suitable for parallelization since they are sequential in nature. The programmers often take advantage of this and hardcode a lot of the variables, e.g. there is exactly one step  $h$  and of course only one model  $m$ . Often many variables are global so that they can be used and changed throughout the whole program. Also, error detection and error handling is also often neglected. While this is acceptable in proof-of-concept implementations, it is unacceptable in software for industrial use, since it makes the software less reliable and less robust. We here present some of the principles that were taken into consideration during the implementation of the algorithm.

- Interfaces allow for future extensions. In our implementation there is only one algorithm implemented for e.g. solving the trust region subproblem (see Equation 12.22). However, the specific implementation of the algorithms is “hidden” behind an interface that allows the implementation of new algorithms for solving the

same problem, but in different ways. If a new algorithm is implemented, then it can be used directly with minimal changes to the surrounding code. Similarly for interpolation models which can also be exchanged in a similar manner. Currently there is only one interpolation model implemented, which implements the second-degree Lagrange polynomials described in Section 12.1, but other interpolation models, e.g. least-squares models or splines, can be implemented in the future.

- Suitable abstractions. Often in implementations of mathematical software, there are few abstractions, which lead to software that is sometimes difficult to understand. As an example, it is common to want to store a point  $x_k$  together with its objective function value  $f(x_k)$ . Although it would be logical to create an abstract type with the two variables, this is rarely done. In our implementation, we have gathered variables that are commonly used together and created abstractions for them. For the previously mentioned example, there is a class `PointAndValue` with two main members: the point and its value. Similar abstractions have been created throughout the software.

This is closely connected to good software engineering practices such as e.g. separation of concerns, encapsulation and abstraction. For more information about this we refer the reader to the practical software engineering books by McConnell [44] and Hunt and Thomas [36].

- State variables gathered in optimization runs (see Section 15.2). All variables that are specific to a particular iteration for a specific problem are gathered into an abstract data structure: the optimization run. Each optimization run contains all variables required for performing an iteration of the algorithm, e.g.  $x_k$ ,  $m_k$ , and  $\Delta_k$ . Since each variable can be set for each optimization run, the values can of course be different for the different optimization runs.

In our implementation, optimization runs are passed as arguments between the functions, allowing the rest of the program to be stateless.

- Stateless algorithms. Since the software is based around the concept of parallelization, the algorithms for e.g. finding the point with the minimal model value within a region, are not allowed to have state variables. The reason for this is that the state is often different for different optimization runs. In a sense the functions are similar to pure functions in functional programming languages, although here the result might also depend on constants that are not supplied as input arguments.
- An optimization run pauses after submitting a point for evaluation. The original UOBYQA algorithm always uses the value in a point in the next iteration, therefore it must wait for the function value to be calculated. Thus, execution can continue until a point is submitted for evaluation. In our implementation, each optimization run must yield and wait until the objective function value has been calculated in all its submitted points. However, other optimization runs may continue executing,

until they in turn submit one or more points for evaluation. Once an optimization run's points have been evaluated, the optimization run may continue executing since the necessary information is now available.

- Explicit representations for viewing and logging. In most cases, there are explicit representations of objects such as gradients or Hessians. Since the program is made with the intention of solving problems where  $n$  is relatively small, we can allow explicit representations. A big advantage of this is that a programmer can look at the code and understand what is going on, without the need to understand a lot of optimization or how the optimization algorithms are implemented. This is also made easier by the fact that there is functionality for printing the values of relevant variables in most classes in the program. Similarly, each optimization run can log e.g. its variables and results from optimization, in each iteration if desired. There are different logging settings that are used to log more or less information in order to allow for a user to follow the optimization during execution.

Should it be necessary in the future, the program can easily be extended with other representations that require less memory etc. since the program is modular and made with expandability in mind.

- Testing is an integral part of the development process. Unit testing and functional testing was performed as an important part of the development. As mentioned earlier, we used the Google test framework [31] for testing and tests were written at the same time as the function they were intended to test. We used both positive and negative tests to test that the functions behaved as expected. Since we thought it very important to test the control flow of the functions, we used a white-box testing approach for the functions. In the most central parts of the algorithm, e.g. building and updating the model, and solving the trust region subproblem, we have tried to get complete code coverage, i.e. that every single line is exercised during testing.

### 18.4 Termination Criteria

Our implementation uses several different termination criteria. The common criterion  $\rho_k \leq \rho_{end}$  from UOBYQA was used. In addition, there is a limit on the number of function evaluations as well as a limit on the number of iterations in the algorithm. That criterion is a safety precaution for the case that no new points are generated (therefore the limit in the number of function evaluations will not be triggered), but the algorithm would still continue.

A non-standard termination criterion is that the user may set a value on the objective function, and if a value lower than or equal this value is found, then the program is terminated. This represents the fact that the user may content with finding a point with a

certain objective function value, and as soon as such a point is found, the program can be terminated since the user is satisfied.

## 18.5 Implementation of Parallelization

For implementation of the parallelization, we used the C++ extensions from the Boost framework [7]. The standalone implementation is intended for testing on synthetic test cases (see Section 19) and parallelization will only use the CPU cores on the machine on which the program executes. We choose to use a quite simple synchronized parallelization for evaluation of points, since the time for evaluating the objective function is typically same regardless of in which point the function is evaluated. Figure 15.1 on page 190 shows an example of synchronous parallelization.

In this implementation, there is one main thread that creates e.g. four threads for evaluating the points. During the evaluation of the points, the main thread waits and only continues to execute once all evaluator threads have finished execution. Naturally, the number of evaluator threads created depends on the number of points that shall be evaluated at the current time, as well as the number of cores available on the specific CPU.

## 18.6 Software Architecture

Figure 18.1 displays the major components of the implementation as well as the connections to a simulator. Utility and storage classes such as the above mentioned PointAndValue classes have been left out for clarity.

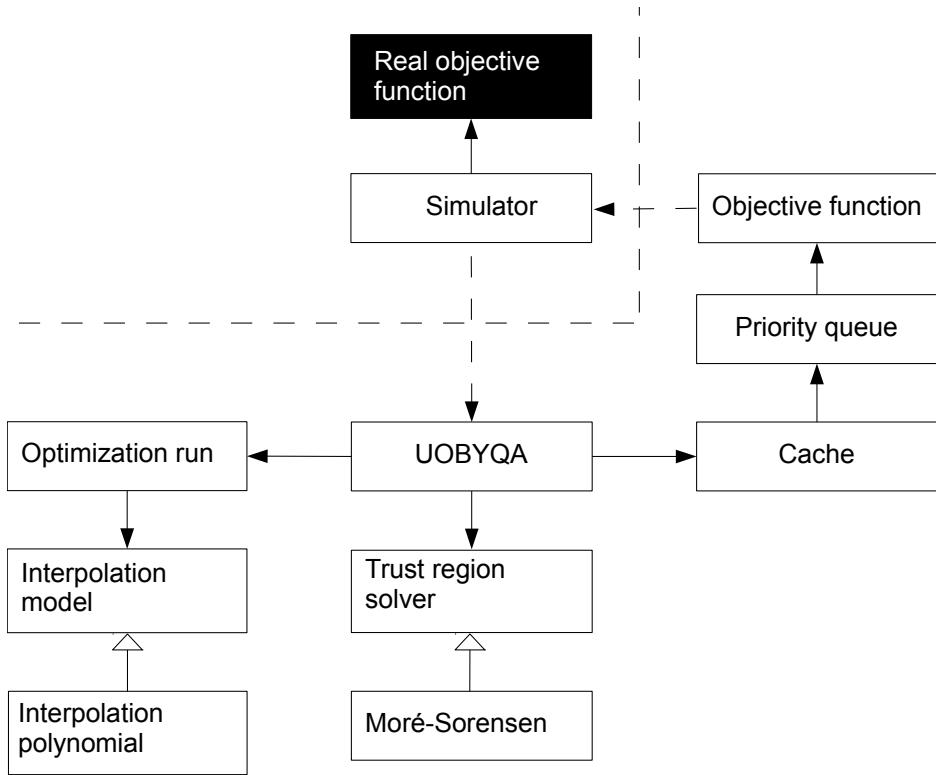
The “use” and “has”-relations are marked by solid arrows, starting in the having/owning objects. Inheritance relations are marked by larger, empty arrows, pointing towards the base class in the inheritance hierarchy. The dashed line marks the border between the simulator and the optimization framework.

When a user specifies an optimization problem, the information is transferred to the optimization framework, concretized by UOBYQA, which creates an optimization run for each starting point. The parameter values are also set at this point. Then UOBYQA executes basically as in Section 13.2.

Each optimization run has an Interpolation model, which is an abstract representation of  $m$ , which is implemented by a second degree Interpolation polynomial (see Section 12.1.1).

For solving the trust region subproblem, there is a Trust region solver interface that facilitates easy implementations of new algorithms. Currently, the only available algorithm is the Moré-Sorensen algorithm.

Once the candidate point has been found, it is checked if the point is already in the Cache. If not, it is added to the Priority queue for evaluation. There is a representation of the Objective function that mainly holds information such as dimension, but also stores



**Figure 18.1:** The major components of the implementation and integration with a simulator.

the point with the lowest value so far, its value etc. When the objective function is to be evaluated, the simulator is called, which starts the evaluation of the Real objective function.

When the optimization is finished, all information regarding evaluated points and their values can be retrieved using interfaces in UOBYQA.

## 18.7 Integration With BEAST

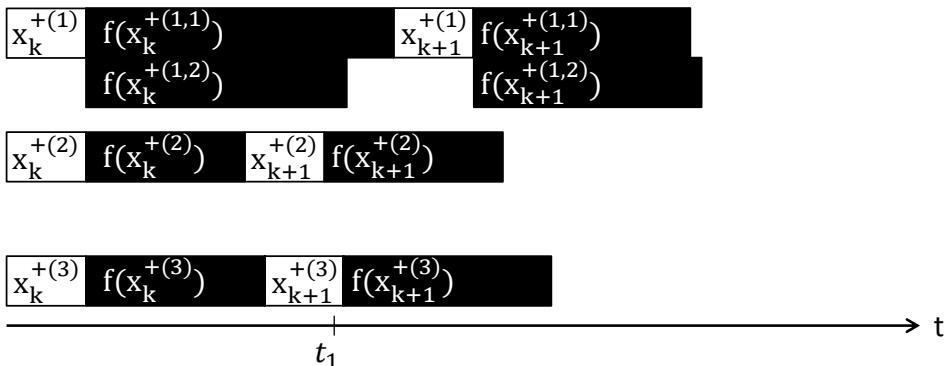
In this section we describe the integration of the optimization framework with the BEAST simulator, and the modifications that have been implemented compared to the standalone version.

### 18.7.1 Asynchronous Parallelization

The standalone version uses synchronous parallelization of evaluation of the points, and all threads are created on the same computer (see Section 18.5). However, this parallelization is not suitable for use in BEAST, and we will here describe why.

BEAST uses asynchronous parallelization; when a BEAST user wants to evaluate two different points, he can request that each simulation is sent to a different computer. Typically each simulation uses several computers in turn, and the calculation time may vary considerably between the points. As soon as one point has been evaluated, the result is returned, even though the other point is still being evaluated.

Naturally, we want the optimization framework to work in the same way, and this means that the synchronous parallelization described in Section 18.5 must be replaced. If synchronous parallelization were used, then the optimization framework would have to wait until both points were finished evaluating before the algorithm could continue.



**Figure 18.2:** Simple example of asynchronous parallelization.

Figure 18.2 displays how the implementation of asynchronous parallelization works, when the optimization software is used in BEAST. There are three different optimization runs, which perform their calculations in parallel and completely independent of each other. In this example, optimization run 1 generates two points for evaluation, and optimization run 2 and 3 generate one point each. The time required for evaluating each of the points may vary significantly, as indicated by the length of the black boxes along the timeline.

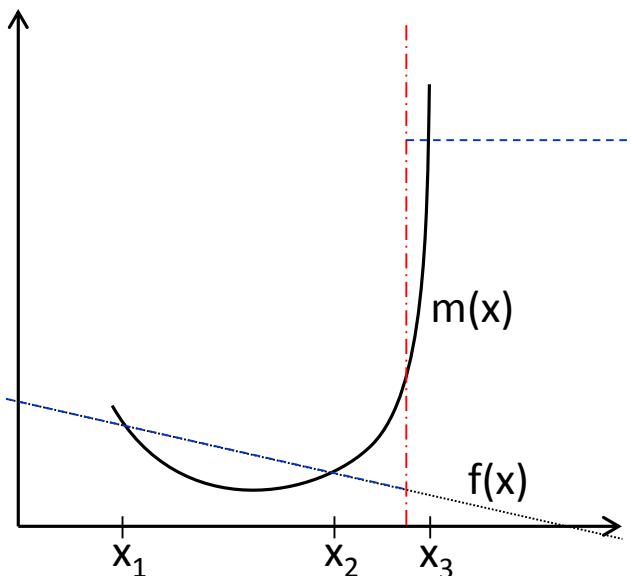
Optimization runs 2 and 3 can continue to execute as soon as the evaluation of their respective points have finished. Since optimization run 1 generates two points, it must wait until both these points have been evaluated before it can continue. Note that this implementation of parallelization really is asynchronous; at time  $t_1$ , optimization run 1 is still evaluating both of its points in iteration  $k$ , while optimization run 2 is evaluating the new point in iteration  $k + 1$  and optimization run 3 is calculating a new point in iteration  $k + 1$ .

### 18.7.2 Constraint Handling

In addition to bounds, the BEAST user may also specify arbitrary Boolean constraints, on the form  $c(x) \leq 0$ . Each such constraint is handled as a black box, which only returns whether the constraint is fulfilled or not. This means that it is not possible to extract how much such a constraint is violated.

To get a unified constraint handling, BEAST handles all constraints, including bounds, by penalty functions. When a point from optimization is suggested for evaluation, it is first checked whether it fulfills all constraints. If so, simulation takes place as normal.

If a point does not fulfill all constraints, it is considered invalid. No invalid point is sent for evaluation and there are several reasons for this. The first and most obvious reason is that it does not fulfill the constraints and thus cannot be a valid solution. Also, some invalid points may produce an invalid geometry, i.e. the bearing will not be physically correct (see also Section 18.7.3). Such an invalid geometry may cause a crash during simulation.



**Figure 18.3:** Example of a situation where penalizing an invalid point will create a model that will lead the trust region optimization algorithm in the wrong direction.

If a point is invalid, a fixed “penalty value” is returned as the simulation result. The point and the value can then be treated as any other point in the optimization algorithm. However, this way of handling constraints has some implications. Since the penalty value is set to be higher than the highest valid value that can occur, and as  $m$  is guaranteed to interpolate all points and their values,  $m$  can be very steep towards an invalid point.

Figure 18.3 shows a schematic example of this. In the figure, the black dotted line

is the objective function  $f(x)$ , which is a function of one variable. The red dotted and dashed line is a constraint and the feasible region is to the left of the constraint. The blue dashed line is what is returned when evaluating a point. It is equal to  $f$  inside the feasible region and takes on a penalty value outside the feasible region. The objective function is evaluated in the points  $x_1, x_2$  and  $x_3$  and these points together with their values are used to create the model  $m(x)$ . The model is drawn in solid black. Since the model is of degree two and interpolates the points, we can see that the minimum is between  $x_1$  and  $x_2$ . Not only will this lead the optimization algorithm in the wrong direction, but it might also be difficult to find an optimum, at the point where the constraint is active. Finding optimum where constraints are active is a general problem when using penalty functions to penalize invalid points, and is even worse here. This problem will remain even if we assume that the algorithm will eventually find points with better objective function values, located between  $x_2$  and the constraint.

Another problem is that if the distance between valid points and invalid points such as  $x_3$  is sufficiently small, it will be impossible to create and update  $m$  due to numerical inaccuracy. In our implementation, we try to detect such problems (see Section 17.2) to avoid the effects of having such models.

### 18.7.3 Implicit Constraints

It is well known that in the design of mechanical products, some combinations of values may create a geometry that is impossible to perform calculations on. While it is obvious that such configurations exist, one might assume that they are removed by the introduction of constraints, and conversely, that all configurations that fulfill all explicit constraints are valid. However, this is not necessarily true. There might also be other constraints, inside the feasible region, that represent illegal configurations. Conn et al. [16] refer to such constraints as *implicit* constraints.

Such implicit constraints may also be present in the production of bearings and the optimization may encounter such configurations during execution. If it turns out that a simulation is impossible to perform for this reason, it is treated as if the point does not fulfill the constraints (see Section 18.7.2). However, in addition to this, the user is alerted that an invalid configuration has been encountered. The reason for this is that the optimization algorithm has encountered something that the user did not expect, since no constraint was added to remove the invalid configuration.

### 18.7.4 Integer variables

So far we have not discussed how integer variables are treated when using the optimization software in BEAST. Since the original UOBYQA algorithm is incapable of handling integer variables, we have decided to use a pragmatic solution to be able to solve problems involving integer variables at all. Depending on the number of values that the integer variable can have, we recommend different strategies to the user.

If the integer variable can only take on a few different values, then the user can create separate optimizations for each integer value. The remaining problems with only continuous variables are solved as normal, possibly with the different problems solved in parallel.

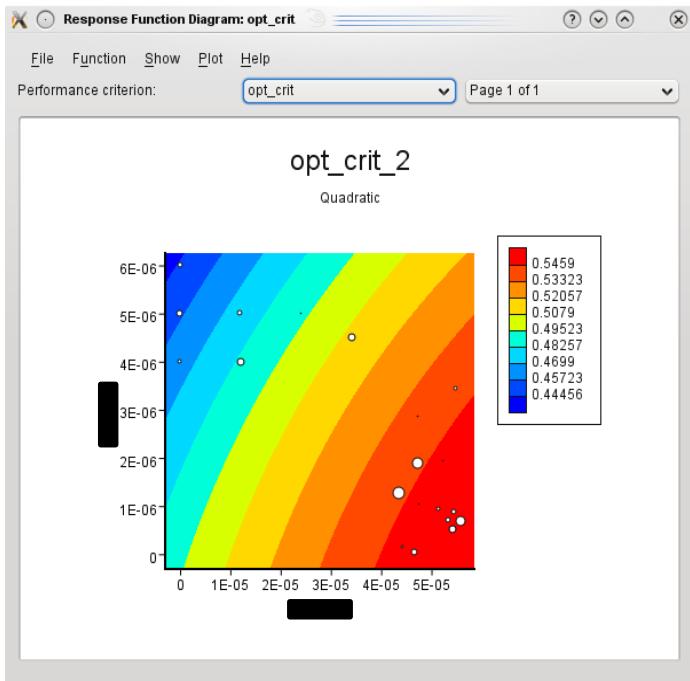
The other case is recommended if the integer variable can take on many different values. In that case, the user marks the variable as integer. After an optimization algorithm calculates a value of such a parameter, it is rounded to the nearest integer value before a simulation is performed. That way, simulations are only performed in points fulfilling all constraints, including integer constraints.

Neither of these strategies is particularly appealing; in the first case, the user is required to do additional work and in the second case, the optimal integer solution may be far from what we find using rounding.

### 18.7.5 Visualization of Results

The Beauty software allows the visualization of results in various different ways, and we will discuss some of them here. Since parameter studies and optimization often take some time (see Section 10.3), the user does not have to wait until the complete optimization has been completed. Instead, the visualization updated when a new point has been evaluated, which allows the user to see gain more information as new information becomes available. While the information in this section is not exactly optimization algorithms, it can nonetheless be useful when optimization algorithms are used in practice. The functionality described in this section has not been implemented by the author of this thesis.

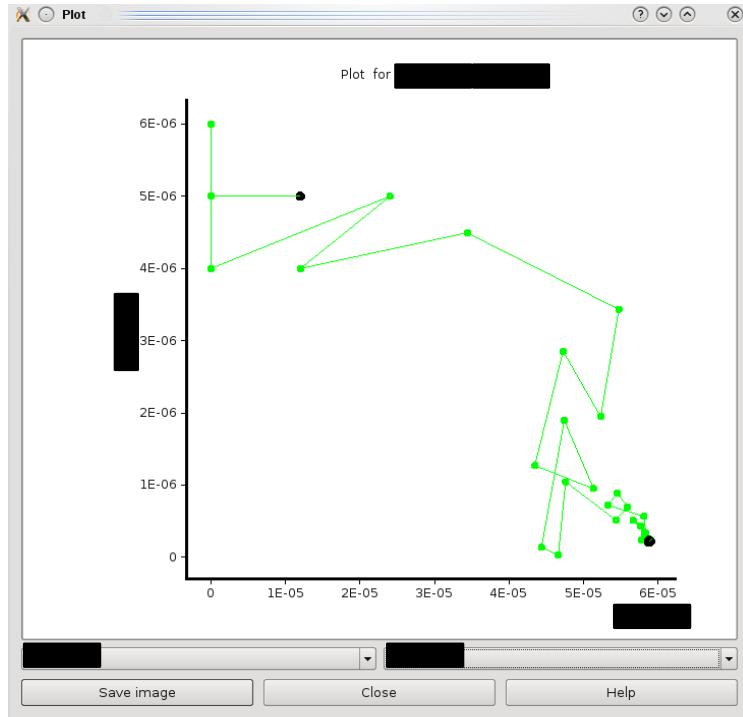
The function surface can be visualized through polynomial approximation, where the polynomials models similar to the models that are presented in Section 12.1 are created. A difference is that the polynomials do not necessarily interpolate the points since the coefficients are determined through a least squares method. If  $n > 2$ , then several plots are created. In that case, all but two of the variables are fixed to values supplied by the user, for each plot. Figure 18.4 shows a polynomial approximation, in this case a quadratic polynomial, of the objective function. The white circles correspond to points in which the objective function has been evaluated, and the size of the circles corresponds to the difference between the polynomial and the approximation.



**Figure 18.4:** A polynomial approximation of the objective function, as displayed in Beauty.

It is also possible to visualize how the optimization algorithm moves in the variable space. The user chooses two variables and essentially gets a top-down view of the variable space, where the points in which the objective function has been evaluated are marked. If the user clicks on a point, a window is displayed that shows the point's coordinates and objective function value. Figure 18.5 shows this such a top-down view.

## 18. Implementation Details



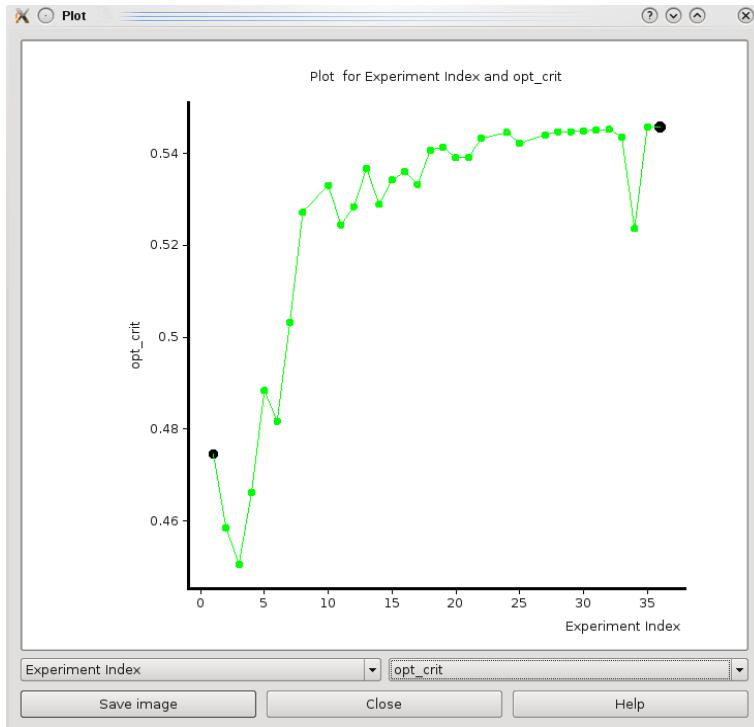
**Figure 18.5:** A top-down view of how the optimization moves in the variable space.

To get more information about the different points and their values, and to see the highest/lowest values of all parameter values as well as the objective function value, the user can use a table that displays all objective function evaluations in the order that they were performed. If a point does not fulfill all constraints, this is also displayed here. An example of such a table is available in Figure 18.6. There, the leftmost table displays the filename of the optimization, the next column displays whether the simulation has been performed or not. Some simulations were not performed since they did not fulfill all constraints. The third column displays the ID number of the optimization run, 1 in this case. The fourth and fifth columns display the variable values for the problem. The sixth and final column displays the objective function value, which was to be maximized, as indicated by the plus sign at the end of the name. The cells with the highest/lowest variable values and objective function values are indicated by red and blue, respectively.

Experiment	Status	StartIdx		opt_crit+
Experiment set: [REDACTED]	Done	1	1.2e-5	0
[REDACTED]	Done	1	0	1e-6
[REDACTED]	Done	1	0	0.4884
[REDACTED]	Done	1	2.4e-5	0
NONE	Out of boundaries	1	0	-1e-6
NONE	Out of boundaries	1	1.2e-5	-1e-6
[REDACTED]	Done	1	3.44054655538247e-5	4.98095585763133e-7
[REDACTED]	Out of boundaries	1	2.51165615650363e-5	-1.3499791425983e-7
[REDACTED]	Done	1	4.63199833195984e-5	6.17243789725162e-7
[REDACTED]	Done	1	5.49727637593721e-5	1.3101117452841e-6
NONE	Out of boundaries	1	6.99618110042698e-5	1.3528261696074e-6
[REDACTED]	Done	1	4.989592467237e-5	2.21620821098588e-6
[REDACTED]	Done	1	4.9296351816066e-5	1.50068357445275e-6
NONE	Out of boundaries	1	6.01901310883412e-5	2.21064758163148e-6
[REDACTED]	Done	1	5.54436838423059e-5	8.11653593337741e-7
[REDACTED]	Done	1	5.30076484039385e-5	9.05407531885315e-7
[REDACTED]	Done	1	5.45692718134822e-5	6.00252782543404e-7
[REDACTED]	Done	1	5.70220504090098e-5	5.0958101639846e-7
NONE	Out of boundaries	1	6.17451829092384e-5	2.97252337333748e-7
[REDACTED]	Done	1	5.85280687384908e-5	6.94646860373035e-7
[REDACTED]	Done	1	5.87396983535236e-5	9.17557108038265e-7
NONE	Done	1	5.67345889811443e-5	7.11471641193212e-7
[REDACTED]	Out of boundaries	1	6.04280496145896e-5	5.36751134618596e-7
[REDACTED]	Done	1	5.99117622973643e-5	8.86230033041012e-7
[REDACTED]	Done	1	5.72696188588681e-5	6.55889791082601e-7
[REDACTED]	Done	1	5.82478682420573e-5	7.38859704299258e-7
[REDACTED]	Done	1	5.89894282910159e-5	7.26613353604863e-7
[REDACTED]	Done	1	5.79384519702758e-5	7.03908514682759e-7
[REDACTED]	Done	1	5.8712364237946e-5	7.42229768199546e-7
[REDACTED]	Done	1	5.86478515439375e-5	6.45653374445529e-7
[REDACTED]	Done	1	5.8360864855232e-5	6.90156462028258e-7

**Figure 18.6:** A table displaying the variable values and the objective function values. The rightmost column displays the objective function value, and the two columns immediately to the left of it displays the values of the two variables. The highest and lowest values are indicated by red and blue shadows of the cells, for the variables and the objective function value.

To see the development of the objective function value as optimization progresses, a plot with the evaluations on the x-axis and the objective function values on the y-axis is available. Such a plot is available in Figure 18.7. In the figure, the objective function was maximized and we can see the typical behavior that the improvement is the largest in the beginning of the optimization and then improvement gradually decreases as optimization progresses.



