universität wien

DIPLOMARBEIT

Titel der Diplomarbeit

„Bi-Criteria decision support for optimizing TV-reach using a genetic algorithm“

Verfasser

Rüdiger Stickler

angestrebter akademischer Grad

Magister der Sozial- und Wirtschaftswissenschaften (Mag. rer. soc. oec.)

Wien, 2008

Studienkennzahl lt. Studienblatt:	**A 175**
Studienrichtung lt. Studienblatt:	**Wirtschaftsinformatik**
Betreuer:	**a.o. Univ.-Prof. Dr. Walter Gutjahr**

Abstract

This thesis deals with the optimization of multi criteria problems using a genetic algorithm. A special kind of the 0-1 knapsack problem with a combinatorial objective function is introduced.

Then the theory of genetic algorithms is described, especially the NSGA-II.

The next part presents the problem of TV optimization, especially the goals of net reach and cost.

Subsequently the implementation of a program to optimize net reach and cost of TV campaigns is described. This program was implemented as a part of this thesis.

The program was used to execute extensive tests to find optimal parameter settings for the genetic algorithm (population size and mutation probability). The results of the tests are evaluated using the Hypervolume Indicator. In addition the development of optimization performance with increasing program run time is analyzed.

Zusammenfassung

Die vorliegende Diplomarbeit beschäftigt sich mit der Optimierung von Mehrzielproblemen unter der Verwendung eines genetischen Algorithmus. Es wird eine spezielle Form des 0-1 Rucksack-Problems mit einer kombinatorischen Zielfunktion eingeführt.

Anschliessend wird die Theorie genetischer Algorithmen beschrieben, speziell des NSGA-II.

Im nächsten Teil wird das Problem der TV Optimierung - speziell hinsichtlich der Netto-Reichweite und der Kosten - dargestellt.

In der Folge wird die Implementierung eines Programmes für die Optimierung von Netto-Reichweite und Kosten beschrieben, wobei das Programm im Rahmen der Diplomarbeit erstellt wurde.

Mit diesem Programm wurden umfangreiche Tests durchgeführt um die optimalen Parameter-Werte (Bevölkerungsgröße und Mutationsrate) für den genetischen Algorithmus zu finden. Die Ergebnisse der Tests werden mit Hilfe des Hypervolume Indicators bewertet. Weiters wird die Entwicklung der performance mit steigender Programm-Laufzeit analysiert.

Table of Contents

Table of Abbreviations

GRP	Gross reach point
NSGA-II	Non-dominated sorting genetic algorithm 2 (a specific genetic algorithm used in the presented application)
NP-hard	nondeterministic polynomial-time hard
GA	Genetic algorithm
EA	Evolutionary algorithm
AI	Artificial intelligence
CPU	Central processing unit, the processor of a computer
TV	Television
HI	Hypervolume indicator, a measurement for the quality of pareto fronts

Index of Figures

Index of Tables

1 Preface

The goal of this thesis is to study the applicability of a genetic algorithm to the problem of TV optimization. The objective of the optimizer is to reach as many different people as possible at the lowest cost.
I will present the theory of genetic algorithms, multi criteria problems and TV optimization. Then I will develop a program implementing a genetic algorithm to optimize TV campaigns.

The main part of the thesis is the analysis of a very exhaustive testing of the algorithm with different parameter combinations. I use the Hypervolume Indicator to compare the different result populations. I also investigate the performance of the algorithm with different run times.

Figure 1: Cartoon illustrating genetic selection (copyright by Gary Larson)

2 Problem

As we want to provide bi-criteria decision support, we actually have two problems to solve. Both are derivatives of the so-called “knapsack problem”, a problem which is very well known to the operations research community. There has been a lot of research targeting this problem and there are lots of books dedicated just to the different solution approaches; for example [Martello, Toth 1990]. The following chapters provide a formal description of the two problems.

2.1 0-1 Knapsack Problem

Suppose a hiker packs his knapsack (or backpack). He has more items than there fit in his backpack. Every item has a value to him and a weight. His problem now is to choose those items that will give him the maximum value under the constraint of a given weight. Figure 2 shows this in a simplified sketch.

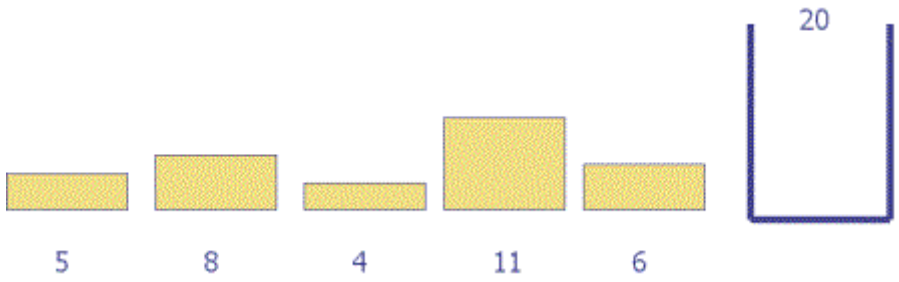

Figure 2: Knapsack Problem (source [www 1])

In this figure the backpack is to the right, with the maximum weight the hiker can carry set as 20. The items are to the left with their respective value underneath. We assume that their weight equals their value. Which of the five items will give the hiker the maximum value while weighing no more than 20? The following Figure 3 shows the solution.

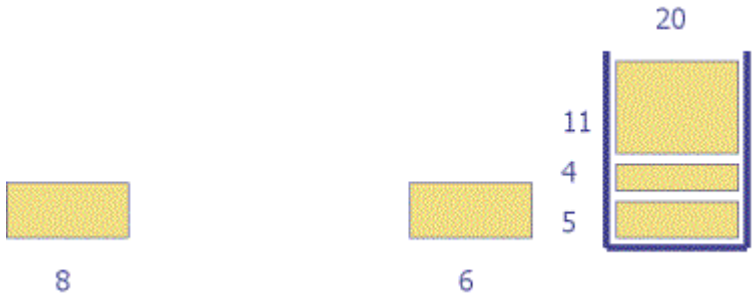

Figure 3: Knapsack Problem solution (source [www 1])

The items with the values 4, 5 and 11 add up to 20 which is the maximum weight the hiker can carry and therefore also the maximum possible value.

Mathematically we can describe the problem as follows: The items are numbered from 1 to n and we introduce a vector of binary variables x_j (j = 1, 2, …, n) with the following meaning:

$$x_j = \begin{cases} 1 & \textit{if object } j \textit{ is selected} \\ 0 & \textit{otherwise} \end{cases}$$

We call the value (or profit) of an item p_j, the weight of an item w_j and the capacity of the knapsack c. The knapsack problem is to select from all the possible binary vectors x that satisfy the constraint

$$\sum_{j=1}^{n} w_j * x_j \leq c$$

the one which maximises this objective function:

$$\sum_{j=1}^{n} p_j * x_j$$

This problem is the simplest form of the knapsack problem and is called 0-1 knapsack problem. The first problem for the bi-criteria decision support takes this form. The next chapter describes the second problem.

2.2 0-1 Knapsack Problem with Combinatorial Value Function

The second problem is also a 0-1 knapsack problem but has an interesting new property: we add another dimension. We have m knapsacks, each of them containing the same items. The items however have different values in different knapsacks. The constraint for the problem stays the same as in the simple case above:

$$\sum_{j=1}^{n} w_j * x_j \leq c$$

The objective function (which we want to maximise) for this problem is

$$\sum_{i=1}^{m} 1 - (\prod_{j=1}^{n} (1 - p_{ij} * x_j))$$

where p_{ij} are probabilities with values between 0 and 1 and x_j is 0 or 1 (see above). This is a rather complex knapsack problem which is not described in current literature as far as I know. In the case of the net reach objective function the index i refers to the viewers of the TV spots, and the index j refers to the TV spots. A more detailed explanation of net reach can be found in chapter 5.4.1 “How are net reach and gross reach measured?”

2.3 NP-Hardness

The two variants of the knapsack problem described above are both NP-hard. This means there is no analytical way to solve the maximisation problem, the true maximum can only be found for sure by brute force algorithms which check all possible vectors. The proof for this can be found here: [Martello, Toth 1990].

3 Multi-Criteria Problems

Many real-world problems don't have a single objective, but rather several non-commensurable and often competing measures of performance, or objectives. Examples of this would be risk versus profit or labour cost versus social security. Formally we can write the problem like this:

$$\min_{x \in C} F(x) = \begin{bmatrix} f_1(x) \\ f_2(x) \\ \vdots \\ f_n(x) \end{bmatrix}$$

Problems of this sort usually don't have a unique, perfect solution, but several equally efficient solutions. These are called the Pareto-optimal set. To find the Pareto-optimal set we need to understand the two concepts of inferiority and non-inferiority. Assuming a maximisation problem, a solution vector u is inferior to a solution vector v if every single objective of u has a lower value than the same objective of v. This can be written as

$$\forall i \in \{1,...,n\}, u_i \leq v_i$$

Solution u is said to be inferior or dominated by v.

Two vectors u and v are called non-dominated by each other or non-inferior if neither v is dominated by u nor vice versa.

This concept is illustrated in the following Figure 4.

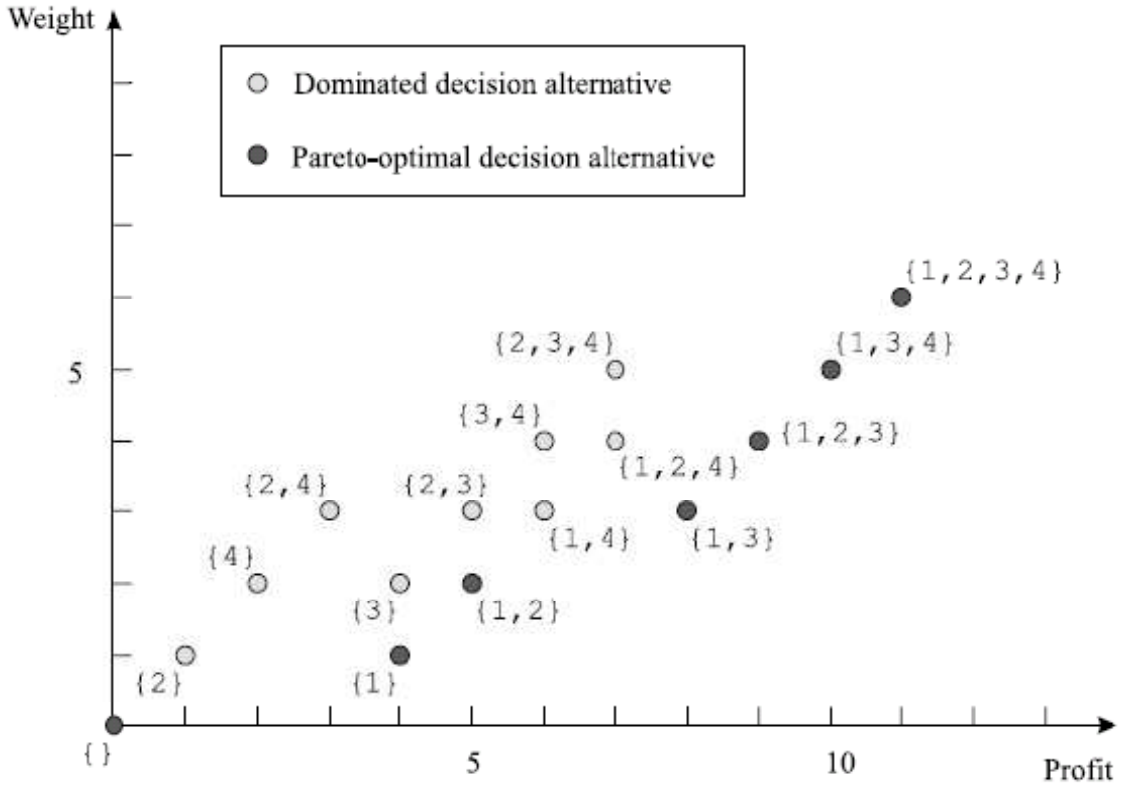

Figure 4: Pareto-Optimal Solutions for the 0-1 Knapsack Problem (source [Laumanns 2003])

This example shows some solutions (solid circles) which dominate others (empty circles). Obviously profit should be maximised and weight should be minimised.

To find a single solution it is necessary to weigh the different objectives against each other and choose from the pareto-optimal set using some sort of preferences. Fonseca and Fleming [Fonseca, Flaming (1995)] cite three different classes for the articulation of preferences.

- **a priori articulation of preferences** – The decision maker aggregates the individual objectives to one single objective function, changing the problem to a single-objective problem before the actual optimisation.
- **a posteriori articulation of preferences** – The decision maker gets a set of non-dominated solutions after the optimisation process and chooses the one best fitting his preferences.
- **progressive articulation** – Optimisation and decision making take alternate turns, so the decision maker can supply partial preference information which might depend on intermediate solutions

The preferences of the decision maker have to be formulated in some sort of utility function. The most widely used ways to do that are the following:

- **weighting coefficients** – a real number is assigned to every objective, which expresses the relative importance to the other objectives. The weighted-sum approach is a very common example.
- **priorities** – an integer is assigned to every objective, designating the order by which the objectives will get optimized.
- **goal values** – for every objective a desired value is given, which could be a minimum, a maximum, or a value which should be attained as closely as possible.

Evolutionary algorithms have become very popular for multi-objective optimisation, because they can calculate several solutions in one simulation run (as opposed to classical algorithms).

4 Genetic Algorithms

A genetic algorithm (GA) is a meta-heuristic used to find approximate solutions to search and optimization problems. It is based on the principles of selection, cross-over and mutation known from classic genetics. It is a general-purpose algorithm and can run without special knowledge of the problem domain. This chapter at first describes GAs in general including a short introduction to classic genetics and then the particular algorithm used for the practical part of this thesis: NSGA-II [Deb 2000].

4.1 Introduction: Classic Genetics

The foundation of classic genetics was laid by Gregor Mendel's laws of inheritance, and by Charles Darwin's theory of variation and natural selection.

Gregor Mendel (1822 – 1884) was an Austrian Monk who studied the nature of inheritance. He cultivated thousands of pea plants over several generations and tested them, carefully documenting the results [Mendel 1865]. He formulated two laws: The law of segregation and the law of independent assortment. According to these laws, each organism inherits two *genes* for each character, one from each parent. Every gene can have different versions, resulting in different characteristics of the same character. These are called *alleles*. If the two alleles are not the same only the dominant allele is expressed in the *phenotype* (the physical appearance). When the organism produces sex cells (*gametes*) the two genes for each character segregate leading to variation in the offspring. Therefore, an organism can bequeath something that is not in his phenotype but only in his genotype (A man for example can pass on the gene for blue eyes to the next generation that he got from his father, even though he himself had his mother's brown eyes).

Charles Darwin (1809 – 1882) was a British naturalist who proposed the theory of variation and natural selection in his famous book "On the origin of species by means of natural selection or the preservation of favoured races in the struggle of life" [Darwin 1859]. The principle of variation says that there is an overproduction of offspring in every species which differ in their phenotype. These differences result from differences in the *genotype*; that is in the genes. Darwin observed these differences but could not explain them. Today we know several reasons: random mutation, recombination, genetic drift and others. Because of the massive overproduction there is a competition over light, water, natural habitat, sexual

partners, etc. Different individuals also have different adaptation to different environmental factors like temperature, wind, salt levels, etc. This "survival of the fittest" leads to better adapted individuals in every generation and finally in the long run to the emergence of new species (macro-evolution).
Mendel's laws failed to explain why all members of a population did not eventually regress to a phenotypic mean, but together with Darwin's principles of variation and natural selection the basic concept of evolution was formed.

Throughout the course of the twentieth century much research was devoted to the mechanisms of genetic inheritance, and our knowledge in this field has grown immensely, with some notable highlights:

- In 1910 the *chromosome* was found to be the carrier of the genes. The position of a gene on the chromosome is called *locus*.
- In 1944 it was shown that the DNA (Deoxyribonucleic acid) holds the gene's information
- In 1953 James Watson and Francis Crick decoded the molecular structure of the DNA.

4.2 Classification of the Genetic Algorithm

Nature was always a fascinating role model for computer scientists and many of them tried to apply her principles in their field. Most notably these were intelligence and the brain (resulting in neural networks), the capability to learn and control the environment (resulting in machine-based learning) and evolution (resulting in evolutionary computing). Together these concepts constitute the field of Artificial Intelligence (AI) (see Figure 5 for an overview).
Evolutionary computing itself can broadly be subcategorized into evolutionary algorithms (EA) and swarm intelligence (comprising ant colony optimization and particle swarm optimization), both of which are metaheuristic optimization algorithms, that is very general algorithms that have no special knowledge of the problem domain.
There are several different evolutionary algorithms:

- Genetic Algorithm (GA): The most popular EA today, a solution of a problem is represented by a string of numbers, on which evolutionary operators are applied (selection, mutation, recombination)

- Evolution strategy: Solutions are represented by vectors of real numbers, mutation rates are usually self-adaptive
- Genetic programming: Solutions have the form of computer programs, their fitness is modelled by their ability to solve a certain problem
- Evolutionary programming: Like genetic programming, but the structure of the program is fixed and only the parameters can change

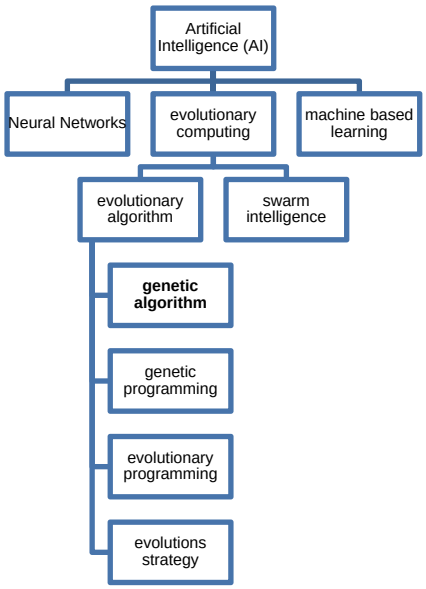

Figure 5: Overview of Concepts of Artificial Intelligence (AI)

4.3 Brief History

The earliest programs that could be called genetic algorithms appeared in the late 1950's and early 1960's, when biologists tried to model some aspects of evolution. It took a while until the idea was born to use those methods for search and optimization.

One of the earliest more successful implementations was a technique called evolution strategy by Ingo Rechenberg of the Technical University in Berlin. He didn't use a population and there was no crossover; one parent was mutated to produce one offspring. The better of the two was used as parent in the next generation. In later versions the idea of populations was included.

In 1966 another important technology was developed: evolutionary programming. Solutions are coded as finite state machines and are randomly mutated. The ones that are better suited to solve a problem are kept for the next generation.

Maybe the best known founding father of genetic algorithms was John Holland. His work beginning in 1962 laid the foundations for many later developments. Most notably he was the first to propose crossover and other recombination operators. In 1975 his book "Adaptation in Natural and Artificial Systems" was published, and laid out a solid theoretical foundation for genetic algorithms, by introducing the concept of schemata, among other things. It was the first to present the evolutionary process including selection, mutation and crossover systematically.

