FH Vorarlberg
University of Applied Sciences

TRAFFYKING

Simulation and Optimisation of Traffic Light Controlled Instersection

Simulation and Optimisation

Vorarlberg University of Applied Sciences
Master's in Mechatronics

Submitted to

Prof. (FH) Dr. Hans-Joachim Vollbrecht

Handed in by

Jan Wenger, Emrah Öztürk, Nicolai Schwartze

Dornbirn, 29.02.2020

Contents

List of Figures

List of Tables

List of Algorithms

1 Task Description

This project is about simulating and optimising a circulative traffic light control. The simulation is done with SUMO which stands for Simulation of Urban MObility. SUMO provides an API that allows external programs to cast and control a simulation. These control structures, simulation evaluation as well as the optimisation algorithms are programmed in Python 3.7.

The project includes the following tasks:

- create infrastructure to call simulation and alter simulation parameters in SUMO
- construct different intersections at different traffic loads
  - Manhattan grid layout: 3x3
  - traffic load: night, day
- four different optimisation algorithms
  - NSGA2
  - Conjugate Gradient Descent
  - Differential Evolution
  - self-created heuristic using Hill-Climbing
- description and evaluation of the results
- precise technical report to ensure repeatability

The simulation returns three parameters:

- overall waiting time calculated as the mean waiting time per car
- overall number of stops calculated as the sum of the stops by all cars
- fairness in waiting time calculated as the variance of the waiting times

We are looking for a solution that minimises all of these parameters.
All of the considered traffic load scenarios are stable. This means, that, under the condition that the optimisation algorithm does not generate impracticable results, the waiting line does not grow arbitrarily large.

2 Scenarios

This chapter is about creating the problem-scenarios in sumo. As stated in the task description, two scenarios on a 3x3 Manhattan gird with different traffic loads for day and night are created.

2.1 Street Network

We used a 3x3 street network for the following simulations.

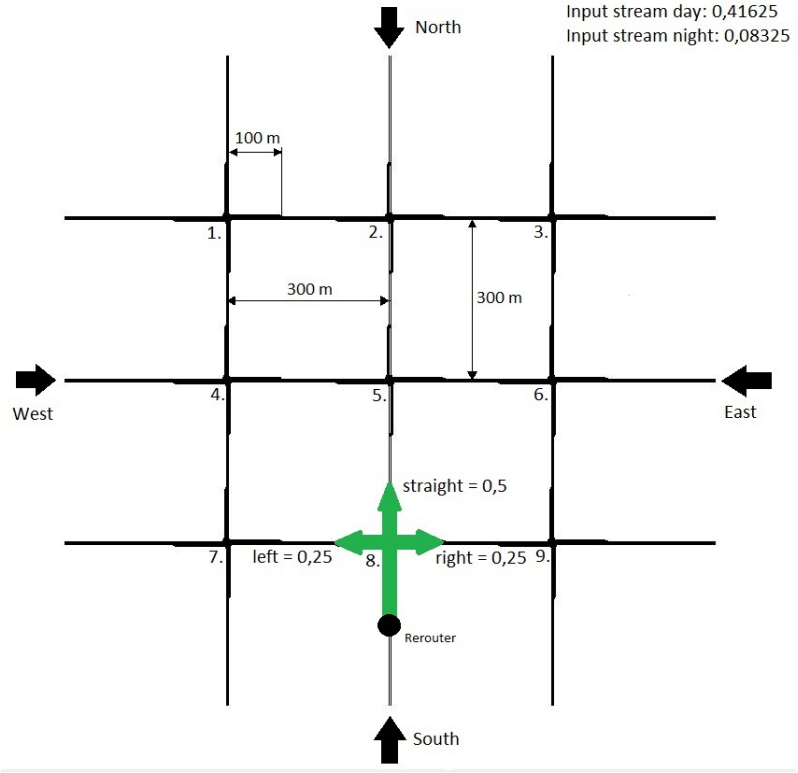

Figure 2.1: 3x3 street network

The street network can be seen in figure 2.1. Only one inflow from each direction (see black arrows) is defined. Rerouters are used for all intersections, which are provided

by sumo (see green arrow). The rerouter is implemented with the following turn probability's left 0.25, right 0.25 and straight 0.5.

2.2 Traffic Load

The traffic load is calculated using the programs Streetnetwork and LPsolve. This is done under the assumption, that the traffic behaves like a fluid (continuous simulation). First the network is created using the program Streetnetwork.