**Figure 18.7:** A plot how the objective function value has changed during the evaluations. The first six evaluations are for building the initial model, since this plot is taken from a two-dimensional problem. Here we have defined our own objective function (see Section 10.3 for more information) that was maximized, whose values are in the interval  $(0, 1]$ .

## 18.8 Summary

Implementation is a very important, but often neglected, part of optimization software. Often only a very rudimentary implementation is done, and it is common that such an implementation ignores error detection and handling of any detected errors. The implementation also affects how easy or difficult it is to extend the optimization software with new algorithms and parallelization.

In this chapter we presented the thoughts and considerations behind the choice of implemented algorithm and third-party software. We also presented the implementation principles that we used throughout the implementation process to get an implementation that is extendable, parallel, well tested and easy to understand. We have also described the termination criteria that we use and the implementation of parallelization. We also gave a brief overview of the software architecture.

Furthermore, we also gave a description of how the standalone implementation was

modified in order to integrate it with the BEAST software. The greatest difference is related to how the parallelization works, since the version that is integrated in BEAST uses asynchronous parallelization. There are several reasons for this, e.g. to make effective use of the available computational power and also the fact that the time required for evaluation of the objective function may vary considerably.

The way constraints are handled in BEAST as well as the implications for optimization algorithms were also discussed in more detail in this chapter. In particular, the fact that points that do not fulfill all constraints are assigned a poor value, is a problem in cases where the optimum is located in a point where one or more constraints are active. Since the model that is used is a second-degree interpolating polynomial, it will in some cases lead the optimization algorithm in the wrong direction.

We have also described implicit constraints and that we handle them by alerting the user, since it turns out that a point was invalid, when it was previously believed that the point was valid.

Finally we described the different ways that information about the results can be visualized to the user, even while the optimization is running. Of special interest is probably the polynomial approximation of the objective function that is created by the Beauty software, as well as the plot where the user can see the objective function values for the different simulations. The approximation of the objective function is interesting to the user since he then can get a visual description of the function. The plot of the objective function values is interesting since the user can see the development of the objective function value over the course of the simulations.

## 18. Implementation Details

---

---

## Testing on Synthetic Test Cases

We have implemented almost all of the suggested parallel extensions suggested in the previous chapters. However, we have not implemented the model building from existing points (see Section 17.3), nor the automatic separability as discussed in Section 17.1. The other extensions have been implemented and here we present test case results from the MVF test case suite [1], which were chosen since they were readily available in the C language. In the following tables, each line represents a test case. For each test case, a random start position was determined. Naturally, within each test case, the start position was the same when comparing different parameter settings etc.

### 19.1 Test Case Specifications

Table 19.1 displays the test case name, number of dimensions and the intervals for the variables. The intervals are not considered bounds, but are rather used to limit the search to an interesting area. Test cases with variable number of dimensions have been marked with ‘var’ in the dimensions column.

The last column is an attempt to characterize the test cases. The characterization is done mostly with regards to whether a function is unimodal (single optima) or multimodal (multiple optima), and whether it is smooth or noisy. Whether a test case is judged to be smooth or noisy are rather subjective, and may not only depend on the amount of high frequency information in the objective function, but also on the size the interval. We have used the term edgy if a function is e.g. a sum of several absolute values. Test cases where we have been unable to analyze the function sufficiently and also unable find any information, are marked by ‘N/A’. A complete formal characterization of the test cases is beyond the scope of this thesis.

**Table 19.1:** Test case specifications.

Name	$n$	Interval	Characterization
Ackley	var	$[-30.0, 30.0]$	Highly multimodal, noisy.
Beale	2	$[-4.5, 4.5]$	Unimodal, smooth.
Bohachevsky1	2	$[-100.0, 100.0]$	Multimodal, fairly smooth.
Bohachevsky2	2	$[-50.0, 50.0]$	Multimodal.
Booth	2	$[-10.0, 10.0]$	Multimodal, smooth.
BoxBetts	3	$x_1 \in [0.9, 1.2]$ $x_2 \in [9.0, 11.2]$ $x_3 \in [0.9, 1.2]$	Multimodal.
Branin	2	$x_1 \in [-5.0, 0.0]$ $x_2 \in [0.0, 15.0]$	Multimodal with three global minima, smooth.
Branin2	2	$[-10.0, 10.0]$	Multimodal.
Camel3	2	$[-5.0, 5.0]$	Multimodal, smooth.
Camel6	2	$[-5, 0, 5.0]$	Multimodal with two global and four local minima, smooth.
Chichinadze	2	$[-30.0, 30.0]$	Unimodal, edgy.
Colville	4	$[-10.0, 10.0]$	Multimodal, smooth.
Corana	4	$[-100.0, 100.0]$	Very multimodal.
Eggholder	var	$[-512.0, 512.0]$	Multimodal, smooth.
FreudensteinRoth	2	$[-15.0, 15.0]$	Multimodal, smooth.
Gear	4	$[12.0, 60.0]$	Multimodal, edgy.
Generalized Rosenbrock	var	$[-100.0, 100.0]$	Unimodal, smooth.
Goldstein-Price	2	$[-2.0, 2.0]$	Multimodal, smooth.
Hansen	2	$[-10.0, 10.0]$	Multimodal, noisy.
Hartman 3	3	$[0.0, 1.0]$	Multimodal with three local minima.
Hartman 6	6	$[0.0, 1.0]$	Multimodal with six local minima.
Himmelblau	2	$[-5.0, 5.0]$	Multimodal with four global minima, smooth.
Holzman 2	var	$[-10.0, 10.0]$	N/A.
Hosaki	2	$x_1 \in [0.0, 5.0]$ $x_2 \in [0.0, 6.0]$	Smooth.
Hyperellipsoid	var	$[-5.12, 5.12]$	Unimodal, smooth.
Kowalik	4	$[-5.0, 5.0]$	Unimodal.
Leon	2	$[-1.2, 1.2]$	Unimodal, smooth.
Levy7	var	$[-32.0, 32.0]$	Multimodal, smooth.
Matyas	2	$[-10.0, 10.0]$	Unimodal, smooth.
Max mod	2	$[-10.0, 10.0]$	Unimodal, edgy.
McCormick	2	$x_1 \in [-1.5, 4.0]$ $x_2 \in [-3.0, 4.0]$	Unimodal, smooth.

*Continued on next page*

Name	$n$	Interval	Characterization
Michalewitz	2	$[0, \pi]$	Multimodal, smooth.
Multimod	var	$[-10.0, 10.0]$	Unimodal, smooth.
Neumaier Perm0	var	$[-1.0, 1.0]$	N/A.
Neumaier Perm	4	$x_i \in [-i, i]$	N/A.
Neumaier Power Sum	4	$[0.0, 4.0]$	N/A.
OddSquare5	5	$[-5\pi, 5\pi]$	N/A.
OddSquare10	10	$[-5\pi, 5\pi]$	N/A.
Plateau	5	$[-5.12, 5.12]$	Unimodal, edgy.
Quartic Noise U	var	$[-1.28, 1.28]$	Multimodal, noisy.
Rana	var	$[-500.0, 500.0]$	Highly multimodal, smooth.
Rastrigin	var	$[-5.12, 5.12]$	Highly multimodal, smooth.
Rastrigin 2	2	$[-5.12, 5.12]$	Multimodal, smooth.
Schaffer 1	2	$[-100.0, 100.0]$	Multimodal.
Schaffer 2	2	$[-100.0, 100.0]$	Multimodal.
Schwefel1_2	var	$[-10.0, 10.0]$	Unimodal, smooth.
Schwefel2_21	var	$[-10.0, 10.0]$	Edgy.
Schwefel2_22	var	$[-10.0, 10.0]$	Unimodal, edgy.
Schwefel2_26	var	$[-512.0, 512.0]$	Highly multimodal, smooth.
Shekel2	2	$[-65.536, 65.536]$	Multimodal, smooth.
Shekel4_5	4	$[0.0, 10.0]$	Multimodal, smooth.
Shekel4_7	4	$[0.0, 10.0]$	Multimodal, smooth.
Shekel4_10	4	$[0.0, 10.0]$	Multimodal, smooth.
Shubert	2	$[-10.0, 10.0]$	Highly multimodal, noisy.
Shubert2	var	$[-10.0, 10.0]$	Highly multimodal, noisy.
Shubert3	2	$[-10.0, 10.0]$	Highly multimodal, noisy.
Sphere	var	$[-10.0, 10.0]$	Unimodal, smooth.
Sphere2	var	$[-10.0, 10.0]$	Unimodal, smooth.
Step	var	$[-100.0, 100.0]$	Unimodal, edgy.
Stretched V	var	$[-10.0, 10.0]$	Highly multimodal, smooth.
Sum Squares	var	$[-32.0, 32.0]$	Unimodal, smooth.
Trecanni	2	$[-5.0, 5.0]$	Unimodal, smooth.
Trefethen4	2	$x_1 \in [-6.5, 6.5]$ $x_2 \in [-4.5, 4.5]$	Highly multimodal, noisy.
Watson	6	$[-10.0, 10.0]$	Multimodal.

## 19.2 Test Case and Algorithm Settings

In the testing of the algorithms on the synthetic test cases, we used the following settings. The start value of the trust region radius  $\rho_{beg}$  was set to a percentage of the variables'

interval (see Section 19.7 for more information about this). If there were different variable intervals, then the smallest interval determined the value of  $\rho_{beg}$ .

For each test case, the starting point was randomized within a region that allowed the complete model to be within the specified interval for each region. This means that the starting point was randomized within a smaller region as the value of  $\rho_{beg}$  increased. The trust region radius  $\Delta$  was set to  $\rho_{beg}$  and  $\rho_{end}$  was set to 10E-7.

We used the three termination criteria:  $\rho_k \leq \rho_{end}$ , at most 1000 NPEs after building the initial model, and at most 1000 iterations in the main loop. If any of these values were fulfilled, then the algorithm was terminated. This means that if the algorithm failed to converge to a stationary point within the limits on function evaluations and iterations, it will be terminated due to the latter criteria.

We chose  $n = 4$  for the test cases with variable dimensionality.

The testing on synthetic test cases was performed on a PC with 4GB RAM and a dual core Intel i5 CPU running at 2.67GHz. Each core has two hardware threads, therefore the operating system sees the CPU as having four separate cores. For this reason, the computer did most four parallel evaluations of the objective function during testing. However, in our tests we sometimes send more than four points for parallel evaluation. In such cases, the points are evaluated in batches of four at a time. Since we use synchronous parallelization in this testing, the algorithm does not know whether all points are really evaluated in parallel.

### 19.3 Interpreting the Test Results

Solving the problems in the setting of interest here is actually a multicriteria optimization in itself. It is desirable to find a low objective function value, but since each objective function evaluation is very time-consuming, it is also desirable to use as few function evaluations as possible. Whether a lower objective function value is “worth” a certain number of evaluations is difficult to determine in advance. In the testing that is presented in this chapter, we will often encounter the situation that one parameter setting will give a lower objective function value than another, but the number of required function evaluations is greater. Here we will favor a lower objective function value, even if this requires more evaluations. However, we are not happy with a large increase in the number of evaluations for a very small improvement in the objective function value, so the tradeoff is somewhat arbitrary. In practical applications, this tradeoff must be done by the user, and can depend on e.g. the particular problem at hand, available time and computational resources.

For the cases where there is a single optimization run that generates a single point in each iteration,  $NPE$  is always equal to one, so we do not distinguish between  $NPE$  and the number of function evaluations, once the initial model has been built.

When we build the initial model, we need to evaluate  $p_1 = \frac{1}{2}(n + 1)(n + 2)$  points. Therefore, when we test building the initial model in one step, we assume that we can

evaluate this many points in parallel, since we must wait until all points have been evaluated before execution can continue.

Due to the scope of the test log files, we elected to not include them in this thesis. Instead, the complete test results are available in Olsson [53].

Some abbreviated test results will be displayed in a table, see e.g. Table 19.3 on page 252. To display how different settings affect the number of function evaluations that are used, we use data profiles [48]. A data profile is a cumulative distribution function, and thus monotonically increasing, step function with range  $[0,1]$ . The x-axis of a data profile displays the number of times one or more points were sent for evaluation (i.e.  $NPE$ ), and the y-axis displays the fraction of test cases. Each setting produces a data profile, and on each line, each marker symbolizes a particular test case. For each number of  $NPE$ , the line's y-value displays the fraction of test cases that were solved using the specified number of function evaluations. However, an algorithm that is aborted as the number of available  $NPEs$  was exhausted is also considered to have solved the problem for the sake of showing the number of  $NPEs$  in a data profile. A user can use data profiles to determine which setting is most likely to solve a certain problem, given a budget on  $NPE$ , since a line with a higher fraction is likely to be better.

The number of function evaluations measure is a relevant measure of the time for solving the kinds of problems that we are interested in since it is assumed that the function evaluations take the majority of the solution time. However, since the data profiles only measure the number of function evaluations, there is no information about the objective function value that a certain algorithm achieved. To display the achieved objective function values, we choose the objective function values achieved by a certain setting as a baseline and compare the objective function values achieved by the other settings to the baseline. An example of such a plot is available in Figure 19.5 on page 250. The x-axis displays the achieved test case results for the baseline setting, and the y-axis displays the results for an alternative setting. Each test case result is displayed by a marker. If the results for a certain test case were equal for both the baseline and the alternative value, the marker would be on the diagonal line. If a lower (better) value was achieved with the alternative setting, the marker is displayed below the diagonal line and if a higher (worse) result was achieved, then the marker is above the line. If the result for the alternative setting was achieved using fewer evaluations, the marker is a cross ( $\times$ ) and if it was achieved with a greater number of evaluations, the marker is a circle ( $\circ$ ).

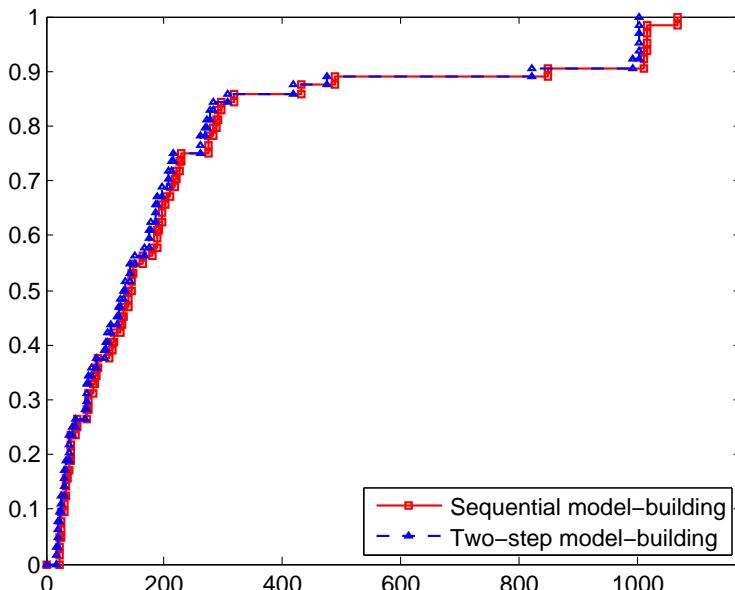
To make the figures of the objective function values clearer, any test cases with significantly higher or lower objective function values have been left out. If there is anything worth commenting about these test cases then this will be mentioned explicitly.

## 19.4 Building the Initial Model in two Steps

We mentioned in Section 15.3 that in the original UOBYQA algorithm, all  $\frac{1}{2}(n+1)(n+2)$  points are evaluated in sequence, even if the placement of the  $n+1$  first points is

deterministic and the remaining points are independent of each other, and only depend on the first  $n + 1$  points. In our implementation, we therefore elected to always evaluate the  $n + 1$  first points in parallel. However, for the sake of completeness, we have made a data profile to display the results if all function evaluations for building the initial model were performed in sequence. This data profile is available in Figure 19.1 and we can see that the two step model building is always slightly better, since  $NPE$  always decrease by  $\frac{1}{2}(n + 1)(n + 2) - 2$ , for each test case. Since the models are identical, the objective function values are the same.

Building the initial model in two steps model instead of  $\frac{1}{2}(n + 1)(n + 2)$  steps yields no disadvantages, so it will be used in all testing in this thesis.

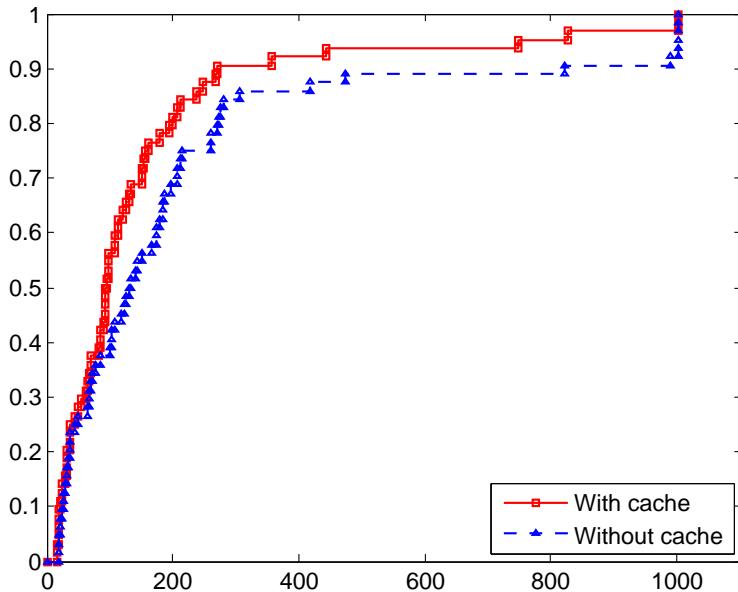


**Figure 19.1:** Data profiles when building the initial model completely sequentially or in two steps.

## 19.5 Effect of a Cache of Evaluated Points

In this section we test the effect of storing all points and their values after evaluation. Then, when a point is added to the queue of points to be evaluated, it is checked if the same point has already been evaluated. If so, the value is returned and the execution can continue. See Section 16.2 for more information. For all test cases here,  $\rho_{beg}$  was set to 1% of the most constrained variable's interval.

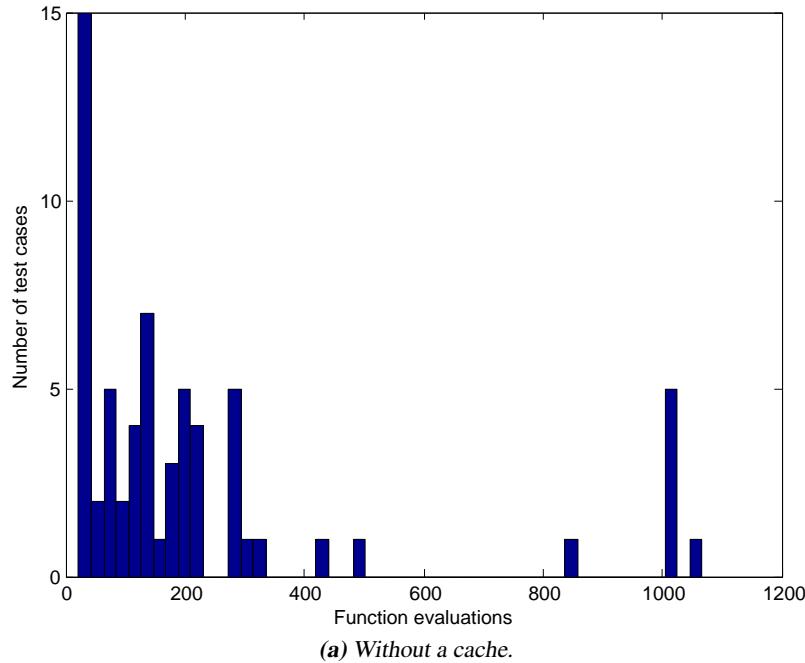
Figure 19.2 displays the data profile for using a cache and not using a cache. From the figure it is obvious that using a cache of points can substantially decrease the number



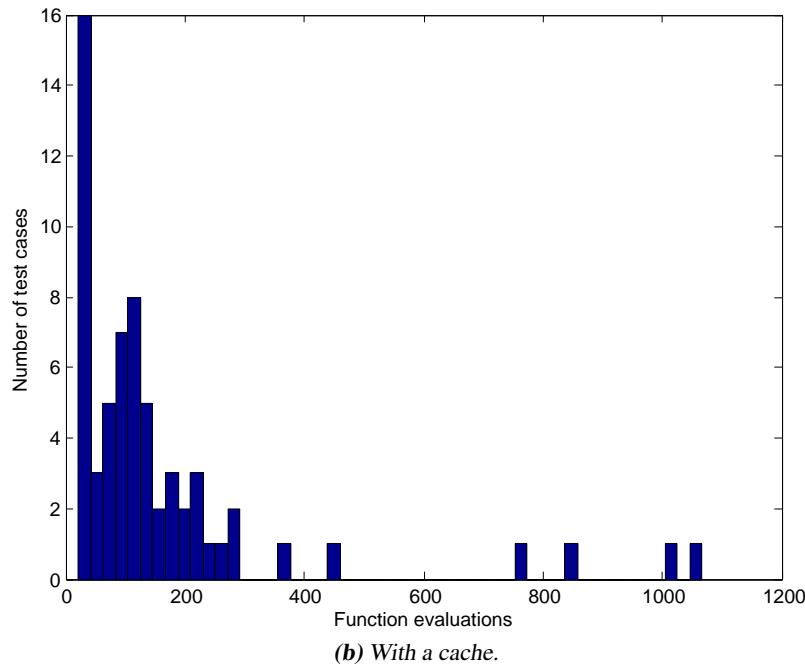
**Figure 19.2:** Data profiles when not using and when using a cache of stored points.

function evaluations. The decrease in the number of function evaluations is lower when few evaluations are required to solve the problem, and there are only a handful test cases where the number of function evaluations do not decrease. This is typically due to the fact that such test cases were not terminated due to convergence to a stationary point, but rather due to the limit on the number of function evaluations. The main decrease is for test cases that fall in between these two extremes. There are also some test cases where the number of function evaluations decrease by a large percentage, e.g. Colville, Hartman 3 and Rana, where the number of function evaluations decrease by 80–90%. Figure 19.3 display histograms of the number of function evaluations for when not using a cache (Figure 19.3a) and when using a cache (Figure 19.3b). The figures show that the test cases are typically solved with considerably fewer function evaluations when using a cache. In all cases an optimum remains the same.

Since using a cache yields no disadvantage in the kind of test cases that we are interested in, we will use it in the remainder of the testing in this thesis.



(a) Without a cache.



(b) With a cache.

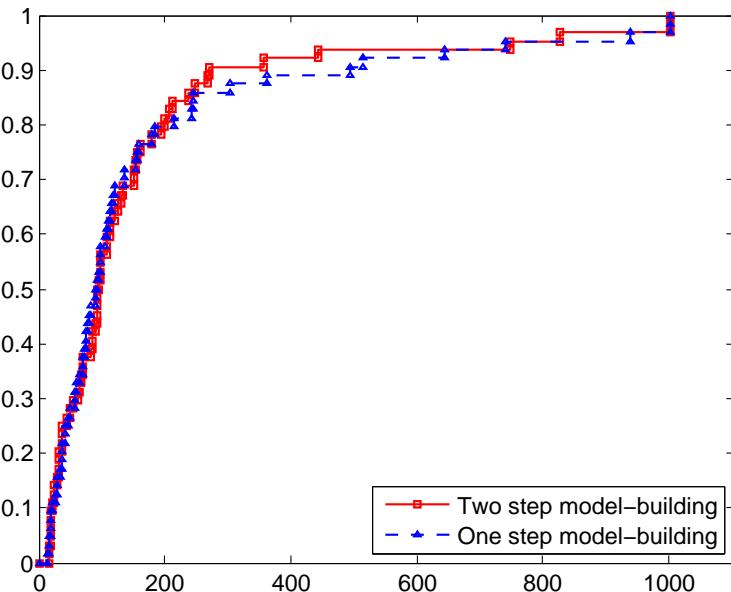
**Figure 19.3:** Histogram of the number of function evaluations without a cache (19.3a) and with a cache (19.3b).

## 19.6 Building the Initial Model in One Step

Here we present the results when building the initial model in one step as discussed in Section 15.3. The positions of all points are determined in advance, and then they are evaluated in parallel. This means that the positions of the points are not biased in directions that are believed to yield better objective function values.

In the original UOBYQA,  $n + 1$  points are placed first and then evaluated in sequence (see Section 19.4), and then the remaining points are placed. This means that the points are evaluated in two batches, if a sufficient number of computers are available.

Figure 19.4 shows data profiles for both one step and two step model building. We can see that the number of function evaluations is quite similar between the two settings, but the two step model building often requires slightly fewer function evaluations.

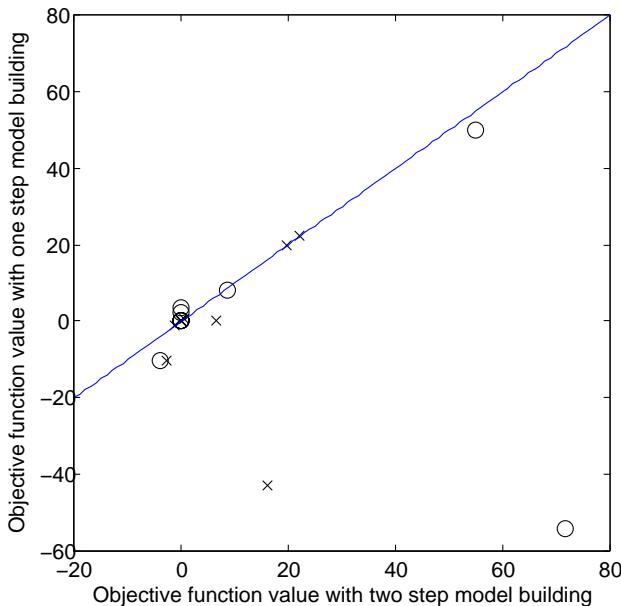


**Figure 19.4:** Data profiles for building the initial model in one or two steps.

Figure 19.5 plots the objective function values for the test cases. We can see that in some cases the one step model building performs better and in some cases it performs slightly worse. Lower objective function values are achieved for some test cases, but sometimes at the expense at a greater number of test cases.

The results of the test cases Corana, Eggholder, Neumaier Perm, Rana and Shekel2 have been removed in order to make the figure clearer, since the magnitude of the objective function values for these test cases are considerably larger than for all other test cases. For the Corana test case, the two step model building achieved a lower objective function value, but required a few more evaluations. In the other cases, the same objective function values were achieved, but the one step model building required fewer evaluations.

Since the initial models are different, the algorithm takes different paths through the search space. This often leads to different number of function evaluations and to different final objective function values. In some cases the one step model building yields fewer objective function evaluations as well as a better objective function value, and in some other cases the opposite applies. Whether any additional function evaluations resulting from an initially worse model are compensated during the execution depends on the particular objective function and the time required for a function evaluation.



**Figure 19.5:** Plot of the objective function values when using two step (x-axis), and one step model building (y-axis).

The only way to be sure that one gets an initial model that is at least as good as when using two step model building is to evaluate all  $2^n$  corner points of the hypercube centered in the starting point, together with all points in the  $n + 1$  dimensional plus sign and choosing the  $\frac{1}{2}(n + 1)(n + 2)$  points with the lowest objective function value that spans the space. Table 19.2 lists the number of required points, the number of points in the  $n$ -dimensional plus sign and the number of corner points in the aforementioned hypercube, for different values of  $n$ . It quickly becomes apparent that even evaluating all corner points requires a huge number of computer as  $n$  grows. We believe that additional information about the objective function is needed when using one step model building, in order to guarantee that we get a model that is at least as good as the model from two step model building. Since we do not have that information, we will not use one step model building in the tests here.