In the 1980's and 1990's these techniques spread more and more to the commercial sector as a result of increases in computing power, decreases in cost and the rise of the internet.

Today GA's are applied in virtually every field (stock market prediction, microchip design, aerospace engineering, biochemistry and scheduling at assembly lines to name a few).

4.4 Basic Mode of Operation

Every genetic algorithm must have the following five components:

- a genetic representation for a (potential) solution to the problem
- a routine that creates the initial population of solutions
- a function that calculates the quality of the solutions (fitness evaluation function)
- functions that implement genetic operators (like selection, mutation and crossover)
- values for several parameters like population size, probability of applying a certain genetic operator, etc

A genetic algorithm starts by creating a population of candidate solution, usually randomly. From this generation, certain individuals get selected based on their fitness. These individuals are modified (mutated and/or recombined) to form a new population. This process repeats itself until a certain quality of solution is achieved or for a certain number of generations. Following is a more detailed description of these processes.

4.4.1 General structure and Terminology

The core of the structure of a genetic algorithm is the potential solution or individual. This is equivalent to an organism in natural evolution. An organism's genes are coded in chromosomes, which in the case of a genetic algorithm are usually coded as strings of numbers. See Table 1: List of Terms of Natural Genetics and their translation to Genetic Algorithms for a list of natural genetic terms and their pendant in genetic algorithms (compare Introduction: Classic Genetics). For most problems it's sufficient to use a binary string

consisting only of 0s and 1s. In the well-known knapsack problem for example you have to fit as many items with different weights as possible in a knapsack without breaking it. In this case every item would be represented by one character in the binary string, 0 meaning this item is not included as part of the solution, 1 meaning it is.

A genetic algorithm uses a population of several individuals to maintain diversity. While there are some rules of thumb for the choice of the population size, there is currently no optimal way today to ensure a good or optimal number. The bigger the population the more variation but this comes at the cost of performance. One has to balance the need for variation against performance, smaller populations allow more generations which could lead to better results.

Natural Genetics	Genetic Algorithm
Chromosome	String
Gene	Feature, character
Allele	Feature value
Locus	String position
Genotype	Population
Phenotype	Alternative solution

Table 1: List of Terms of Natural Genetics and their translation to Genetic Algorithms terminology

4.4.2 Initialization

Usually the first population is generated completely randomly, so the entire search space is represented. In some cases however it can be useful to seed solutions in an area that is likely to contain the optimal solution.

The population size is usually set at hundreds or thousands of individuals. There are some pointers on how to set the population size optimally (e.g. [Gotshall S, Rylander B 2003]), but there is no consensus about this in the research community.

4.4.3 Fitness evaluation (selection)

Every individual of the population gets evaluated based on a fitness-function. This measures how good the candidate solution is. Better solutions are preferred to worse ones obviously, this is called selection pressure. There are many different implementations for the selection process; there are also methods which include bad individuals to maintain a bigger diversity.

Other algorithms use different techniques to ensure diversity and choose only the best individuals.

The fitness evaluation is usually the most time consuming part in the algorithm and should therefore be optimized for speed the most. If it is still too slow one can try to choose an entirely different fitness function.

4.4.4 Genetic Operators (reproduction)

After the best individuals have been selected a child generation is generated. The two most important ways to do this are crossover and mutation. For a crossover two parent-individuals are selected and give different parts of their chromosome to the child. For mutation the chromosome is altered randomly, but only in a small part because big mutations tend to yield very bad results.

In this way a complete new generation is generated which typically has a better average fitness than the parent generation because only the best individuals of the parent generation have been selected to breed the new one.

4.4.5 Termination

This process is repeated again and again until certain conditions are met. These conditions usually are one or more of the following:

- a certain number of generations has been reached
- a certain quality of the solution has been reached
- a certain computing time has been reached
- the solutions have reached a plateau and new iteration don’t bring better solutions
- manual termination

4.5 Properties

Genetic Algorithms combine two important properties for search algorithms: exploration (stochastic search) and exploitation (directed search). Exploration is the ability to search a big part of the search space as opposed to exploitation which is the ability to find the optimum from a particular starting point (exploiting this point). The traditional hill climbing algorithm, which always moves toward the steepest incline until it reaches a peak is very good at exploitation. Random search where random points in the search space are looked at until some condition is reached would be an example of an algorithm with good exploration ability.

While there are classic algorithms which combine these two important properties (like random restart hill-climbing), the genetic algorithm is particularly good at combining these two seemingly opposed properties which makes it more robust than other algorithms.
Another important property of genetic algorithm is the population. Classic algorithms usually only look at one solution at a time whereas the GA uses a whole population of solutions at the same time. This can be used to maintain diversity which is especially useful in multi-criteria optimizing where one might want to find a set of pareto-optimal solutions instead of just one solution.

4.6 Criticism

While genetic algorithms today are widely accepted both by researchers and by practitioners to be successful in many problem domains there is also some criticism. As an example the "No free lunch" Theorem is mentioned here. It was first presented in this context in a paper by Wolpert and Macready [Wolpert, Macready 1995]. It says that *"...all algorithms that search for an extremum of a cost function perform exactly the same, when averaged over all possible cost functions. In particular, if algorithm A outperforms algorithm B on some cost functions, then loosely speaking there must exist exactly as many other functions where B outperforms A."* This suggests that specialized algorithms which incorporate some knowledge of the problem domain should always be preferred to general-purpose algorithms like genetic algorithms (see Figure 6). Of course there are other things to consider too, like the time it takes to implement an algorithm. Also there are cases where no knowledge from the problem domain can be applied in an algorithm.

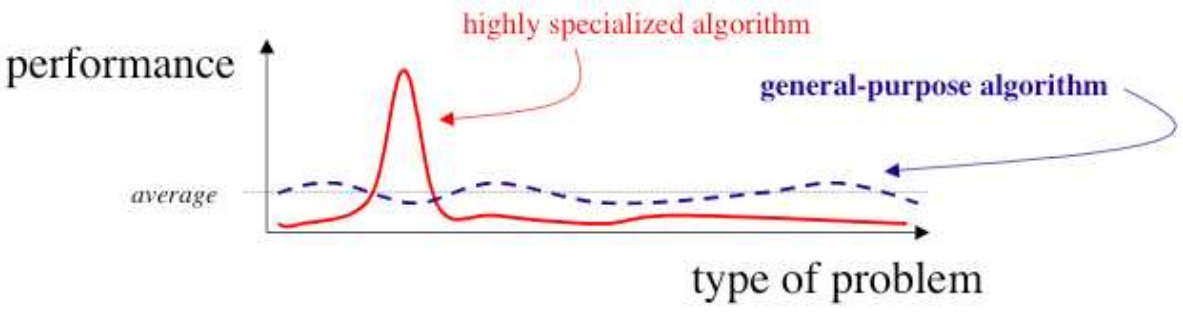

Figure 6: "No free lunch": Performance of specialized versus general-purpose algorithm

4.7 NSGA-II

The NSGA-II (non-dominated search genetic algorithm II) is a particular genetic algorithm which is especially good at multi-criteria optimization. It was published in 2000 by Kalyanmoy Deb [Deb 2000]. It maintains a population of pareto-optimal solutions, that is solutions where one cannot say that one is better than another without preferences on the different criteria.

The two main components of this algorithm are the non-dominated search which finds sets of pareto-optimal solutions and the diversity preservation which ensures a good exploration of the search space and a very good convergence to the true pareto-optimal front (the set of all pareto-optimal solutions in the search space). After explaining these two components in more detail the main loop of the algorithm will be described.

4.7.1 Non-dominated Sort

We are working with multi-criteria problems, so each solution is actually a solution vector with values for each objective. As we have seen in chapter 3 “Multi-Criteria Problems” a solution is dominated by another solution if the values for all objectives are inferior.

The non-dominated sort of the NSGA-II finds all solutions which are not dominated by other solutions. This set of solutions is called a non-dominated front. After this front was found the search is performed again and finds all non-dominated solutions from the remaining solutions, they are the second non-dominated front. The search is repeated until there are enough solutions for a child population, i.e. at least half the solutions.

To sort a population according to its non-domination each solution has to be compared with every other solution in the population on every criterion. This implies a computational complexity of O(M * N) for one solution where M is the number of objectives and N is the number of individuals in the population. Doing this process for all the solutions in a population takes a complexity of O ($M * N^2$). This will find the first non-dominated front, to find all non-dominated fronts it takes O ($M * N^3$) in the worst case which is when there is only one individual in every front.

The NSGA-II uses a bookkeeping strategy to reduce the complexity to O ($M * N^2$). It takes one solution at a time and includes it in a set temporarily. Then it compares this solution with all the other solutions in this temporary set. If the new solution dominates another solution in the set that solution gets deleted from the set. If the new solution is dominated by a member

of the set the new solution gets deleted from the set. When all the solutions of the population are checked, the first non-dominated front is found. This procedure is repeated until enough non-dominated sets are found. Following is the implementation of this procedure in pseudo-code.

```
P' = find-nondominated-front(P)
P' = {1}                                  include first member in P'
for each p ∈ P ∧ p ∉ P'                   take one solution at a time
    P' = P' ∪ {p}                         include p in P' temporarily
    for each q ∈ P' ∧ q ≠ p                compare p with other members of P'
        if p ≺ q, then P' = P'\{q}         if p dominates a member of P', delete it
        else if q ≺ p, then P' = P'\{p}    if p is dominated by other members of P',
                                           do not include p in P'

F = fast-non-dominated-sort(P)            F is a set of non-dominated fronts
i = 1                                     i is the front counter and is initialized to one
until P ≠ ∅
    F_i = find-nondominated-front(P)      find the non-dominated front
    P = P\F_i                             remove non-dominated solutions from P
    i = i + 1                             increment the front counter
```

Figure 7: Overview of the non-dominated-sort in pseudo code (source [Deb 2000])

4.7.2 Diversity Preservation

The result of the repeated non-dominated search is a convergence to the optimal solution. Additionally we want to have a good spread of solutions in the set. The NSGA-II uses a density estimation to achieve this. It measures the distance between the two neighbours of a point along each objective to get an estimate of the smallest enclosing cuboid that doesn't include any other solution (marked by the dashed line in Figure 8).

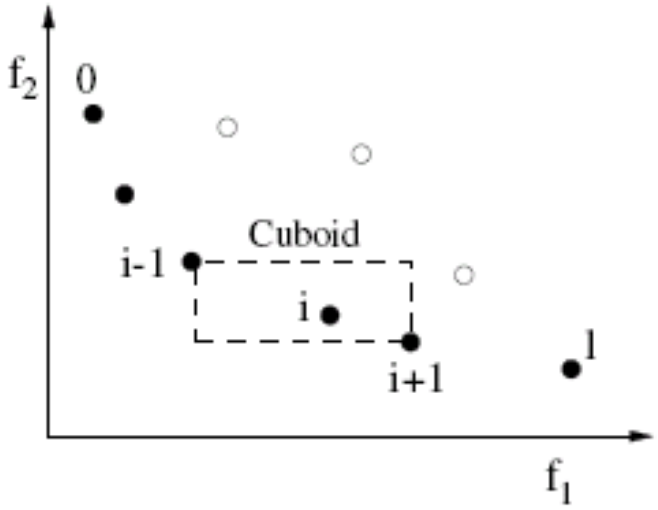

Figure 8: Sketch of NSGA II's Diversity Preservation Mechanism (source [Deb 2000])

This distance measurement requires the sorting of all solutions according to each of the objectives. The following pseudo-code shows how.

```
crowding-distance-assignment(I)
l = |I|                                  number of solutions in I
for each i, set I[i]_distance = 0        initialize distance
for each objective m
  I = sort(I, m)                         sort using each objective value
  I[1]_distance = I[l]_distance = ∞      so that boundary points are always selected
  for i = 2 to (l − 1)                   for all other points
    I[i]_distance = I[i]_distance + (I[i + 1].m − I[i − 1].m)
```

Figure 9: Overview of the diversity preservation in pseudo code (source [Deb 2000])

The complexity of this part of the algorithm is governed by the particular sorting algorithm that is used, which is usually O (N * log(N)) (for example “quicksort”). The overall complexity for sorting according to each of the M objectives therefore is O(M * N * log(N)). The distance measurement for every point is the cumulated distance measurement for all objectives. If one solution has a higher value than another this means it is more crowded which makes it less favourable. That’s the reason the authors of the algorithm call this procedure crowding-distance-measurement.

4.7.3 Main loop

Here we will see how the non-dominated search and the crowding-distance measurement will work together to form an efficient algorithm. At first a random population is created. It gets

sorted according to non-domination, resulting in several pareto-optimal fronts. To select the better half of the population we take the first non-dominated fronts until we reach the one that crosses the line to the second half of the population. This set gets sorted by its crowding-distance and we only take the least crowded solutions. With the resulting half population we use the usual genetic operators (mutation, cross-over, etc) to form a new child generation. These two get recombined and the next generation is performed (see Figure 10 for a graphic representation of this process).

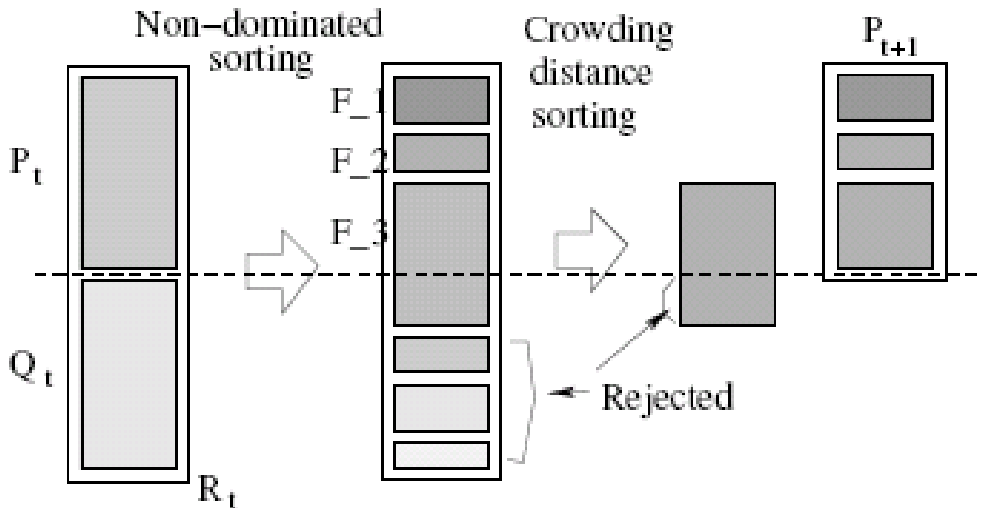

Figure 10: Mock-up of the NSGA-II (source: [Deb 2000])

5 Problem Instance

The specific problem discussed in this thesis is that of TV-advertisement-optimization by Media Agencies. This chapter describes in more detail the heritage, goal and importance of TV optimizers.

5.1 Why advertisement and how does it work?

The ultimate goal of all advertisement is to maximize sales. There are other, long-term goals like building brand equity but in the end it is sales that count.

Historically there are two different views about how advertisement works [Broadbent 1997]:

1. Learning

   This theory states that consumers have to be taught about products.

2. Relevance

   This view sees relevance of the advertisement to the consumer as the driving factor.

Today most experts agree on the second theory. Consumers are not passively waiting for messages on TV; they do other activities while watching TV, not paying attention when there is nothing of interest.

Relevance is the key factor [Ephron 1998]. A message is relevant to a consumer if he is ready to buy a certain product. An example would be a person who just ate the last cereal and wants to buy some more. To him an advertisement of Kellogg's Corn Flakes is relevant and it could influence his brand selection when buying the product.

This theory implies that it is more important to reach as many different people as possible with an advertisement as opposed to the classical approach of having as many contacts as possible (where reaching the same person several times counts as contact). There is no way to find the persons who are ready to buy a product, so the first approach to reach as many different people as possible maximizes the possibility of reaching those persons. This goal is called Reach or Recency, the goal of getting as many contacts as possible is Frequency.

As a side note, it actually is possible to find people who are more likely to be ready to buy a product. As an example Weight Watchers commercials are usually aired after the holidays, stop-smoking commercials in the morning with the first coffee, etc. This kind of scheduling is communication strategy and should be used regardless of reach or frequency goals.

The actual impact of advertising is still a source of discussion. One of the more prominent discussions was about the banning of cigarette advertising. The cigarette manufacturers claimed that advertisement would only influence brand selection for people who smoke. The opponents of cigarette-advertising say that it increases consumption as well. This case was won by the cigarette-advertising opponents, but there are still many debates in different contexts and there is no common opinion among experts.

5.2 How to measure advertisement results?

A well known quote in the advertising industry goes as follows: "I know half of my dollars are wasted, but I don't know which half." (The quote is generally attributed to either John Wanamaker or William Lever).

This quote illustrates how hard it is to reach potential customers using traditional advertising. Today we can look at big databases of advertising cost, sales and other data. MMA is one of the leading marketing-mix modelling firms. Their database also includes factors like price, competition, weather, trade support and marketing [www 2]. As a measure for the effectiveness of advertisement they use the following formula:

$$\frac{\text{total brand sales} - \text{cost of goods}}{\text{cost of advertising}}$$

This is the equivalent of advertising-delivered "profit before taxes" [Ephron, 2003].

With this model it is possible to find out the short-term effects of advertisement and it's ROI (if the data is available for a market).

5.3 What is a Media Agency?

When a company wants to advertise its products it usually has to go to two agencies: A creative agency and a media agency. The creative agency gets briefed about the target audience, the goals of the campaign and the used media to create an ad (like a TV-spot or magazine-ad). Some years ago they went on to plan a schedule and buy the advertising space, but recently this planning and buying process has become so complex that special media agencies were formed, mostly by the planning departments of creative agencies. Nowadays there are a lot more TV-stations, radio-stations, magazines and other media than there used to be ten years ago. There are also many new special advertisement media (in Austria for example ContainAd, Rolling Board, etc).

Media Agencies have a much broader field of operation today than just scheduling and buying:

- **Market Research**

  They find out as much as possible about the consumers. Who will be most interested in buying the product? What do these people like? What media do they use and when? How can they be reached best?

- **Brand Research**

  What are the goals of the brand? How does the client want to develop the brand?

- **Communication Strategy**

  What message should be communicated?

- **Other Research**

  How does advertising work in general?

  Media Agencies also have to constantly look for new ways to stand out from their competition. They come up with new unusual formats, interactivity, etc.

- **Reporting**

  There is a rising need for qualitative and quantitative measures of the impact of advertisement. Media agencies try to come up with better and better measures, so they can prove their value added to the client and show that they are better than their competitors.

- **Tools**

  Media Agencies are constantly developing new tools to optimize the planning and/or buying process (like TV-Optimizers) and to get new insights from their data.

The process of an advertisement campaign is roughly as follows:

1. Briefing

The client tells the media agency about his goals. Some clients are very specific, others want consulting in some areas.

2. Strategy

The media agency develops a strategy suited to the needs of the client. They conduct market research, draw on the results of their regular research and try to come up with some creative new ideas.

3. Planning

The planning department of the agency develops a schedule for the campaign. They might use several tools, like TV-optimizers or forecast-tools.

4. Buying

   The goal of the buying department is to buy advertising space as closely to the planned schedule as possible.