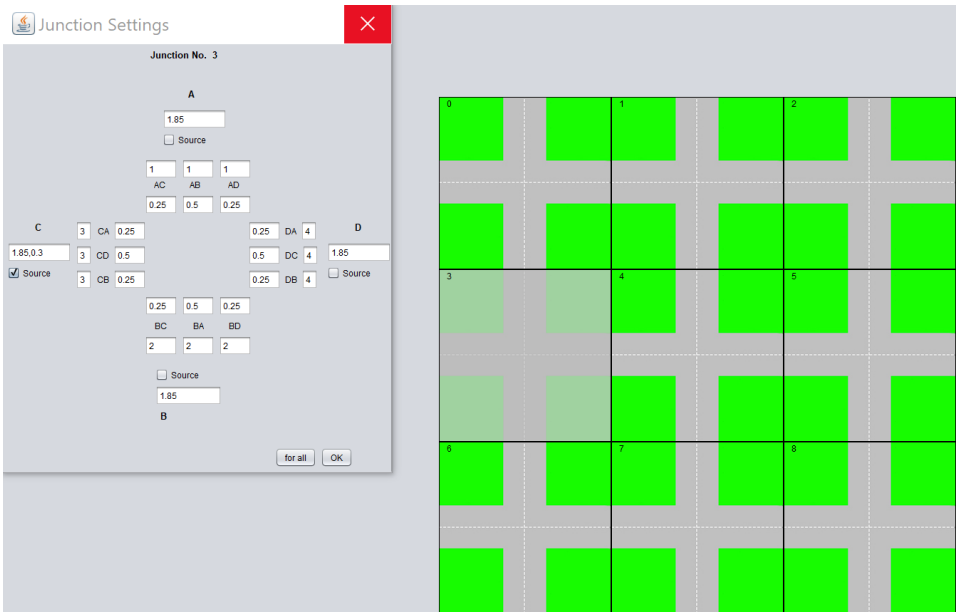

Figure 2.2: creation of street network

Figure 2.2 shows the creating process. All inflow values are set to the value 0.3 and all outflow values are set to the value 1.85. We defined a day ($\epsilon = 0.5$) and a night ($\epsilon = 0.9$) scenario. The program then crates a script, which can be used in LPsolve for the calculation of the c value.

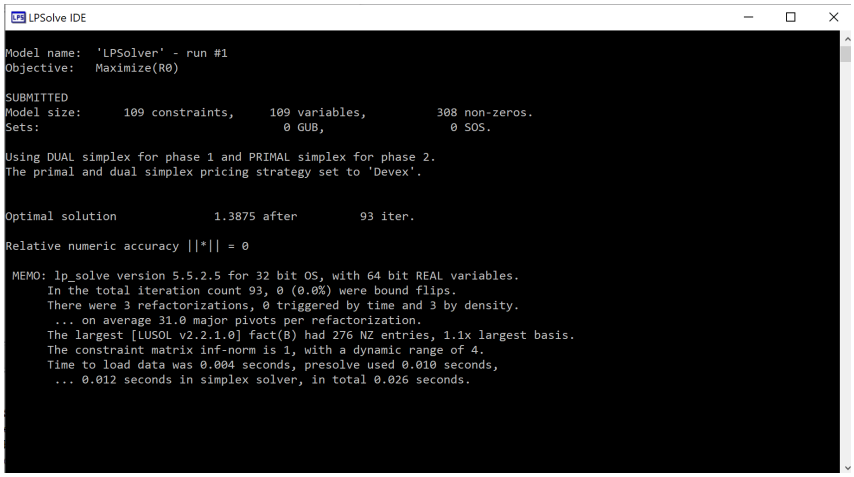

Figure 2.3: calculation with $\epsilon = 0.5$

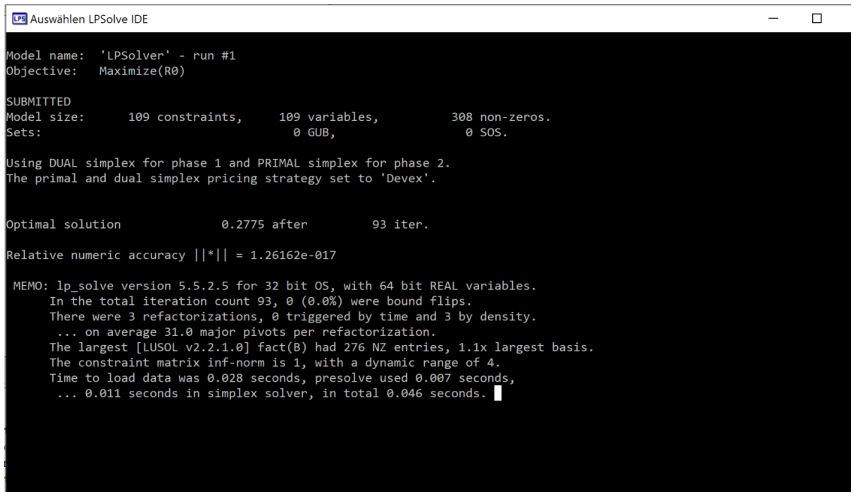

Figure 2.4: calculation with $\epsilon = 0.9$

Figure 2.3 and figure 2.4 show the output of for the c values calculated with LPsolve. The used inflow values for day 0.41625 and night 0.08325 are then calculated by multiplying the chosen inflow value (0.3) with the calculated c values.