$n$	$\frac{1}{2}(n+1)(n+2)$	$2n+1$	$2^n$
2	6	5	4
3	10	7	8
4	15	9	16
5	21	11	32
6	28	13	64
7	36	15	128
8	45	17	256
9	55	19	512
10	66	21	1024
12	91	25	4096
15	136	31	32768

**Table 19.2:** Numbers related to model building in different dimensions.

## 19.7 Different Values of the Initial Trust Region Radius

Choosing the value of the initial trust region radius  $\rho_{beg}$  can potentially have a large effect on the performance of the algorithm. A greater value allows the model to capture information from a larger part of the interval for each variable, but it might model the objective function worse since the interpolation points are farther from each other.

It is common to use a fixed value, e.g.  $\rho_{beg} = 1$ , regardless of the size of the region. However, we feel that this is questionable since this in some cases covers a very large part of the feasible region while in other cases it only covers a very small part. To determine the effect, if any, of different values of  $\rho_{beg}$ , we have performed testing where we set  $\rho_{beg}$  to a fraction of the smallest interval of any variable's interval. In the test cases used here, the variables' intervals are often similar for different variables, thus there should be no big imbalance between them.

We have tested with  $\rho_{beg} = 1, 2, 5, 10, 15, 20, 25$  and 30% of the minimum variable space. Since the values 2, 5, 10, 15 and 20% were dominated in most cases, they are left out of the discussion in this section.

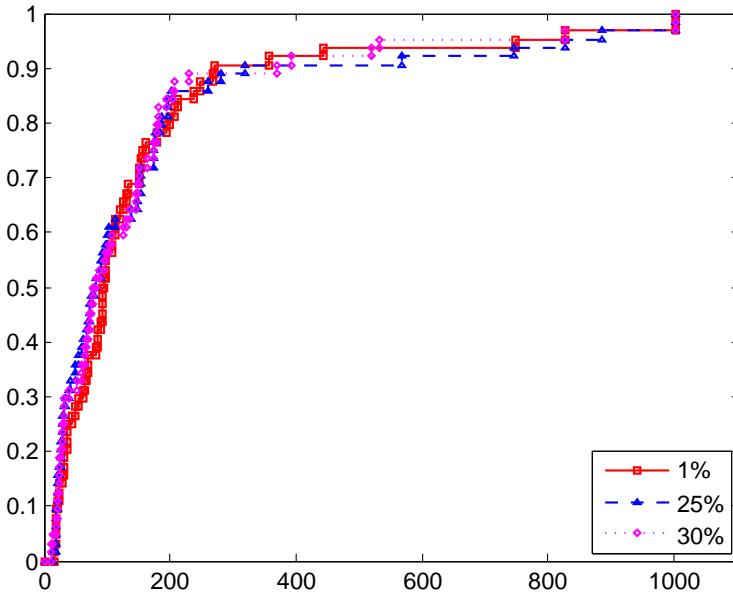
Table 19.3 displays some abbreviated results from these tests. The first line displays the total number of evaluations for all test cases and the second row displays the median number of function evaluations for the test cases. In all tables such as this one, the median number of function evaluations was rounded to the closest integer. The third line displays the number of test cases that each setting solved with the fewest number of function evaluations. If two or more values of  $\rho_{beg}$  solved a test case with the same minimal number of evaluations, that test case was added to all those values of  $\rho_{beg}$ .

Naturally, the number of function evaluations does not tell the complete picture, the

final objective function value is also important and the fourth line displays the number of test cases for which each setting yielded the lowest objective function value. Just like previously, if two or more values of  $\rho_{beg}$  provided the same minimal value for a particular test case, that test case was added to all such settings. The same was done if several values of  $\rho_{beg}$  lead to the same objective function value.

Initial radius $\rho_{beg}$	1%	25%	30%
Total evaluations	10791	11154	10405
Median evaluations	105	90	86
Cases with fewest evaluations	25	37	38
Cases with lowest values	22	52	53

**Table 19.3:** Results when varying the initial trust region radius.



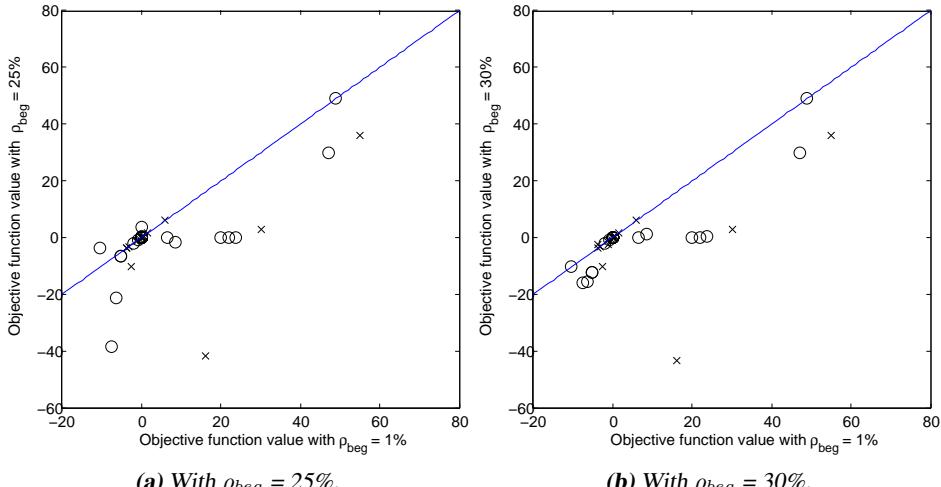
**Figure 19.6:** Data profiles for when varying the initial trust region radius.

Data profiles for varying the value of  $\rho_{beg}$  is available in Figure 19.6, where we can see that varying  $\rho_{beg}$  makes a difference in basically all of the test cases, but it is not obvious which value of  $\rho_{beg}$  that is better. In many of the test cases, a higher value of  $\rho_{beg}$  is better, but there are also a few cases where  $\rho_{beg} = 1\%$  is superior.

Figure 19.7 displays more information about what objective function values different values of  $\rho_{beg}$  achieved. The objective function values for  $\rho_{beg} 25\%-30\%$  are plotted versus the values for  $\rho_{beg} = 1\%$ . We use  $\rho_{beg} = 1\%$  of the most constrained variable space to get

a common baseline. We can see that numerous objective function values are improved, and also that the number of function evaluations for those test cases decrease. We note that both settings of  $\rho_{beg}$  improve a great number of test cases, without any significant degradation in other test cases.

Just like in the previous plots, we omit the objective function values for Corana, Eggholder, Neumaier Perm, Rana and Shekel2 from the plots.



**Figure 19.7:** Objective function values for  $\rho_{beg} = 25\%$  and  $30\%$ , plotted against the values for  $\rho_{beg} = 1\%$ .

By looking at Table 19.3, Figure 19.6 and Figure 19.7 together, one can see a quite clear trend that increasing the initial trust region radius yields better objective function values, while the number of function evaluations do not exhibit an obvious trend. This is not surprising since a greater value of  $\rho_{beg}$  decreases the region in which the initial starting point is randomized, forcing it towards the center of the feasible region. For each test case, the interval (see Table 19.1) is often set so that the global optimum close to the middle of the interval. Thus, a greater value of  $\rho_{beg}$  provides an initial position that is closer to the global optimum, and it also allows longer steps towards it. Since many of the test cases are multimodal, it is not surprising that starting closer to the global optima yields lower objective function values in many cases. For these reasons, we do not want to draw too much conclusions from these results.

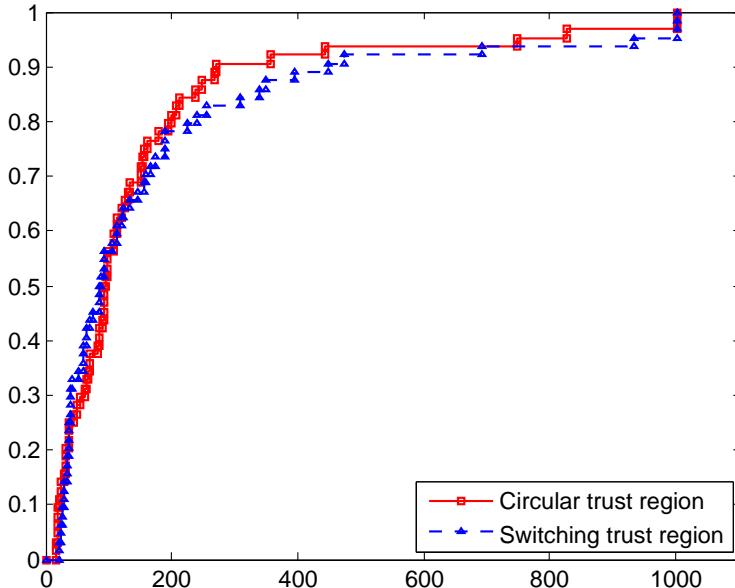
## 19.8 Non-spherical Trust Region Shapes

Here we present results from trust region shapes other than the standard hyper spherical. The value of  $\rho_{beg}$  was set to 30% of the most constrained variable's interval, and the other general settings were the same as in Section 19.2.

We will not present any results from testing static axis-aligned trust regions (see Section 15.4.1), since we in general cannot in advance determine along which axes the ellipses should be aligned.

### 19.8.1 Switching Elliptical Trust Region

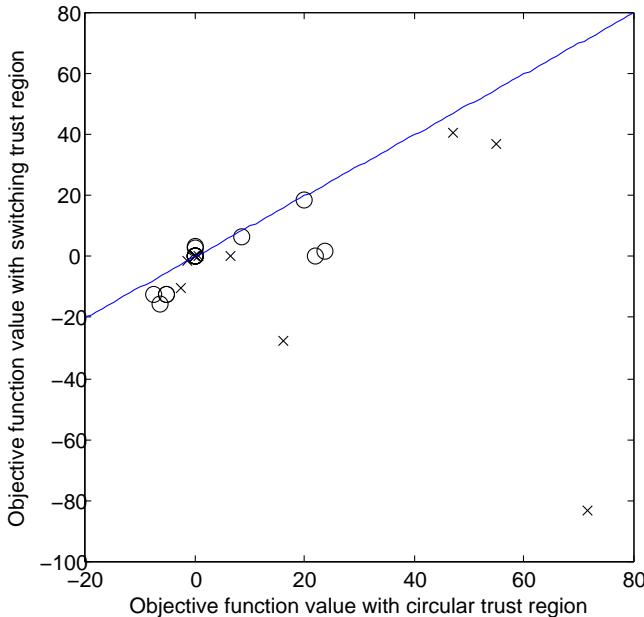
We have tested using an elliptical switching trust region, as discussed in Section 15.4.2. In our testing, the trust region in every iteration switched axis, from being elongated along the  $i$ :th coordinate axis to the  $(i + 1)$ :th coordinate axis in the next iteration, and so on. We used  $l_{\min} = 1$  and  $l_{\max} = 3$  in the testing.



**Figure 19.8:** Data profile using the switching trust region.

In Figure 19.8 we can see that the switching trust region solves some problems using a fewer number of iterations, but these are in general the test cases that require few objective function evaluations. For the more difficult test cases, that require a greater number of function evaluations, the circular trust region setting performs better.

From Figure 19.9 shows that for quite a few test cases, better objective function values were found, while in some other cases, the setting with a switching trust region shape ended up with worse objective function values, as compared to the circular trust region. A notable test case was Corona, where the number of function evaluations increased from 155 to 1016, but at the same time the objective function value decreased from 184197 to 57674.2.



**Figure 19.9:** Objective function values when using a switching trust region plotted versus objective function values when using the standard spherical trust region.

### 19.8.2 Elliptical Trust Regions

Here we present results from when testing an elliptical trust region, as described in Section 15.4.2. To avoid too elongated ellipses, we set  $l_{\min} = 1$  and  $l_{\max} = 3$ . As the “step”  $d_k$  in Algorithm 15.2, we used the movement of the best point, and the transformation matrix  $T$  was thus updated when there was a new best point, i.e. when  $f(x_k^+) < f(x_k)$ .

To measure the effect of different values of  $\gamma$ , we tested with  $\gamma = 0.0, 0.05, 0.15, 0.25, 0.35, 0.50, 0.6, 0.75$  and  $1.0$ . A value  $\gamma = 0$  is equal to using a circular trust region and is included for reference. A value  $\gamma = 1.0$  means the shape of the trust region depends completely on the latest movement of the best point, and all history is disregarded. The values  $\gamma = 0.15, 0.35$  and  $0.60$  are inferior most of the time. For this reason, they have been left out of the rest of this section.

Some abbreviated test results are available in Table 19.4. In the table, if several values of  $\gamma$  found the same lowest objective function value, then the test case was added to all such values of  $\gamma$ . For 25 test cases, the same optimal value was found regardless of the value of  $\gamma$ . To focus on the test cases where not all values of  $\gamma$  yielded the same objective function value, these 25 test cases have not been included in ‘Cases with lowest unique values’. Thus, that line only contains the test cases where not all values of  $\gamma$  yielded the same objective function value.

Looking at Table 19.4, the multiobjective nature of user’s problem becomes apparent.

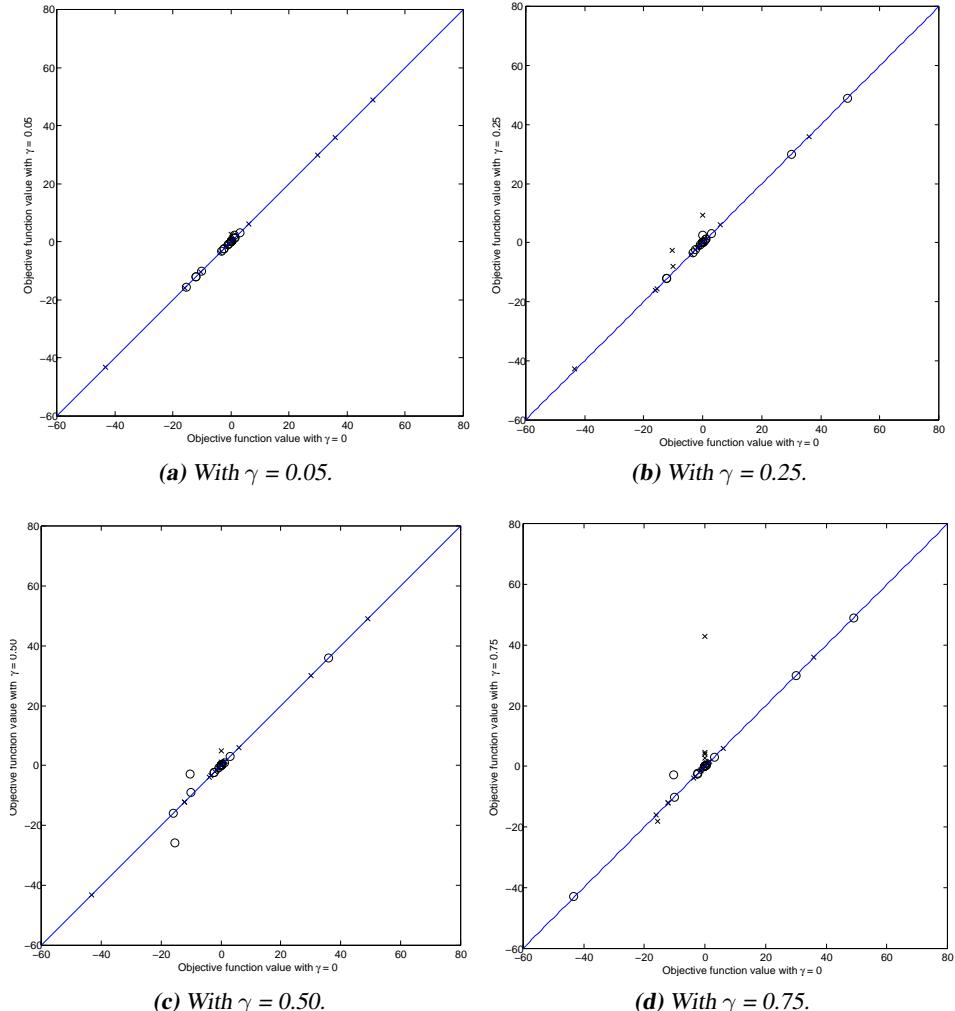
We can see that choosing  $\gamma$  greater than zero in all cases resulted in a lower total number of function evaluations, compared to the standard trust region whereas the median number varied. All different values of  $\gamma$  gave better objective function values than the other values of  $\gamma$  in at least two cases, but the standard trust region often converged to points with better objective function values. Thus, the standard trust region seems to be all-round, since it offers decent performance in all cases.

$\gamma$	0	0.05	0.25	0.50	0.75	1.00
Total evaluations	10405	9630	7999	8170	8372	6922
Median evaluations	86	100	94	88	87	80
Cases with fewest evaluations	18	13	19	18	23	23
Cases with lowest unique values	12	14	6	7	8	8

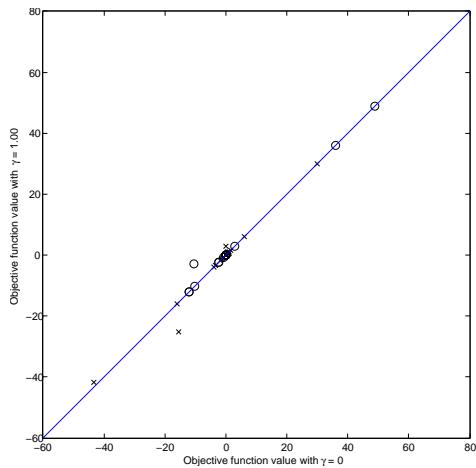
**Table 19.4:** Results when varying the forgetting factor  $\gamma$ .

Figure 19.11 displays data profiles for different values of  $\gamma$ . We can see that using a non-circular trust region is often very beneficial in that it requires fewer function evaluations. The objective function values, in Figure 19.10, show that there is no big difference in the achieved objective function values between the different values of  $\gamma$ . Some settings, e.g. 1.00, perform slightly worse for a few test cases, but overall there is no big difference. Just like in the previous plots, we omit the objective function values for Corana, Eggholder, Neumaier Perm, Rana and Shekel2 from the plots of achieved objective function values.

It is difficult to draw any definite conclusions from these tests. While increasing the value of  $\gamma$  can decrease the number of function evaluations, it might do so at the expense of the function value. An example of this is the results for  $\gamma = 0.25$  (Figure 19.10b). On the other hand, one can look at the results for  $\gamma = 0.50$  (Figure 19.10c), where many results are achieved with fewer evaluations, and when the objective function value is worse, the impairment is small.

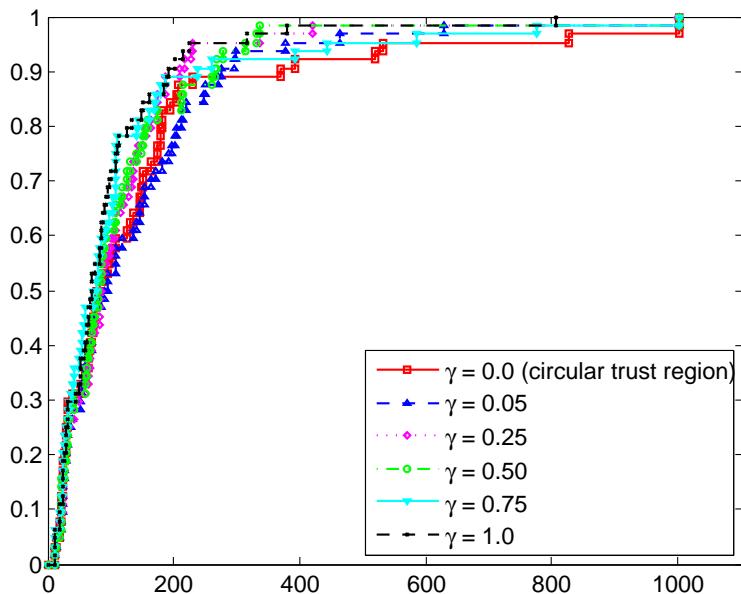


**Figure 19.10:** This figure shows the objective function values for different values of the forgetting factor  $\gamma$ .



(e) With  $\gamma = 1.00$ .

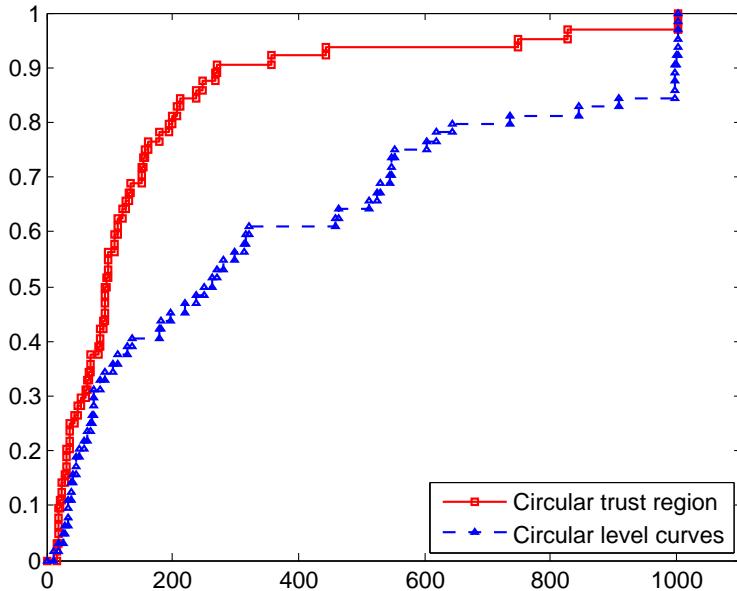
**Figure 19.10:** Objective function values for different values of the forgetting factor  $\gamma$ , cont.



**Figure 19.11:** Data profile when varying the forgetting factor  $\gamma$ .

### 19.8.3 Circular Level Curves

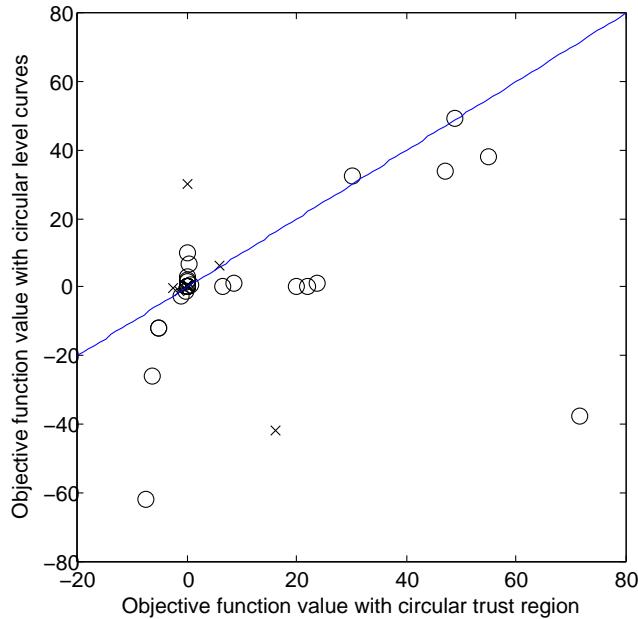
We have tested the transformation for creating circular level curves from Section 15.4.2.



**Figure 19.12:** Data profiles for when using the transformation that creates circular level curves, compared with the normal circular trust region.

Figure 19.12 displays the data profiles for the transformation that creates circular level curves as well as the original circular trust region, for comparison. It is obvious that the transformed problem requires a greater number of function evaluations. There are also a greater number of problems that are terminated for reasons other than convergence to a stationary point, as indicated by the blue triangles at around 1000 function evaluations. However, when looking at Figure 19.13, which displays the objective function values, we can see that the transformed trust region often achieves better objective function values than the circular trust region. The degradation in objective function values is typically small when the achieved objective function values are worse.

We conclude that the transformation that creates circular level curves often requires a greater number of function evaluations than the standard circular trust region, but achieved lower objective function values for quite a few test cases. Therefore, this transformation can be very useful, especially if it is important to achieve low objective function values and a large number of function evaluations can be performed.



**Figure 19.13:** Function values when using a normal circular trust region compared to using a transformed trust region to create circular level curves.

## 19.9 Testing of Parallel Extensions

In the following sections we present results of testing of the most of the parallel extensions that we have presented previously in this thesis.

### 19.9.1 Evaluating Points in Several Models

To test whether evaluating a point in several models makes any difference, we created an additional optimization run in addition to the first. The starting point of the first optimization run was the same as in the previous testing, and the second optimization run's starting point was randomly determined. We are interested in the effect that the second optimization run has on the first, therefore we show results only for the first optimization run. Table 19.5 displays some abbreviated results. Both optimization runs were allowed to execute concurrently, one iteration at a time, and instead of using only the first optimization run's model, the candidate points  $x_k^+$  was evaluated in both models, as described in Section 16.1. After evaluation in the two models, the points predicted value was used as before, i.e. to determine any change in the trust region radius the next iteration. All optimization runs used a spherical trust region.

We have tested with different values of the model weight exponent  $\alpha$ . For clarity, the first column in Table 19.5 displays the original results when using a single optimization

run with a circular trust region (we have symbolically marked the value of  $\alpha$  as infinite in this case). The other columns display the results for different values of  $\alpha$ . Higher values of  $\alpha$  decreases the influence of the second optimization run's model, and a sufficiently high value will cancel the influence completely.

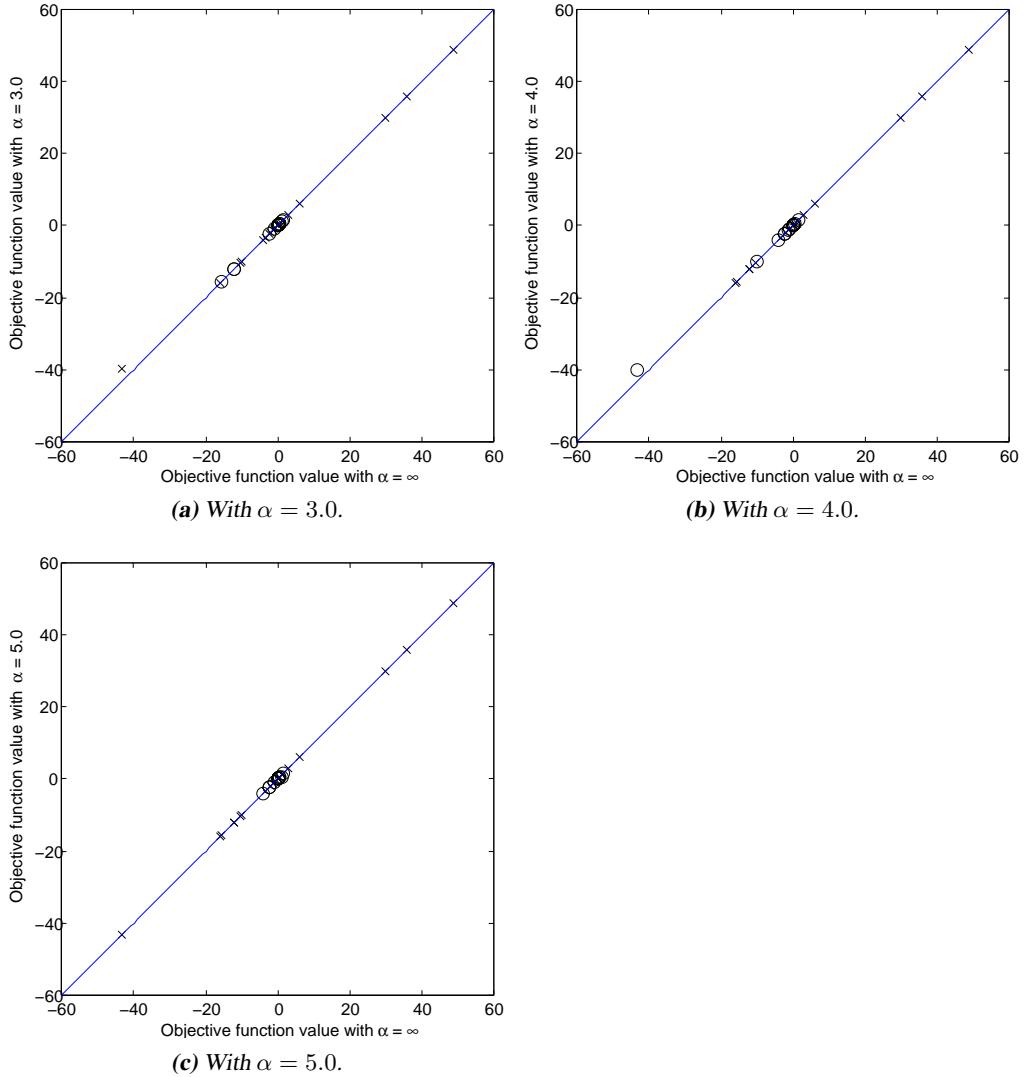
There were 30 test cases for which the best objective function value was equal for all values of  $\alpha$ , these have been excluded from the rows displaying the number of test cases with lowest unique values in Table 19.5.