Because of the huge amounts of money the media agencies spend for their clients they get quantity discounts which they can pass back to their clients, allowing their clients to reach more people with the same amount of money.

5.3.1 How are media agencies positioned?

All major media agencies are part of big networks that own several creative agencies, media agencies and all kinds of agencies who specialise in certain industries or functions (like healthcare, direct marketing, interactive, etc). A list of the Top Ten Marketing Networks worldwide can be found in Table 2.

Rank	Marketing Organization	Worldwide Revenue in (Mio $)	% Change to previous year
1	Omnicom Group	9,747.2	+13.1
2	WPP Group	9,370.1	+16.2
3	Interpublic Group of Cos.	6,200.6	+5.8
4	Publicis Groupe	4,777.3	+8.4
5	Dentsu	2,851.0	+19.1
6	Havas	1,866.0	-0.6
7	Aegis Group	1,373.6	+28.7
8	Hakuhodo DY Holdings	1,372.4	+16.4
9	Asatsu-DK	473.3	+14.3
10	Carlson Marketing Group	346.9	+7.6

Table 2: Top Ten Marketing Networks by Revenue in 2004 source: [Advertising Age Feb 2006])

The big networks did a lot of acquisitions and mergers; the market is dominated by few very big ones. This trend seems to continue.

5.4 What is TV-Optimization?

For many blue chip companies, TV is the most important media vehicle. TV quicky builds reach, creates a world through audio visual elements and entertains. Unfortunately, this popularity is reflected in the high cost of TV airtime.

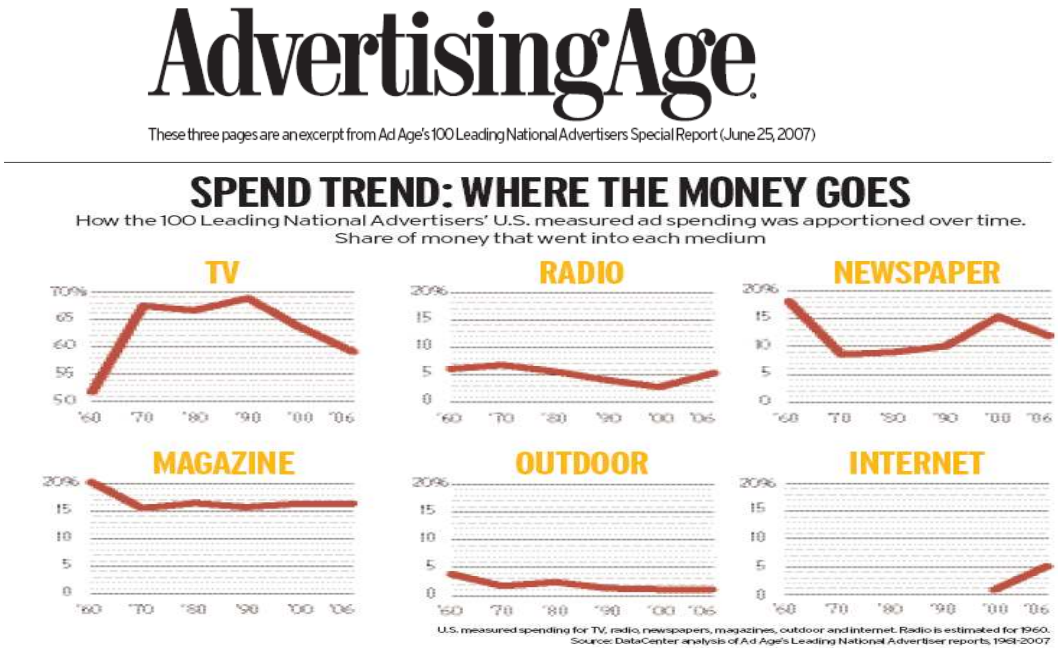

Figure 11: US advertisement spending by medium in 2007 (source: [Advertising Age 2007])

The cost of a single thirty-second TV spot during the annual Super Bowl in the United States has reached $2.6 million (as of 2007) [www 3].

As we saw in chapter "5.1 Why advertisement and how does it work?" a very important advertising goal is net reach, that is reaching as many different persons as possible [Kotler 1997, p649ff]. This goal is mathematically difficult to solve as we saw in chapter "2.2 0-1 Knapsack Problem with Combinatorial Value Function".
This chapter will show how net reach is measured and what data is needed.

5.4.1 How are net reach and gross reach measured?

Net reach is measured as a percentage of a target audience. In this paper I use a target audience of women between 18 and 38 which is a typical target audience. A net reach of 50 would mean 50% of all women between 18 and 49 have seen the spot (at least once).

Gross reach is called GRP (gross reach point) and is measured in the same way. 200 GRP would mean twice as many contacts as there are people in the target audience (which doesn't mean every single person in the target audience was reached).

A campaign could have 200 GRP and a net reach of 20, and it could have 200 GRP and a net reach of 40. In the first case every person who saw the spot, saw it ten times on average, in the second case five times (but twice as many different persons).

The industry standard for calculating reach (which I use) is actually slightly different: It is the actual time of watching divided by the possible running time. This is also called p-value and is the probability of a person to see a certain TV spot. The reason for this is that the available data only tell us the time span the viewer watched an advertisement block (containing several TV spots) but not if and how long he saw the one we are concerned with.

Example: If an advertisement block lasts 5 minutes and a person watches 1 minute of this block (consecutively or not) she has a p-value of 1/5 or 0.2. This can be seen as having a 20% chance of having seen a certain TV spot in this block.

To get the net reach for one person watching multiple advertisement blocks we use the following formula:

$$1-\prod_{j=1}^{n}(1-p_j * x_j)$$

where pj is the p-value of advertisement block j and xj is 1 if the block is booked, 0 if not.

The result is the probability that the person has seen the spot at least once. The formula for the cumulated net reach for all persons is then

$$\sum_{i=1}^{m} 1-(\prod_{j=1}^{n}(1-p_{ij} * x_j))$$

In the given case test-data from Austria are used, provided by Fessel GFK, a marketing research company.

For every person in the test panel there is a weight for each day which is defined so that the persons in the panel are representative (with regard to gender, education, income and others) of the total population of Austria.

To get the total net reach for a campaign we sum up the net reach value for each person (weighed accordingly for each day). The weights can change from day to day as new persons enter the panel while others leave it and because the total population changes over time.

Gross reach can easily be computed per advertisement block as the sum of all p-values multiplied with their respective weight for the day.

5.4.2 Why are optimization tools needed?

As explained in chapter 2.2 “0-1 Knapsack Problem with Combinatorial Value Function” the net-reach problem is NP hard and therefore there cannot be found a solution with analytical means. Only by trying all possible combinations the campaign with the best net reach can be found. This is not possible today or in the near future because of the huge number of possible combinations.

The only solutions are either expert knowledge (“educated guesses”) or meta-heuristics, one of which is proposed in this paper.

5.5 What Data Do I Use?

The public TV in Austria is analyzing the TV audience via an ongoing study conducted by Fessel GFK. A panel of 1500 people who represent the Austrian population (with respect to age, gender, location, social class) is equipped with special remote controls which document which person watches which channel at what time.

Another data source I use is monthly market data from Focus GmbH. These contain the cost of TV spots.

Note: For the presented application I use ex-post data, which means I optimize using the statistical data after the campaign is over. In real life we have to use forecasts which add another layer of complexity and which is out of scope of this paper.

5.6 Further Aspects Not Covered By the Presented Application

The optimization algorithm uses ex-post data about the viewing behaviour of the population. To use the application in real life a prognosis of this would be necessary. This is another complex topic and is not in the scope of this thesis.

There are many more factors to consider when planning a campaign [Arpino, 2008]:

- Volume discounts by TV stations (where certain volume criteria have to be met to get a higher discount for all bookings)
- Rebate in kind
- Other incentives by TV stations
- Client preferences

- Relevance (some TV programs typically will be more in tune with a product than others).

Ideally, these factors should be taken into consideration in the optimization process; this is also not in the scope of this work.

6 Implementation

This chapter describes in detail the implementation of the presented application. It also explains the choice of technologies and describes the methods I used to optimize the source code for optimal performance.

6.1 Technologies used

The application was implemented in the language C. This choice was made because performance is absolutely critical to achieve good results. In C every line of code is the equivalent of only a few assembly language commands (machine code). Higher level languages like Java often produce a lot of code in the background without the programmer's knowledge and often out of the programmer's control. C also gives the programmer full control of computer memory, so the programmer can choose exactly when memory is allocated and de-allocated.

PostgreSQL is used as the relational database, mainly because it is free and features stored procedures. PLSQL is used as the language for stored procedures.

Java is used for the data import routines, because it has a better database support than C.

6.2 Initialization

The raw data is available in text files only and there are different sources for the data about the cost of the spots and the viewer behavior. To work with the data it was helpful to import it into a relational database. As a next step the data from the two sources must be linked. This was done by a code identifying each TV spot ("Block code"). In both data sources these codes are invalid or missing for about one to three percent of the records. In these cases I used date and time to match the spots. This was implemented as a stored procedure in the database to optimize performance.

When all the data is stored in the database an application routine can be called to extract the relevant data from the time frame the user wishes to use and store it in database tables

especially tailored to the optimization application. These steps are not part of the main program and are therefore not described in more detail.

When the application is started it takes three input parameters: Population size, mutation probability and Number of CPU clock cycles until the program terminates. These parameters are saved internally and used in several parts of the application.
Next the program calculates the number of TV spots and the number of persons in the timeframe which will be optimized. These numbers are in turn used to allocate memory for the data structures and to de-allocate the memory after the program has finished. Finally the various data structures holding all the necessary data are filled from the database.

Next a random population is generated. The number of individuals is defined by the parameter population size. For every individual (which represents a campaign) a random set of spots is chosen for initialization. I arbitrarily chose 1% as the percentage of spots which are chosen for the initial individuals. Different percentages didn't change the results in several experiments, an indication for a well working diversity preservation in the optimization algorithm.

6.3 Data structures

The main data structure used in the application is called "individual". It represents a TV campaign and contains an array of TV spots (these are the genes in genetic algorithm terminology). Each spot is represented by an Integer and can be either 1, which means the spot is part of the campaign; or 0, which means the spot is not part of the campaign. (Note: I chose Integer as data type because it is very simple to use and is also the fastest data structure in terms of access and computations on the 32 Bit x86 platform. A self-made bit data type would be more memory efficient but it would not be faster.)
This data structure also stores the fitness values (Cost and net reach).
Furthermore it stores the crowding distance and the rank of the individual, which are used for the non dominated sort (see chapter 4.7.1 Non-dominated Sort).
Lastly it stores some temporary data used for a high performance fitness calculation and the number of booked spots (number of genes with the value 1).
All individuals of a simulation run are stored in a population structure.

Further data structures used:
The p-values used to calculate the net reach are stored in an array together with the person and the spot: data[person][spot] = 1-p-value
The cost of each spot is stored in an array: cost[spot]
The weight of each person per day of year is stored in an array: weight[person][day of year] = weight
The GRP (gross reach points) of every spot are stored in an array: grp[spot]=grp
The date of each spot is stored in an array: spot_date[spot]=day of year.

6.4 Main algorithm

This chapter shows the inner workings of the optimization routine. It consists of code fragments in pseudo-code followed by a short explanation.
For a basic theoretical overview of the algorithm see chapter 4.7 NSGA-II.
The algorithm gets three parameters from outside when the program is started: The population size, the mutation probability and the allocated runtime in CPU clock cycles.

6.4.1 Main loop

The following code fragment shows the outer loop of the optimization algorithm, the loop implementing a generation of the genetic algorithm.
The algorithm uses mutation as its only genetic operator. The crossover operator is not used because the net reach of an individual is dependent on all the genes. A combination of genes which results in a high net reach cannot be expected to result in better than average results when it is applied to a different individual with a different chromosome. It would just equal a very big mutation, which doesn't produce good results.

Code fragment: Main loop

`initRandomPop(MixedPop)`	Initialize random start population
`calcFitness(MixedPop)`	Calculate fitness of start population
`parentPop = nonDominatedSort(MixedPop)`	Do a non-dominated sort and store the better half of the population as parent population
`copyPop(parentPop, childPop)`	Copy parent population to child population

```
mutate(child)                          Mutate child population.
//start generations
loop while(clockcycles < limit)        Loop until a specified runtime limit is
                                       reached
     mixedPop = parentPop + childPop   Combine parent and child population
                                       to a mixed population
     parentPop = nonDominatedSort (mixedPop)   Do a non-dominated
                                       sort and store the better half of the
                                       population as parent population.
     copyPop(parentPop, childPop)      Copy parent population to child
                                       population.
     mutate(childPop)                  Mutate child population.
     calcFitness(childPop)             Calculate fitness of child population.
end while loop //end generations
```

Explanation

The above code fragment is an overview of the optimization algorithm on a very general level. The subroutines are explained in more detail below. One loop in the code represents one generation of the genetic algorithm. Parent and child population are combined to a mixed generation. This mixed generation is then sorted by non-domination (see chapter 4.7.1 Non-dominated Sort). After sorting the better half of the mixed generation is the new parent population. The parent population is then copied; the copy represents the new child population. Finally the child population is mutated and the fitness values of its individuals are calculated. This concludes a generation and the next generation starts by combining the mutated child population with the parent population. This loop is run until the clock cycles used by the program exceed the allowed CPU clock cycles for a simulation run (these are provided as an input parameter to the application).

6.4.2 Fitness calculation

This chapter describes the routine which calculates the fitness values of a single individual.

Code fragment: calcFitness

```
//calculate cost
loop for (all genes)                    Loop over all genes of a individual.
     cost = cost + cost[gene]           Add the cost which is saved per spot in the
                                        data structure cost[] at initialization time
end for loop

ind.saveSpotCount                       Calculate the number of booked spots and
                                        save it

//calculate net reach
loop for (all persons)
     temp_nrw=1                         set temporary variable temp_nrw to 1
     temp_weight=0                      set temporary variable temp_weight to 0
      loop for (all genes)
          if (gene == 1)
              temp_weight=temp_weight                    +
                   weight[person][spot_date[gene]]
               temp_nrw = temp_nrw * data[person][gene]
          end if
     end for loop
     ind.person_nrw[person] = temp_nrw
     ind.person_nrw_weight[person] = temp_weight
     temp_nrw = (temp_weight/spot_count) * (1 – temp_nrw)
     nrw = nrw + temp_nrw
end for loop
ind.cost = cost                         saves the calculated cost in the
                                        individual data structure
ind.nrw = nrw                           saves the calculated net reach in
                                        the individual data structure
```

Explanation

The first part of this routine calculates the total cost of a campaign. This is straightforward as the cost of each spot is saved in a data structure at initialization time and they are simply summed up to determine the total cost of a campaign. Also the number of booked spots is calculated and saved.

The second part computes the net reach. The calculation of the net reach is explained in detail in the chapters 2.2 0-1 Knapsack Problem with Combinatorial Value Function (Description of the abstract problem) and 5.4.1 How are net reach and gross reach measured? (calculation in practice).

The basic formula for the net reach per person is $1 - [(1 - \mathrm{p1}) * (1 - \mathrm{p2}) * ... * (1 - \mathrm{pn})]$ multiplied with the cumulative weight per person per spot date divided by the number of spots. The sum of this value of all persons is the final net reach for the campaign and is saved in the individual data structure.

The algorithm above first calculates the probability of one person to have seen the spot (in a loop over all persons). The p-values are stored in the data structure data[person][spot] at initialization time as (1 - p-value). These values are multiplied for each spot of the campaign that is booked (each gene with a value of “1”) and stored in the variable temp_nrw. After all genes are processed for one person the value is stored in the variable person_nrw in the individual data structure. Also the weight of each person is calculated in this loop and saved in the variable nrw_weight[person] in the individual data structure. These values are saved, because they are later used by the mutation routine for a high performance fitness calculation. The fitness values of the individual is not calculated newly after a mutation, but instead the changes to the existing fitness values are calculated (see chapter 6.4.6 Mutation) using these temporary variables.

6.4.3 Non dominated sort

This chapter describes the routine used to sort a population by non-domination (see chapter 4.7.1 Non-dominated Sort). It is divided into the non-dominated sort itself and the crowding distance assignment.

6.4.4 Non dominated sort main routine

The following code fragment describes the main sort routine for the search of non-dominated pareto fronts (see chapter 4.7.1 Non-dominated Sort). It works with the combined parent and

mutated child population and returns the best half of the individuals. It stops sorting after half the individuals are sorted.

Code fragment: nonDominatedSort

```
loop while (while there are less than half the individuals
ranked)
     put first free individual in the current front array
     loop for (i=1; i < pop_size*2; i++)
          add next free element to array
          loop for (elements in array)
               compare individuals in current front with new
               individual
               if (new element dominates array-element)
                    remove array-element
                    add new element to array
               if (array element dominates new element)
                    remove new element from array
               else (no domination)
                    do nothing
          end for loop
     end for loop

     loop for(elements in array)
          assignRank(element, number of pareto front)
     end loop

     if (sizeOf(array) < parentPop.getFreeSpace)
          parentPop.addArrayElements(array)
     else
          assignCrowdingDistance(array)
     end if

end do loop
```

Explanation

The routine starts by putting the first free (not yet sorted individual) in an array that represents the current pareto front. Then it loops over all individuals in the mixed population (which are not yet part of a pareto front), adds them temporarily to the array and compares them with all the elements already in it. There are three possible outcomes:

- If the newly added individual dominates an element in the array the dominated individual is removed from the array and the new individual stays in the array.
- If the newly added individual is dominated by an element in the array it is removed from the array.
- If there is no domination the new element stays in the array as well.

At the end of the loop the array contains a complete pareto front of non-dominated individuals. If there are less than half the elements sorted it is repeated to calculate another pareto front until there are as least half the elements sorted.
If more than exactly half the individuals are sorted in pareto fronts the crowding distance is calculated for the individuals in the last pareto front to achieve as good a diversity as possible.

6.4.5 Diversity preservation (crowding distance)

This chapter describes the calculation of the crowding distance and the following assignment to the target population for the individuals with the highest crowding distance (see chapter 4.7.2 Diversity Preservation). This is done for the last pareto front resulting from the non-dominated sort if it doesn't fit in the result population.

Code fragment: assignCrowdingDistance

```
crowding_array = new array
```

Add all elements to a new array together with their net reach and cost

```
loop for (elements in pareto front array)
     crowding_array.add(element)
     crowding_array.element.saveNrw(element)
     crowding_array.element.saveCost(element)
end for loop

//nrw
```

```
crowding_array.sort(nrw)                    Sort the array by net reach
crowding_array[0].crowd_dist = infinity     Assign crowding distance
                                            of infinity to first and last
                                            element in array
crowding_array[last].crowd_dist = infinity
loop for (rest of individuals in crowding_array)
     ind.crowding = ind.crowding + (crowding_array[next].nrw -
          crowding_array[prev].nrw)
end for loop

//cost
crowding_array.sort(cost)
crowding_array[0].crowd_dist = infinity     Assign crowding distance
                                            of infinity to first and last
                                            element in array
crowding_array[last].crowd_dist = infinity
loop for (rest of individuals in crowding_array)
     ind.crowding = ind.crowding + (crowding_array[next].cost -
          crowding_array[prev].cost)
end for loop

//sort array by crowding distance
crowding_array.sort(crowding distance)
loop while (parentPop.size < population_size)
     copyInd(crowding_array.element, parentPop)
end while loop
```

Explanation

An array is created which holds the objective values of all the individuals in the pareto front (net reach and cost).