2.3 Calculate Relative Green

The following image 2.5 shows the green phases for each intersection.

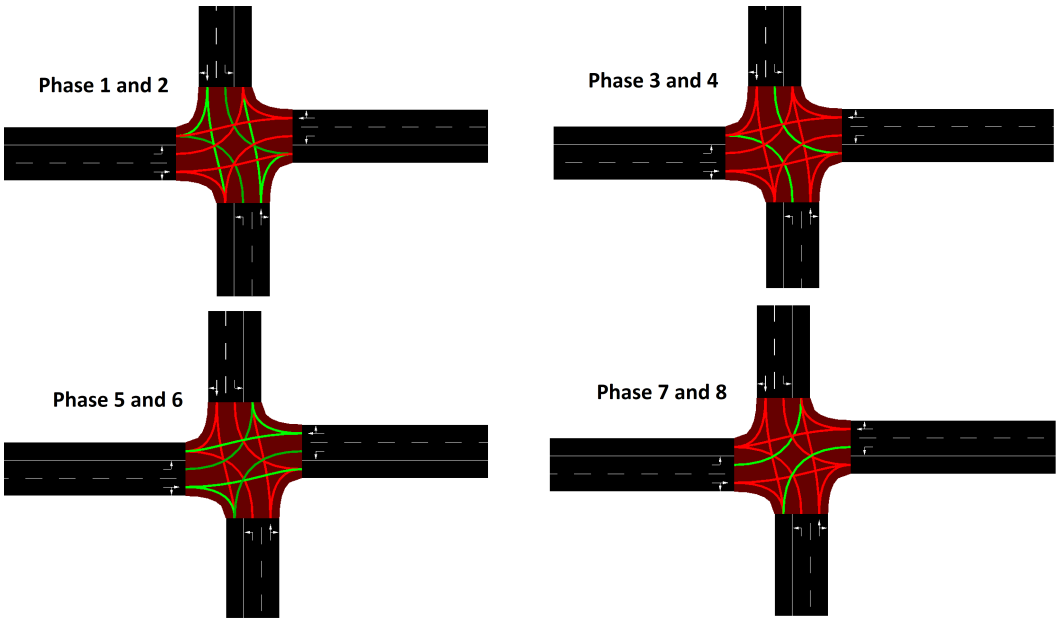

Figure 2.5: traffic light phases for each intersection

The calculation of the relative greens phases is related to the input stream into the intersection. If we call the input stream from the street i I_i and the turn probability from the i-th street to the j-th street p_{ij}, then we can determine the relative green time using an weighted average. Note that the streets are numerated from 1 (north) increasing clockwise to 4 (west). For the first relative green time (seen in figure 2.5, upper left corner) we get the following calculation:

$$rg_{phase1} = \frac{max(I_1(p_{13} + p_{14}) + I_3(p_{31} + p_{32}))}{\sum_{i=1}^{4} phase_i} \tag{2.1}$$

This is because, the input stream from street 1 divided (according to the turn probabilities) into street 3 ($I_1 p_{13}$) and street 4 ($I_1 p_{14}$). The same is done for the input stream I_3. Then this value is divided by the sum of all green phases. This procedure is then used to calculate all four phases.

3 Algorithms

This chapter describes the pseudocode and the characteristics of the implemented optimisation algorithms. Altogether four different optimisers are compared: Differential Evolution (DE), Non Dominated Sorting Genetic Algorithm (NSGA II), Conjugate Gradient Descent (CGD) and a self-created heuristic based on the Hill-Climbing (HC) principal. All optimiser have been tested to work on deterministic multidimensional sphere functions. The table 3.1 shows the implementation of this testfunction in 2D space. Since the NSGA-II is designed for multicriteria optimisation, the testfunction must be adapted: two overlapping spheres are optimised.