$\alpha$	$\infty$	3	4	5
Total evaluations	10405	11262	11760	11620
Median evaluations	86	88	90	93
Cases with fewest evaluations	37	32	32	30
Cases with lowest unique values	8	19	21	15

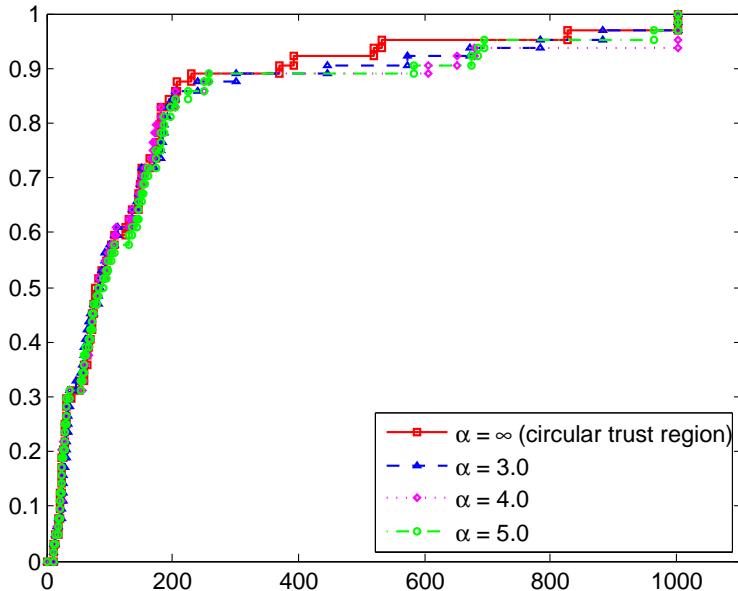
**Table 19.5:** Results when varying the model weight exponent  $\alpha$ .

Figure 19.15 displays the data profiles for the different values of  $\alpha$ . Evaluating a point in several models does not make a big difference in many test cases, but the difference increases when more than 200 function evaluations are required. Then, using several models seems to degrade the performance of the algorithm, at least when considering only the number of function evaluations. Comparing with Figure 19.14, which shows the objective function values, we can see that there are no big differences in objective function values between the different values of  $\alpha$ . The best results seem to be achieved with  $\alpha = 3 - 4$ , which have high number of test cases with unique lowest values, and a marginal increase in the median number of function evaluations. Using  $\alpha = 5$  not only increases the median and total number of function evaluations, it also leads to fewer cases with unique lowest values.

In some test cases the objective function values are decreased and many test cases are solved with fewer evaluations. Since we are difficulty seeing a clear trend in the data, we abstain from using evaluation in several models as the default in the rest of the testing.



**Figure 19.14:** Objective function values for different values of the model weight exponent  $\alpha$ .



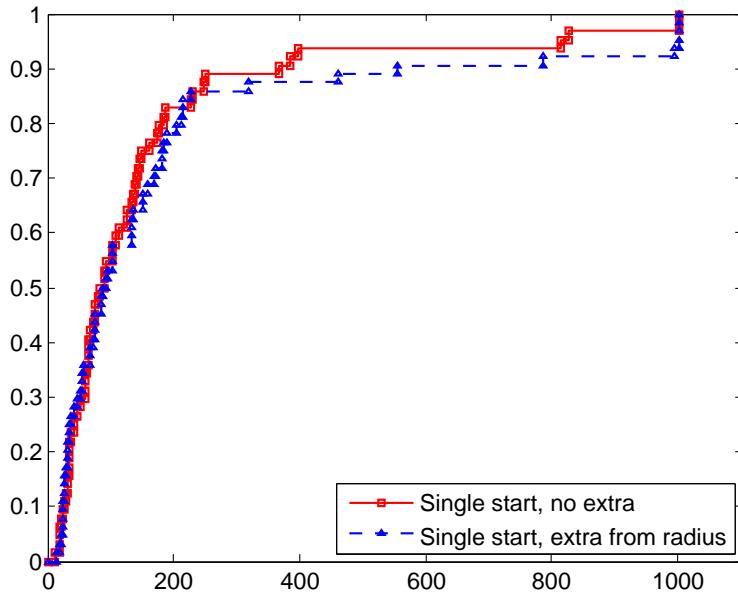
**Figure 19.15:** Data profiles when varying the model weight exponent  $\alpha$ .

## 19.9.2 Solving the Trust Region Subproblem With Different Radii

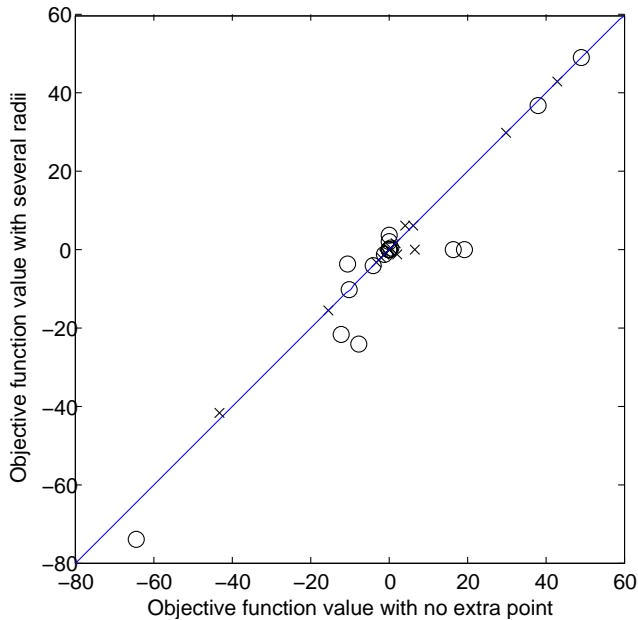
In this section we present and discuss the result of solving the trust region subproblem with several radii, as presented in Section 15.5.1. In that section we came to the conclusion that we could not see any advantage of using a smaller trust region radius than the original. Therefore, the trust region subproblem was only solved if the updated radius was larger than the original radius. The point with the lowest objective function value was included in the model in the next iteration, and the other point was ignored.

Data profiles of the results when solving with different radii according to above, is available in Figure 19.16. It is compared to the standard, with all settings equal. Up to about 100 NPEs, the settings are about quite similar. After that, the original setting is better.

When looking at the plot of objective function values (Figure 19.17), we see that this setting seems to be more efficient than just solving the subproblem with one radius. In many test cases a better objective function value is achieved, but at a greater number of evaluations. In the cases where the achieved objective function value is worse than the original setting, then the number of function evaluations is less.



**Figure 19.16:** Data profile when generating several points through solving the trust region subproblem with several radii.

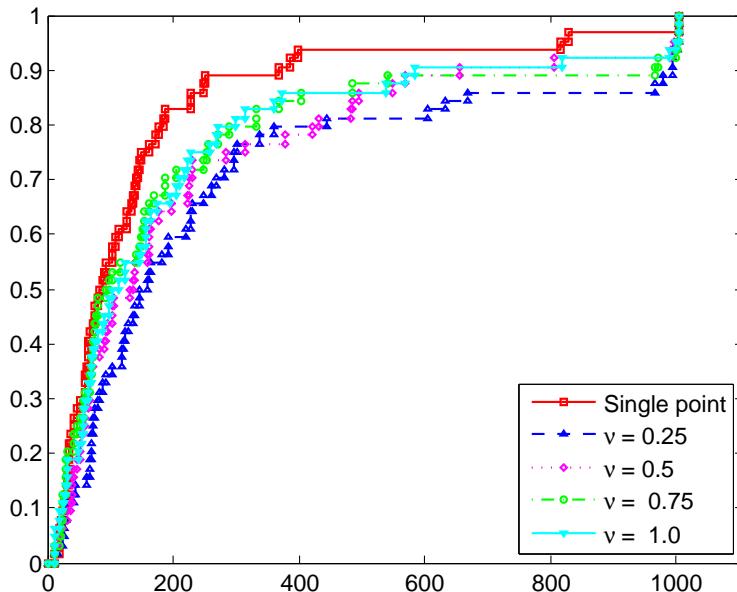


**Figure 19.17:** Objective function values when generating several points through solving the trust region subproblem with several radii.

### 19.9.3 Generation of Several Points in a Plus Sign Pattern

The results for generating several points in a plus sign centered in the original point are presented in this section.

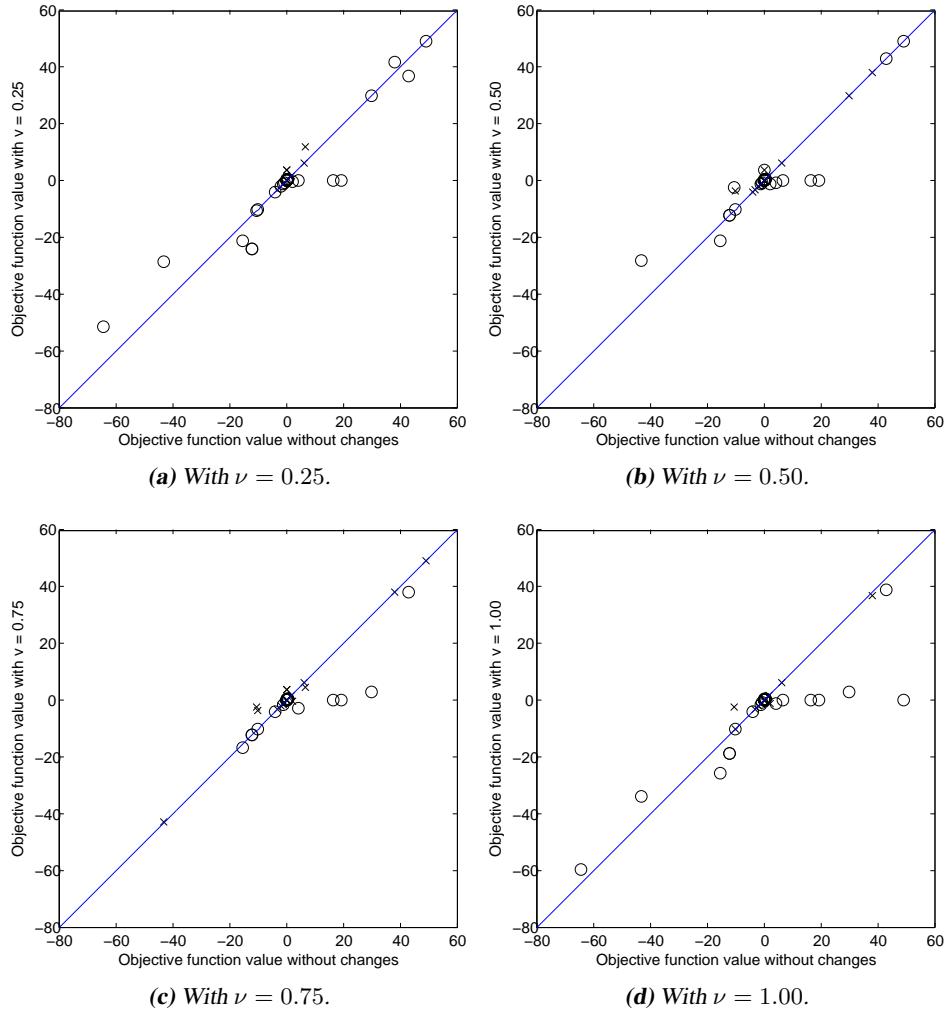
Figure 19.18 displays the data profile for the values 0.25, 0.50, 0.75 and 1.00 of the factor  $\nu$ , see Section 15.5.2 for more information about this value. As visible in the figure, we can see that generating a total of  $2n + 1$  points in each iteration actually increases the number of NPEs until the algorithm converges. Also, a greater number of test cases are aborted because the number of allowed function evaluations has been exhausted. It also seems like if one is to generate several points like this, a high value of  $\nu$  shall be used.



**Figure 19.18:** Data profiles when generating several points in a plus sign, with different values of the scale factor  $\nu$ .

In Figure 19.19 we can see that the resulting values in some cases are worse when several points are generated. For  $\nu = 0.75$ , there are equally many test cases that are improved as are impaired and for the other values, a majority are impaired. For  $\nu = 0.25$ , the ratio between improved and impaired is about 1:2 and for the other two tested values is the ratio about 2:3. For the test cases whose values are not displayed in Figure 19.19, the test results are the opposite, a majority of test cases were improved, for all values of  $\nu$  except 0.25, and in some cases the improvement was very large. For  $\nu = 0.25$ , a majority of the test cases got the same value.

A potential problem with using the plus sign is that it is axis-aligned. It could potentially be better to rotate it so that it extended in the direction of  $h$  from  $x$ .

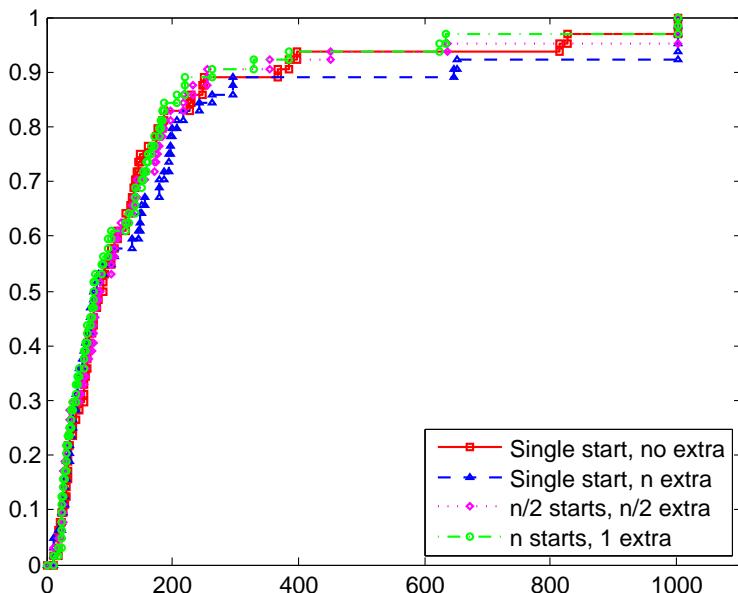


**Figure 19.19:** Objective function values for different values of the scale factor  $\nu$ .

### 19.9.4 Generation of a Fixed Number of Points

The results for generation of several points as described in Section 15.5.2 are presented in this section.

With this setting, it is possible to generate between 1 and  $n$  points in each iteration, per optimization run. Here we have tested the generation of a total of  $n + 1$  points in each iteration, where the number of points is achieved in different ways. The two extremes are: one optimization run that generates  $n$  extra points, and the other extreme is  $n$  optimization runs, where a single extra point is allocated where it is thought that they are the most beneficial. In between these extremes we have tested with  $n/2$  optimization runs, with  $n/2$  extra points. We had no restriction on the number of extra points generated per optimization run.



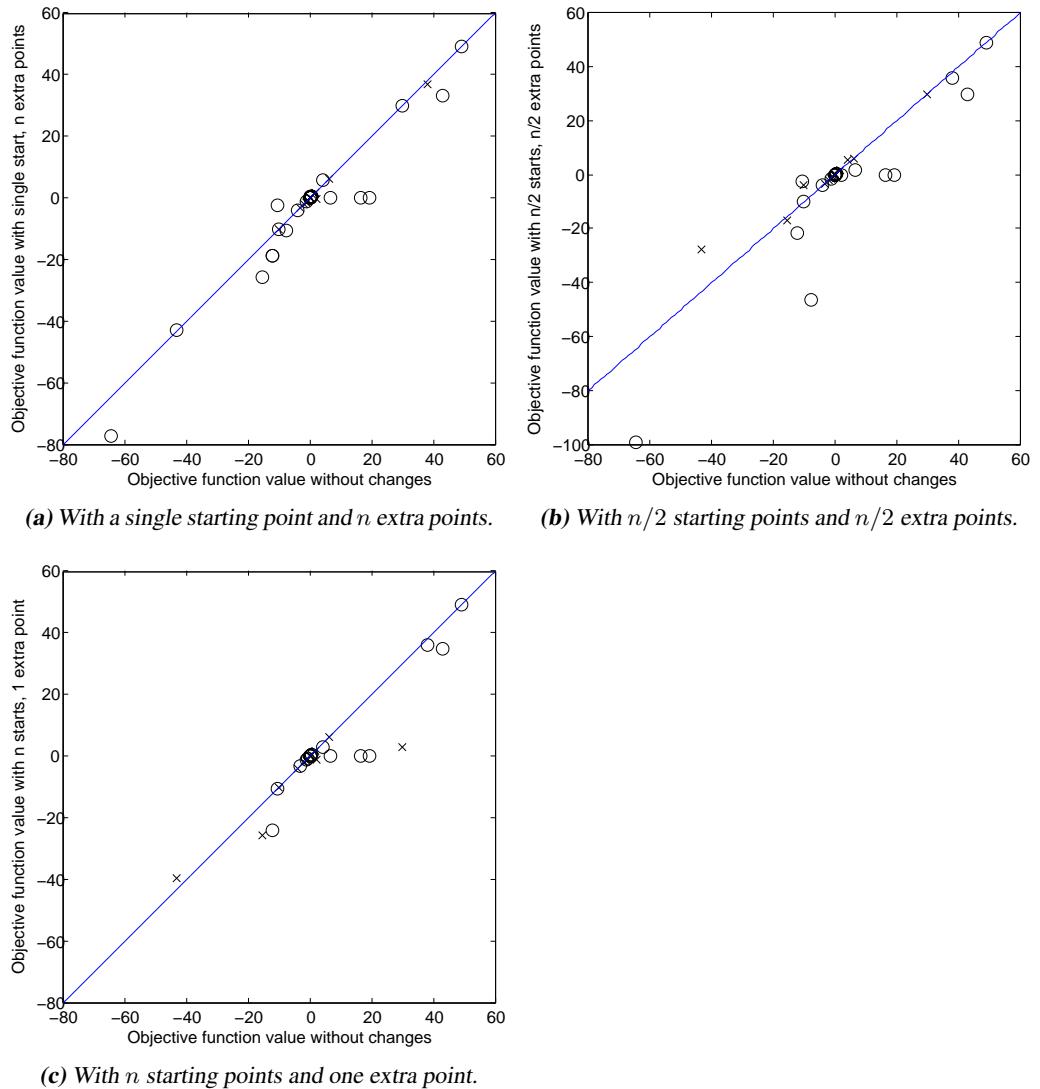
**Figure 19.20:** Data profiles when generating  $n + 1$  points in each iteration, in comparison with generation of a single point in each iteration.

For comparison, we also include the baseline of not creating any additional points at all. The data profiles are available in Figure 19.20 and we can see that the generation of several points in each iteration does not cause the algorithm to converge faster. Judging from the data profiles, it seems like one should prefer using several optimization runs instead of generating several points for each optimization run. In fact, the worst setting seems to be using a single optimization run that generates  $n$  additional points.

When looking at the data profiles, it seems clear that using a single optimization run that generates  $n$  points in each iteration is best, but it is more difficult to see any clear trends when we look at the achieved objective function values (Figure 19.21). After fur-

ther analysis, we can see that using  $n$  and adding a single point is superior. For that setting, the ratio between improved and impaired is about 3:2, while the opposite ratio applies of using one starting point and generating  $n$  additional points. When using  $n/2$  starting points and equally many additional points, the ratio is 1:1.

From this we draw that the conclusion that generating additional points in the way suggested in Section 15.5 yields little benefit. Using several starting points yields better objective function values and requires fewer function evaluations.



**Figure 19.21:** Objective function values for different ways to generate  $n + 1$  points in each iteration.

### 19.9.5 Several Starting Points

Here we present results of testing using several optimization runs, for the same problem, and compare the results to when using a single starting point. Each optimization run had a different starting point, but all other variables were equal. Here we let the four logical cores of the test computer simulate  $C = 4$ . In order to determine whether several starting points make any difference we use several different criteria.

1. The number of  $NPE$  required to achieve an objective function value that is at least as good as when using a single starting point. This answers the question 'If an objective function value of  $f$  is sufficient, what is the number of  $NPEs$  that is required to achieve this, when having access to  $C$  computers?'
2. The best objective function value achieved by any of the parallel starting points, using as many  $NPE$  as the single starting point did before it terminated. That is, if the single starting point used 100  $NPEs$  before it terminated, what is the best objective function value achieved by the optimization runs when  $NPE = 100$ ? This answers the question 'What is the best objective function value that we can achieve with a budget of  $NPE$  objective function evaluations, when using  $C$  computers?'

Data relevant to the first criterion is available in Table 19.6 and data relevant to the second criterion is available in Table 19.7.

Since the information displayed in the tables is the result of extended logging of every test case, we have elected to not include them in the Appendix.

Just like in the previous testing, we randomized all starting positions. To spread the optimization runs over a larger search space, we use  $\rho_{beg} = 5\%$  of the interval of the most constrained variable. As several optimization runs were used, the starting point of each optimization run was required to be at least 10% of the most constrained variable's interval away from each other.

When using several starting points, we performed several trials; one in which there was no information sharing and one with information sharing. In the latter trial, each candidate point  $x_k^+$  was evaluated in all available models (see Section 16.1) with  $\alpha = 4$ .

In Table 19.6, the first column displays the objective function name. Columns two and three display the achieved objective function value  $f$  and the value of  $NPE$  required, when using a single starting point. Column number four displays the  $NPE$  required to find a value at least as good as  $f$ , when using several optimization runs and no information sharing. The next column displays the change in percent in the number of evaluations, rounded to one decimal place. A decrease in the number of required function evaluations is a value less than zero and an increase is a value greater than zero.

The last two columns display the same data as the two previous columns, but with information sharing, where each candidate point is evaluated in all available models. If, for a certain test case, none of the optimization runs achieved a value that was at least as good as  $f$ , then the columns contain 'N/A'.

**Table 19.6:** The number of NPEs required to achieve a certain objective function value when using one or several starting points.

Objective function	<i>f</i>	NPE	NPE	Change	NPE	Change
Ackley	19.4364	184	184	0	166	-9.8
Beale	4.55759e-15	75	72	-4	N/A	N/A
Bohachevsky1	0	34	26	-23.5	41	+20.6
Bohachevsky2	1.29227e-11	29	29	0	29	+0
Booth	6.53614e-25	12	11	-8.3	11	-8.3
BoxBettis	1.67461e-14	89	89	0	N/A	N/A
Branin	0.397887	23	22	-4.3	25	+8.7
Branin2	6.50309	31	31	0	33	+6.5
Camel3	0.298638	33	25	-24.2	25	-24.2
Camel6	2.10425	35	27	-22.9	31	-11.4
Chichinadze	-43.3159	75	75	0	N/A	N/A
Colville	9.06405e-18	397	397	0	N/A	N/A
Corana	184197	142	142	0	153	+7.7
Eggholder	-1627.68	80	80	0	194	+142.5
Freudenstein-Roth	48.9843	64	63	-1.6	59	-7.8
Gear	2.85776e-08	139	139	0	138	-0.7
Generalized Rosenbrock	1.91252e-13	367	330	-10.1	560	+52.6
Goldstein-Price	30	27	25	-7.4	35	+29.6
Hansen	-7.61849	21	21	0	22	+4.8
Hartman 3	-3.86278	52	49	-5.8	71	+36.5
Hartman 6	-3.322	137	120	-12.4	93	-32.1
Himmelblau	3.5564e-14	20	20	0	20	+0
Holzman2	2.31249e-23	386	238	-38.3	209	-45.9
Hosaki	-1.12779	66	24	-63.6	16	-75.8
Hyperellipsoid	6	104	12	-88.5	12	-88.5
Kowalik	0.00159405	251	240	-4.4	308	+22.7
Leon	1.93787e-15	114	60	-47.4	N/A	N/A
Levy7	-64.6282	126	126	0	N/A	N/A
Matyas	2.41059e-26	41	12	-70.7	12	-70.7
Max Mod	1.25419e-09	65	65	0	N/A	N/A
McCormick	-1.91322	43	16	-62.8	16	-62.8
Michalewitz	-0.801303	60	53	-11.7	47	-21.7
Multimod	2.01106e-08	229	228	-0.4	412	+79.9
Neumaier Perm0	1.78131e-15	162	162	0	165	+1.9
Neumaier Perm	1.8328e-15	228	227	-0.4	N/A	N/A
Neumaier power sum	6.21618e-05	1003	767	-23.5	1002	-0.1
Odd square5	-1.39764	828	828	0	N/A	N/A
Odd Square10	-1.3585	1003	1002	-0.1	1002	-0.1
Plateau	43	173	155	-10.4	155	-10.4
Quartic Noise U	1.56688	249	163	-34.5	157	-36.9
Rana	-765.143	91	62	-31.9	65	-28.6

*Continued on next page*

## 19. Testing on Synthetic Test Cases

---

Objective function models)	$f$ models)	NPE	NPE	Change	NPE	Change
Rastrigin	37.9028	68	68	0	66	-2.9
Rastrigin2	0.121099	34	33	-2.9	39	+14.7
Schaffer1	0.487077	46	46	0	50	+8.7
Schaffer2	0.000307125	82	68	-17.1	83	+1.2
Schwefel1_2	4.69129e-25	133	119	-10.5	119	-10.5
Schwefel2_21	3.0254e-08	177	177	0	203	+14.7
Schwefel2_22	16.4176	144	144	0	159	+10.4
Schwefel2_26	319.836	61	61	0	77	+26.2
Shekel2	500	108	94	-13	72	-33.3
Shekel4_5	-10.1532	63	54	-14.3	56	-11.1
Shekel4_7	-10.4029	93	74	-20.4	64	-31.2
Shekel4_10	-10.5364	59	56	-5.1	63	+6.8
Shubert	-12.1543	18	18	0	20	+11.1
Shubert2	-15.5105	186	178	-4.3	148	-20.4
Shubert3	-12.1543	18	18	0	20	+11.1
Sphere	2.72574e-25	104	23	-77.9	23	-77.9
Sphere2	2.36716e-24	72	47	-34.7	47	-34.7
Step	0	125	125	0	131	+4.8
Stretched V	7.98293e-17	147	130	-11.6	143	-2.7
Sum Squares	1.04744e-24	149	129	-13.4	129	-13.4
Trecanni	4.43456e-13	38	28	-26.3	26	-31.6
Trefethen4	4.03313	35	25	-28.6	29	-17.1
Watson	0.00228767	815	555	-31.9	759	-6.9

When using several optimization runs and no information sharing, the number of  $NPE$  was decreased in around two thirds of the test cases and the median improvement was 13.4%. The remaining test cases were unchanged. When using several optimization runs and information sharing, 31 cases were improved and the median improvement was 20.4%. For nine test cases, this setting failed to find an objective function value at least as low as for a single starting point. For 22 test cases, this setting required a greater number of  $NPEs$  than the single starting point. For these cases the median increase in  $NPE$  was 11.1%.