The array is then sorted by net reach. The first and the last element in the array get assigned a crowding distance of infinity as they add the highest diversity possible and are therefore the most desirable for the result population (in absence of dominating individuals).

For the rest of the individuals the crowding distance is calculated as the difference in net reach between the following and the previous element in the array. This value is saved for all the individuals.
Next the array is sorted by cost.
Again the first and the last element in the array get a crowding distance of infinity assigned.
For the rest of the elements the crowding distance is calculated analogous to the first objective as the difference in cost between the next element in the array and the previous one. This value is added to the previously calculated crowding distance and saved.
Now all elements have the cumulative crowding distance of both objectives assigned. The array is sorted by crowding distance and the elements with the highest crowding distance are moved to the new parent population until it is full.

6.4.6 Mutation

This chapter describes how the routine for mutating an individual works. This routine is called for all individuals of a newly created child population when the mutation takes place.
Following this code fragment there is another one showing the adaptation of the fitness value after mutation.

6.4.6.1 Main mutation routine

This routine flips genes (TV spots) randomly according to the mutation probability parameter.

Code fragment: mutatePop

`loop for (all individuals)`	loop over all individuals (TV campaigns) of the population
`loop for (all genes)`	loop over all genes of the population (genes represent the TV spots of a campaign)
`if (random() < MUTATION_PROB)`	Generates a random number between 0 and 1 and checks if it is smaller than the mutation probability.
`flipSpot(gene)`	If it is, the current spot is flipped, that is if it was part of the

```
                                    campaign before it will be
                                    removed and if it was not part
                                    before, it will be added. (0 is set to
                                    1, 1 is set to 0).
            UpdateFitness(ind)      The fitness values of the individual
                                    is updated.
        end if
    end for loop
end for loop
```

Explanation

The mutation routine is fairly simple. Based upon the mutation probability the individual genes (TV spots) are flipped from booked to not booked and vice versa. After flipping a gene the fitness values of the individual are re-calculated. This turns out to be multiple times faster than recalculating the fitness values after the mutation process is finished.

6.4.6.2 Updating of fitness values during the mutation process

This part of the application was created during the optimization phase, when it became apparent that the fitness calculation and especially the calculation of the net reach value took up the major part of CPU time. To improve the performance temporary variables were introduced which store the net reach value for each person in the individual data structure. It turned out to be multiple times faster to recalculate this value for each gene-flipping (based only on the one gene that is changing) than to calculate the net reach from scratch after the mutation process.

There is one routine for adding a gene (change from 0 to 1) and one routine for removing a gene (change from 1 to 0). The routines take the individual data structure and the gene as input parameters. The following code fragment describes the routine for adding a gene, the routine for removing a gene works analogous.

Code fragment: updateFitness, add gene

```
nrw = 0
ind.spotcount += 1
loop for (all persons)
```

```
    ind.person_nrw[person] *= data[person][gene]
    ind.person_nrw_weight[person]                    +=
        weight[i][spot_date[gene]];
    temp_nrw   =   (   ind.person_nrw_weight[person]   /
        ind.spotcount ) * ( 1 – ind.person_nrw[person])
    nrw += temp_nrw
end for loop
ind.nrw = nrw
ind.cost += cost[gene]
```

Explanation

The routine loops over all persons and multiplies the net reach value for the person (saved by the initial fitness-calculation described in chapter 6.4.2 Fitness calculation) with the (1 – p-value) of the gene that is added. Then it adapts the weight of the person. The new net reach values and weights of all persons are saved back to the individual data structure for future fitness updates and the final net reach value and cost value are updated as well.

6.5 Data Output

All relevant data from an optimization run is written to a database, so the results can be examined and compared to real life campaigns.

The general information of a simulation run contains the following information: Run time (in CPU clock cycles), number of generations, mutation probability, population size, CPU clock cycles per second (so the run time in seconds can be calculated) and the date and time of the run.

For every individual in the final parent population the following data is saved: net reach, cost, gross reach, rank and crowding distance. This data represents the quality of the results of the simulation and is later used to calculate the hypervolume indicator to compare different results.

As a last step all the spots of an individual are saved to the database, they are the actual result. They are used to verify the results by the media agency OMD with their regular tools.

Optionally the application can write performance-debugging-information to the console (see next chapter).

6.6 Performance optimization

I used pre-processing directives to conditionally compile the program with performance-information only when the debug-parameter is set. If the code is compiled with the debug parameter not set, there is no performance-code in the final executable at all, so there is no effect on performance.

I saved the following information for performance debugging:

- Time spent in each procedure, or part of procedure for critical parts.
- Number of additions, subtractions, multiplications, divisions, generation of random numbers and array index accesses. This information is stored per procedure.

I used this information to find out which parts of the algorithm took the most time and was able to make substantial performance improvements by reviewing the relevant sections and by using the data gathered by the debug program unit.

The debugging framework allowed me to quickly assess performance changes after small program changes (like the usage of different data types or changes in the algorithm). This was a very important part of creating the final application and a surprisingly large amount of time was spent trying different algorithms and data structures.

7 Test

This chapter shows the results of the presented application.
It is divided into five parts:

- First I lay out a test methodology. Here I define a way to do simulation runs in such a way that I get enough results so I can draw meaningful conclusions out of them.
- Then I show how to compare them with each other (every run results in a whole population of results and these results might not be clearly better or worse than others (i.e. no domination)).
- Next I will show the actual results in a graphical way and discuss them.
- I go on to show the effects of rising run time on the performance
- In the last part I will compare the quality of the optimizer results with real life campaigns.

7.1 Test Methodology

7.1.1 Goals

The systematic test of the optimizer has two goals:

1) Quantitative Comparison of the performance of different configurations of the genetic algorithm
2) Quantitative Comparison of the results with „real life“ TV campaigns.

There are many ways to measure the value of a TV campaign (cost, net reach, GRPs, see chapter 5.4.1) but I focus on the two objectives that I am optimizing: net reach and cost.

7.1.2 Configuration of the Simulation Runs

The presented application takes three parameters: Population size, mutation probability and allowed runtime for one simulation run. As of this day, there is no known way to find optimal (or even good) parameter settings for genetic algorithms other than by trial and error. To find the best setting for this particular implementation I first try a broad range of different configurations. After reviewing the results I do some further finer grained testing in those areas that delivered the best results in the first test.

For each configuration I do 10 simulation runs. To compare the configurations I take the average of the 10 runs and compare their Hypervolume Indicator HI (see chapter 7.2.1 Basic Overview of the Hypervolume Indicator).
Every simulation run has the same amount of time to finish. Time is checked at the beginning of each generation and it includes the time needed for initialization (which is higher for bigger populations).
Time is not measured in seconds but in CPU clock cycles for two reasons:

- It eliminates differences from the amount of CPU-power the machine is giving to the program which could change because of screen-savers, virus-checks and other things.
- It allows running simulations on different machines and comparing the results (the simulation runs are very time intensive).

A simple batch-file does the program-calls with the desired configuration (mutation probability and population size), calling each configuration a total of ten times.

7.2 Analysis

To compare different populations with each other I do three things:
First I scale the two objectives to the interval $[0 \dots 1] \in R$
To scale the costs I will take the cost of all spots in the given time period as 1.
Reach is already expressed as a percentage and thus is in the interval [0 … 1].
To compare the performance of simulation runs I need both objectives to have the same goal (minimization or maximization). To this end I invert the reach-objective, so that both objectives are to be minimized.
Finally I quantitatively measure the resulting pareto-fronts of each optimization run. I use the well known Hypervolume Indicator first used by Zitzler [Zitzler 1999].

7.2.1 Basic Overview of the Hypervolume Indicator

The Hypervolume Indicator (HI) was introduced by Zitzler to allow a comparison of the performance of multi-objective optimization algorithms or to be exact a comparison of pareto-optimal fronts.
The Hypervolume is an indicator number that can be computed for a pareto-optimal front with two or more objectives. If all the objectives are maximized it is the area or volume (depending on the number of objectives) comprised by the pareto front and the axes of the coordinate system.

If the objectives are minimized, a worst-case point is taken and the Hypervolume Indicator is calculated as the area (or volume) between the worst case point and the pareto front.
The higher the value of the hypervolume indicator, the better the quality of the solution front.
The following graphic illustrates the use of the hypervolume indicator on the example of two pareto fronts and two maximization objectives:

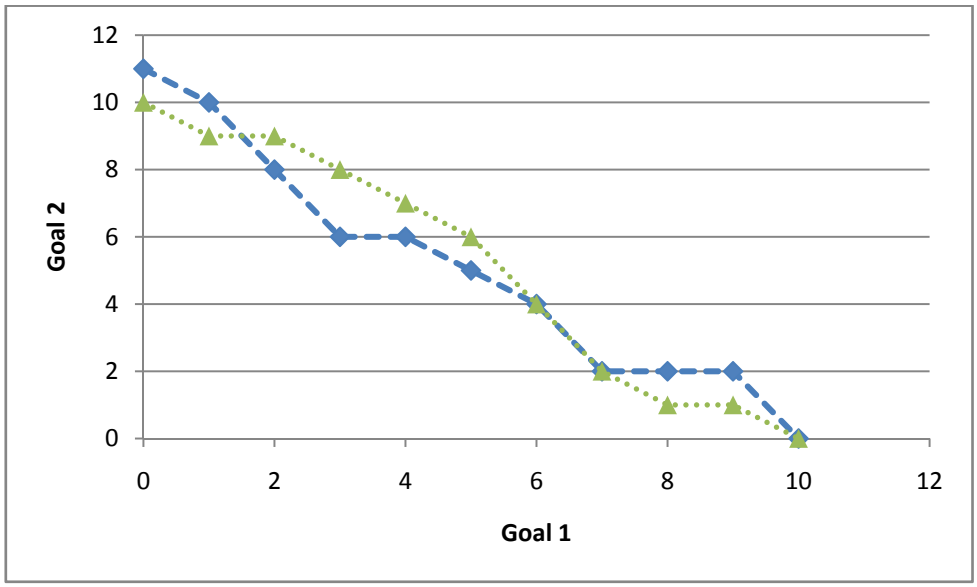

Figure 12 Two overlapping pareto fronts

In the following figure the hypervolume indicator of one of the fronts is illustrated.

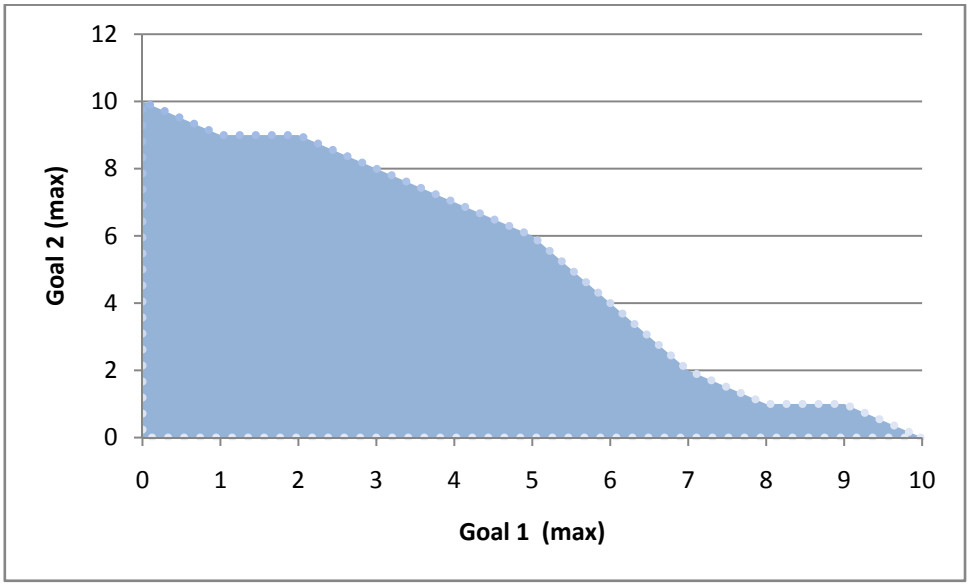

Figure 13: Pareto front with Hypervolume indicator (marked area)

As you can see the two pareto fronts are overlapping so there is no immediate way to tell which is the better one. The hypervolume is the area comprised by a pareto front and the axes. This can be seen as all solutions that are dominated by the solutions in the pareto front. The bigger this area, the better is the quality of the pareto front.

7.2.2 Application of the hypervolume indicator

The presented problem has two goals:

1) Minimize cost
2) Maximize reach

To use the hypervolume indicator, I need to have two minimization goals. Therefore I invert the reach goal. Reach is a percentage between 0 and 1, so I just calculate (1 – reach).
I divide the cost of the campaign by the cost of all spots, so it is in the range of 0 – 1 as well.
Finally I calculate the hypervolume indicator for a pareto front using the following algorithm:

```
x_hyper := 0.0;
x_temp  := 1.0;
            FOR rec_ind IN
                  SELECT 1-(nrw/100.0) as r_nrw, cost/2575767.0 as r_cost
                   FROM opti_ind_info
                   WHERE run_id=p_run_id
                   ORDER BY r_nrw DESC
            LOOP
                  x_hyper := x_hyper + ((1 - rec_ind.r_cost) * (x_temp - rec_ind.r_nrw));
                  x_temp := rec_ind.r_nrw;
            END LOOP;
```

7.3 Example of a pareto front

The following scatter diagram shows the final pareto front from one of the simulation runs. The following parameters were used:
Mutation probability: 0.0004
Population size: 500
Clock cycles until termination: 250.000

In the following figure net reach is plotted on the horizontal axis and cost is plotted on the vertical axis.

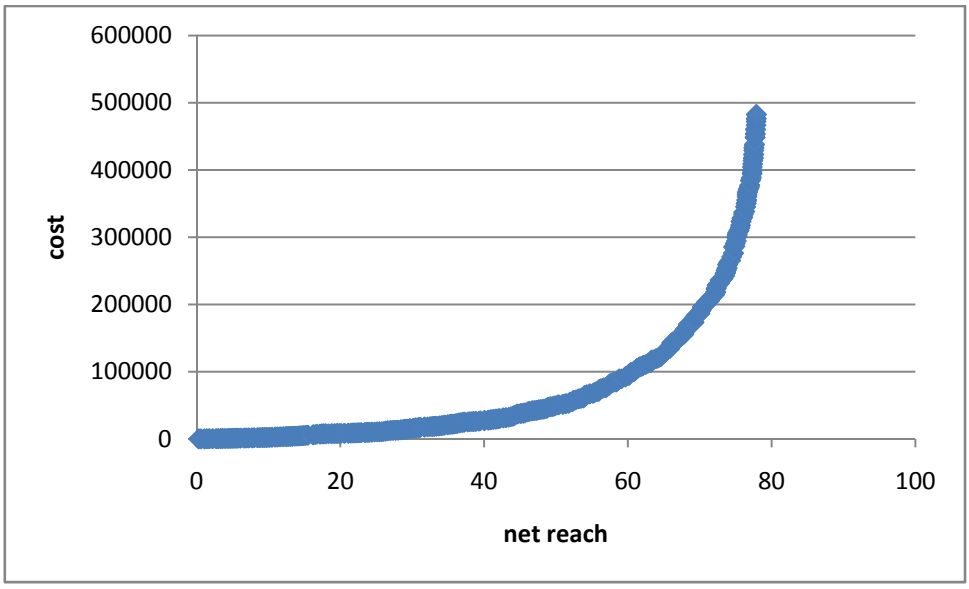

Figure 14 Example of pareto front

For the calculation of the hypervolume indicator the net reach objective is changed to a minimization objective (1 – net reach). This is shown on the following figure.

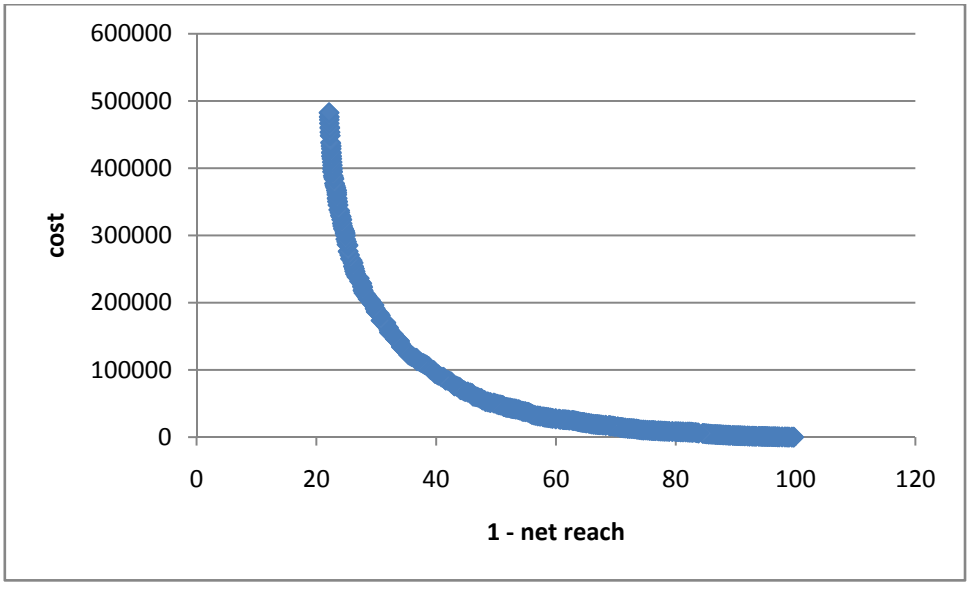

Figure 15: Example of a pareto front (2)

It can be seen that the diversity preservation of the used algorithm works very well: The line is very smooth and goes from one extreme value (zero cost and zero reach) to the other extreme (about € 500,000 cost and 80% reach). The cost of € 500.000 is not actually the

extreme value (the total cost of all spots in the time frame is € 2,575,767), but the net reach does not rise significantly beyond that point.

For the computation of the hypervolume indicator both objectives are scaled to the same interval [0 .. 1]. The following graphic shows the pareto front with scaled objective values.