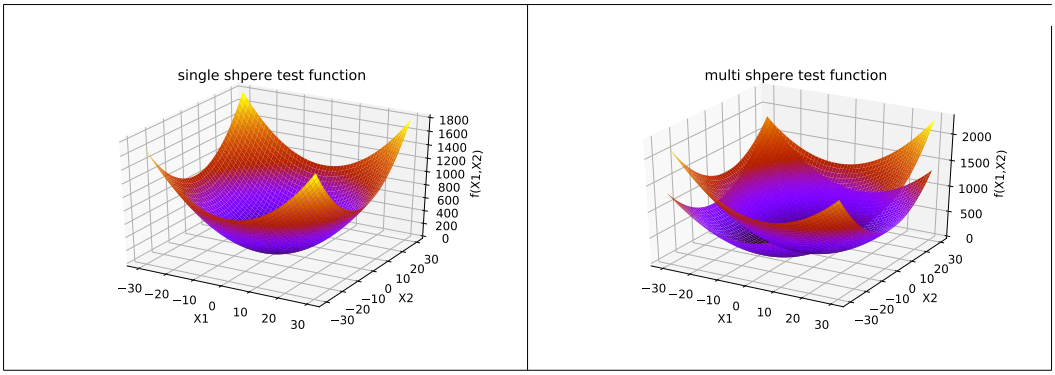

Table 3.1: comparison of single-criteria and multi-criteria sphere function

3.1 Differential Evolution

Differential Evolution was first published by Storn and Price 1997. New forms and adaptions are constantly developed which belong to some of the best performing optimisation algorithms. The "framework" of Differential Evolution is described in algorithm 3.1.

The framework allows for different definitions of mutation and crossover. In this implementation the mutation $rand\_1$ is used. The following equation 3.1 describes this mutation operator. The subscript i hereby stands for the current individual in the loop. The indices $r1$ to $r3$ denote random but different individuals in the population.

Algorithm 3.1: Differential Evolution

```
1  population ← initialization
2  while g < G_max do
3      for individual x_i in population do
4          v_i = muation(x_i, population, F)
5          u_i = crossover(x_i, v_i, CR)
6          if function(u_i) < function(x_i) then
7              x_i = u_i
8          end
9      end
10     g = g + 1
11 end
```

$$v_i = x_{r1} + F(x_{r2} - x_{r3}) \tag{3.1}$$

The parameter F is also called the scalefactor and can be chosen by the user. Typically this is set to be a number in the intervall $F = [0...1]$.

Further a binomial crossover strategy is used. For every coordinate j in the vector, either an element of the old vector x_i or a new vector v_i is chosen. This again depends on a user defined parameter CR called the crossover rate. To ensure, that at least on single element from the new vector is chosen, a random coordinate K must be transferred.

$$u_{ij} = \begin{cases} v_{ij}, & \text{if } j = K \vee rand[0,1] \leq CR \\ x_{ij}, & \text{otherwise} \end{cases} \tag{3.2}$$

The following figure 3.1 shows the verification of the differential evolution algorithm on the sphere function.

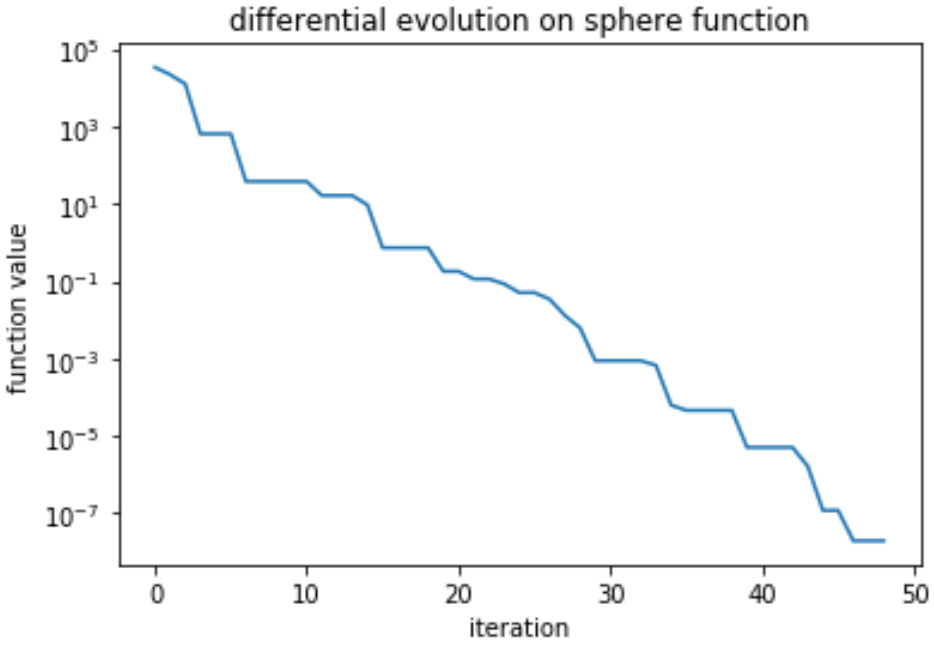

Figure 3.1: testrun of differential evolution on the sphere, shows how the function value depends on the iteration

3.2 Nondominated Sorting Genetic Algorithm II

The NSGA-II was proposed by Kalyanmoy Deb et al. 2002. In contrary to the DE, this algorithm is able to optimise multiple functions at the same time. This is called a multi-criteria optimiser. The result of this algorithm are multiple individuals in a population. Ideally this population describes the pareto front. These are all feasible solutions, the user can then decide which optima is best suited for that specific application.