In Table 19.7, the first two columns display the objective function name and the number  $NPEs$  that the single starting point used before termination. The third column displays the best achieved objective function value when using several starting points and no more  $NPEs$  than for a single starting point. Each  $x_k^+$  was evaluated only in the owning optimization run's model. The last column displays the best achieved objective function values for several starting points when each candidate point  $x_k^+$  was evaluated in all optimization runs' models, again when using no more than  $NPEs$  than for a single starting point. This table yields no information whether the number of starting points used less  $NPEs$  than the single starting point did, to achieve the same or a better value.

**Table 19.7:** Objective function values for testing of several starting points.

Objective function	NPE	$f$	$f$ (single model)	$f$ (multiple models)
Ackley	184	19.4364	1.50883e-08	2.66702e-08
Beale	75	4.55759e-15	2.02611e-16	1.13234e-13
Bohachevsky1	34	0	0	0
Bohachevsky2	29	1.29227e-11	1.66533e-16	1.11022e-16
Booth	12	6.53614e-25	6.45288e-28	6.45288e-28
BoxBettts	89	1.67461e-14	1.67461e-14	3.14223e-14
Branin	23	0.397887	0.397887	0.397887
Branin2	31	6.50309	7.81138e-13	5.95183e-09
Camel3	33	0.298638	0.298638	0.298638
Camel6	35	2.10425	-1.03163	-0.453195
Chichinadze	75	-43.3159	-43.3159	-42.8686
Colville	397	9.06405e-18	9.06405e-18	6.16017e-13
Corana	142	184197	925.881	3061.04
Eggholder	80	-1627.68	-1627.68	-1215.85
Freudenstein-Roth	64	48.9843	48.9843	48.9843
Gear	139	2.85776e-08	2.85776e-08	9.15375e-09
Generalized Rosenbrock	367	1.91252e-13	7.2351e-14	0.072424
Goldstein-Price	27	30	3	3
Hansen	21	-7.61849	-116.743	-174.749
Hartman 3	52	-3.86278	-3.86278	-3.86278
Hartman 6	137	-3.322	-3.322	-3.322
Himmelblau	20	3.5564e-14	3.5564e-14	2.91269e-15
Holzman2	386	2.31249e-23	2.05946e-31	1.66924e-30
Hosaki	66	-1.12779	-2.34581	-2.34581
Hyperellipsoid	104	6	6	6
Kowalik	251	0.00159405	0.00159405	0.00157641
Leon	114	1.93787e-15	3.49535e-17	0.02581
Levy7	126	-64.6282	-64.6282	-34.0514
Matyas	41	2.41059e-26	7.91543e-29	7.91543e-29
Max Mod	65	1.25419e-09	1.25419e-09	3.49353e-09
McCormick	43	-1.91322	-1.91322	-1.91322
Michalewitz	60	-0.801303	-0.801303	-0.801303
Multimod	229	2.01106e-08	2.01106e-08	2.5323e-08
Neumaier Perm0	162	1.78131e-15	2.97776e-16	3.54466e-16
Neumaier Perm	228	1.8328e-15	1.8328e-15	239.863
Neumaier power sum	1003	6.21618e-05	3.35193e-09	1.35278e-07
Odd square5	828	-1.39764	-1.39764	-1.39763
Odd Square10	1003	-1.3585	-1.3585	-1.40648
Plateau	173	43	36	36
Quartic Noise U	249	1.56688	1.56688	1.56688

Continued on next page

## 19. Testing on Synthetic Test Cases

---

Objective function	<i>NPE</i>	<i>f</i>	<i>f</i> (single) model)	<i>f</i> (multiple models)
Rana	91	-765.143	-955.56	-955.56
Rastrigin	68	37.9028	37.9028	37.9028
Rastrigin2	34	0.121099	0.121099	0.121099
Schaffer1	46	0.487077	0.396102	0.22769
Schaffer2	82	0.000307125	0.000173227	0.000282183
Schwefel1_2	133	4.69129e-25	2.15211e-26	2.15211e-26
Schwefel2_21	177	3.0254e-08	3.0254e-08	2.68483e-07
Schwefel2_22	144	16.4176	8.18755e-05	7.80496e-05
Schwefel2_26	61	319.836	319.836	319.836
Shekel2	108	500	-1.56962e+10	-1.57122e+11
Shekel4_5	63	-10.1532	-10.1532	-10.1532
Shekel4_7	93	-10.4029	-10.4029	-10.4029
Shekel4_10	59	-10.5364	-10.5364	-10.5364
Shubert	18	-12.1543	-15.7632	-13.2272
Shubert2	186	-15.5105	-25.7418	-25.7418
Shubert3	18	-12.1543	-15.7632	-13.2272
Sphere	104	2.72574e-25	3.35384e-27	3.35384e-27
Sphere2	72	2.36716e-24	3.116e-28	3.116e-28
Step	125	0	0	0
Stretched V	147	7.98293e-17	2.61688e-17	3.32816e-17
Sum Squares	149	1.04744e-24	2.08683e-26	2.08683e-26
Trecanni	38	4.43456e-13	5.82936e-18	-2.7734e-15
Trefethen4	35	4.03313	-0.630303	-0.630303
Watson	815	0.00228767	0.00228767	0.00228767

We can see that when using several starting points and evaluating  $x_k^+$  only in one model, better objective function values were achieved in half of the test cases and in the remaining cases, the achieved objective function value was the same as when a single starting point was used. When several starting points and each  $x_k^+$  was evaluated in all available models, then 33 test cases were improved, for 18 test cases the same value was achieved, but there are also 13 test cases where this setting failed to perform as well as a single starting point.

In light of these test results we think that if there are several computers available, several starting points is a very good choice. In many cases we saw considerable improvements, regardless of which of the two criterion that we used to measure improvement in. Using several models to evaluate the candidate point in is more uncertain as compared to using only the owning optimization run's model; in some cases it gave the very best values but in other cases it failed to perform as well as a single starting point.

We conclude that if there are several computers available, then we think that several starting points shall be used. The safer way to go is to use several starting points and using

only a single model to evaluate the candidate point, since we typically find better objective function values in fewer iterations when using this setting, as compared to evaluating the point in several models.

## 19.10 Testing of Strategies for Optimization Control

We have tested the different strategies for terminating and creating new optimization runs dynamically (see Section 17.4). For all testing here, we used  $n+4$  optimization runs. The initial trust region radius  $\rho_{beg}$  was 5% of the most constrained variable's interval. When elliptical trust regions were used, we used  $l_{\min} = 1$  and  $l_{\max} = 3$ . In occurring cases, we used  $\gamma = 0.25$ .

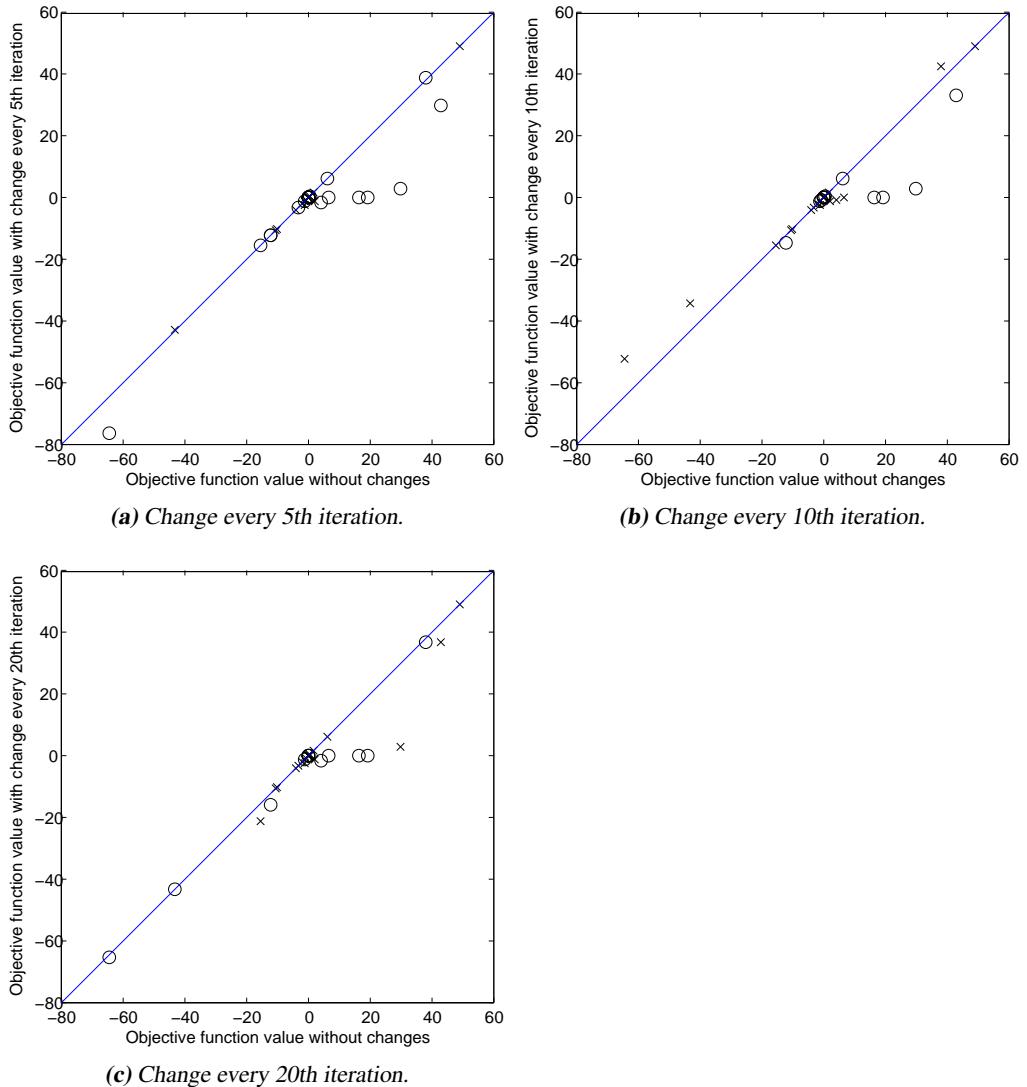
As the number of optimization runs varied in this testing, we performed as many function evaluations as possible in parallel, and the remaining points had to wait until all ongoing evaluations were finished. Then new function evaluations were started, until all function evaluations had been performed. Thus, even though it seemed like it to the algorithm, all function evaluations were not performed in parallel within each iteration, during this testing.

### 19.10.1 Keep Single Best Optimization Run

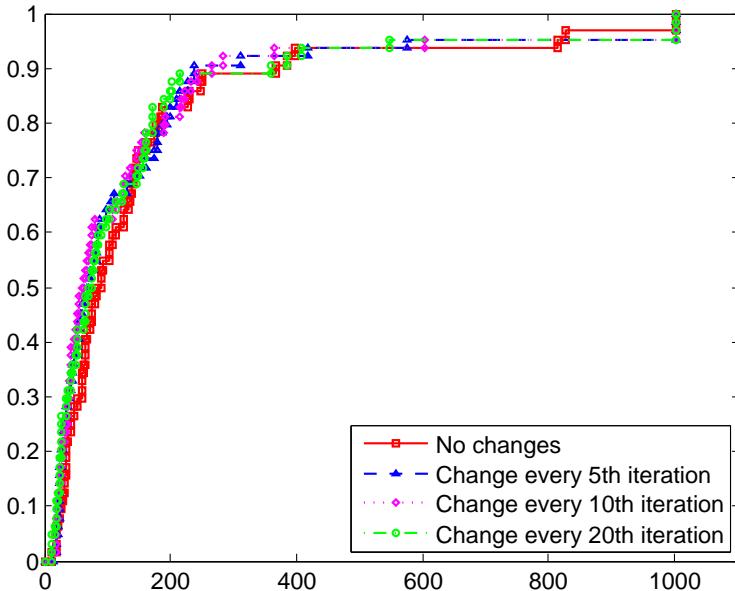
With this setting, there was initially one each of circular, an elliptical from correlation, a trust region that creates circular level curves, a switching elliptical as well as an elliptical trust region along each coordinate axis. After a certain number of iterations, all optimization runs except the one with the currently best objective function value were terminated. From the sole remaining optimization run, optimization runs with all trust region shapes that were different from the parent's, were created. We have tested with varying the number of iterations between the terminations/creations. The values that we have tested are 5, 10 and 20 iterations.

Figure 19.23 displays data profiles for the standard setting with no terminations/creations as well as for the above mentioned termination/creation intervals. Up to about 200 function evaluations, terminating and then creating new optimization runs perform better than the standard, passive, control strategy. After that, the difference is not so big, but it is not until about 800 evaluations that the standard strategy outperforms the other.

Figure 19.22 displays the achieved objective function values for the standard setting (x-axes) and the alternative control strategies (y-axes). For all strategies, we see that they quite often achieve better objective function values than the standard setting, and from Figure 19.23 we know that this is done with fewer function evaluations in many cases. This is somewhat difficult to see in Figure 19.22, since many of the test cases that were solved with fewer evaluations have objective function values that are close to zero.



**Figure 19.22:** Objective function values for terminating and creating new optimization runs with varying frequency, using the control strategy that keeps the best optimization run and then creates new optimization runs from it.



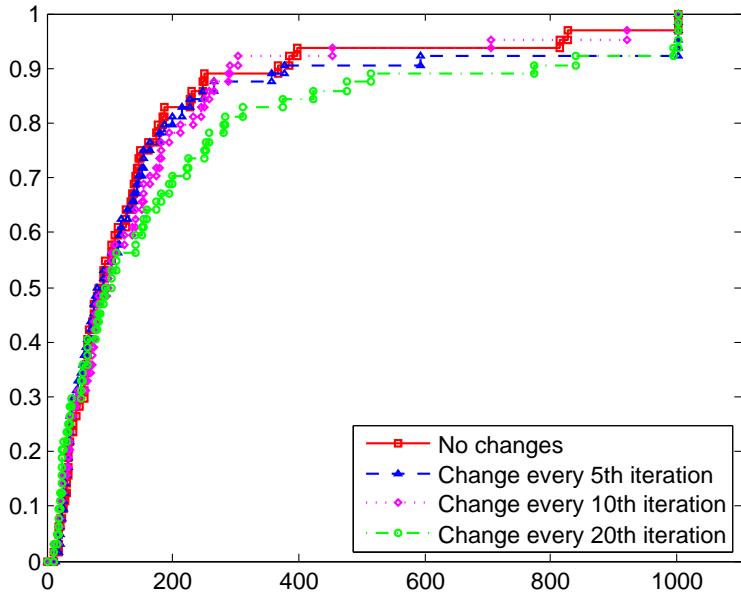
**Figure 19.23:** Data profile for the control strategy that keeps the best optimization runs and creates children from it.

### 19.10.2 Dynamic Weight Adjustment

Here we present testing of the dynamic weight adjustment, as presented in Section 17.4.1. The weight of exploration was set to 1.0 and was gradually decreased to 0.0 over the course of the allowed 1000 objective function evaluations. This means that in some test cases, exploitation was highly prioritized during the complete process. However, this should not cause any major problems since such test cases are solved using a small number of function evaluations.

When optimization runs were terminated, half of the optimization runs were terminated. The remaining half was allowed to have one child each, to bring the number of optimization runs back to the original number. The new optimization run had either a circular or an adaptive elliptical trust region where the trust region shape for the new optimization run was always different from the parent's. The fact that half of the optimization runs were terminated means that the number of concurrent optimization runs was different from the testing in Section 19.10.1.

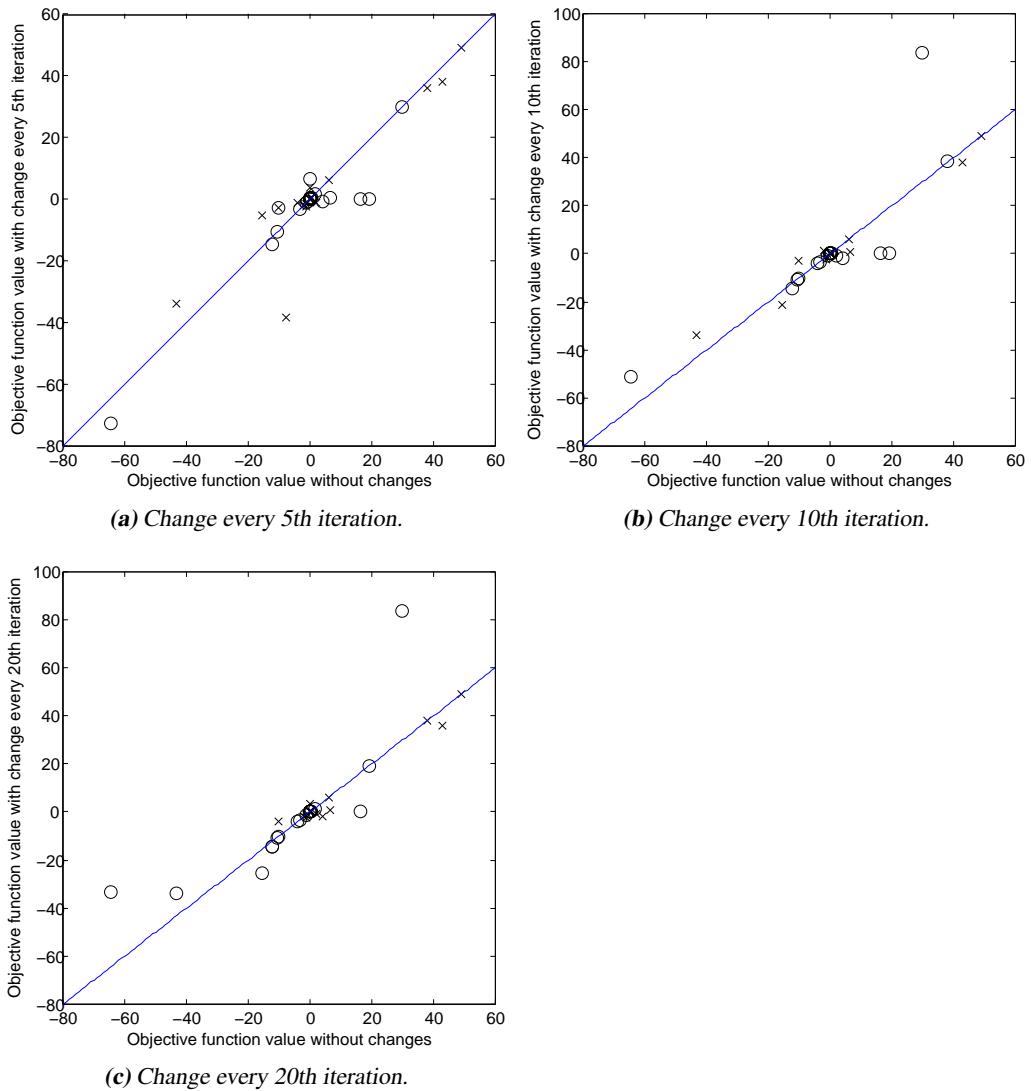
When looking at the data profile in Figure 19.24, the improvement is not that obvious. All settings perform about the same up to about 100 function evaluations. After that, terminating and creating new optimization runs every 10th, and especially every 20th iteration performs considerably worse. As the number of iterations increase, terminating and creating every 10th iteration performs better, and is slightly better than terminating and creating every 5th iteration. Terminating and creating every 10th iteration seems to



**Figure 19.24:** Data profile for the dynamic control strategy.

be the best setting together with the normal, passive control strategy.

When considering the achieved objective function values (Figure 19.25), we can see that for all the settings tested here, there were both improvements and impairments. We think that the best results are achieved when optimization runs are terminated and created every 5th iteration (Figure 19.25a), but also with that setting, there are some test cases that are impaired. However, there are also some cases that are improved in both aspects, i.e. both the objective function value is improved, and fewer objective function evaluations are required. For the other two settings (Figure 19.25b and 19.25c), there are more test cases that are impaired and fewer that are improved.



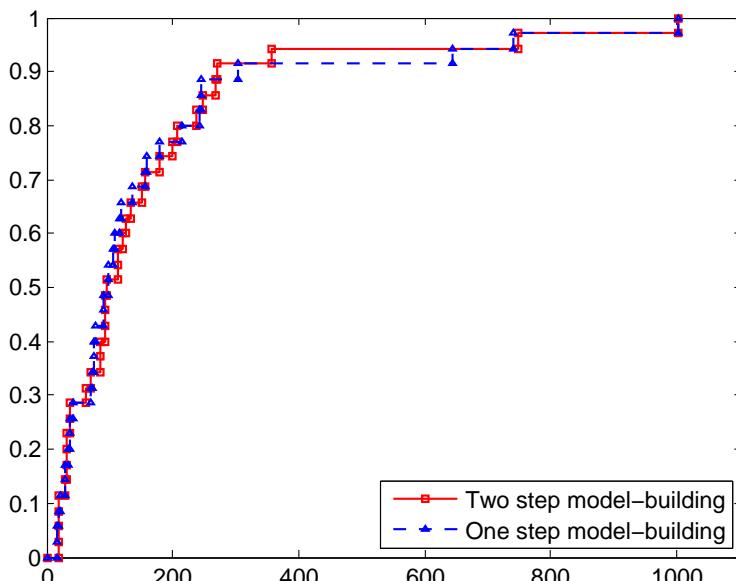
**Figure 19.25:** Objective function values for terminating and creating new optimization runs with varying frequency when using the dynamic control strategy.

## 19.11 Further Analysis and Discussion

Since quite a few of the test cases in the MVF test case suite are very multimodal and/or noisy, we perform further analysis of a subset of the test cases in this section. The test cases were chosen after the regular testing had been performed and since basically all settings achieved very similar objective function values, it should be easier to judge the effect of the different changes that we have suggested.

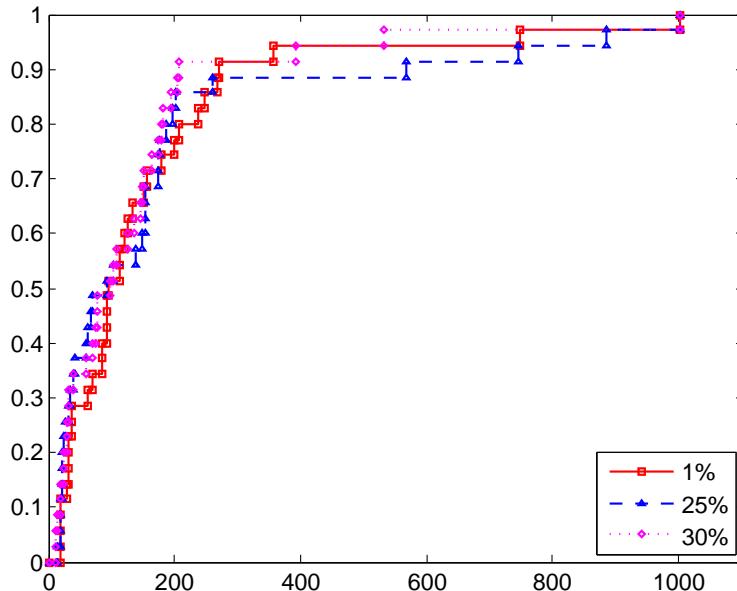
The test cases included in this analysis are the following: Bohachevsky1, Bohachevsky2, Booth, BoxBent, Branin, Branin2, Gear, Himmelblau, Holzman2, Hyperellipsoid, Kowaliak, Leon, Matyas, Max mod, McCormick, Michalewitz, Multimod, Neumaier Perm0, Neumaier Perm, Neumaier power sum, Quartic Noise U, Rastrigin2, Schaffer1, Schaffer2, Schwefel1\_2, Schwefel1\_21, Schwefel2\_22, Schwefel2\_26, Sphere, Sphere2, Step, Stretched V, Sum squares, Trecanni and Watson. We have some characteristics of most of these test cases, see Table 19.1. Recall from the table that some of these test cases are still multimodal and that some are labeled edgy.

We begin with discussing the effect of building the initial model in one step (see Sections 15.3 and Section 19.6). From the data profiles in Figure 19.26, we can see that one step model building performs well: in a few cases it performs slightly better than the two step model building and in many other cases are the results very similar. It is only when the number of NPEs exceed 300 that it is outperformed. These results make it more interesting to use model building in one step.



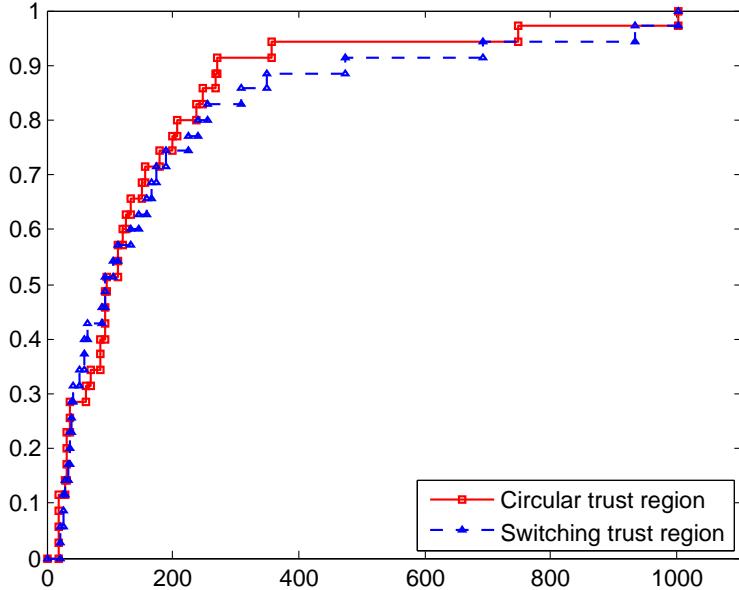
**Figure 19.26:** Data profiles for building the model in one or two steps.

Next we investigate the effect of varying the initial trust region radius  $\rho_{beg}$  (see Section 19.7). The data profiles in Figure 19.27 are quite interesting: the number of function evaluations show no monotonic dependency on the initial trust region radius. It actually makes the setting  $\rho_{beg} = 25\%$  seem worse in relation to the other, when the above subset of test cases is used. It is only the best setting for very few test case (around the values 0.4 – 0.5 on the vertical axis). The setting  $\rho_{beg} = 1\%$  is somewhat unreliable, it sometimes performs very well (around the value 0.6 on the vertical axis), whereas in other cases it is far from the best setting. While we cannot completely explain the results, we have some ideas. These ideas are as follows: a small radius makes the model an accurate depiction of the objective function in the region where it is located. This in turn yields good candidate points, which leads to larger trust region radii, thus allowing longer steps towards an optimum. For the largest radius, the points are spread further apart, and this is likely to yield some point(s) with low objective function value. Since the algorithm starts the optimization there, it has a head start as compared to the other settings. For this reason, the model does not have to be as accurate, in order to still provide good performance. It has an additional advantage in that it may also take longer steps. For the settings in between, it seems like they have less of these advantages: their models are not accurate enough to permit the longer steps and their head start is sufficient to get a good total performance.