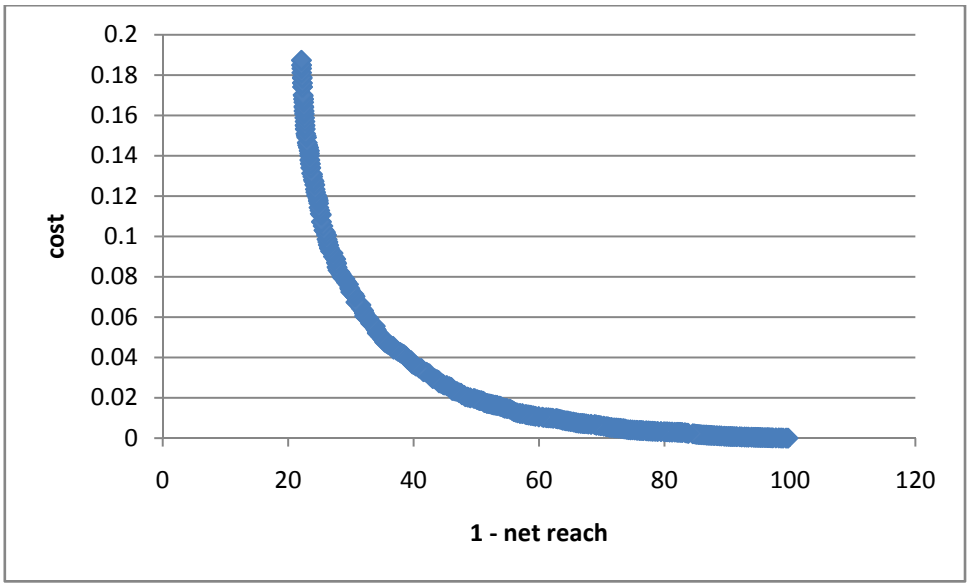

Figure 16: Example of pareto front with scaled objective values

7.4 Results of different parameter configurations

A test case for the simulation runs a typical campaign goal from the agency OMD was used.

- The timeframe of the campaign is one week (09/04/2006 – 09/10/2006).
- The target group is women aged between 18 and 49 years (a standard target group)
- The number of TV spots in this time frame is 2,449.
- The number of persons who are part of the target group is 628

The number of TV spots and persons in this time frame is actually higher, but all persons who did not fall into the target group and all spots that were not watched by any person in the target group were removed before the test runs.

This campaign was actually planned by OMD. In chapter 7.6 “Comparison with real life campaigns” I compare cost, net reach and GRPs from four campaigns that were planned by OMD with the results of the simulation runs.

For the source of the data see chapter 5.5 “What Data Do I Use?”.

This chapter shows the results of the simulation runs using different parameter settings. For each parameter setting ten runs were done, and the average Hypervolume Indicator of these is reported in the table. The ten best results of each run are marked red in the table.
A short note regarding the clock-cycles before termination: 250.000 clock-cycles are used in all the runs, this equals about five minutes on a Pentium dual core 2GHz with 1GB RAM (where one CPU is used completely by the simulation program and the other one not at all).
I used 10 different population sizes and 10 different mutation probabilities per run. This equals 100 combinations, each of which is run 10 times. So the total time for one run was about 100*10*5 minutes which is about 83 hours or 3.5 days.

7.4.1 Test Run 1

This run used the following parameters:

Clock cycles until termination: 250.000

Mutation probabilities: 10, 20, 30, 40, 50, 60, 70, 80, 90, 100

Population size: 10, 25, 50, 75, 100, 125, 150, 175, 200, 225, 250

The following table lists the results which were achieved in ten runs:

	10	20	30	40	50	60	70	80	90	100
10	0.586	0.570	0.567	0.575	0.550	0.554	0.516	0.518	0.518	0.490
25	0.514	0.495	0.512	0.566	0.562	0.537	0.548	0.531	0.555	0.549
50	0.542	0.573	0.549	0.543	0.546	0.544	0.559	0.543	0.558	0.532
75	0.573	0.548	0.554	0.543	0.541	0.545	0.545	0.536	0.541	0.557
100	0.546	0.548	0.565	0.549	0.552	0.541	0.541	0.541	0.535	0.533
125	0.556	0.571	0.548	0.569	0.570	0.547	0.547	0.550	0.539	0.546
150	0.555	0.548	0.466	0.466	0.512	0.468	0.454	0.530	0.559	0.521
175	0.563	0.497	0.500	0.479	0.523	0.472	0.495	0.478	0.480	0.431
200	0.445	0.458	0.467	0.450	0.475	0.450	0.429	0.458	0.452	0.436
225	0.434	0.432	0.436	0.447	0.441	0.438	0.446	0.458	0.440	0.447
250	0.420	0.419	0.413	0.419	0.427	0.429	0.475	0.568	0.565	0.562

Table 3: Results of test run 1

The following illustration shows the results graphically:

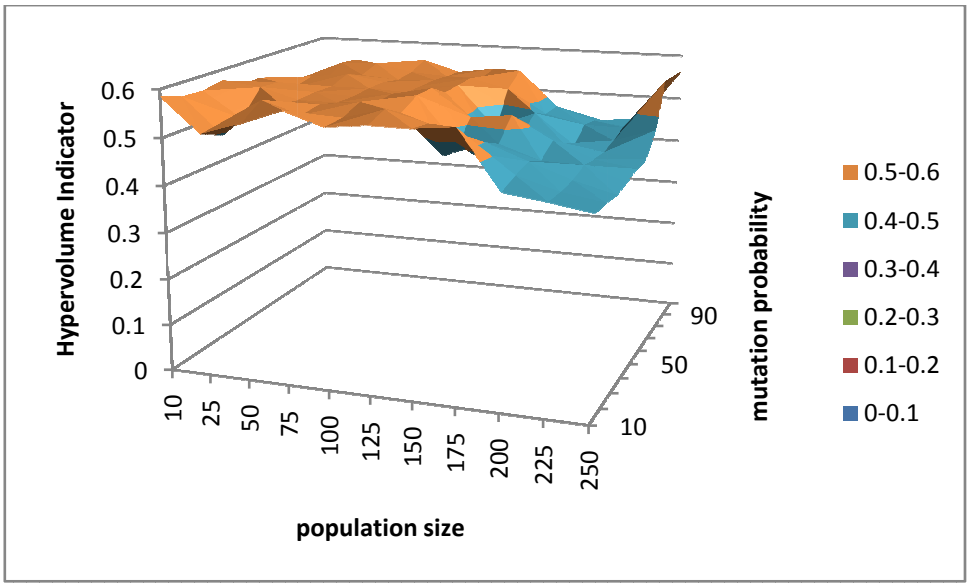

Figure 17: Results of test run 1

Mutation probabilities of 50% or lower seem to yield better results than higher mutation probabilities.

7.4.2 Test Run 2

This run used the following parameters:

Clock cycles until termination: 250.000

Mutation probabilities: 2.5, 5, 7.5, 10, 12.5, 15, 17.5, 20, 22.5, 25

Population size: 1, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100

The following table lists the results which were achieved in ten runs:

	2.5	5	7.5	10	12.5	15	17.5	20	22.5	25
1	0.123	0.092	0.087	0.106	0.114	0.112	0.102	0.126	0.103	0.114
10	0.665	0.591	0.528	0.513	0.538	0.519	0.489	0.493	0.481	0.468
20	0.522	0.505	0.492	0.464	0.483	0.482	0.467	0.463	0.460	0.513
30	0.525	0.455	0.473	0.479	0.475	0.494	0.475	0.514	0.525	0.474
40	0.483	0.537	0.496	0.488	0.461	0.433	0.415	0.372	0.420	0.384
50	0.444	0.401	0.426	0.396	0.398	0.365	0.397	0.375	0.390	0.390
60	0.402	0.404	0.396	0.418	0.398	0.392	0.395	0.391	0.376	0.396
70	0.451	0.387	0.408	0.390	0.428	0.425	0.381	0.401	0.398	0.384
80	0.389	0.377	0.413	0.378	0.449	0.458	0.456	0.458	0.437	0.457
90	0.431	0.439	0.459	0.438	0.431	0.397	0.383	0.389	0.366	0.374
100	0.408	0.371	0.383	0.365	0.370	0.362	0.372	0.530	0.534	0.607

Table 4: Results of test run 2

The following illustration shows the results graphically:

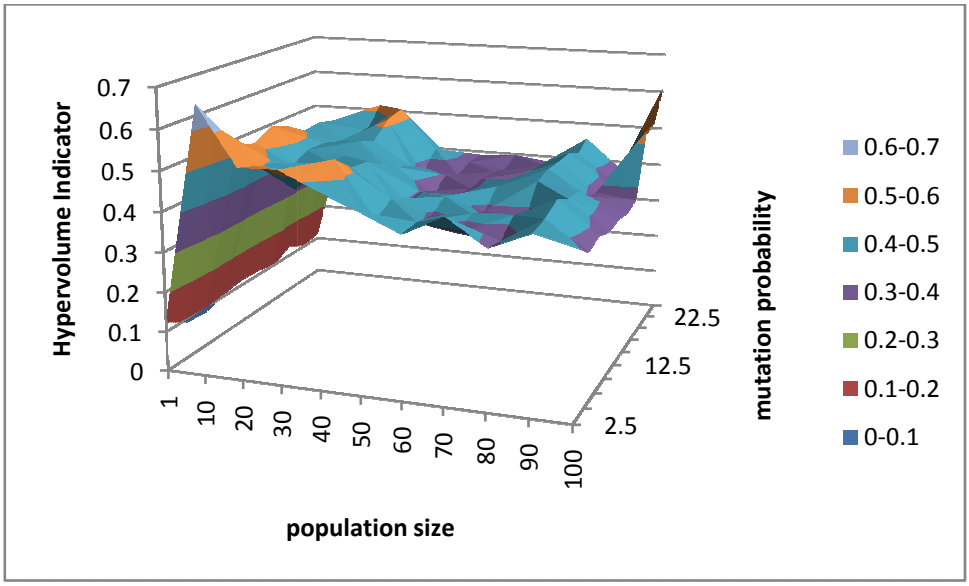

Figure 18: Results of test run 2

Mutation probabilities of around 25% and lower than 12.5% seem to yield slightly higher than average results.

7.4.3 Test Run 3

This run used the following parameters:

Clock cycles until termination: 250.000

Mutation probabilities: 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5

Population size: 10, 20, 30, 40, 50, 60, 70, 80, 90, 100

The following table lists the results which were achieved in ten runs:

	0.5	1	1.5	2	2.5	3	3.5	4	4.5	5
1	0.143	0.175	0.150	0.107	0.151	0.096	0.108	0.114	0.113	0.114
10	0.653	0.634	0.673	0.687	0.693	0.692	0.694	0.695	0.696	0.694
20	0.659	0.645	0.677	0.698	0.699	0.702	0.697	0.706	0.707	0.708
30	0.664	0.646	0.679	0.701	0.702	0.704	0.706	0.710	0.710	0.711
40	0.676	0.656	0.683	0.700	0.705	0.706	0.708	0.710	0.713	0.712
50	0.675	0.653	0.683	0.700	0.706	0.706	0.709	0.710	0.712	0.713
60	0.673	0.655	0.685	0.701	0.707	0.707	0.709	0.712	0.713	0.713
70	0.673	0.656	0.682	0.704	0.707	0.708	0.708	0.711	0.712	0.714
80	0.676	0.657	0.682	0.702	0.707	0.708	0.709	0.710	0.712	0.714
90	0.676	0.655	0.684	0.702	0.705	0.706	0.709	0.712	0.713	0.713
100	0.674	0.658	0.681	0.704	0.706	0.708	0.708	0.712	0.712	0.713

Table 5: Results of test run 3

The following illustration shows the results graphically:

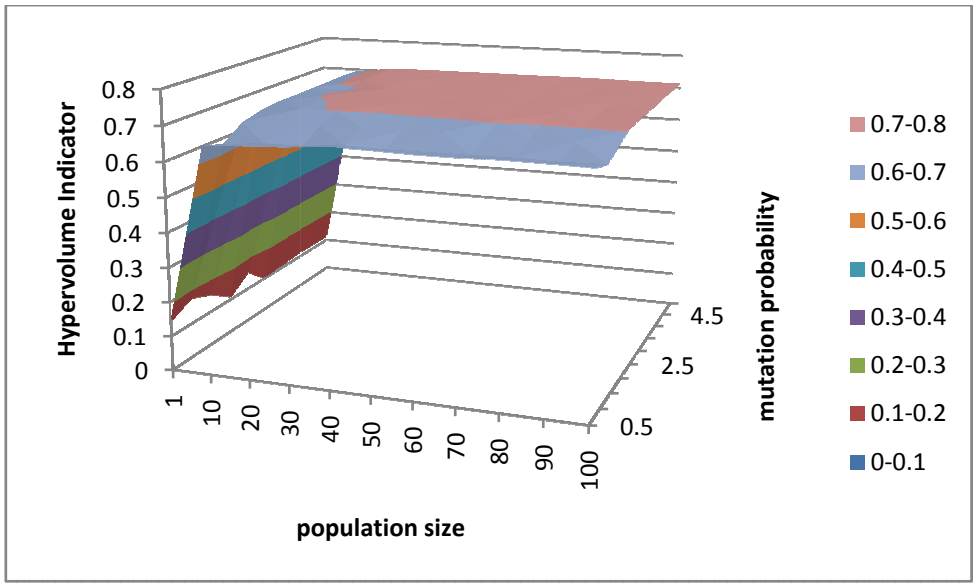

Figure 19: Results of test run 3

Mutation probabilities around 5% have the best results so far. Mutation probabilities lower than that also have good results.

7.4.4 Test Run 4

This run used the following parameters:

Clock cycles until termination: 250.000

Mutation probabilities: 0.5, 1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5

Population size: 500, 1000, 1500, 2000, 2500, 3000, 3500, 4000, 4500, 5000

The following table lists the results which were achieved in ten runs:

	0.5	1	1.5	2	2.5	3	3.5	4	4.5	5
500	0.582	0.579	0.316	0.234	0.254	0.239	0.247	0.313	0.278	0.320
1000	0.532	0.467	0.309	0.314	0.329	0.345	0.306	0.311	0.331	0.319
1500	0.361	0.324	0.341	0.285	0.309	0.357	0.325	0.303	0.333	0.303
2000	0.347	0.335	0.308	0.311	0.309	0.322	0.307	0.285	0.177	0.206
2500	0.334	0.188	0.168	0.117	0.105	0.080	0.101	0.096	0.094	0.043
3000	0.247	0.136	0.099	0.101	0.088	0.117	0.073	0.079	0.155	0.113
3500	0.138	0.113	0.104	0.146	0.081	0.093	0.083	0.067	0.131	0.129
4000	0.114	0.118	0.144	0.137	0.112	0.124	0.142	0.098	0.074	0.132
4500	0.123	0.127	0.100	0.098	0.095	0.128	0.128	0.135	0.133	0.055
5000	0.073	0.029	0.054	0.068	0.212	0.269	0.193	0.247	0.179	0.190

Table 6: Results of test run 4

The following illustration shows the results graphically:

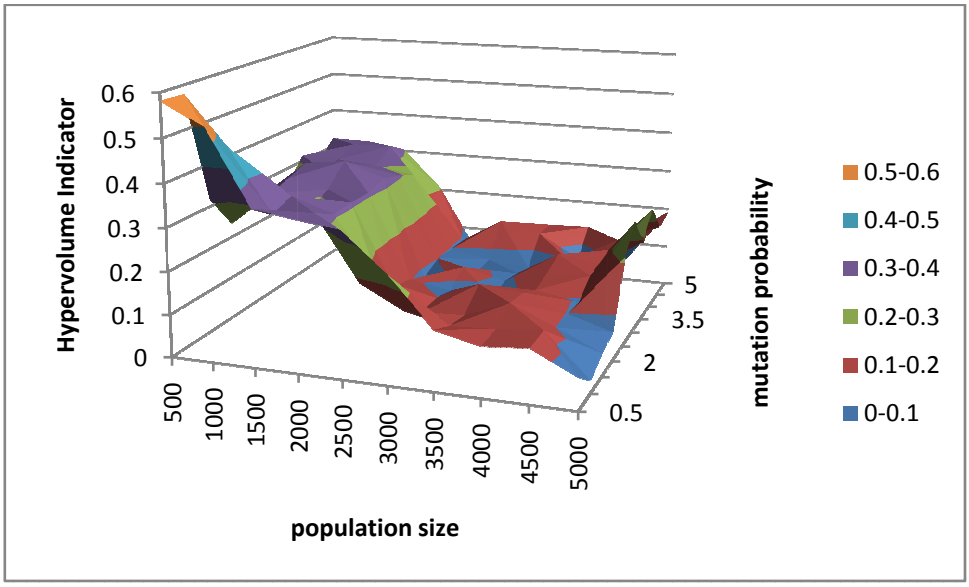

Figure 20: Results of test run 4

The lower mutation probabilities of this test run yield the best results. Population sizes over 2000 don't have good results, apparently the number of generations gets too small with the given time.

7.4.5 Test Run 5

This run used the following parameters:

Clock cycles until termination: 250.000

Mutation probabilities: 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0

Population size: 10, 25, 50, 75, 100, 125, 150, 175, 200, 225, 250

The following table lists the results which were achieved in ten runs:

	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
10	0.702	0.679	0.667	0.658	0.652	0.645	0.641	0.640	0.641	0.629
25	0.716	0.690	0.679	0.669	0.666	0.663	0.656	0.653	0.654	0.645
50	0.716	0.697	0.682	0.676	0.672	0.668	0.660	0.662	0.656	0.659
75	0.720	0.697	0.684	0.679	0.675	0.671	0.665	0.662	0.662	0.656
100	0.721	0.699	0.691	0.676	0.668	0.672	0.665	0.664	0.664	0.655
125	0.717	0.699	0.683	0.678	0.676	0.671	0.668	0.665	0.665	0.661
150	0.719	0.696	0.686	0.677	0.673	0.669	0.669	0.663	0.662	0.663
175	0.719	0.695	0.686	0.677	0.671	0.670	0.665	0.667	0.668	0.666
200	0.715	0.696	0.687	0.680	0.674	0.670	0.665	0.669	0.666	0.661
225	0.716	0.696	0.686	0.677	0.674	0.670	0.668	0.668	0.665	0.666
250	0.715	0.693	0.684	0.679	0.678	0.670	0.670	0.667	0.665	0.665

Table 7: Results of test run 5

The following illustration shows the results graphically:

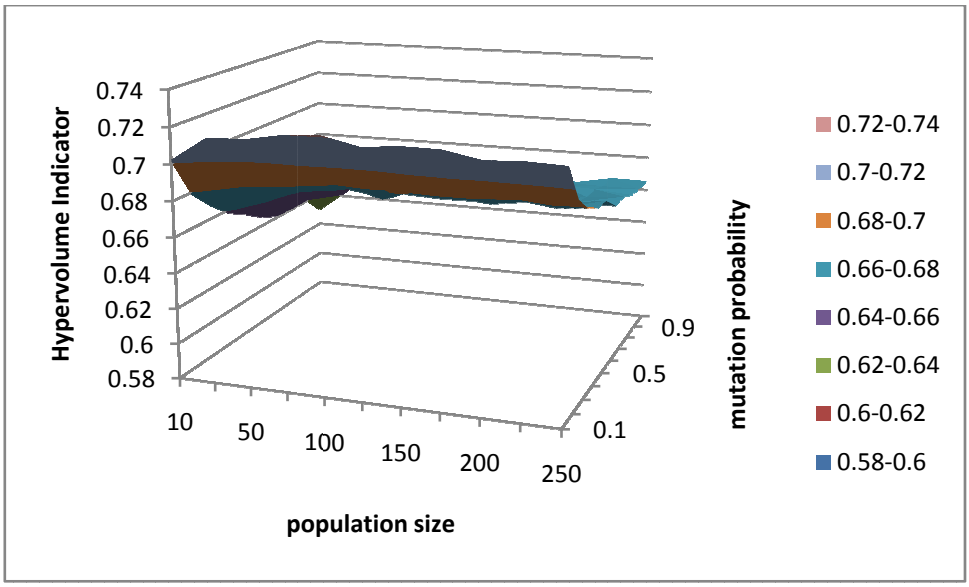

Figure 21: Results of test run 5

A mutation probability of 0.1% yields the best results so far. Even lower probabilities seem promising. Population size is not a big factor in the results.