The implementation was done in conjunction with the algorithm description in appendix B. For a more convenient implementation and readability, every individual in the NSGA II is an object of the class *individual* which holds the most important attributes of a candidate. The following pseudocode 3.2 outlines the general steps to

perform the NSGA II algorithm.

Algorithm 3.2: NSGA II

```
1  population ← initialization
2  while g < G_max do
3      for i in population/2 do
4          p1, p2 = tournament_selection(Pt)
5          q1, q2 = crossover(p1, p2)
6          q1 = mutation(q1)
7          q2 = mutation(q1)
8          Qt = Qt ∪ (q1, q2)
9      end
10     Rt = Rt ∪ (Qt)
11     Rt = fast_non_dominated_sort(Rt)
12     Pt = crowding_distance_sorting(Rt, Pt.size)
13     g = g + 1
14 end
```

Figure 3.2 describes a simple testrun of the algorithm on a multi-criteria sphere function. The plot clearly shows that the algorithm is able to find a valid pareto front.

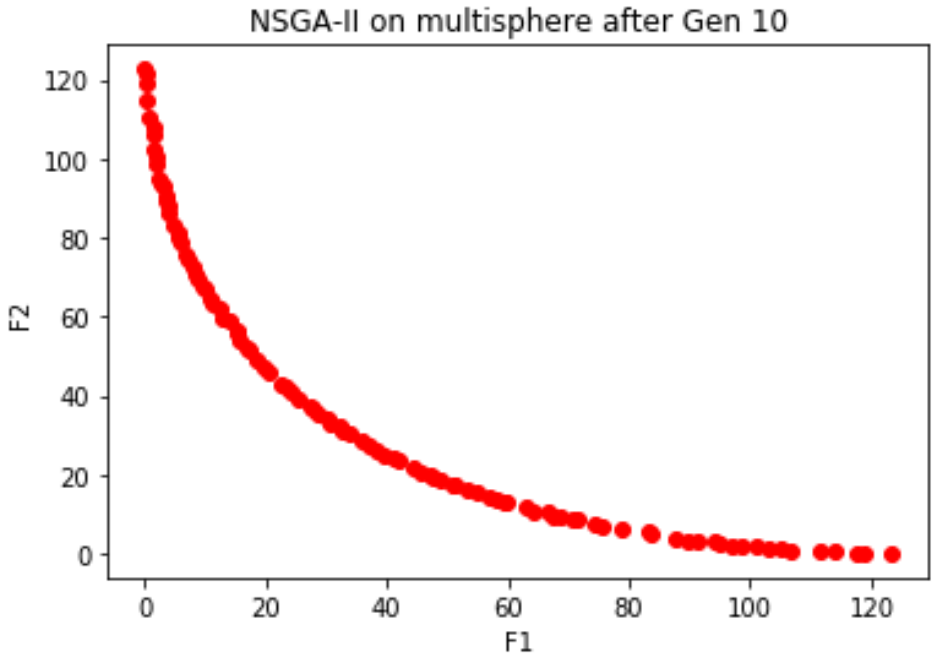

Figure 3.2: testrun of the NSGA-II algorithm on the multi-criteria sphere, shows the found solutions in function space

3.3 Conjugate Gradient Descent

Compared to the previous algorithms, the Conjugate Gradient Method is a deterministic algorithm which uses a numeric approximation of the gradient. This method is again capable of optimising scalar valued functions.
The original idea dates back to Hestenes and Stiefel 1952. It is specially constructed for optimising quadratic problems of the form $f(x) = \frac{1}{2}x^T Hx + b^T x$. If the problem can be represented in this form, the CGD takes exactly n iteration where n is the dimensionality of the function $f : \mathbb{R}^n \to \mathbb{R}$.
The following pseudocode 3.3 outlines the implemented steps of the CGD. This algorithm also holds two subfunctions: *numGrad* calculates the numeric gradient with the central differencing scheme and the *linsearch* implements a golden section search starting from point x and along a vector d.

Algorithm 3.3: Conjugate Gradient Method

```
1  x, d ← initialization
2  while ||∇f(x)|| < ε OR ||η̂d|| < ε do
3      grad = numGrad(x)
4      d = −grad + ||grad||² / (||grad_old||² d)
5      if grad d / (||grad|| ||d||) > −α then
6          d = -grad
7      end
8      η̂ = linesearch(f(x, d))
9      x_old = x
10     grad_old = grad
11     x = x + η̂d
12 end
```

The plot in figure 3.3 shows the function value per iteration of a CDG algorithm on a 2D sphere. As the sphere function is a quadratic function, the CDG is able to optimise it within the exact number of iterations as the dimension of the function.