**Figure 19.27:** Data profiles for varying the initial trust region radius  $\rho_{beg}$ .

When looking at using an elliptical trust region that changes direction every iteration, we see that there is no big change as compared to our analysis for all test cases (see



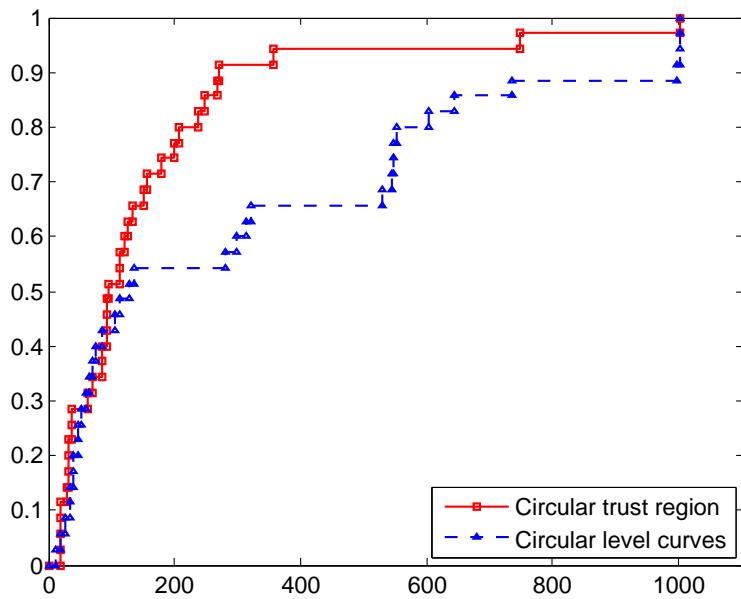
**Figure 19.28:** Data profiles for using a trust region shape versus keeping the standard circular trust region.

Section 19.8.1). There are some cases where the switching trust region performs better (0.3-0.45 on the vertical axis in Figure 19.28) but the normal circular setting performs better in most cases. The same conclusions hold for the transformation that creates circular level curves (see Section 15.4.2 and Section 19.8.3) as visible from the data profile in Figure 19.29. Up to about 0.5 on the vertical axis, the settings are very similar. After that, the standard setting performs considerably better. For this reason, we cannot make a general recommendation to use the transformation.

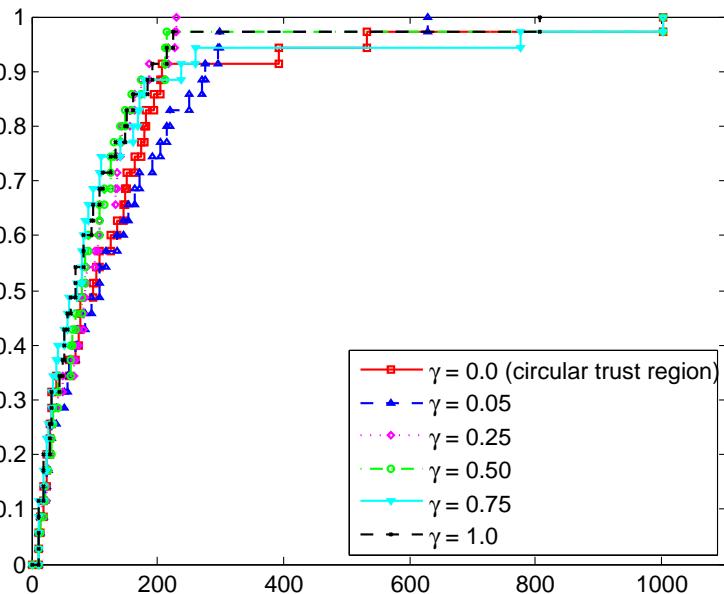
Results for the adaptive elliptical trust region are displayed in Figure 19.30. This is interesting, since most settings outperform the standard circular setting. Furthermore, using  $\gamma = 0.75$  is superior for test cases that require up to about 200 NPEs, but after that  $\gamma = 0.75$  is the worst setting, together with using a circular trust region.

If we were to give a general suggestion regarding whether to use elliptical trust regions, it would be to use them and use  $\gamma = 0.25$ . The reason for this is that it performs well most of the time, and never requires a great number of function evaluations. This means that for an all-round setting, the weight of the existing transformation matrix is 0.75, whereas the matrix created from the new step gets weight 0.25 in the convex combination. That is, we shall keep the most of the existing matrix, since it will allow longer steps in directions where we are likely to want to go next.

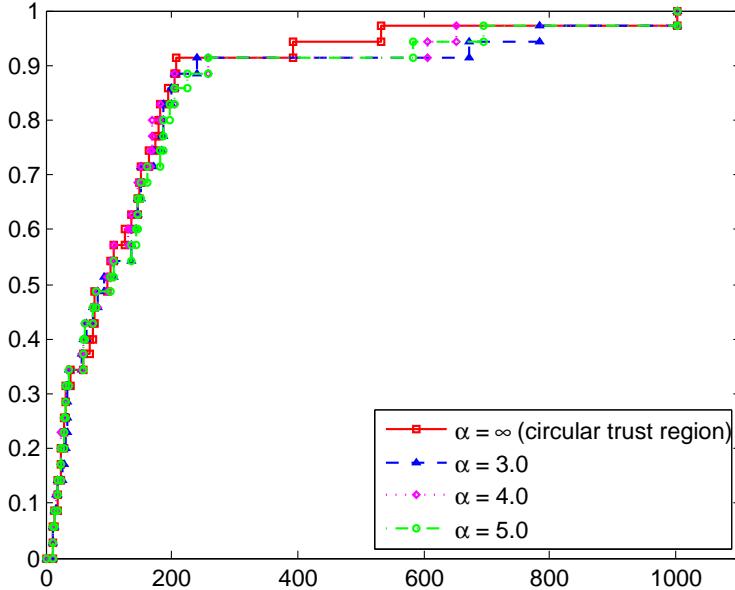
Figure 19.31 displays data profiles for using several models to predict the candidate point's value. Different values of the model weight exponent  $\alpha$  were used and we can see



**Figure 19.29:** Data profiles for using a transformation that creates circular level curves versus keeping the standard circular trust region.



**Figure 19.30:** Data profiles for the adaptive elliptical trust region with different values of the forgetting factor  $\gamma$ , versus the standard circular trust region.

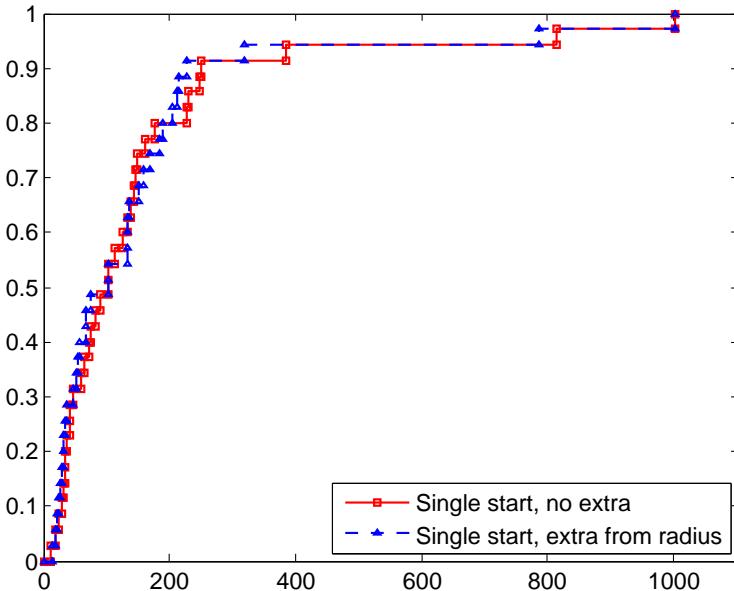


**Figure 19.31:** Data profiles for different values of the model weight exponent  $\alpha$ , compared to the standard circular trust region.

that here we must change the conclusion from Section 19.9.1. Here, it points to the fact that using a higher value of  $\alpha$  is better. Perhaps a higher value than five would yield better results. The standard setting is slightly better than  $\alpha = 5$ . Their performance is about the same up to about 400 *NPEs*, where the standard setting is slightly better until about 650 *NPEs*. After that, they are equal again, for the few test cases that remain.

These results show that valuating the candidate point  $x_k^+$  in several models is not as successful as we had hoped. The influence of the other models seem to bring little improvement. This is probably from the fact that as the distance from the points in the interpolation set increases, the value of the polynomials often increase or decrease a great deal. The model  $m$  is a weighted sum of the polynomials. We tried to counteract this with the model weight exponent that is intended to decrease the influence of models that are far away from  $x_k^+$ . It seems like the values that we used in the tests were not enough to have a positive influence. If another kind of model were used in the optimization runs, created using e.g. least-squares method, then this approach could perhaps be of greater benefit. Also, if there are many optimization runs, then the model weight weight should also be increased to prevent a situation where the optimization runs influence each another's predicted point values too much

When looking at the data profiles in Figure 19.32, we can see that generating two points by solving the trust region subproblem with two different radii seems to be slightly better than just generating a single point. Assuming that one wants to use one starting

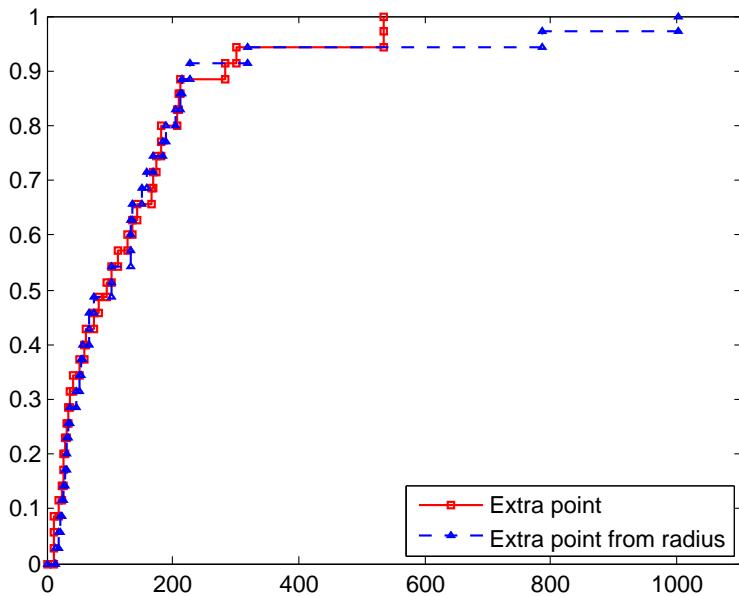


**Figure 19.32:** Data profiles for generating several points in each iteration by solving the trust region subproblem with different radii, compared to the standard circular trust region.

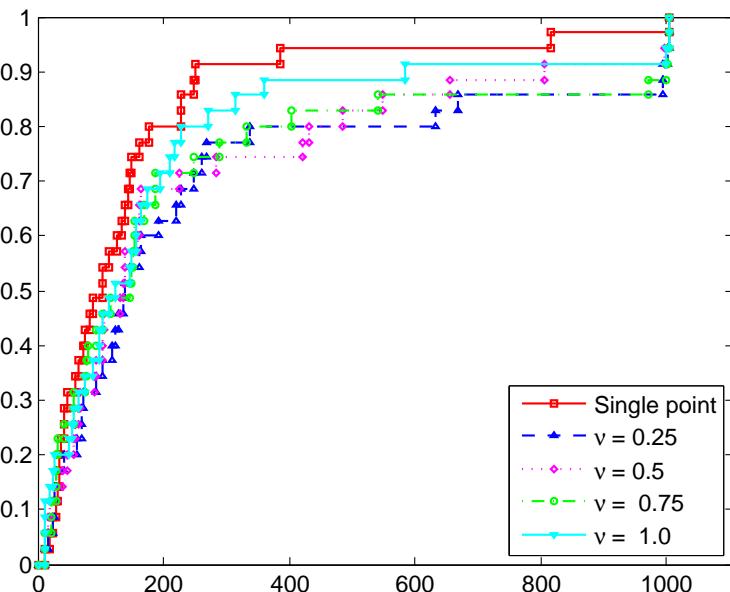
point and generate two points, is it better to solve the trust region subproblem several times or just adding an additional point using the algorithm in Figure 15.4 in Section 15.5? To determine this we ran such tests and the data profiles are available in Figure 19.33. We can see that the two alternatives perform about equally well until about 550 NPEs. The big difference is visible there; the extra point setting never requires more NPEs than this, while solving the trust region subproblem does. On the basis of this, we recommend adding an extra point instead of solving the subproblem several times with different radii.

Whether one should generate additional points by placing them in a plus sign pattern centered in  $x_k^+$  is our next topic of discussion. Data profiles are presented in Figure 19.34 and we can see that the original setting that generates a single point in each iteration is still the best, but that the difference is less here as compared to Figure 19.18 on page 265. In both figures,  $\nu = 1.0$  performs the best, but the difference between that setting and the other settings is less here. Thus, if several points shall be generated in this way, then they shall be placed far away from  $x_k^+$ , since this yields better performance.

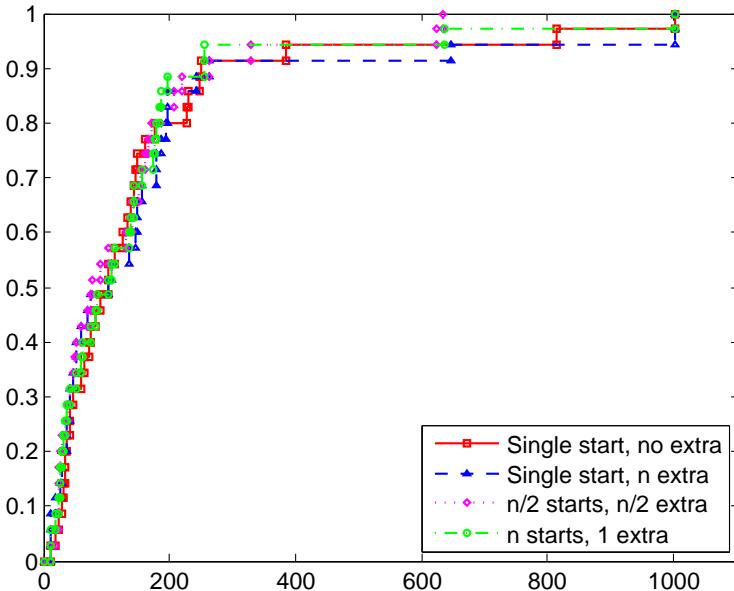
The other way to generate several point in each iteration, i.e. generating a fixed number of points, is our next topic. Data profiles for different ways to generate  $n+1$  points are available in Figure 19.35. Just like in Section 19.9.4, using  $n$  optimization runs and then adding a single additional point is the best option. However, we note that the difference is smaller here, and using  $n/2$  optimization runs and the same number of additional points



**Figure 19.33:** Data profiles for either solving the trust region subproblem with different radii, or adding one extra point.



**Figure 19.34:** Data profiles for generating several points in each iteration by placing  $2n$  extra points in a plus sign pattern centered in  $x_k^+$ .



**Figure 19.35:** Data profiles for various ways of generating several points.

is actually superior in a few test cases here, which it was not in Figure 19.20. The original setting, of not generating any additional points, is also superior in a couple of cases. We think that using  $n$  optimization runs is the best option here and recommend that this is used.

About the generation of several points for each optimization run, we believe that there are two important things to consider. The first is how to place the new points. In our implementation, we placed them along the coordinate axes for simplicity. It might be better to place the points further away from  $x_k$ , beyond  $x_k^+$ , but in approximately the same direction.

The other thing is to decide which point(s) to include in the model in the next iteration. We only included one, which was the point with the lowest objective function value. This might lead to the model becoming too convex, and thus unable to accurately model the real objective function. This may also bias the algorithm towards local optimization, which is undesirable.

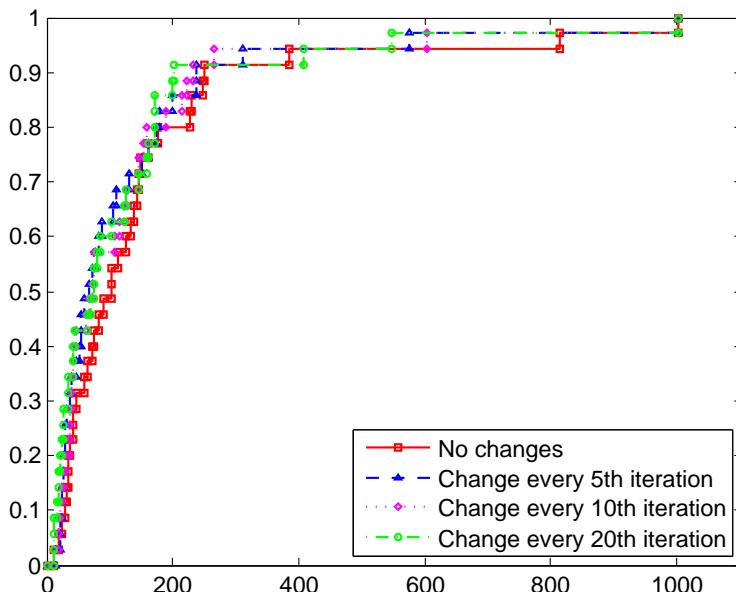
We believe that more advanced criteria for determining which point(s) to include in the model will need to be developed in the future, and that such criteria may include not only the objective function value, but also distance to the other points in the interpolation set, the ratio  $r$  from Equation 12.23 and more.

Also, we updated the trust region radius only if the included point was the original candidate point  $x_k^+$ , in all other cases, the radius was kept unchanged. Our motivation for this was to avoid seemingly random changes in the radius due to less reliable models

farther from the trust region center. However, as the effect of this is that it might cause the algorithm to require a greater number of function evaluations, since the radius does not decrease in the same pace as before. Some other options are to always adjust the radius as well to use some surrogate value created from several of the points' values and use that value as input for adjustment of the trust region radius.

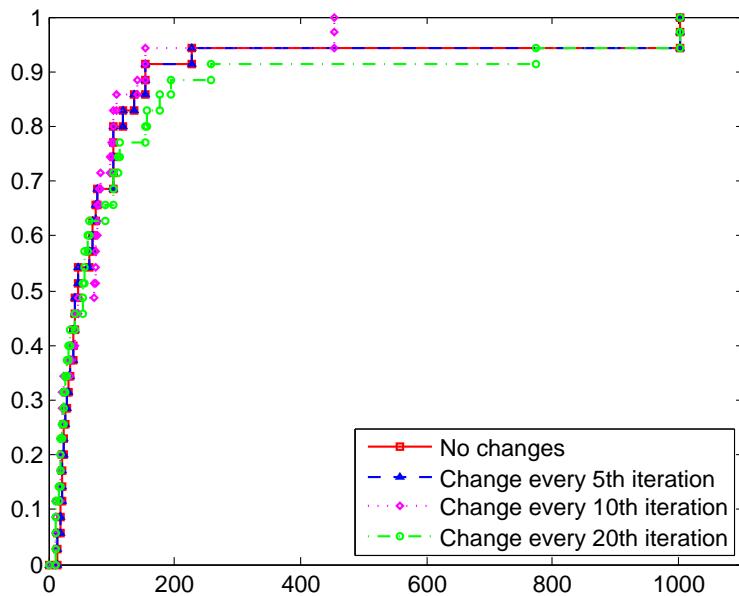
Regarding the control strategies, data profiles for keeping the best optimization run (see Section 17.4.2 and Section 19.10.1) are available in Figure 19.36 and data profiles for the dynamic control strategy (see Section 17.4.2 and Section 19.10.2) are available in Figure 19.37. Regarding the former, we think that performing the change (i.e. terminating and creating new optimization runs) every 20th iteration is the best option. This strategy outperforms the original strategy of no changes.

For the dynamic control strategy, performing the change every 10th iteration is the better choice. It is outperformed in just a few test cases and is the best choice in the vast majority of cases. The other conclusion here is that changing every 20th iteration is the seemingly worst choice. It is about the same as the other options until about 150 *NPEs*, after which it is the worst choice.

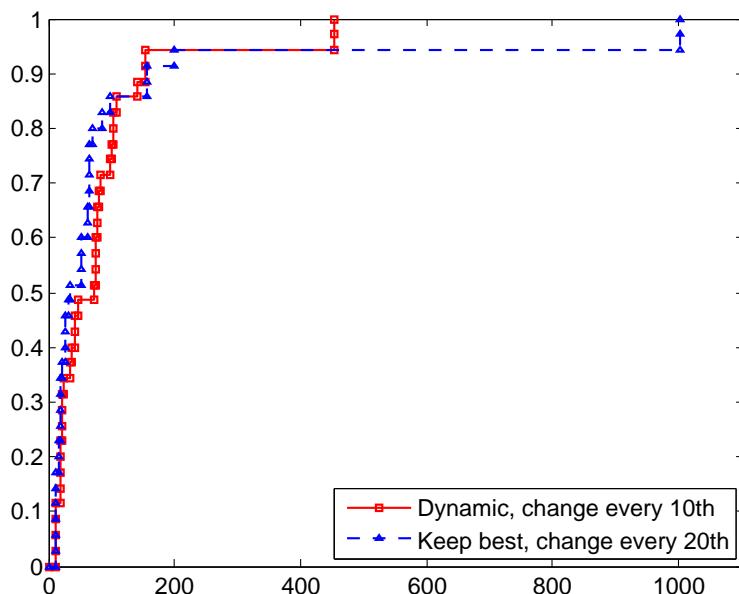


**Figure 19.36:** Data profiles for the control strategy of keeping the best optimization run and terminating the other.

When it is settled that both control strategies can outperform the standard strategy, the question is which of the two is superior. Looking at the data profile in Figure 19.38, we can see that keeping the best optimization run and creating new optimization runs every 20th iteration is slightly better until about 150 *NPEs*, after which the two are very similar



**Figure 19.37:** Data profiles for the dynamic control strategy.



**Figure 19.38:** Data profiles for the best control strategies of each kind.

until about 450 *NPEs*, since the dynamic control strategy never requires more *NPEs* than this. Due to the more consistent performance, we recommend the dynamic strategy.

## 19.12 Testing with Limited Execution Time

We conclude the testing on the MVF test case suite by displaying results from testing with a budget of 100 *NPEs*. If we assume that each function evaluation takes 2 hours, this would take a complete week and 8 hours, if running calculations nonstop around the clock!

The goal of the testing is to determine which setting that utilizes the limited time in the best way. As a baseline, we used a single optimization run with a circular trust region. A setting with four starting points with a circular trust regions, and another setting with elliptical trust regions with  $\gamma = 0.75$ . The objective function values for the different settings are available in Figure 19.8

**Table 19.8:** Objective function values for 100 NPEs.

Objective function	$n$	$f$	$f$	$f$
		One start (circular)	Four starts (circular)	Four starts (elliptical)
Ackley	4	0.000285314	9.97586e-05	0.000233799
Beale	2	0.570409	1.72731e-15	6.75095e-18
Bohachevsky1	2	7.99361e-15	0	0
Bohachevsky2	2	0	0	0
Booth	2	3.484e-26	1.27795e-28	1.15174e-28
BoxBettis	3	1.56498e-14	1.42741e-15	1.79947e-14
Branin	2	0.397887	0.397887	0.397887
Branin2	2	7.25683e-15	7.25683e-15	1.78604e-15
Camel3	2	0.298638	5.91777e-15	5.05714e-14
Camel6	2	2.10425	-1.03163	-1.03163
Chichinadze	2	-42.8686	-43.3159	-43.3159
Colville	4	3.68478	1.8568	3.85165
Corana	4	16155.7	111.372	5.48663
Eggholder	4	-1627.68	-1627.68	-1627.68
Freudenstein-Roth	2	48.9843	48.9843	48.9843
Gear	4	1.02647e-08	1.02647e-08	2.40735e-08
Generalized Rosenbrock	4	25618.3	2.58663	9.48735
Goldstein-Price	2	3	3	3
Hansen	2	-47.5604	-145.478	-176.542
Hartman 3	3	-3.86278	-3.86278	-3.86278
Hartman 6	6	-3.322	-3.322	-3.322
Himmelblau	2	5.02543e-15	5.02543e-15	7.52654e-15
Holzman2	4	1.36154e-06	1.55662e-13	1.76237e-05

*Continued on next page*

Objective function	$n$	$f$	$f$	$f$
		One start (circular)	Four starts (circular)	Four starts (elliptical)
Hosaki	2	-1.12779	-2.34581	-2.34581
Hyperellipsoid	4	6	6	6
Kowalik	4	0.00464795	0.00132973	0.00254685
Leon	2	1.38316e-14	1.38316e-14	3.44361e-17
Levy7	4	-44.334	-117.076	-106.876
Matyas	2	1.005e-27	2.22557e-30	2.35968e-30
Max Mod	2	4.58623e-09	2.34802e-09	3.17715e-09
McCormick	2	-1.91322	-1.91322	-1.91322
Michalewitz	2	-0.801303	-0.801303	-0.801303
Multimod	4	0.000335308	0.000125044	0.337487
Neumaier Perm0	4	8.29745e-13	1.13225e-14	1.29563e-12
Neumaier Perm	4	44.0364	5.18194e-14	8.70195
Neumaier power sum	4	0.00566086	2.1988e-06	0.207383
Odd square5	5	-1.39764	-1.39764	-1.3973
Odd Square10	10	-1.35838	-1.35838	-1.3918
Plateau	5	31	30	30
Quartic Noise U	4	1.56688	1.56688	1.56688
Rana	4	-933.357	-933.357	-1205.69
Rastrigin	4	36	36	36
Rastrigin2	2	0.121099	0.121099	0.121099
Schaffer1	2	0.495946	0.312103	0.272743
Schaffer2	2	0.000163926	9.27235e-05	0.000260182
Schwefel1_2	4	6.9617e-29	3.24038e-30	9.49098e-31
Schwefel2_21	4	0.000524645	4.80383e-05	0.000347042
Schwefel2_22	4	0.00323112	0.000190367	0.00456153
Schwefel2_26	4	754.12	754.12	773.889
Shekel2	2	500	-1.87114e+10	-2.38012e+09
Shekel4_5	4	-10.1532	-10.1532	-10.1532
Shekel4_7	4	-3.7243	-10.4029	-10.4029
Shekel4_10	4	-2.42173	-10.5364	-10.5364
Shubert	2	-12.1543	-18.9894	-14.6909
Shubert2	4	-21.3887	-21.3887	-21.3887
Shubert3	2	-12.1543	-18.9894	-14.6909
Sphere	4	3.80073e-27	1.34106e-29	1.14385e-29
Sphere2	4	9.20995e-28	1.36079e-29	1.5657e-29
Step	4	0	0	0
Stretched V	4	1.04797e-16	1.04797e-16	4.40478e-15
Sum Squares	4	5.63247e-27	1.28387e-27	1.22471e-27
Trecanni	2	1.46809e-16	2.98472e-17	1.112e-17
Trefethen4	2	0.454589	-1.0497	-0.562926
Watson	6	3.0179	2.37418	0.112711

All settings achieved the same objective function values in 16 cases. The settings with four starting points with circular trust regions achieved the best objective function values in 22 cases and four starting points with elliptical trust regions achieved the best results in 14 cases. In the remaining cases, two settings achieved the same lowest value.

A few test cases deserve particular mention. For the Corana test case, both settings with four starting points achieved considerably better values than the single starting point, and the adaptive elliptical was superior. The same applies for the Generalized Rosenbrock and Shekel2 test cases. For the Watson test case, the elliptic trust region performs the best, and the standard circular setting requires 597 *NPEs* before it is finished. Allowing an additional 497 *NPEs* decreases the objective function value to only slightly better than the best optimization run with elliptical trust region achieves in 100 *NPEs*.

The Beale test case also shows why we consider parallelization and multiple starting points. During normal testing, the setting with a circular trust region required 976 *NPEs* to achieve an objective function value of 0.45343, whereas both settings with several starting points achieved values lower than 1.73E-15 after 100 *NPEs*.