7.4.6 Test Run 6

This run used the following parameters:

Clock cycles until termination: 250.000

Mutation probabilities: 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5

Population size: 10, 25, 50, 75, 100, 125, 150, 175, 200, 225, 250

The following table lists the results which were achieved in ten runs:

	0.05	0.1	0.15	0.2	0.25	0.3	0.35	0.4	0.45	0.5
25	0.732	0.717	0.704	0.694	0.687	0.681	0.679	0.676	0.672	0.667
50	0.739	0.722	0.709	0.697	0.693	0.688	0.680	0.679	0.674	0.669
75	0.742	0.722	0.709	0.699	0.691	0.685	0.682	0.679	0.676	0.672
100	0.742	0.723	0.709	0.702	0.694	0.689	0.686	0.679	0.677	0.676
125	0.741	0.722	0.710	0.701	0.693	0.684	0.682	0.679	0.678	0.673
150	0.741	0.720	0.708	0.700	0.692	0.690	0.687	0.679	0.679	0.677
175	0.743	0.719	0.708	0.697	0.693	0.689	0.684	0.679	0.678	0.675
200	0.741	0.719	0.707	0.696	0.691	0.688	0.684	0.680	0.676	0.677
225	0.740	0.717	0.707	0.699	0.693	0.686	0.683	0.680	0.680	0.679
250	0.741	0.718	0.705	0.697	0.692	0.686	0.684	0.680	0.679	0.673

Table 8: Results of test run 6

The following illustration shows the results graphically:

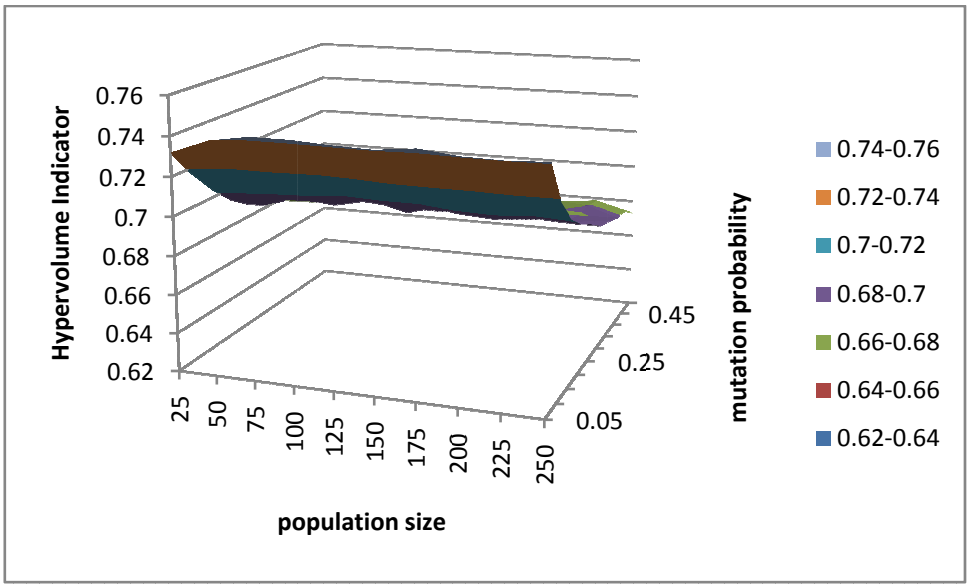

Figure 22: Results of test run 6

This simulation run shows a clear tendency of better results with lower mutation probabilities. Population size again is not a big influence.

7.4.7 Test Run 7

This run used the following parameters:

Clock cycles until termination: 250.000

Mutation probabilities: 0.0004, 0.0008, 0.0012, 0.0016, 0.002, 0.0024, 0.0028, 0.0032, 0.0036, 0.040

Population size: 2, 10, 80, 90, 100, 110, 500, 1000, 1500, 2000

The following table lists the results which were achieved in ten runs:

	0.0004	0.0008	0.0012	0.0016	0.002	0.0024	0.0028	0.0032	0.0036	0.04
2	0.507	0.504	0.541	0.505	0.517	0.518	0.516	0.515	0.548	0.546
10	0.712	0.710	0.711	0.699	0.667	0.698	0.710	0.712	0.709	0.713
80	0.745	0.734	0.698	0.742	0.747	0.748	0.748	0.748	0.749	0.735
90	0.678	0.641	0.689	0.721	0.730	0.722	0.656	0.691	0.749	0.687
100	0.643	0.713	0.681	0.733	0.700	0.654	0.655	0.748	0.679	0.682
110	0.739	0.706	0.738	0.719	0.716	0.681	0.727	0.705	0.706	0.740
500	0.661	0.711	0.693	0.730	0.701	0.706	0.708	0.709	0.731	0.708
1000	0.751	0.750	0.680	0.682	0.731	0.689	0.695	0.745	0.739	0.700
1500	0.685	0.683	0.715	0.678	0.683	0.731	0.689	0.731	0.739	0.729
2000	0.673	0.672	0.590	0.702	0.663	0.627	0.716	0.713	0.708	0.712

Table 9: Results of test run 7

The following illustration shows the results graphically:

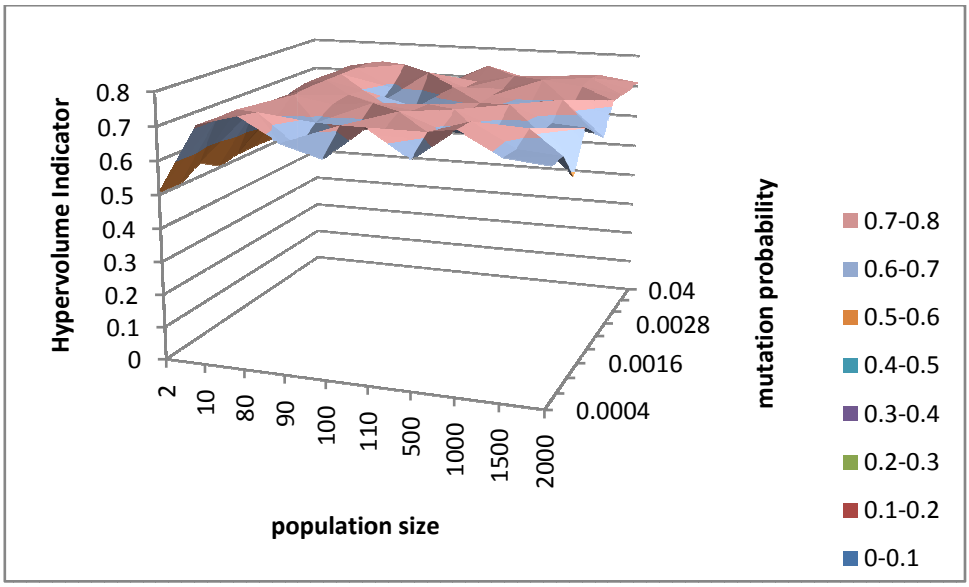

Figure 23: Results of test run 7

A plateau in the results seems to be reached. The mutation probability of 0.0004 is the smallest possible for the given data set (there are about 2,500 spots possible and one out of 2,500 equals about 0.0004). Mutation probabilities of 0.05 and lower seem to have equally good results. Population sizes of 80 and of 1,000 also seem to yield slightly better results than the other ones used.

7.4.8 Conclusion from the test runs

A summary of the results of the test runs is displayed in the following table.

Test Run	Mutation probability	Population size	Best Hypervolume Indicator
1	10 – 100	10 – 250	0.586
2	2.5 – 25	1 – 100	0.665
3	0.5 – 5	10 – 100	0.714
4	0.5 – 5	500 – 5000	0.582
5	0.1 – 1	10 – 250	0.721
6	0.05 – 0.5	10 – 250	0.743
7	0.0004 – 0.04	2 – 2000	0.751

Table 10: Summary of test run results

A very small mutation probability near the minimum (one spot out of all possible spots per mutation on average) seems to yield the best results. The population size is not a very big factor; although there are certain population sizes which give slightly better results (80 and 1,000 had the best results). Population sizes bigger than 1,500 don’t produce good results with the given run time of five minutes.

These parameters will be used in the following chapter to test different run times and their impact on the quality of the results.

7.5 Test of different run times

After the optimal parameter configuration was found, different run times were tested using these parameters. The following run times were tested:

Clock cycles	Approximate run time
8,333	10 seconds
25,000	30 seconds
50,000	1 minute
250,000	5 minutes
500,000	10 minutes
1,000,000	20 minutes
1,500,000	30 minutes
3,000,000	1 hour
6,000,000	2 hours
12,000,000	4 hours
24,000,000	8 hours

Table 11: Evaluated run times

The mutation probability was set to 0.0004

The population size was set to 80 or 1.000, depending on the run time. A population size of 1,000 does not produce good results with a run time of less than five minutes, but better results than 80 at higher run times.

The following data is reported for each run time:

- Number of generations of each run
- Hypervolume Indicator of each run
- Average number of generations
- Average Hypervolume Indicator
- A scatter graph of the results of one of the runs

7.5.1 Ten seconds

The following table lists the results which were achieved in ten runs:

generations	population size	mutation probability	HI
283	80	0.0004	0.664
281	80	0.0004	0.676
280	80	0.0004	0.666
275	80	0.0004	0.669
271	80	0.0004	0.669
261	80	0.0004	0.666
267	80	0.0004	0.667
269	80	0.0004	0.653
271	80	0.0004	0.670
275	80	0.0004	0.673
AVG 273	**80**	**0.0004**	**0.667**

Table 12: Results of simulation runs with run time of ten seconds

The results of a single run can be seen in the following graphic:

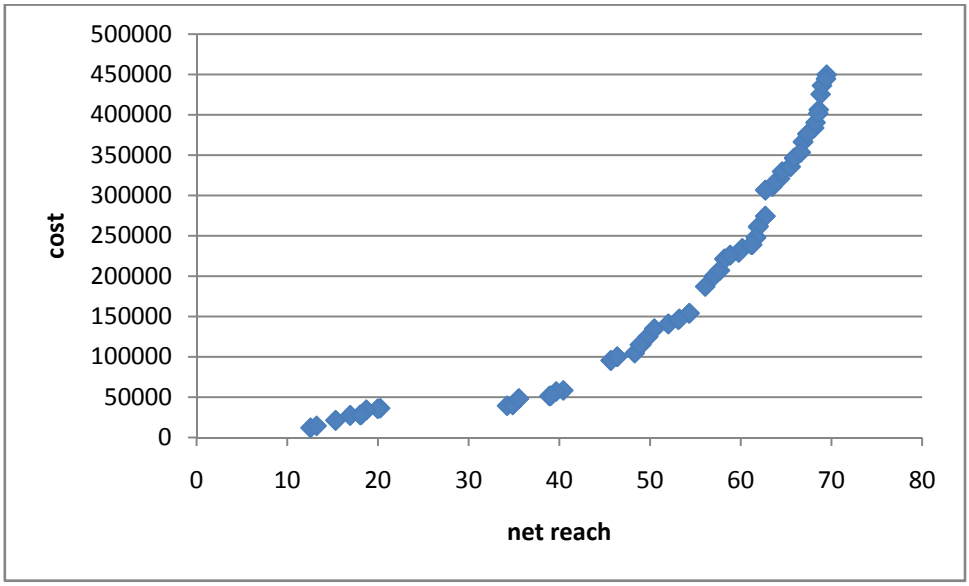

Figure 24: Pareto front of a test run with run time of ten seconds

7.5.2 30 seconds

The following table lists the results which were achieved in ten runs:

generations	population size	mutation probability	HI
1,323	80	0.0004	0.697
1,227	80	0.0004	0.691
1,316	80	0.0004	0.685
1,303	80	0.0004	0.692
1,322	80	0.0004	0.697
1,325	80	0.0004	0.688
1,256	80	0.0004	0.702
1,294	80	0.0004	0.703
1,283	80	0.0004	0.683
1,278	80	0.0004	0.706
AVG 1,293	**80**	**0.0004**	**0.694**

Table 13: Results of simulation runs with run time of 30 seconds

The results of a single run can be seen in the following graphic:

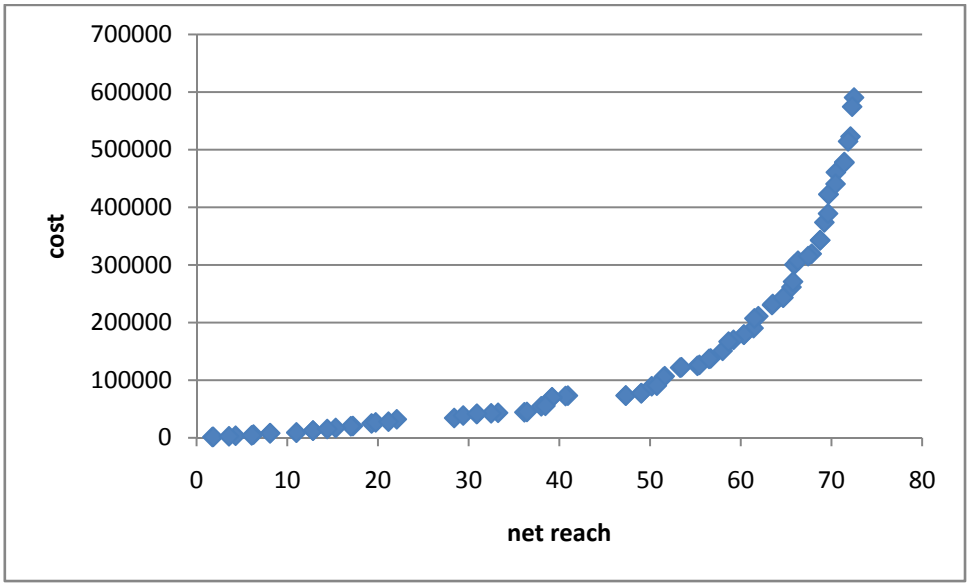

Figure 25: Pareto front of a test run with run time of 30 seconds

7.5.3 One minute

The following table lists the results which were achieved in ten runs:

generations	population size	mutation probability	HI
2,795	80	0.0004	0.704
2,739	80	0.0004	0.722
2,882	80	0.0004	0.706
2,862	80	0.0004	0.702
2,796	80	0.0004	0.710
2,881	80	0.0004	0.708
2,884	80	0.0004	0.706
2,893	80	0.0004	0.717
2,842	80	0.0004	0.712
2,753	80	0.0004	0.716
AVG 2,833	**80**	**0.0004**	**0.710**

Table 14: Results of simulation runs with run time of one minute

The results of a single run can be seen in the following graphic:

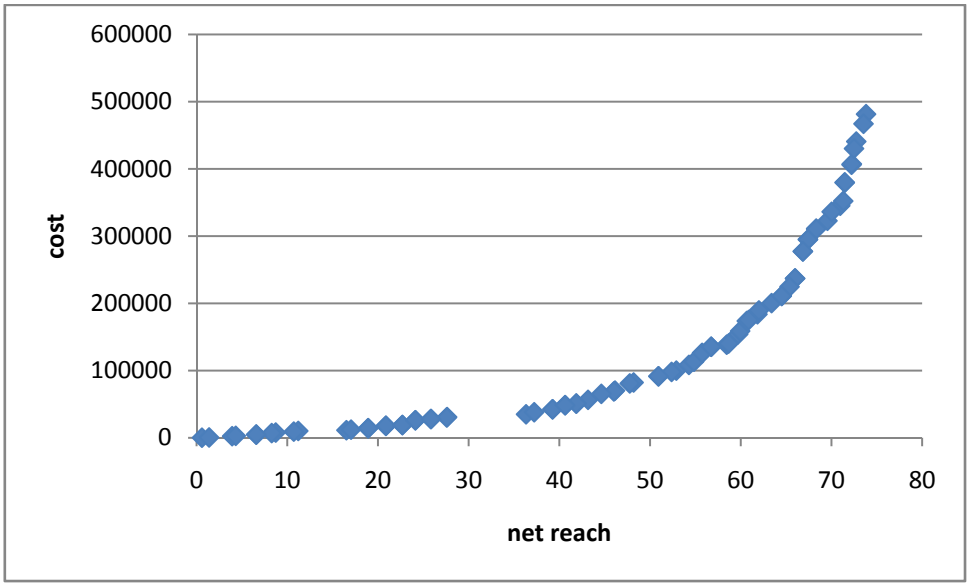

Figure 26: Pareto front of a test run with run time of one minute

7.5.4 Five minutes (population size 1,000)

The following table lists the results which were achieved in ten runs:

generations	population size	mutation probability	HI
693	1000	0.0004	0.727
700	1000	0.0004	0.732
694	1000	0.0004	0.728
703	1000	0.0004	0.728
603	1000	0.0004	0.733
572	1000	0.0004	0.719
660	1000	0.0004	0.725
600	1000	0.0004	0.725
666	1000	0.0004	0.734
666	1000	0.0004	0.725
AVG 656	**1000**	**0.0004**	**0.728**

Table 15: Results of simulation runs with run time of five minutes (population size 1,000)

The results of a single run can be seen in the following graphic:

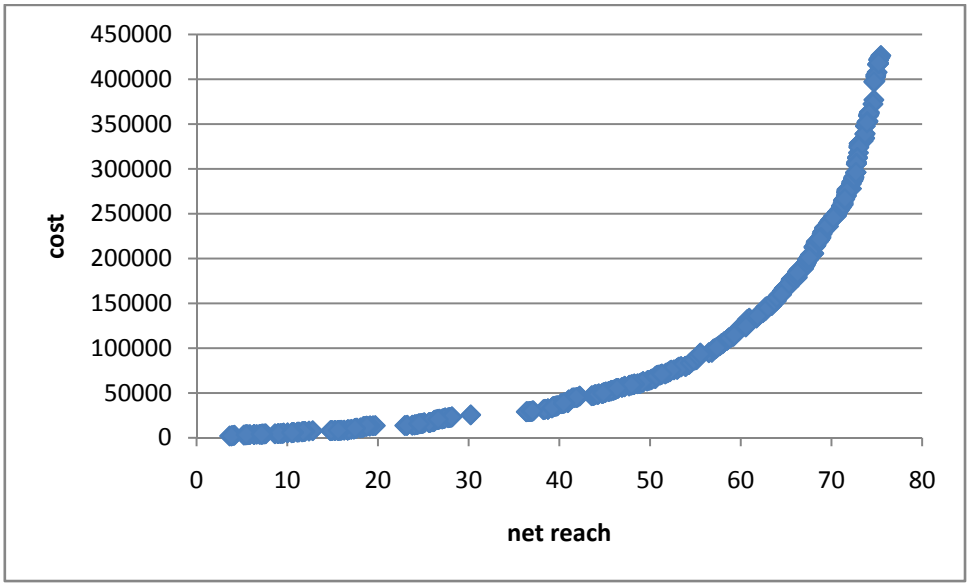

Figure 27: Pareto front of a test run with run time of five minutes (population size 1,000)