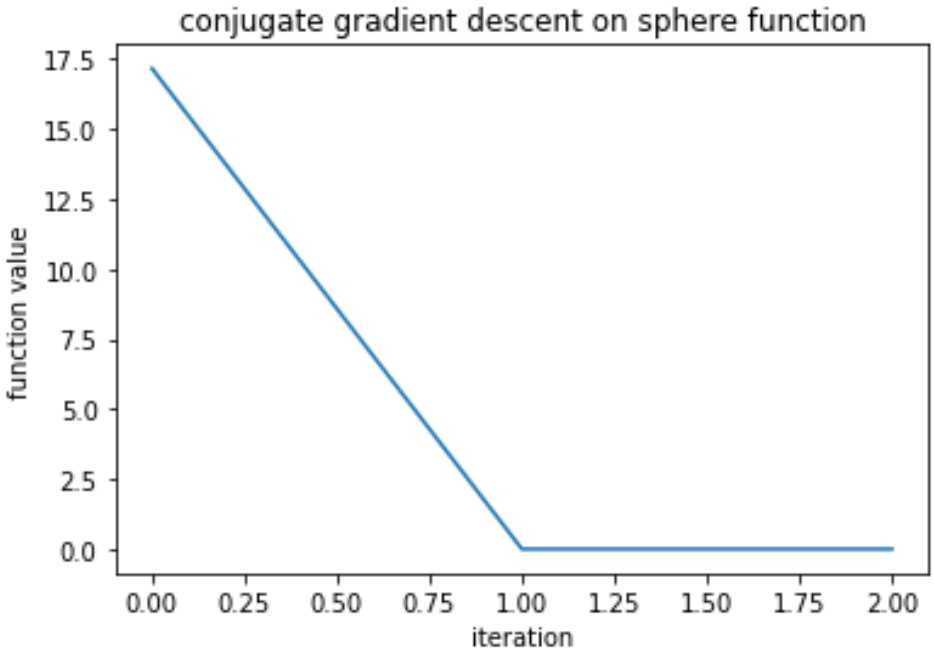

Figure 3.3: testrun of the CDG algorithm on a 2D sphere

3.4 Hill Climbing

The Hill Climbing method is a very simple from of optimisation algorithm. This method again optimises scalar valued problems. The general idea is to sample points around the current location, and select the new location based on the fitness values of these points. There are two main concepts, that need to be figured out to make the process work are:

- generating new solutions
- stepsize adaption

In this implementation, new solutions are produced by going a step in positive and the negative direction of each coordinate. If two consecutive steps do not result in a better solution, the stepsize is adapted by bisecting it. The main issue with this method is, that it performs a strong local search and ultimately leads to a premature convergence.

The following pseudocode 3.4 describes the necessary steps.

Algorithm 3.4: Hill Climbing

```
1  x_start ← initialization
2  while fe < #FE_max do
3      for d in Dim do
4          z[d] = step
5          y_{1,2} = x ± step
6          f = funcion(y)
7          fe = fe + 1
8      end
9      if function(y_d) < function(x) then
10         x = y_d
11     end
12     else
13         step = 0.5 * step
14     end
15 end
```

The figure 3.4 shows how the HC heuristic optimises the sphere function. It is clearly seen, that this algorithm needs many more steps for the same function than the DE or CDG algorithm.