Our conclusion from the testing with a limited execution time is that using several optimization runs is very beneficial also in this case. In about one quarter of the test cases were the values the same, and in all other cases were the objective function values improved by using several optimization runs, and in some cases were the improvements considerable.

## 19.13 New Test Cases

In this section we present some new test cases and the results of applying the algorithms on them. The test cases are described in Table 19.9. The functions are designed by taking a fairly simple basic structure and then adding higher-order disturbances as we believe that many realistic problems will look like this.

Name	dim	Intervall	Function
Snake1	2	$[-10, 10]$	$0.5\sqrt{2r} + 2d^2 + 0.2r$ , where $r = x_1^2 + x_2^2$ and $d = 3\sin(x_1) - x_2$
Snake2	2	$[-40, 40]$	$2d^2 + 0.1r$ , where $r = x_1^2 + x_2^2$ and $d = 3\sin(x_1) - x_2$
Superbowl	2	$[-10, 10]$	$0.5\sqrt{2r} + 2\sin(3r) + 0.2r$ , where $r = (x_1^2 + x_2^2 + 0.001)$
Quadbasin	2	$[-5, 5]$	$x_1^4 - 16x_1^2 + 5x_1 + x_2^4 - 16x_2^2 + 5x_2 + 10\sin(10x_1x_2)$

**Table 19.9:** The new test functions.

Both the Snake test cases are multimodal with optimum in  $(0, 0)^T$  with  $f^* = 0$ . Superbowl is noisy and multimodal with a local optimum at  $(0, 0)^T$  with  $f = 0$  and

several global optima, e.g.  $(0.422, 1.168)^T$  and  $(-1.219, 0.238)^T$ . The global optima are spread symmetrically around the origin, and all have  $f^* = -0.806$ .

The Quadbasin test case is based on a scaled version of the Styblinski-Tang test case, with added noise. It is multimodal with the global optimum located in  $(-2.939, -2.939)^T$  with  $f^* = -166.576$  and there are several local optima:  $(-2.860, 2.692)^T$  and  $(2.692, -2.860)^T$ , both with  $f = -138.239$ , and  $(2.717, 2.717)^T$  with  $f = -110.067$ . The intervals are not lower and upper bounds: they are used to limit the area of interest for optimization.

Three-dimensional plots and level curves for Snake1 are given in Figure 19.39, for Snake2 in Figure 19.40, for Superbowl in Figure 19.41 and for Quadbasin in Figure 19.42.

We used the following settings during testing on these problems:  $\rho_{beg}$  was set to 5% of the interval,  $\rho_{end} = 10E - 7$  and we allowed 50NPEs after building the initial model.

The motivation of these tests is an additional comparison of promising settings for certain relevant functions. The settings were chosen because we believe that they would perform well when using a very strict budget of NPEs. For this reason, we only compare function values. The three settings that we have tested are: a single optimization run that works as the baseline case, five optimization runs that generate one point each, and a single optimization run that, in addition to  $x_k^+$ , generates additional four points in plus-sign pattern centered in  $x_k^+$ .

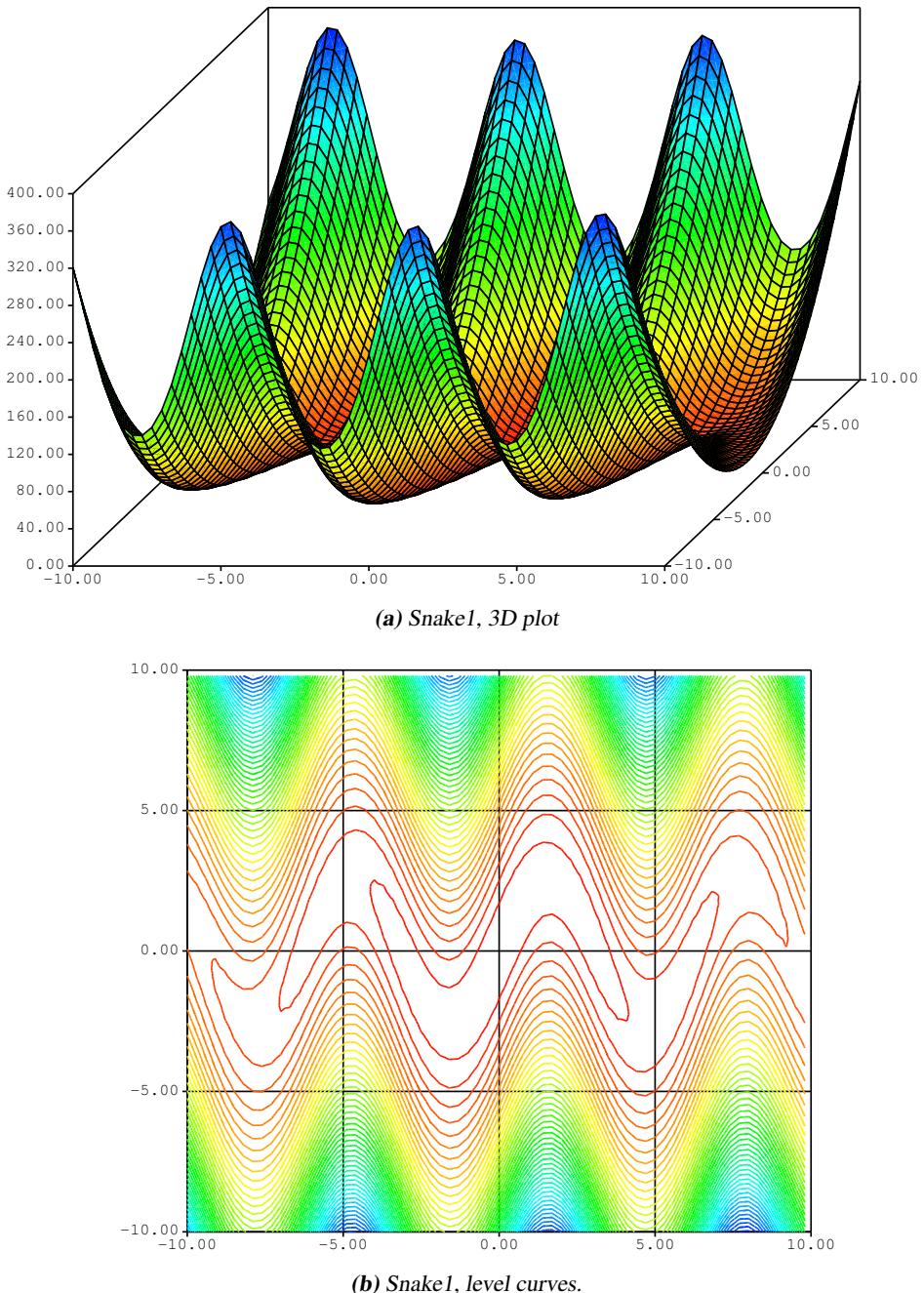
Function	$f$ (One start)	$f$ (Five starts)	$f$ (One start, +)
Snake1	3.82324	1.12614E-5	0.00145146
Snake2	67.1368	4.98105E-13	14.9276
Superbowl	4.09283	0.00769107	-0.806106
Quadbasin	-166.576	-166.576	-166.263

**Table 19.10:** Results from new test functions.

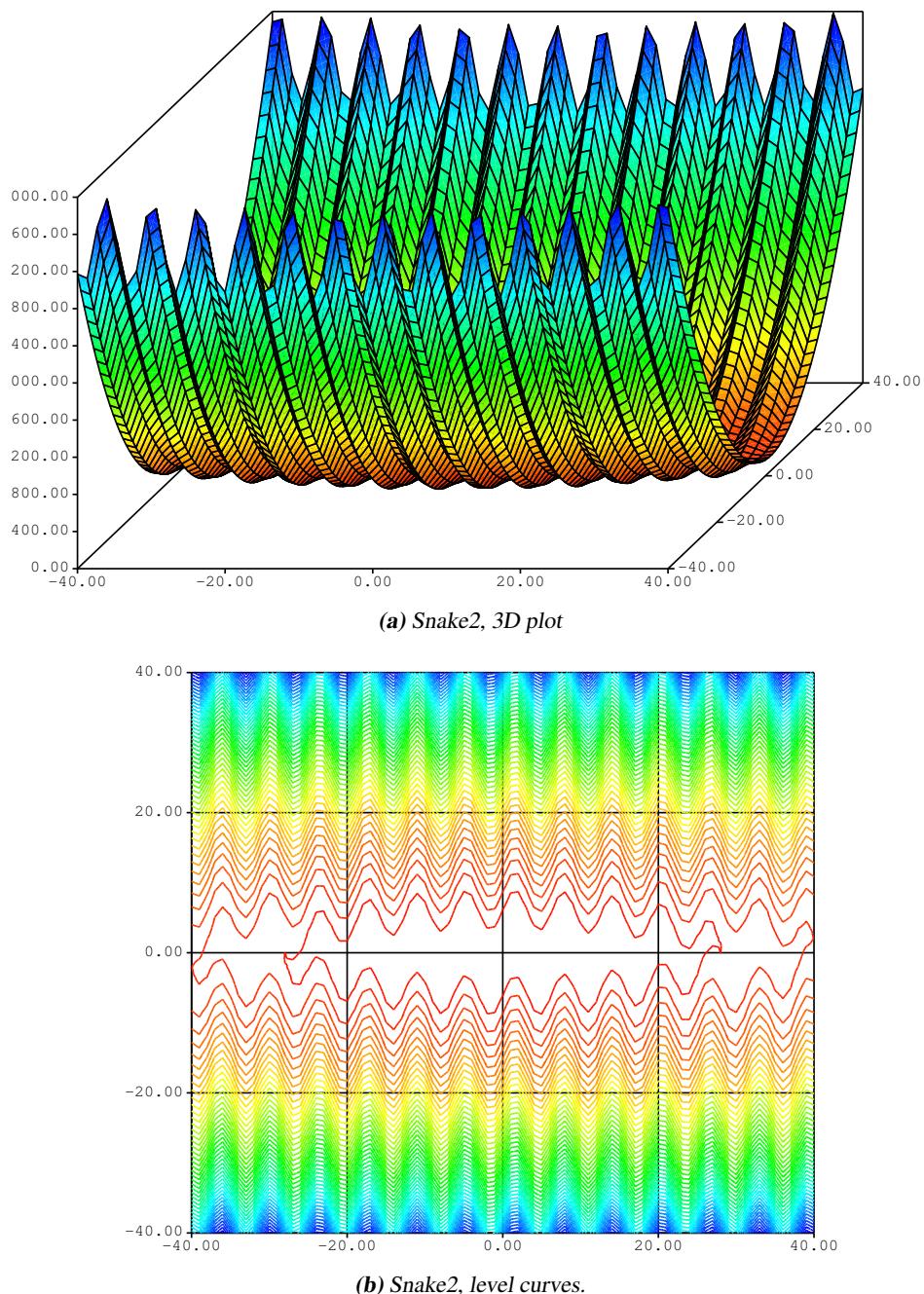
Table 19.10 displays the achieved objective function values for the different settings. For several optimization runs, we show the best objective function value.

To begin with, we can see that it is clearly beneficial to use either several starting points or to generate several points in each iteration, even if the budget of NPEs is low. It is only the Quadbasin test case that is not improved as compared to the base case.

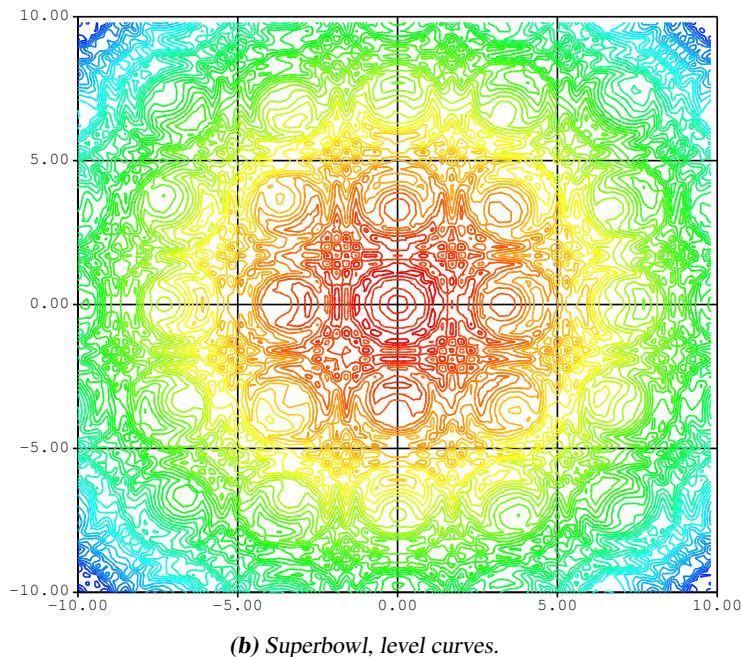
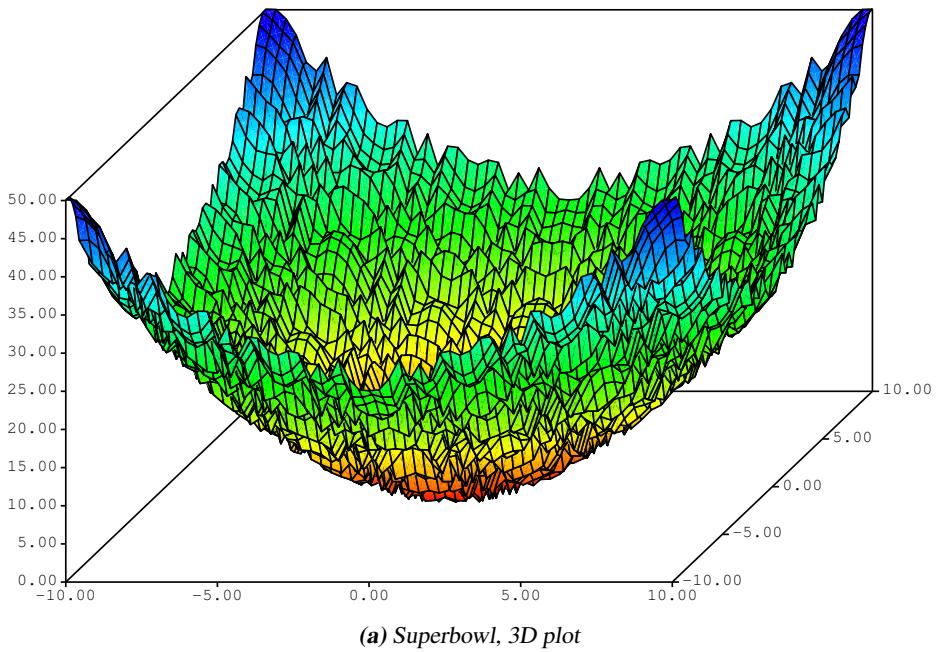
Between using several optimization runs or a single one that generates several points, we can see that there is no big difference for Snake1 and Quadbasin. For Snake2, using several starting points is superior while the opposite applies for Superbowl. The interesting case here is the Superbowl test case, where adding extra points in a plus-sign is better. This is probably due to the fact that the function is approximately monotonous and that  $m$  models the function fairly accurately.



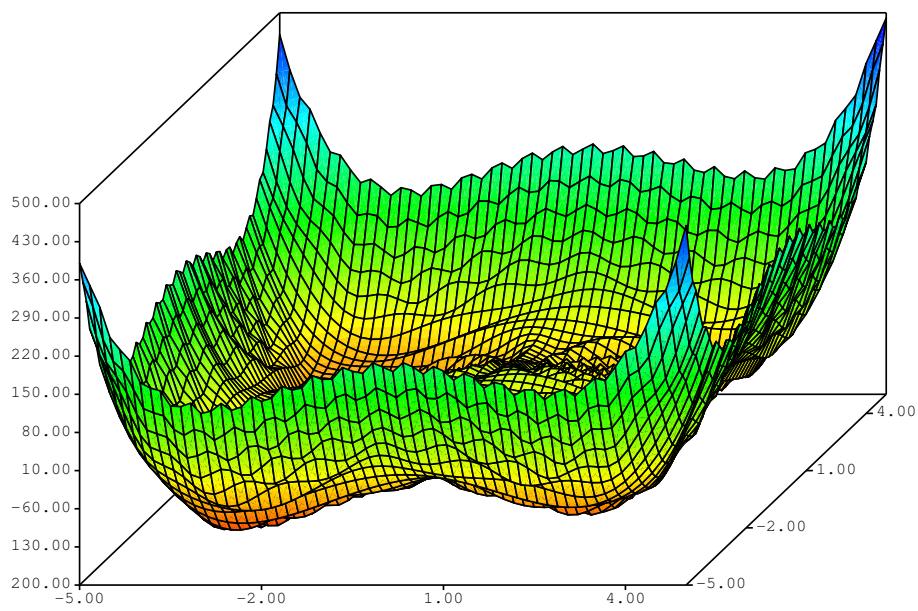
**Figure 19.39:** Pictures of the Snake1 test function.



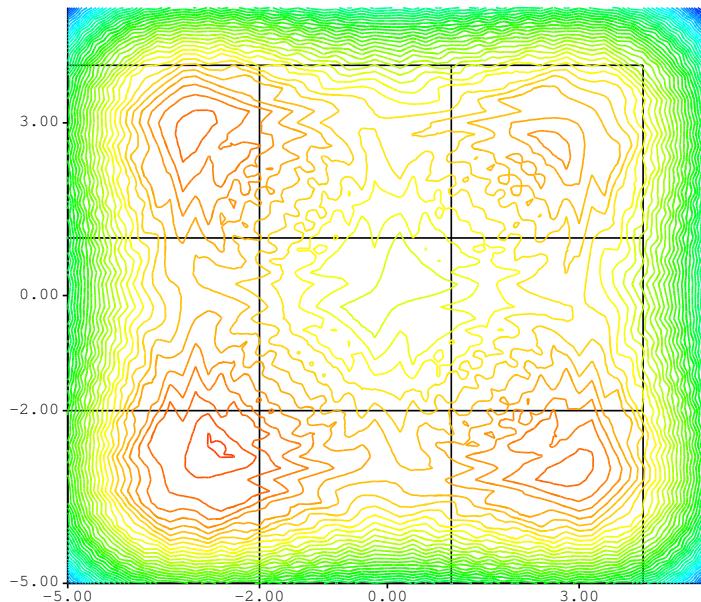
**Figure 19.40:** Pictures of the Snake2 test function.



**Figure 19.41:** Pictures of the Superbowl test function.



(a) Quadbasin, 3D plot



(b) Quadbasin, level curves.

**Figure 19.42:** Pictures of the Quadbasin test function.

## 19.14 Summary

We have presented results of testing the different algorithms on a suite of low-dimensional test cases. Since the test cases vary greatly regarding e.g. modality and noisiness, we find that it is in many cases difficult to draw definite conclusions. We believe that this is mainly due to the test cases. Some of them are extremely difficult since they are to some extent made for global optimization. In global optimization, the objective function is known and the number of function evaluations and the time required until the optimum has been found is of lesser importance. For this reason, we have performed a closer analysis of a subset of the test cases in order to be able to draw more accurate conclusions.

But even looking at all the test case results, there are some things that the results indicate should always be done. The first and easiest thing, that has no relevant negative side effects, is to use a cache of stored points. The next thing is to never build the initial model completely in sequence, instead evaluate the points in at most two batches. In our further analysis, we see that the difference between the one and two step model building is not big. Therefore we recommend that two step model building is used if the evaluation time is very long, in order to decrease the time before the optimization can start.

Regarding the initial trust region radius, we achieved the very best results with  $\rho_{beg} = 30\%$  of the most constrained variables interval. However, using  $\rho_{beg} = 1\%$  also gave very good results, while the settings in between gave worse results. That the largest value performs best here is no surprise, since it spreads in the points over a large region. On the other hand the smallest tested value performs very well, while the second biggest value performs poorly and the very biggest value performs the best. From this we do not want to draw strong conclusions and we believe that it is quite difficult to give an advice regarding the initial trust region radius that is likely to work well in all situations.

In our further analysis of the adaptive elliptical trust regions, we saw that keeping the weight of the rank one matrix at  $\gamma = 0.25$ , led to the best results. In effect, this causes the transformation matrix to have quite a bit of inertia since it does not change so much in each iteration. Since this setting outperforms the standard circular trust region, we can recommend this for general use. As far as we know, the results presented here are the first results of applying non-circular trust regions to this kind of optimization.

When performing testing with a stricter limit on the number of *NPEs*, we saw that using several optimization runs was very beneficial. We achieved improvements in objective function values in about three quarters of the test cases and in some cases were the improvements considerable.

We performed similar testing, but with an even stricter limit on the number of *NPEs*, on our new test cases Snake1, Snake2, Superbowl and Quadbasin. The results were very similar: using several optimization runs led to improvement in the objective function values in three of the four cases.

We can without hesitation recommend that several starting points, in our case implemented through optimization runs, are used. Several optimization runs not only help in finding an optima with good values, but also helps with finding several optima, that would

otherwise not be found. Remember that we typically display the best value found, even if the optimization runs have converged to different optima. The only disadvantage that we can come to think of with this approach is that the time for building the initial model increases, if the number of available computers is insufficient. To some extent this can be ameliorated by using one step model building, however. Once the initial model has been built, we never lose anything by using several starting points. For this reason we recommend that several starting points are always used.

Using several starting points proved superior to the other tested ways to generate several points for evaluation in each iteration, i.e. several points in a plus sign as well as distributing a certain number of additional points where it is believed that they would prove the most useful. The case where we can recommend using the additional points is when the amount of unused computers is not sufficient for adding one more starting point. In that case we recommend using the distribution of the additional points as mentioned above.

It is perhaps not so surprising that many starting points outperform using a small number of starting points and adding additional points to them. The former can be spread out over a larger fraction of the search space, and should then, from a purely statistical standpoint, have a greater chance of finding good values. Therefore, if there are  $C$  computers available, we think that the best way to use them is with the same number of starting points, provided that we can afford the waiting time until the initial models have been built.

When analyzing the results of testing of the control strategies, we saw that the results when comparing dynamically decreasing the exploration weight every 10th iteration during execution, and keeping the single best optimization run and killing the rest in every 20th iteration were quite close. The advantage of the first strategy is that it never required a great number of function evaluations, i.e. the worst case was not that bad. For test cases that required less than about 100 *NPEs*, the latter strategy was the best. This is probably due to the fact that these test cases are fairly easy, and that the best optimization run is probably making good progress towards an optimum. If we were to give a general recommendation, it would be to use the first strategy, since it was able to avoid the worst cases and deliver a consistent performance.



# 20

---

## Testing On Industrial Test Cases

This chapter presents results from using the optimization algorithm on industrial applications.

### 20.1 Test Results from Frontway

Frontway has used the optimization software described in this thesis to minimize the cost of a new production line, with parameters such as the tank capacity and pump capacities, for a certain production program. The constraints were that the production program must be fulfilled and that the plant must run all the time, i.e. no stops or breaks in production due to overflows/empty tanks or due to insufficient pump capacity.

Frontway employees say that they not only saved a lot of time, they also got a better solution, when using the algorithms developed in this thesis. Furthermore, they believe that they are not able to perform the same work as the optimization algorithm does, for several reasons. To begin with, interpreting the results of each simulation and then deciding which variable(s) to change takes time, and then the new input must be set. Since the simulation takes some time, the user works with something else at the same time, and is then forced to switch tasks when the simulation is finished, in order to set values so that the next iteration in the simulation can be performed. Using the optimization software, the user can set initial values and then check the results at his convenience, since what was previously done by hand is now automatic.

Due to the results that they achieved and the time they saved, they see a great potential in the optimization algorithms and want to continue to use them in the future.

Since the specific facility that was optimized belongs to a customer, Frontway is unwilling to disclose further information about the results due to competition and the customer's identity.

## 20.2 Test Results from SKF

We have applied the optimization algorithms in this thesis to several problems at SKF and here we discuss some of the results.

### 20.2.1 Testing on a Two Variable Problem

To create a test problem with two variables, we used an existing bearing and changed two variables. We incorporated two performance criteria that were somewhat contradictory into a custom objective function that was to be maximized. The value of the objective function was normalized to the interval  $(0, 1]$ . In this test case we used one optimization run.

The figures in Section 18.7.5, beginning on page 234, are taken from this testing. In Figure 18.5, each function evaluation is indicated by a dot, where the dot corresponding to the first experiment is indicated by a small black dot and the last evaluation is marked by a larger black dot. All other experiments are marked by green dots. The lines between the dots are an attempt to trace the order in which the points were evaluated. Note that the lines are just an indication since some function evaluations were performed in parallel.

We can clearly see the first six points that are used to create the original model, since they are placed along the coordinate axes in the upper left corner. After that, the points are the results of trust region iterations and model iterations, where the iterations of the first kind are performed in order to find better objective function values and the latter ones are performed with the intention of finding points that can improve the model.

We can see that the algorithm basically moves from the upper left corner of the search space to the lower right corner, and that the step lengths decrease as the optimization progresses. Eventually the trust region radius becomes so small that the algorithm terminates.

Figure 18.6 displays some of the function evaluations and we can see that the best objective function value that the algorithm finds is approximately 0.5437. The remaining function evaluations are failed attempts to find points with better objective function value, as well as model iterations. This is also visible in Figure 18.7, where we can see that after the initial large improvement in objective function value, the improvement decreases and eventually ceases. Towards the end, many iterations yield approximately the same objective function value. A model iteration is performed late in the process, which is indicated by the considerably worse objective function value, approximately 0.52. Eventually the algorithm terminates since the trust region radius becomes small enough for termination.

Due to the contradictory performance criteria in the objective function, we knew that we were not going to achieve a very high objective function value. An experienced BEAST user that was involved in setting up the problem said that the found solution was reasonable given the objective function, and that it was not obvious what to change to achieve higher objective function values than what the algorithm achieved.

## 20.2.2 Testing on a Four Variable Problem

One of the problems that we have solved is optimization of an existing bearing that is often used in wind mills. The problem had four variables and the objective function was to be maximized. Like in the previous case we used a custom objective function whose values were normalized to  $(0, 1]$ . In this case we were very ambitious and set both the minimum and maximum required values high. For this reason, many points got an objective function value of zero and the other got low objective function values. This leads to a difficult optimization problem since the objective function becomes quite flat and the algorithm gets little information about promising directions.

Figure 20.1 displays quadratic approximations of the objective function, which are taken from Beauty (see Section 18.7.5 for more information about the visualization of results). We have exchanged the real variable names for Var 1, Var 2, Var 3 and Var 4. The current product has the values  $(0, 0, 0, 0)^T$ . The best point found by the optimization algorithm is marked by a red circle in the plots.

In this particular test case we used four starting points spread throughout the search space and performed a great number of function evaluations in order to provide enough information to create the plots. In addition to the objective function, we also compared the current product to the found points with regard to two other criteria. These criteria were not included in the objective function and the reason for including them is to see that the found points are not extremely poor in some other important aspect. In this case the optimization algorithm found several points that not only achieved a higher objective function value than the original product, but also gave better values of the two other criteria.