7.5.5 Five minutes (population size 80)

The following table lists the results which were achieved in ten runs:

generations	population size	mutation probability	HI
15,169	80	0.0004	0.745
15,496	80	0.0004	0.744
15,279	80	0.0004	0.745
15,356	80	0.0004	0.741
15,224	80	0.0004	0.744
15,452	80	0.0004	0.741
15,219	80	0.0004	0.745
15,918	80	0.0004	0.745
15,868	80	0.0004	0.744
15,743	80	0.0004	0.742
AVG 15,472	**80**	**0.0004**	**0.744**

Table 16: Results of simulation runs with run time of five minutes (population size 80)

The results of a single run can be seen in the following graphic:

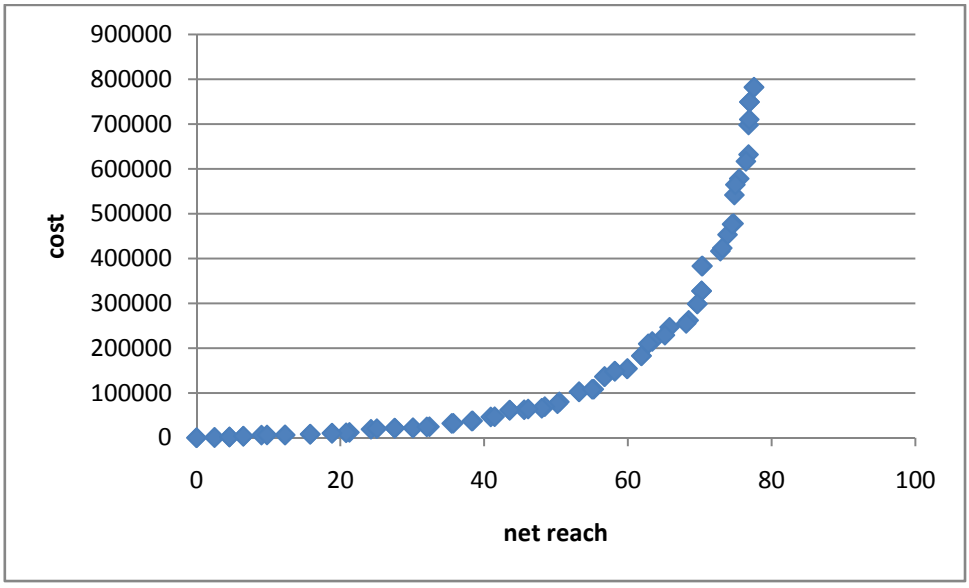

Figure 28: Pareto front of a test run with run time of five minutes (population size 80)

7.5.6 Ten minutes (population size 80)

The following table lists the results which were achieved in ten runs:

generations	population size	mutation probability	HI
28,335	80	0.0004	0.748
27,749	80	0.0004	0.749
26,923	80	0.0004	0.746
27,210	80	0.0004	0.746
27,952	80	0.0004	0.748
26,832	80	0.0004	0.745
27,666	80	0.0004	0.746
27,933	80	0.0004	0.746
26,764	80	0.0004	0.747
26,644	80	0.0004	0.746
AVG 27,401	**80**	**0.0004**	**0.747**

Table 17: Results of simulation runs with run time of ten minutes (population size 80)

The results of a single run can be seen in the following graphic:

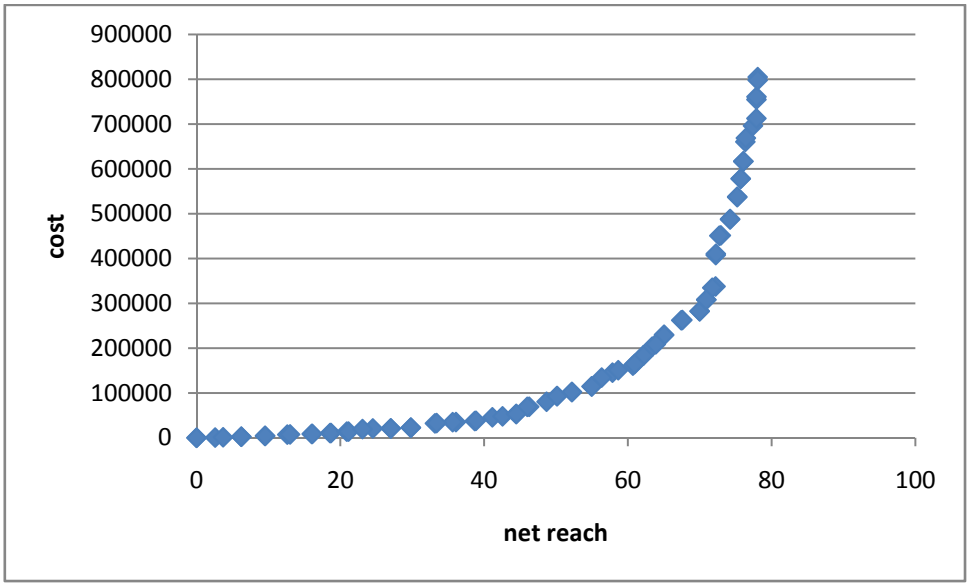

Figure 29: Pareto front of a test run with run time of ten minutes (population size 80)

7.5.7 Ten minutes (population size 1000)

The following table lists the results which were achieved in ten runs:

generations	population size	mutation probability	HI
1,816	1,000	0.0004	0.739
1,698	1,000	0.0004	0.742
1,716	1,000	0.0004	0.748
1,746	1,000	0.0004	0.751
1,716	1,000	0.0004	0.745
1,771	1,000	0.0004	0.745
1,791	1,000	0.0004	0.748
1,799	1,000	0.0004	0.744
1,749	1,000	0.0004	0.749
1,679	1,000	0.0004	0.748
AVG 1,748	**1,000**	**0.0004**	**0.746**

Table 18: Results of simulation runs with run time of ten minutes (population size 1,000)

The results of a single run can be seen in the following graphic:

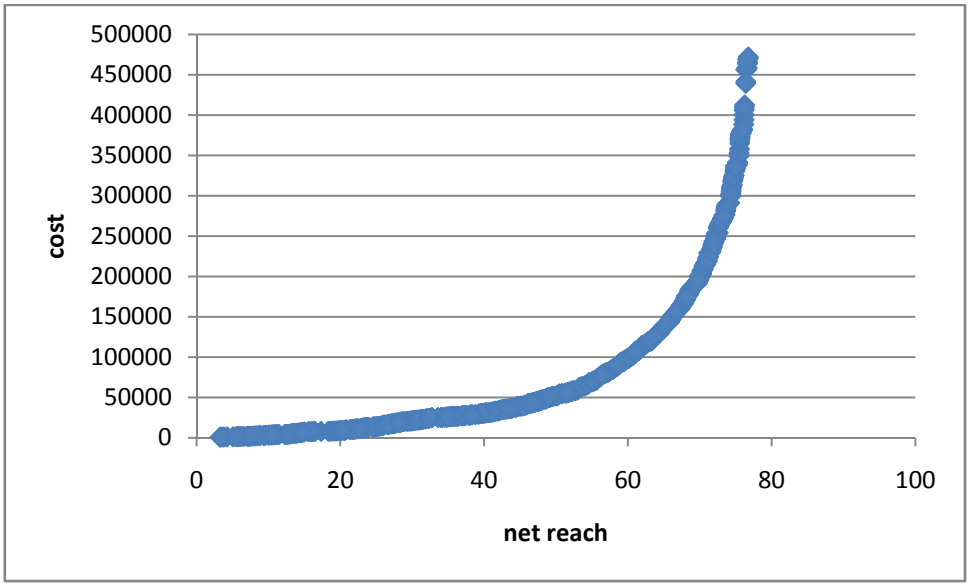

Figure 30: Pareto front of a test run with run time of ten minutes (population size 1,000)

7.5.8 20 minutes (population size 80)

The following table lists the results which were achieved in ten runs:

generations	population size	mutation probability	HI
62,799	80	0.0004	0.747
62,331	80	0.0004	0.746
62,896	80	0.0004	0.748
61,602	80	0.0004	0.748
61,974	80	0.0004	0.747
62,989	80	0.0004	0.749
61,616	80	0.0004	0.746
61,673	80	0.0004	0.747
62,306	80	0.0004	0.749
62,609	80	0.0004	0.747
AVG 62,280	**80**	**0.0004**	**0.747**

Table 19: Results of simulation runs with run time of 20 minutes (population size 80)

The results of a single run can be seen in the following graphic:

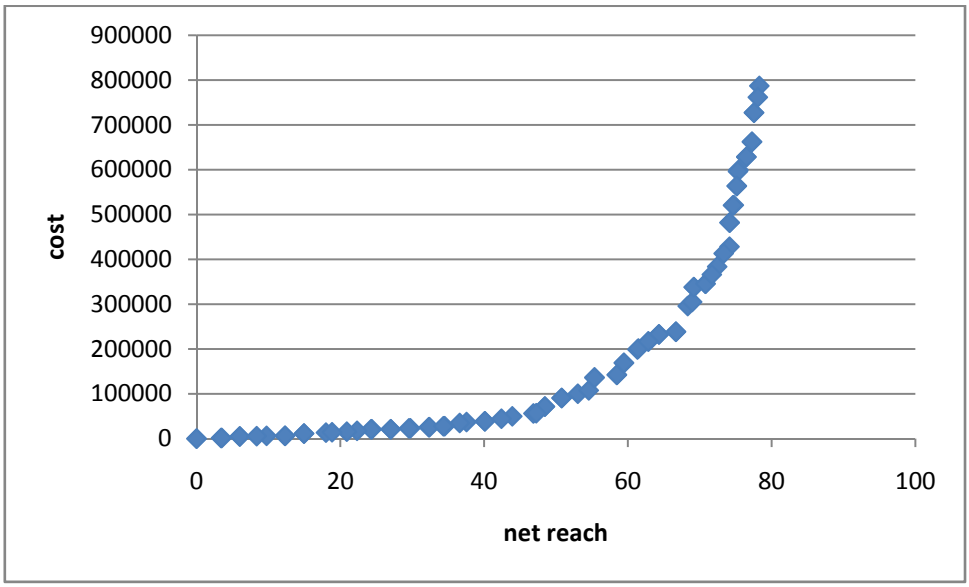

Figure 31: Pareto front of a test run with run time of 20 minutes (population size 80)

7.5.9 20 minutes (population size 1,000)

The following table lists the results which were achieved in ten runs:

generations	population size	mutation probability	HI
3,369	1,000	0.0004	0.754
3,139	1,000	0.0004	0.750
3,258	1,000	0.0004	0.751
2,984	1,000	0.0004	0.753
2,925	1,000	0.0004	0.751
3,044	1,000	0.0004	0.753
3,464	1,000	0.0004	0.756
2,806	1,000	0.0004	0.750
3,011	1,000	0.0004	0.754
2,831	1,000	0.0004	0.755
AVG 3,083	**1,000**	**0.0004**	**0.753**

Table 20: Results of simulation runs with run time of 20 minutes (population size 1,000)

The results of a single run can be seen in the following graphic:

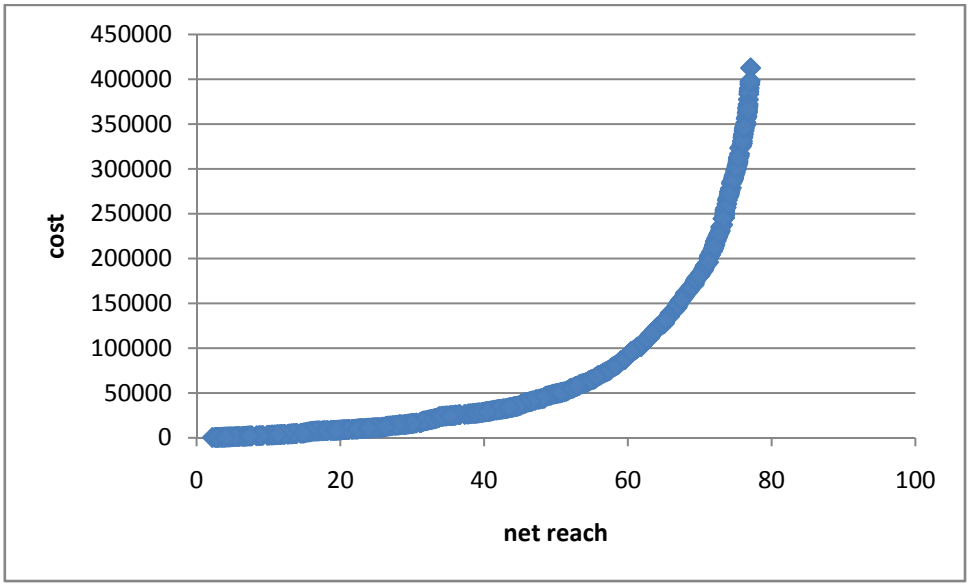

Figure 32: Pareto front of a test run with run time of 20 minutes (population size 1,000)

7.5.10 30 minutes (population size 80)

The following table lists the results which were achieved in ten runs:

generations	population size	mutation probability	HI
76,079	80	0.0004	0.747
76,693	80	0.0004	0.749
81,350	80	0.0004	0.748
80,982	80	0.0004	0.748
82,248	80	0.0004	0.749
78,051	80	0.0004	0.749
76,959	80	0.0004	0.747
77,012	80	0.0004	0.749
82,501	80	0.0004	0.750
76,955	80	0.0004	0.748
AVG 78,883	**80**	**0.0004**	**0.748**

Table 21: Results of simulation runs with run time of 30 minutes (population size 80)

The results of a single run can be seen in the following graphic:

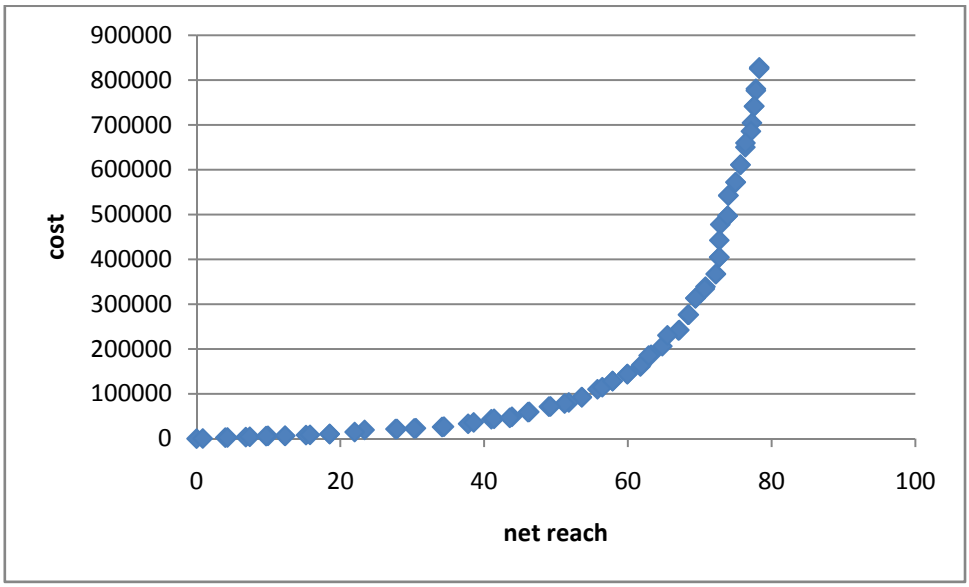

Figure 33: Pareto front of a test run with run time of 30 minutes (population size 80)

7.5.11 30 minutes (population size 1,000)

The following table lists the results which were achieved in ten runs:

generations	population size	mutation probability	HI
4,627	1,000	0.0004	0.758
4,712	1,000	0.0004	0.756
4,933	1,000	0.0004	0.756
5,205	1,000	0.0004	0.758
5,411	1,000	0.0004	0.759
5,458	1,000	0.0004	0.757
4,677	1,000	0.0004	0.755
4,624	1,000	0.0004	0.759
4,525	1,000	0.0004	0.758
5,043	1,000	0.0004	0.757
AVG 4,922	**1,000**	**0.0004**	**0.757**

Table 22: Results of simulation runs with run time of 30 minutes (population size 1,000)

The results of a single run can be seen in the following graphic:

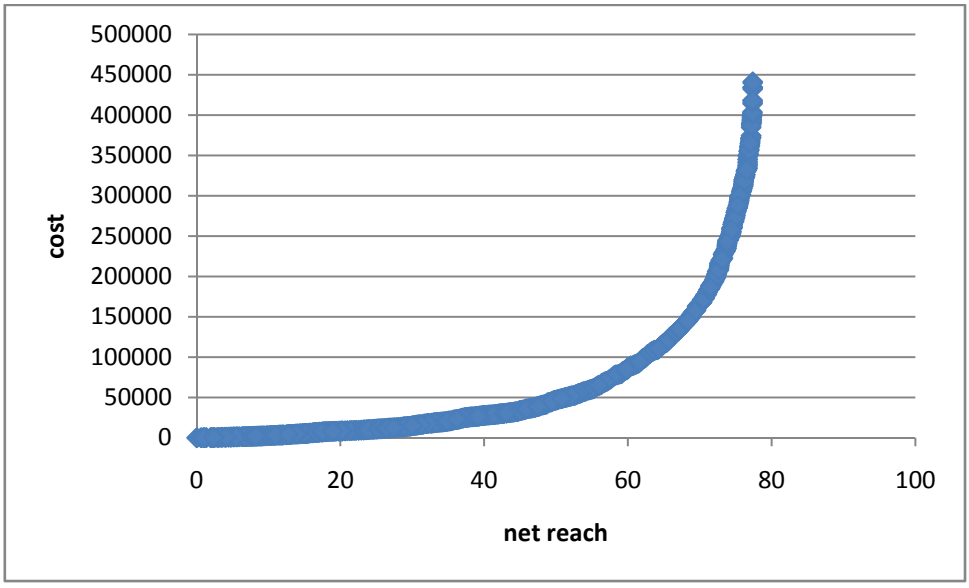

Figure 34: Pareto front of a test run with run time of 30 minutes (population size 1,000)

7.5.12 One hour (population size 80)

The following table lists the results which were achieved in ten runs:

generations	population size	mutation probability	HI
164,561	80	0.0004	0.748
166,615	80	0.0004	0.748
166,949	80	0.0004	0.749
154,448	80	0.0004	0.747
167,857	80	0.0004	0.749
170,091	80	0.0004	0.747
175,021	80	0.0004	0.748
175,070	80	0.0004	0.749
175,939	80	0.0004	0.748
175,345	80	0.0004	0.748
AVG 169,190	**80**	**0.0004**	**0.748**

Table 23: Results of simulation runs with run time of one hour (population size 80)

The results of a single run can be seen in the following graphic:

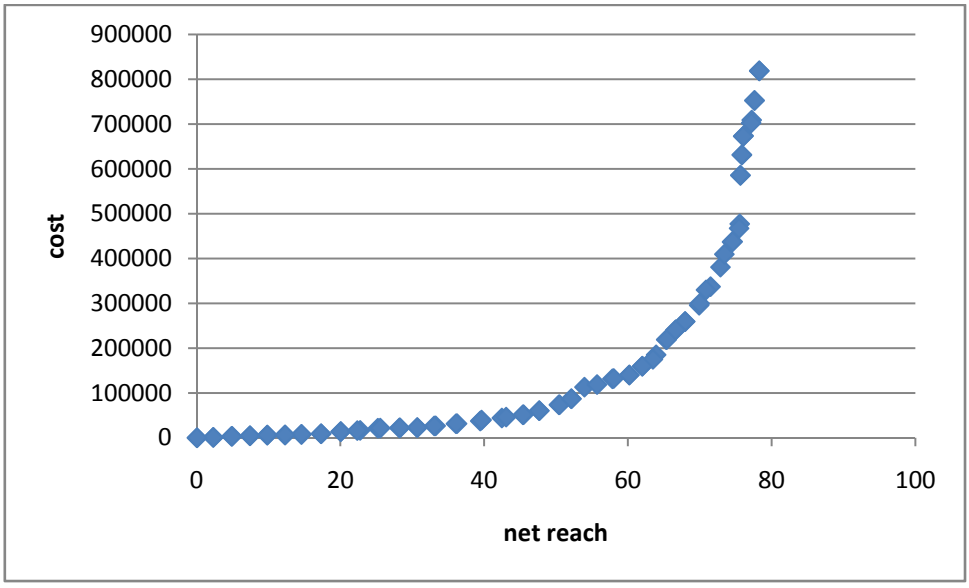

Figure 35: Pareto front of a test run with run time of one hour (population size 80)

7.5.13 One hour (population size 1,000)

The following table lists the results which were achieved in ten runs:

generations	population size	mutation probability	HI
10,900	1,000	0.0004	0.763
11,352	1,000	0.0004	0.762
11,092	1,000	0.0004	0.764
10,810	1,000	0.0004	0.763
9,959	1,000	0.0004	0.761
11,064	1,000	0.0004	0.763
11,381	1,000	0.0004	0.763
11,021	1,000	0.0004	0.762
11,276	1,000	0.0004	0.763
11,076	1,000	0.0004	0.762
AVG 10,993	**1,000**	**0.0004**	**0.763**

Table 24: Results of simulation runs with run time of one hour (population size 1,000)

The results of a single run can be seen in the following graphic:

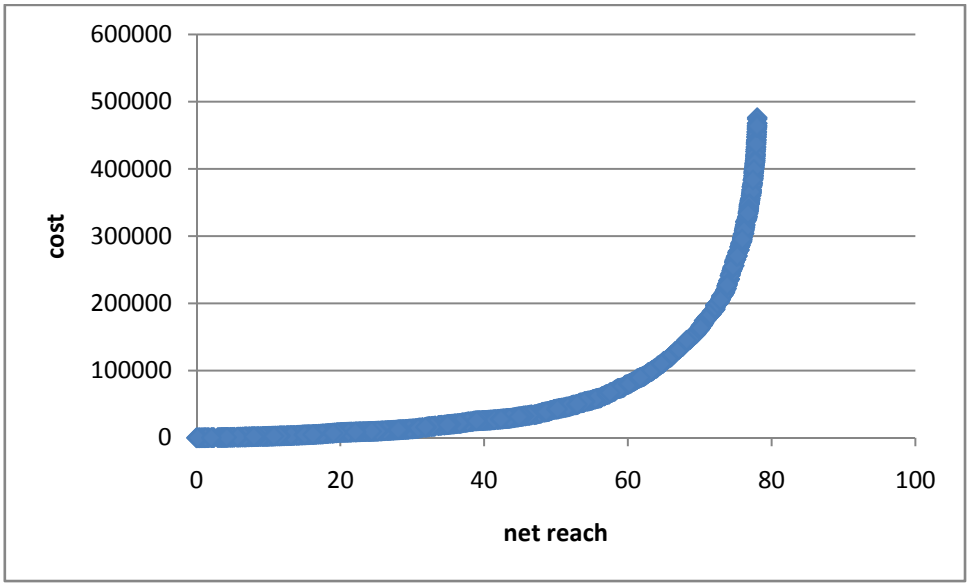

Figure 36: Pareto front of a test run with run time of one hour (population size 1,000)

7.5.14 Two hours

The following table lists the results which were achieved in five runs:

generations	population size	mutation probability	HI
22,473	1,000	0.0004	0.765
21,661	1,000	0.0004	0.764
21,711	1,000	0.0004	0.765
22,107	1,000	0.0004	0.765
22,289	1,000	0.0004	0.764
AVG 22,048	**1,000**	**0.0004**	**0.765**

Table 25: Results of simulation runs with run time of two hours

The results of a single run can be seen in the following graphic:

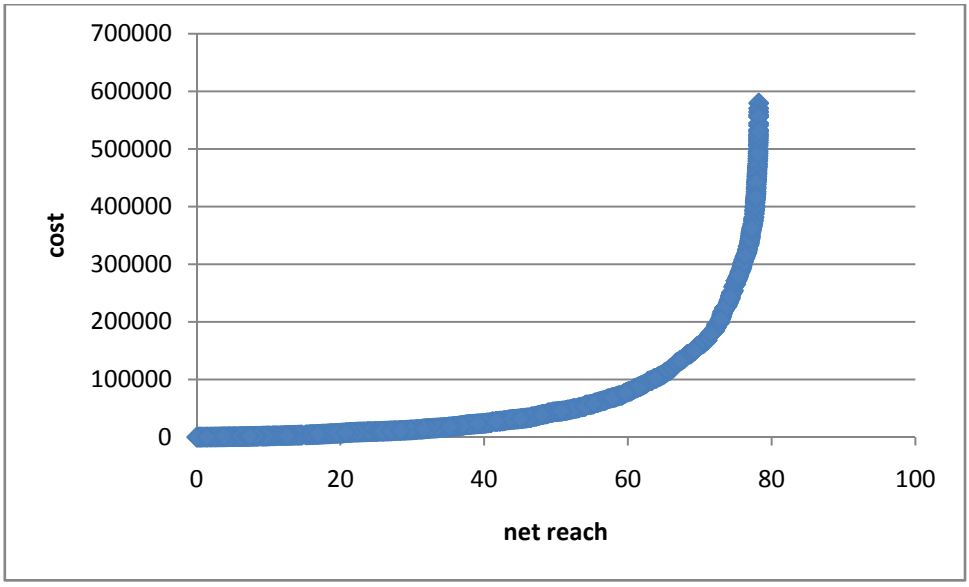

Figure 37: Pareto front of a test run with run time of two hours

7.5.15 Four hours

The following table lists the results which were achieved in five runs:

generations	population size	mutation probability	HI
44,109	1,000	0.0004	0.766
45,236	1,000	0.0004	0.766
43,617	1,000	0.0004	0.766
44,237	1,000	0.0004	0.766
45,616	1,000	0.0004	0.766
AVG 44,563	**1,000**	**0.0004**	**0.766**

Table 26: Results of simulation runs with run time of four hours

The results of a single run can be seen in the following graphic:

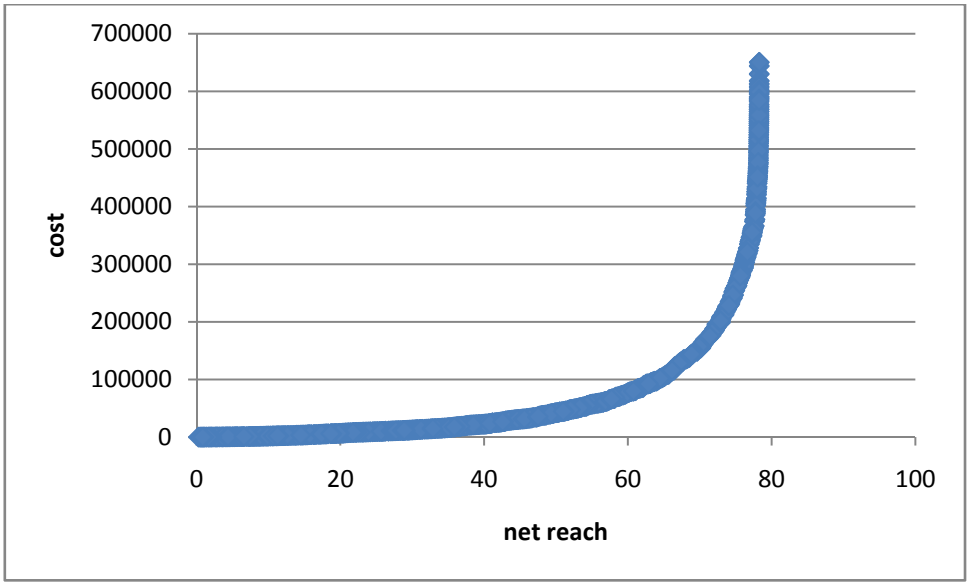

Figure 38: Pareto front of a test run with run time of four hours

7.5.16 Eight hours

The following table lists the results which were achieved in three runs:

generations	population size	mutation probability	HI
85,586	1,000	0.0004	0.766
89,919	1,000	0.0004	0.766
90,519	1,000	0.0004	0.766
AVG 88,675	**1,000**	**0.0004**	**0.766**

Table 27: Results of simulation runs with run time of eight hours

The results of a single run can be seen in the following graphic:

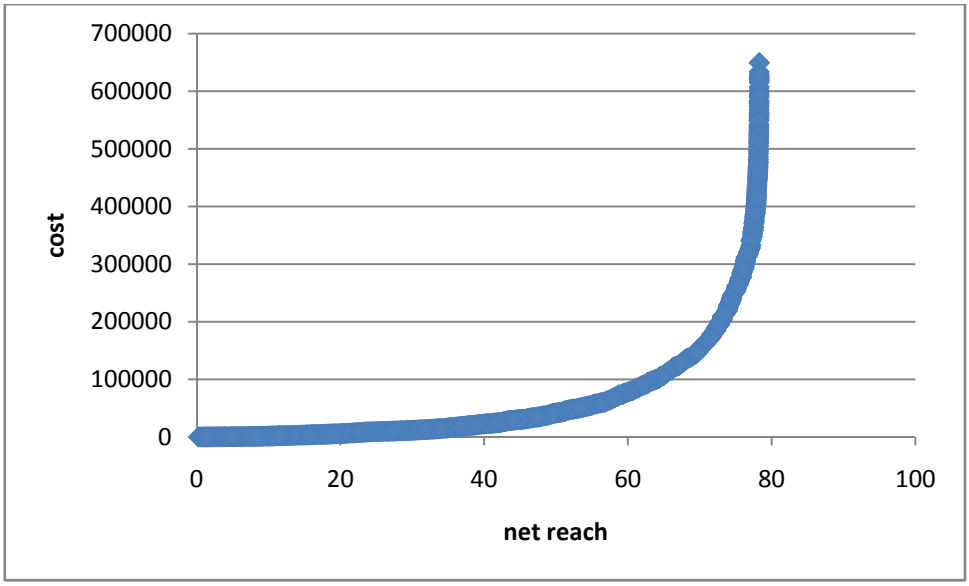

Figure 39: Pareto front of a test run with run time of eight hours

7.5.17 Conclusion

The results of simulation runs with different run times improve rapidly up to a run time of about five minutes. From there on, longer run times only achieve slightly better results, as can be seen in the following table and graphic.

Also it can be seen a population size of 80 yields good results up to a run time of about ten minutes, whereas with run times of ten minutes and higher a population size of 1,000 delivers better results. The data indicates that the results stop improving significantly after about 25,000 generations (see following table).

	Population size 1,000		**Population size 80**	
Run time	**Average number of generations**	**Average HI**	**Average number of generations**	**Average HI**
10 seconds	-	-	273	0.667
30 seconds	-	-	1,293	0.694
1 minute	-	-	2,833	0.710
5 minutes	656	0,728	15,472	0.744
10 minutes	1,748	0.746	27,401	0.747
20 minutes	3,083	0.753	62,280	0.747
30 minutes	4,922	0.757	78,883	0.748
1 hour	10,993	0.763	169,190	0.748
2 hours	22,048	0.765		
4 hours	44,563	0.766		
8 hours	88,675	0.766		

Table 28: Results of different run times with population sizes of 80 and 1,000

The following illustration shows these results with two different lines, one for a population of 80 and one for a population of 1,000.

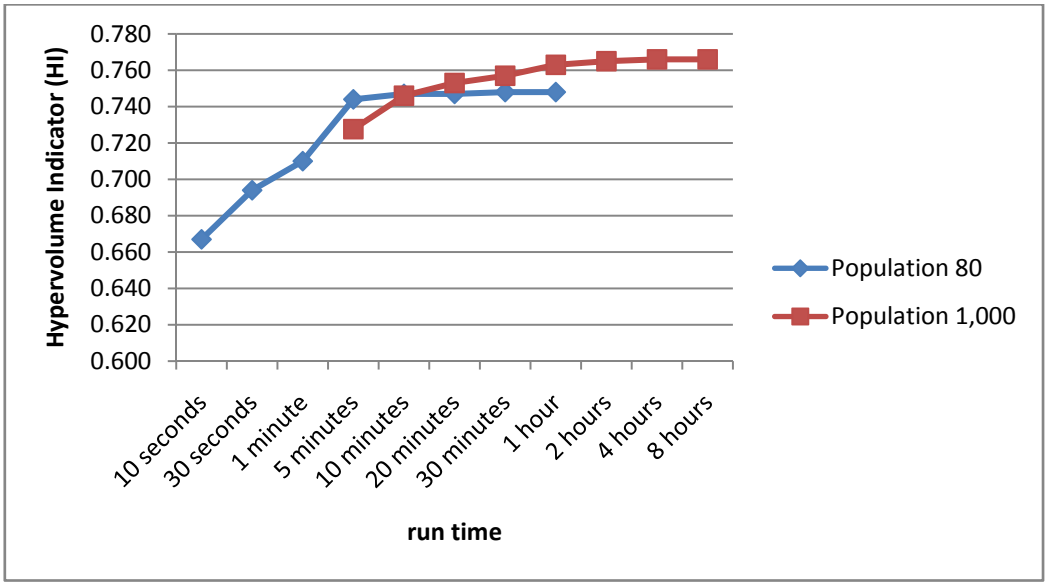

Table 29 Results with different run times

7.6 Comparison with real life campaigns

To evaluate the quality of the results I obtained some real life TV campaigns along with their objective values. These campaigns ran in the same time frame and with the same target group as the campaigns created by the presented application. This real life case is therefore identical to the test case used for optimization (see chapter7.4 “Results of different parameter configurations”).

For the calculation of net reach and GRP (gross reach points) see chapter 5.4.1 ”How are net reach and gross reach measured?”.

	cost	reach	GRP
Plan 1	89,879	33.4	76.0
Plan 2	99,762	28.9	93.2
Plan 3	66,771	27.1	46.7
Plan 4	84,663	36.2	67.9

Table 30: Four real life campaigns and their objective values

First I show four plans with equal cost to the real life campaigns and their net reach and gross reach values.

cost	reach	GRP
90,295	62.53	102.64
100,040	64.16	103.17
67,106	58.33	86.53
85,717	61.74	98.40

Table 31: Comparison of real life campaigns with results from simulation runs (1)

The following table shows the variance:

cost	reach	GRP
0%	87%	35%
0%	122%	11%
1%	115%	85%
1%	71%	45%

Table 32: Variance in %

Next I show four plans with equal net reach to the real life campaigns and their cost and gross reach values.

cost	reach	GRP
16,718	33.43	41.51
11,929	28.86	31.96
10,834	27.29	29.94
20,224	36.13	45.52

Table 33: Comparison of real life campaigns with results from simulation runs (2)

The following table shows the variance:

cost	reach	GRP
-81%	0%	-45%
-88%	0%	-66%
-84%	1%	-36%
-76%	0%	-33%

Table 34: Variance in %

The results of the campaigns created by the simulation run are vastly better than the real life campaigns.
With cost as a constant, the reach doubles.
With reach as a constant, the costs sink by around 80%.

As already mentioned in chapter “5.6 Further Aspects Not Covered By the Presented Application” the real life campaigns cannot directly be compared against the results of the optimization routine, because I used ex-post viewing data. Still the results show that the presented algorithm does a very good job at optimizing.

8 Conclusion

The optimization application presented in this thesis yielded excellent results compared to four campaigns which were run by a media agency for international clients.

As already mentioned above the results from the optimizer and the real life campaigns are not fully comparable for several reasons. First the real life campaigns were created before the viewing data was available; the optimizer uses ex-post data. Second there are some constraints like client preferences, volume rebates by TV stations, etc.

Still; the results were so much better (double the reach with the same cost, 80% less cost for the same reach) that further research in this area is definitely warranted.

For an optimizer application to be used in a media agency there are a lot of problems to be solved yet, but it seems they also bring opportunities to improve the current way of campaign planning and buying.

9 Bibliography

Broadbent S. (1997), “Accountable Advertising”, NTC Publications

Darwin C. (1859), “On the origin of species by means of natural selection or the preservation of favoured races in the struggle of life”

Deb K. (2000), „A Fast and Elitist Multi-Objective Genetic Algorithm: NSGA-II“, KanGAL report 200001, Indian Institute of Technology, Kanpur, India

Ephron, E. (1995), “Want to bug an agency president?”, originally published in “Inside Media”

Ephron, E. (1998), “Recency Planning“, originally published in Mediaweek

Ephron, E. (2003), “Finding the other half”, published at www.ephrononmedia.com

Fonseca C., Fleming P. (1995), “Multiobjective Optimization and Multiple Constraint Handling with Evolutionary Algorithms I: A Unified Formulation”, University of Sheffield, UK

Gotshall S., Rylander B. (2003), “Optimal Population Size and the Genetic Algorithm”

Holland J. (1975), “Adaptation in Natural and Artificial Systems”, MIT Press

Kotler P. (1997), “Marketing Management”, Prentice Hall

Laumanns M. (2003), “Analysis and Application of Evolutionary Multiobjective Optimization Algorithms”, dissertation at the Swiss Federal Institute of Technology Zurich

Martello S., Toth P. (1990), “Knapsack Problems – Algorithms and Computer Implementations“

Mendel G. (1865), „Versuche über Pflanzen-Hybriden“

Wolpert D.H., Macready W.G. (1997), “No Free Lunch Theorems for Search”, The Santa Fe Institute, USA

Zitzler E. (1999), “Evolutionary Algorithms for Multiobjective Optimization: Methods and Application“, habilitation thesis at the Swiss Federal Institute of Technology Zurich

Internet-Sources

[www 1] http://www.gym1.at/schulinformatik/aus-fortbildung/fachdidaktik/vk-01/jahresplan-pohlig/Tag27/knapsack.htm (accessed in June 2008)

[www 2] http://www.mma.com/marketing_mix.htm (accessed in June 2008)

[www 3] http://www.washingtonpost.com/wp-dyn/content/article/2007/01/30/AR2007013001534.html (accessed in June 2008)

Magazines

"Advertising Age", published February 27, 2006 by Crain Communications

"Advertising Age", Erwin Ephron Article, 9.2.1998

"Advertising Age", Advertising Age's 100 Leading National Advertisers Special Report, June 25, 2007

Interviews

Arpino D. (2008), several interviews with Deborah Arpino, CEO of OMD Austria, held in 2008