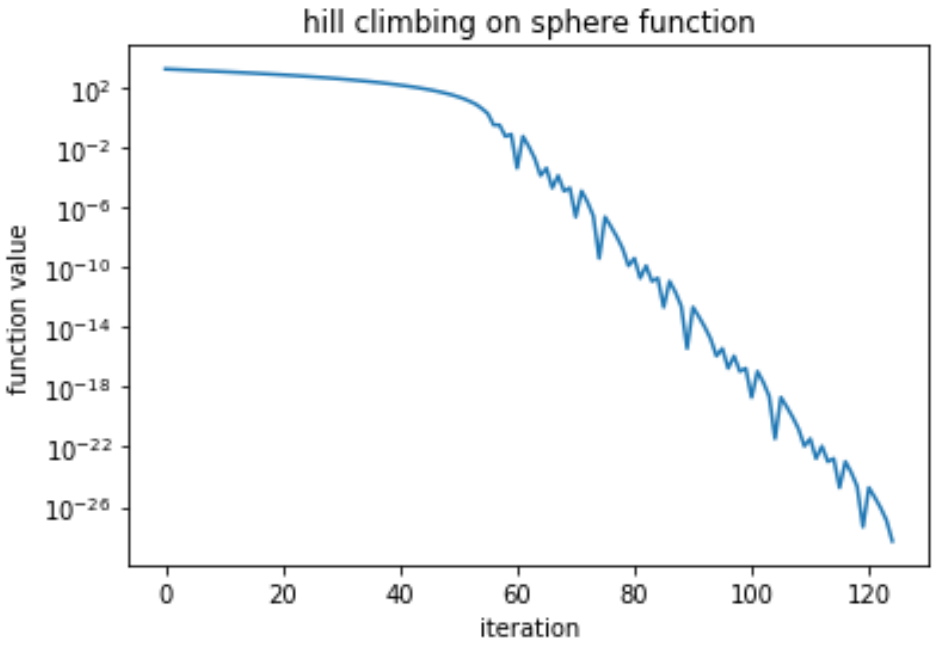

Figure 3.4: testrun of the HC heuristic on a sphere function

4 Simulation Infrastructur

The whole project is programmed in Python 3.7. The choice for that programming language was mainly that Sumo provides a rich API in Pyhton. Although the API is mainly designed to be used with Python 2.7, all necessary features also work in Pyhton 3.

All optimisation algorithms take a function handler as an input, which acts as the function to be optimised. In this case this is the Sumo simulation. Two versions of simulation function are created: a single- and a multi-criteria function. The multi-criteria function returns the three parameters "stop count" (sC in equation 4.2), "waiting time" (wT in equation 4.1) and "fairness" (F in equation 4.3) as a tuple. The waiting time is calculated by the mean over all waiting times for every car in the simulation. Similar the stop count is the sum of all stops in the whole simulation, but without normalisation of the number of cars. The fairness is defined as the variance over all waiting times meaning a smaller fairness results in a better fitness value.

Only the NSGA-II can handle these values separately. For all other algorithms, the single-criteria function is used. The three parameters are intermingled by a weighted mean (as seen in equation 4.4). For the sake of simplicity, the weights are kept at 1. These three parameters can be read from the tripinfo file, which is automatically created by Sumo after every simulation.

Problem: Since the cars can take a turn at every intersection, it might be possible that car gets trapped in the grid for a longer period. This artificially increases the number of stop counts and the waiting time. The optimiser can not improve this situation because the traffic light has no impact on this issue. To prevent that this effect weakens the found solutions, the function evaluation must be adapted. This is done by normalising the waiting time as well as the stop count with the length of the trip (rL) by that specific car. However, this also implies, that the grid is equally spaced.

$$wT = \frac{1}{N} \sum_{c=0}^{N} \frac{wT_c}{rL} \tag{4.1}$$

$$sC = \sum_{c=0}^{N} \frac{sC_c}{rL} \tag{4.2}$$

$$fair = \frac{1}{N} \sum_{c=0}^{N} (wT_c - wT)^2 \tag{4.3}$$

$$singleCriteria = \frac{wWT \cdot wT + wSC \cdot sC + wF \cdot F}{wWT + wSC + wF} \tag{4.4}$$

Since this simulation is haevily subjected to randomness, one simulation would not be meaningful. Thus, a mean over 10 replications is calculated.

The following flowchart in 4.1 shows the necessary steps to execute a full experiment. The state "do optimizer steps" is a placeholder for all other steps done by the optimisation algorithm.

The individuals are coded so that the problem to be optimised is 9-dimensional. On the first position of the vector is the cyclic time for a single traffic light. The following 8 parameters are the temporal offsets of each traffic light to the reference traffic light (counting from east to west).