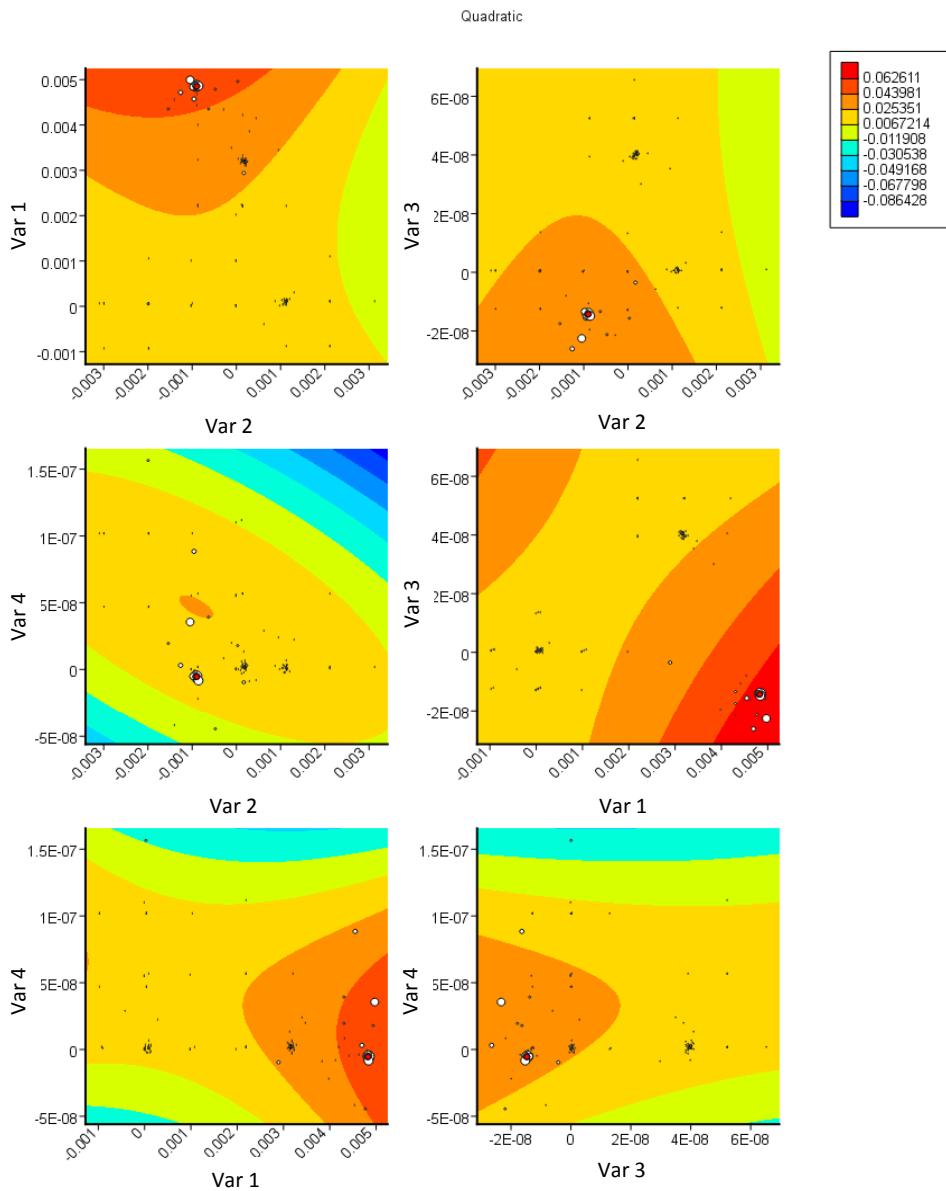
As is visible in the plots, both the best point as well as several other points found by the optimization algorithm yield considerably better objective function value than the point currently used in the product. The current product has a value of 0.0, the best point has an objective function value of about 0.159 and around half of the points in the optimization have an objective function value higher than zero. In the plot with Var 1 and Var 3, it actually looks like the current product lies in a local minima.

In this case, the optimization algorithm found several solutions that were better than the current product. What is perhaps even more interesting is the fact that not only were the objective function values better, but the points were also better with respect to the two other criteria.

## 20.3 Summary

In this chapter we have presented some results from industrial test cases. The algorithm has been applied in as diverse areas as optimization of tank volumes and pump capacities in a paper mill, and optimization of bearings. In all cases have the algorithm performed as expected in the sense that it finds an extremum of the objective function.

Due to competition and company secrecy, the companies that we have cooperated with



**Figure 20.1:** Quadratic approximations of the objective function as created by the Beauty software. The real variable names have been exchanged.

are only willing to provide a few details about the problems and the results, but both say that they are happy with the results and want to continue to use the optimization software in the future.



# 21

---

## Future Work

In our specific implementation, one area where we think that it is significant room for improvement is the constraint handling. As described in Section 10.3 and 18.7.2, constraints are handled, outside the algorithms, by penalty functions that yield a constant value regardless of the amount of violation and the number of violated constraints. In the future, this should be exchanged for a better way of handling constraints, e.g. an augmented Lagrangian algorithm [3, 52]. We prefer augmented Lagrangian methods to SQP methods since the former do not require a linearization of any non-linear constraints.

Another area that deserves more work is the handling of integer variables. However, this area is even more difficult than the constraint handling. To the best of our knowledge, there is only a single algorithm published that handles integer variables in the context of derivative-free optimization within a trust region, i.e. the MISQP algorithm [20]. A problem here is that if the trust region radius is less than one, then illegal candidate points will be generated in at least every other iteration, since the step length is insufficient to move directly between valid points.

A third area that might benefit from more work is having the option of building and updating the model in the style of the NEWUOA algorithm (see Section 13.3). We elected not to include this as  $n$  is small in the problems that we are interested in, and the benefit would be quite limited. However, if it is desirable to solve problems involving a greater number of variables in the future, then this could be a worthwhile modification.

Regarding future research directions, we believe that two connected research areas deserve more attention. The first area is whether one should use few starting points that each generates a greater number of points in each iteration, or a greater number of starting points that each generate a smaller number of points in each iteration. This is an optimization problem in itself. The answer is by no means obvious since it depends on the particular problem, the number of available computers, whether synchronous or asyn-

chronous parallelization is used, etc. We believe that choosing the correct consideration between the number of starting points and the number of points generated in each iteration can make a large difference in the result and we conjecture that this problem will attract considerable attention in the future.

The other research area is control of optimization algorithms on a more abstract level. In the discussions regarding this (see Section 17.4) and global optimization (see Section 15.6), we mentioned that it is difficult both to know when to terminate optimization runs as well as when and where to create new ones. This is of course related to the fact that we know very little about the objective function. Here we would like to draw the reader’s attention to the area of *auto tuning* or *self tuning* in computer science [80, 49, 18]. Very briefly, auto tuning is concerned with finding the best algorithm (or algorithm settings) for solving a particular problem instance, using some information about the problem data as well as about the hardware that the algorithm is running on.

We believe that this could be beneficial in the area of derivative free optimization as well. To perform auto tuning, we would like to store information gathered about the objective function during execution, such as the achieved objective function values, where in the variable space they are achieved, all estimates of Lipschitz values as well as where they are achieved, the ratio between predicted and achieved improvement in objective function value and more. However, we would also like to store meta information such as the execution time of evaluations and whether this varies significantly between different points in the variable space. If this is the case, then it might be less attractive to generate many points for each optimization run, since the optimization run must wait until all points have finished evaluating. Instead, using more optimization runs that each generates few points might be more attractive, especially if this is combined with asynchronous parallelization.

These are just a few examples of information that we think could be useful to gather during the execution of the algorithm and provide some guidance for higher level optimization control. We believe that auto tuning has been attempted in optimization in a more or less structured manner earlier, and that it will be increasingly important in the future as better algorithms for solving currently troublesome areas, e.g. constrained optimization, are developed.

As we hope has been apparent in this chapter, we believe that as the “easier” areas of derivative free optimization are solved in the future, the focus will gradually shift towards areas with a higher level of abstraction. We believe that one of the most important areas is the consideration between using many starting points that generate one or a few points each, or fewer starting points that generate many points each. Another example is self tuning of the algorithm, based on information gained during execution. The information gathered could be used both to guide the above consideration as well as for higher level control of the algorithm. In our opinion, these areas are fairly closely related and both are challenging research areas in themselves that require and deserve considerable attention in the future.

## Conclusions

In this thesis, we mainly focused on different ways to use parallelization in order to improve performance of model building algorithms for time-consuming objective functions.

We have analyzed how such algorithms work in order to find what is possible to parallelize. Starting from the building the initial model, to having several starting points that each generate one or more candidate points in each iteration, to evaluating the candidate points in several models to hopefully be able to better predict to objective function value in the points. When a candidate point is sent for evaluation, we first control whether the point is in the cache. If the candidate point is equal to any of the cached points, then the value can be returned immediately. For the points that must be evaluated, we evaluate as many as possible in parallel, to use the available computational resources efficiently. To move in the direction towards global optimization and to take the first steps towards a higher level control of optimization algorithms, we have developed different ways to determine which optimization runs to terminate, and then children are created from the remaining, since they are believed to be more promising. We want to stress that the extensions and modifications that we have developed are not limited to the specific algorithm that we used as a starting point. They can be used most of the existing model building algorithms for derivative-free optimization.

Testing has been performed using a test case suite that includes both uni- and multimodal, smooth as well as noisy problems. The big differences between the test cases makes it difficult to draw clear conclusions from the results. For this reason we further analyzed a subset of the test cases. From this we can see that some of the parallel extensions performed very well whereas some others did not fare as well.

Our algorithms have been implemented for use in industrial settings, and there is both a standalone implementation and a version that has been integrated into a simulator for dynamic multibody simulation. Both versions have been used to solve industrial

problems. Due to competition and company secrecy, we are unable to disclose much information about the results, other than the fact that the algorithms work as expected also in industrial settings and that the users are happy with the results, since the optimization algorithms find high quality solutions with little input from the user. Both companies that have been our industrial partners during the project want to continue to use the software developed in this thesis in the future.

Optimization in industrial settings is considerably different from academic applications: algorithms alone will rarely provide the complete solution. Nor will just supplying more computational power. New optimization algorithms that take advantage of the hardware's capabilities are necessary. Indeed, powerful optimization software, if possible tuned to the problem at hand, running on state of the art hardware will provide the best chances of finding a solution in reasonable time. However, the softer side of the problems shall not be ignored. The most powerful optimization algorithms will not be helpful if it is not used. The algorithm must be robust, the implementation documented, interface understandable, it must be easy for the user to set up the problem and the optimization algorithm etc. If these requirements are not fulfilled, then the algorithm might not be used since the user is not comfortable using it. Making sure that these requirements are fulfilled are time-consuming tasks that are typically ignored in academia, but they have great impact on the users' perception of the methods as well as their ability to use them.

If we were to give recommendations for industrial users, the following would be a list of things to absolutely do: use a cache of stored points, parallelize the model building as much as possible while still being sure that you do not get a model that is worse than in sequential model building, use as many starting points as possible and also to investigate whether it is beneficial to generate several points in each iteration, from each starting point. When the evaluation time of the objective function is so long that it would be difficult to use optimization at all, then we would also recommend to build the initial model in one step, to decrease the time until the optimization can begin.

Also, the adaptive elliptical transformation with a fairly low weight on the change performed well in our testing. Therefore we can recommend that not only circular trust regions are used. Similarly, the active control strategies led to improvements over the standard strategy, so we can also recommend those for use on practical problems.

In our opinion, the quest for parallelization of optimization algorithms is far from over, we have just scratched the surface here and there is plenty more to do. We have scrutinized the algorithms from initial model building, to the generation of points in each iteration, and finally to parallel evaluation of the points. Nonetheless, we think that there is much more to do and hope that future researchers will be inspired by our work and hopefully get some ideas for their own research.

# A

---

## Terms

Here we briefly describe some of the terms that are used throughout the thesis.

**Table A.1:** The major terms used in the thesis.

Term	Description
$n$	Number of dimensions of the problem, $n \geq 2$ .
$f$	Computationally expensive function that is to be minimized, $f \in R^n$ .
$k$	Current iteration.
$x_k$	Iteration point in iteration $k$ , $x_k \in R^n$ .
$h_k$	Step (direction and length) from $x_k$ to next point in iteration $k$ , $h_k \in R^n$ .
$m$	Model function, typically a polynomial, that is used to approximate $f$ .
$r$	Ratio between the achieved improvement and the expected improvement in the model and in the objective function value.
$d$	Degree of interpolating polynomial.
$Y$	Set of points used to create $m$ .
$p_1$	Number of interpolation points in $Y$ .
$g$	Approximation of $f$ 's gradient, $g \in R^n$ .
$H$	Approximation of $f$ 's Hessian. Symmetric matrix in $R^{n \times n}$ .
$x_k^+$	Point that in iteration $k$ is a candidate for inclusion in $Y$ , $x_k^+ \in R^n$ .
$x_r$	A point that is removed from $Y$ when $x_k^+$ is added, $x_r \in R^n$ .
$x_m$	Trust region midpoint, $x_m \in R^n$ .
$\Delta$	Trust region radius, $\Delta \in R^+$ .
$e(x)$	The expected objective function value of some point $x$ , $e(x) \in R$ .
$O$	Number of optimization runs that exist simultaneously.
$C$	Number of computers that are available for evaluation of the objective function.

*Continued on next page*

## A. Terms

---

Term	Description
$NPE$	Number of parallel evaluations, i.e. number of times one or more points are sent for evaluation.
$NE$	Number of function evaluations.
$T$	Transformation matrix, $T \in R^{n \times n}$ .
$l_{\min}$	The distance on the ellipse that is closest to the ellipse's midpoint, as measured in the number of $\Delta s$ . Typically, $l_{\min} = 1$ applies and $l_{\min} \in R^+$ .
$l_{\max}$	The distance on the ellipse that is farthest away from the ellipse's midpoint, as measured in the number of $\Delta s$ . Typically, $l_{\max} > 1$ and $l_{\max} > l_{\min}$ applies and $l_{\max} \in R^+$ .
$\gamma$	The fraction of the old transformation matrix that is “forgotten” when the new transformation matrix is calculated, $\gamma \in [0, 1]$ .
$G_k^i$	Number of points generated by an optimization run $i$ in iteration $k$ .
$\nu$	The fraction of the step length that the additional points are moved from the candidate point, $\nu \in R^+$ .

---

## Bibliography

- [1] Ernesto P. Adorio. MVF-multivariate test functions library in c for unconstrained global optimization., 2005. <http://adorio-research.org/wordpress/?p=13451>. Accessed April 23, 2012.
- [2] Ma. Belén Arouxét, Nélida Echebest, and Elvio A. Pilotta. Active-set strategy in Powell's method for optimization without derivatives. *Computational and Applied Mathematics*, 30(1):171–196, 2011.
- [3] Mokhtar S. Bazaraa, Hanif D. Sherali, and C. M. Shetty. *Nonlinear Programming: Theory and Algorithms*. John Wiley & sons, 2006.
- [4] R.K. Beatson, J.B. Cherrie, and C.T. Mouat. Fast fitting of radial basis functions: Methods based on preconditioned GMRES iteration. *Advances in Computational Mathematics*, 11(2):253–270, 1999.
- [5] Frank Vanden Berghen and Hugues Bersini. CONDOR, a new parallel, constrained extension of Powell's UOBYQA algorithm: Experimental results and comparison with the DFO algorithm. *Journal of Computational and Applied Mathematics*, 181(1):157 – 175, 2005.
- [6] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.
- [7] Boost. <http://www.boost.org/>. Accessed November 27, 2012.
- [8] M.J. Box. A new method of constrained optimization and a comparison with other methods. *Computer Journal*, pages 42–52, 1965.

## Bibliography

---

- [9] Martin D. Buhmann. *Radial Basis Functions: Theory and Implementations*. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press, 2003.
- [10] Richard H. Byrd, Jorge Nocedal, and Richard A. Waltz. Knitro: An integrated package for nonlinear optimization. <http://www.ziena.com/papers/integratedpackage.pdf>. Accessed March 7, 2013.
- [11] C++11. [http://www.iso.org/iso/iso\\_catalogue/catalogue\\_ics/catalogue\\_detail\\_ics.htm?ics1=35&ics2=60&ics3=&csnumber=43289](http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?ics1=35&ics2=60&ics3=&csnumber=43289). Accessed January 15, 2013.
- [12] Andrew R. Conn, Nicholas I.M. Gould, and Luis N. Vicente. *Trust-region Methods*. MPS-SIAM Series on Optimization. Society for Industrial and Applied Mathematics, 2000.
- [13] Andrew R. Conn, Katya Scheinberg, and Philippe L. Toint. DFO. <https://projects.coin-or.org/Dfo>. Accessed October 22, 2012.
- [14] Andrew R. Conn, Katya Scheinberg, and Philippe L. Toint. *LANCELOT: a Fortran package for large-scale nonlinear optimization (Release A)*. Number 17 in Springer Series in Computational Mathematics. Springer Verlag, Heidelberg, Berlin, New York, 1992.
- [15] Andrew R. Conn, Katya Scheinberg, and Philippe L. Toint. Recent progress in unconstrained nonlinear optimization without derivatives. *Mathematical Programming*, 79(1):397–414, 1997.
- [16] Andrew R. Conn, Katya Scheinberg, and Philippe L. Toint. A derivative free optimization algorithm in practice. In *Proceedings of the 7th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization*, September 1998.
- [17] Andrew R. Conn, Katya Scheinberg, and Luis N. Vicente. *Introduction to Derivative-free Optimization*. MPS-SIAM Series on Optimization. Society for Industrial and Applied Mathematics, 2008.
- [18] A. Davidson, Yao Zhang, and J.D. Owens. An auto-tuned method for solving large tridiagonal systems on the GPU. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 956–965, 2011.
- [19] R. D. Dony. Karhunen-Loëve transform. In Kamisetty Ramam Rao and Pat Yip, editors, *The Transform and Data Compression Handbook*. CRC Press, 2000.
- [20] Oliver Exler and Klaus Schittkowski. A trust region SQP algorithm for mixed-integer nonlinear programming. *Optimization Letters*, 1(3):269–280, 2007.

- 
- [21] Giovanni Fasano, José Luis Morales, and Jorge Nocedal. On the geometry phase in model-based algorithms for derivative-free optimization. *Optimization Methods and Software*, 24(1):145–154, 2009.
  - [22] Ronald Fisher. *The Design of Experiments, 9th Edition*. Macmillan, 1971.
  - [23] Richard Fletcher. *Practical Methods of Optimization*. John Wiley & sons, 1987.
  - [24] Dag Fritzson, Lars-Erik Stacke, and Jens Anders. Dynamic simulation - building knowledge in product development. *Evolution*, 1, 2014. ISSN:1104-8166. AB SKF, S-41550 Göteborg, Sweden.
  - [25] Frontway. [www.frontway.se](http://www.frontway.se). Accessed January 17, 2014.
  - [26] Keinosuke Fukunaga. *Introduction to Statistical Pattern Recognition, Second Edition*. Academic Press, 1990.
  - [27] Philip E. Gill, Walter Murray, and Michael Saunders. <http://www.sbsi-sol-optimize.com/>. Accessed October 22, 2012.
  - [28] Philip E. Gill, Walter Murray, and Margaret H. Wright. *Practical Optimization*. Emerald Group Publishing Limited, 1982.
  - [29] Fred Glover. Tabu search – Part 1. *ORSA Journal on Computing*, 1(2):190–206, 1989.
  - [30] Fred Glover. Tabu search – Part 2. *ORSA Journal on Computing*, 2(1):4–32, 1990.
  - [31] GoogleTest. <http://code.google.com/p/googletest/>. Accessed November 27, 2012.
  - [32] Hans-Martin Gutmann. A radial basis function method for global optimization. *Journal of Global Optimization*, 19(3):201–227, 2001.
  - [33] Lixing Han and Guanghui Liu. On the convergence of the UOBYQA algorithm. *Journal of Applied Mathematics and Computing*, 16(1–2):125–142, 2004.
  - [34] Robert Hooke and T. A. Jeeves. “Direct search” solution of numerical and statistical problems. *Journal of the ACM*, 8(2):212–229, April 1961.
  - [35] Reiner Horst and Panos M. Pardalos (editors). *Handbook of Global Optimization*, volume 2 of *Nonconvex optimization and its applications*. Kluwer Academic Publishers, 1995.
  - [36] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, 1999.
  - [37] Steven G. Johnson. The NLOpt nonlinear-optimization package. <http://ab-initio.mit.edu/nlopt>. Accessed April 30, 2011.

## Bibliography

---

- [38] Donald R. Jones, C.D. Perttunen, and E. Stuckman. Lipschitzian optimization without the Lipschitz constant. *Journal of Optimization Theory and Applications*, 79(1):157–181, 1993.
- [39] E.J. Kansa and Y.C. Hon. Circumventing the ill-conditioning problem with multi-quadratic radial basis functions: Applications to elliptic partial differential equations. *Computers & Mathematics with Applications*, 39(7-8):123–137, 2000.
- [40] Franz Kappel and Alexei V. Kuntsevich. An implementation of Shor’s r-algorithm. *Computational Optimization and Applications*, 15(2):193–205, February 2000.
- [41] Napsu Karmitsa, Adil Bagirov, and Marko M. Mäkelä. Comparing different non-smooth minimization methods and software. *Optimization Methods and Software*, 2010. DOI: 10.1080/10556788.2010.526116. Accessed May 10, 2011.
- [42] S. Kirkpatrick, C. D Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.
- [43] LAPACK. <http://www.netlib.org/lapack/>. Accessed November 27, 2012.
- [44] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction, Second Edition*. Microsoft Press, 2004.
- [45] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1998.
- [46] Douglas C. Montgomery. *Design and Analysis of Experiments, Eighth Edition*. John Wiley & sons, 2012.
- [47] Jorge J. Moré and D. C. Sorensen. Computing a trust region step. *SIAM Journal on Scientific Computing*, 4(3):553–572, 1983.
- [48] Jorge J. Moré and Stefan M. Wild. Benchmarking derivative-free optimization algorithms. *SIAM Journal on Optimization*, 20(1):172–191, 2009.
- [49] Rajib Nath, Stanimire Tomov, and Jack Dongarra. An improved MAGMA GEMM for Fermi graphics processing units. *International Journal of High Performance Computing Applications*, 24(4):511–515, 2010.
- [50] John A. Nelder and Roger Mead. A simplex method for function minimization. *Computer Journal*, 7(4):308–313, 1965.
- [51] Stefan Nilsson. Preparing and running experimental designs with BEAST, 11.0. A DOE tutorial, 2011. SKF Engineering & Research Centre.
- [52] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, 2006.

- 
- [53] Per-Magnus Olsson. Test results from parallelization of model building algorithms for derivative-free optimization. Technical Report LITH-MAI-R-20014/01-SE, Department of Mathematics, Linköping University, 2014.
  - [54] Panos M. Pardalos and H. Edwin Romeijn (editors). *Handbook of Global Optimization - Volume 2: Heuristic Approaches*, volume 62 of *Nonconvex optimization and its applications*. Springer-Science+Business Media, B.V., 2002.
  - [55] M.J.D. Powell. The theory of radial basis function approximation in 1990. In W.A. Light, editor, *Advances in Numerical Analysis*, volume 2: Wavelets, Subdivision Algorithms and Radial Basis Functions, pages 105–210. Kluwer Academic, 1992.
  - [56] M.J.D. Powell. A direct search optimization method that models the objective and constraint functions by linear interpolation. In Susana Gomez and Jean-Pierre Hennart, editors, *Advances in Optimization and Numerical Analysis*, pages 51–67. Kluwer Academic, 1994.
  - [57] M.J.D. Powell. Recent research at Cambridge on radial basis functions. In M. D. Buhmann, M. Felten, D. Mache, and M. W. Müller, editors, *New Developments in Approximation Theory*, pages 215–232. Birkhäuser, Basel, 1999.
  - [58] M.J.D. Powell. UOBYQA: Unconstrained optimization by quadratic approximation. Technical Report Technical Report No. DAMTP2000/14, Department of Applied Mathematics and Theoretical Physics, University of Cambridge, 2000.
  - [59] M.J.D. Powell. UOBYQA: unconstrained optimization by quadratic approximation. *Mathematical Programming*, 92(3):555–582, 2002.
  - [60] M.J.D. Powell. The NEWUOA software for unconstrained optimization without derivatives. In Gianni Pillo and Massimo Roma, editors, *Large-Scale Nonlinear Optimization*, volume 83 of *Nonconvex Optimization and Its Applications*, pages 255–297. Springer, 2006.
  - [61] M.J.D. Powell. The BOBYQA algorithm for bound constrained optimization without derivatives. Technical Report NA2009/06, Department of Applied Mathematics and Theoretical Physics, Cambridge, 2009.
  - [62] M.J.D. Powell. The LINCOA algorithm, 2013. <http://www.mat.univie.ac.at/~neum/glopt/powell/LINCOA.txt>. Accessed December 18, 2013.
  - [63] Thomas H. Rowan. *Functional Stability Analysis of Numerical Algorithms*. PhD thesis, University of Texas at Austin, May 1990.
  - [64] Conrad Sanderson. Armadillo: An open source C++ linear algebra library for fast prototyping and computationally intensive experiments. Technical report, NICTA, 2010.

## Bibliography

---

- [65] Thomas Sauer. Computational aspects of multivariate polynomial interpolation. *Advances in Computational Mathematics*, 3(3):219–237, 1995.
- [66] Thomas Sauer and Yuan Xu. On multivariate Lagrange interpolation. *Mathematics of Computation*, 64(211):1147–1170, 1995.
- [67] Khalid Sayood. *Introduction to Data Compression, Second Edition*. Morgan Kaufmann Publishers, 2000.
- [68] K Scheinberg and Ph L Toint. Self-correcting geometry in model-based algorithms for derivative-free unconstrained optimization. *SIAM Journal on Optimization*, 20(6):3512–3532, 2010.
- [69] Katya Scheinberg. Geometry in model-based algorithms for derivative-free unconstrained optimization, 2009. Optima 79.
- [70] Robert B. Schnabel. A view of the limitations, opportunities, and challenges in parallel nonlinear optimization. *Parallel Computing*, 21(6):875–905, 1995.
- [71] Yaroslav D. Sergeyev and Dimitri E. Kvasov. Lipschitz global optimization. In James J. Cochran, Louis A. Cox, Pinar Keskinocak, Jeffrey P. Kharoufeh, and J. Cole Smith, editors, *Wiley Encyclopedia of Operations Research and Management Science*. John Wiley & Sons, Inc., 2010.
- [72] I. V. Sergienko and P. I. Stetsyuk. On N. Z. Shor’s three scientific ideas. *Cybernetics and Systems Analysis*, 48(1):2–16, 2012.
- [73] Naum Z. Shor. *Minimization Methods for Non-Differentiable Functions*. Springer Verlag, New York, 1985.
- [74] Naum Z. Shor and N. G. Zhurbenko. The minimization method using space dilatation in direction of difference of two sequential gradients. *Kibernetika*, 3:51–59, 1971.
- [75] Bruno O. Shubert. A sequential method seeking the global maximum of a function. *SIAM Journal on Numerical Analysis*, 9(3):379–388, 1972.
- [76] SKF. <http://www.skf.com/group/our-company/index.html>. Accessed January 5, 2014.
- [77] Erwin Stinstra, Dick den Hertog, Peter Stehouwer, and Arjen Vestjens. Constrained maximin designs for computer experiments. *Technometrics*, 45(4):340–346, 2003.
- [78] Anke Tröltzsch. *An active-set trust-region method for bound-constrained nonlinear optimization without derivatives applied to noisy aerodynamic design problems*. PhD thesis, University of Toulouse, 2011.

- 
- [79] Edwin van Dam, Dick Den Hertog, Bart Husslage, and Gijs Rennen. Space filling designs. [www.spacefillingdesigns.nl](http://www.spacefillingdesigns.nl). Accessed October 22, 2012.
  - [80] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178 – 194, 2009. Special Issue: Revolutionary Technologies for Acceleration of Emerging Petascale Applications.

## Bibliography

---