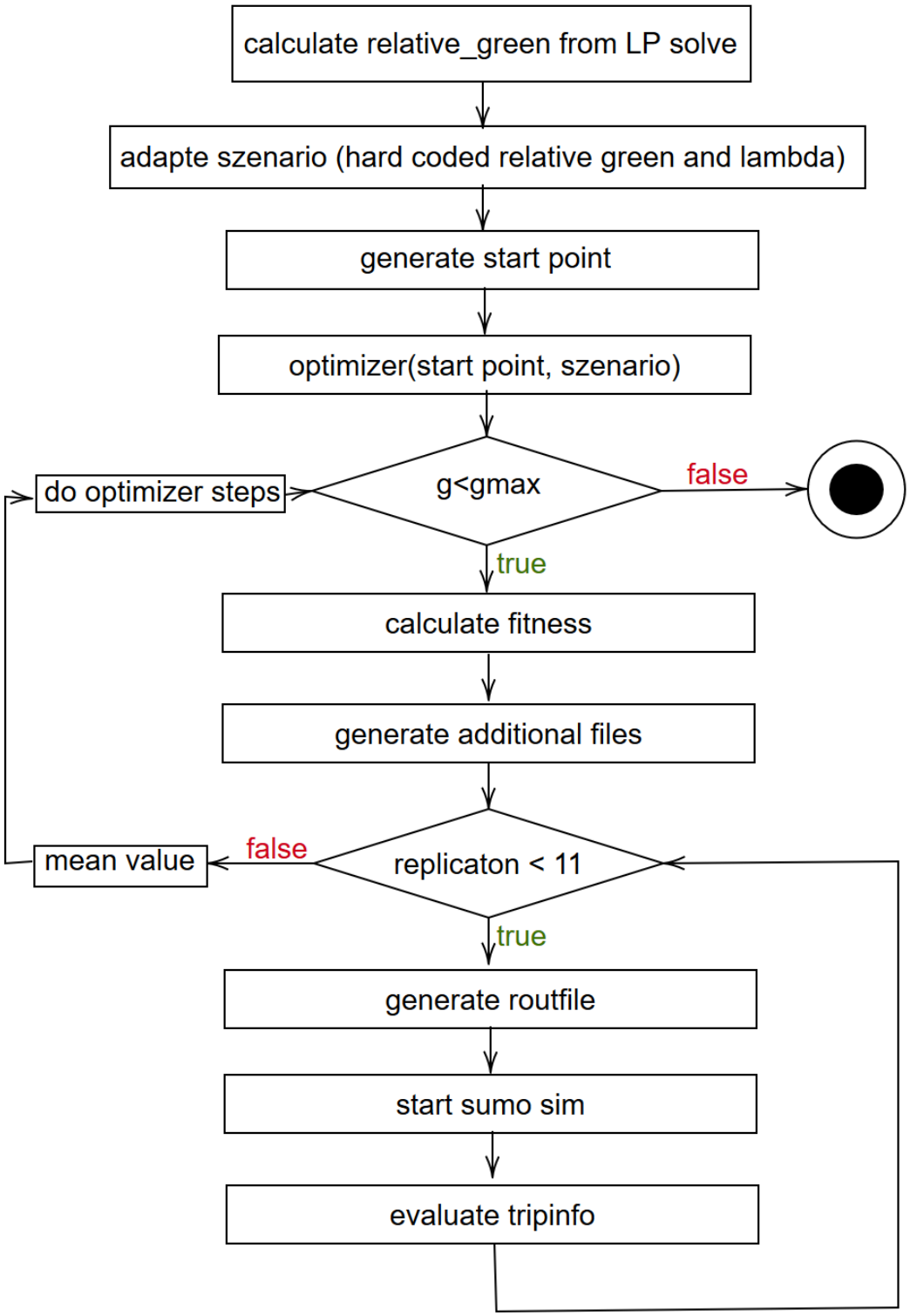

Figure 4.1: flowchart of a complete experiment including the setup and optimisation

5 Experiments

This chapter describes all performed experiments. Since there are 4 optimisation algorithms and 2 scenarios, combining all results in 8 different experiments. These are described in the following chapters.

5.1 NSGA-II night

The table 5.1 below shows the setup for this experiment.

Parameter	Value
Population Size	30
Initialisation min	$\vec{x}^T = [5 \cdots 5]$
Initialisation max	$\vec{x}^T = [70 \cdots 70]$
Initialisation Strategy	uniform random
Generation max	15
Simulation time	3600 steps = 1h
Scenario layout	symmetric
Replications	10
Input Stream λ	0.0832499999999292

Table 5.1: parameter table of experiment NSGA-II at night

The following two figures 5.1 and 5.2 show the distribution of the solutions in the function space. Without looking at the actual values, it can be concluded that the solutions tend to spread further apart with advancing number of generations. At the very least this proves that the NSGA-II algorithm is optimising the simulation.

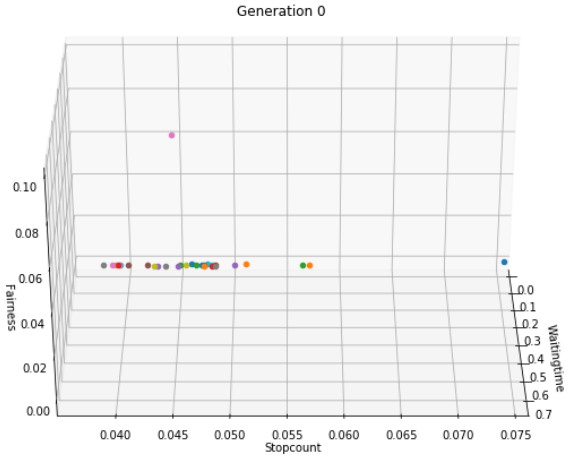

Figure 5.1: sample result of the NSGA-II on the night scenario after initialisation

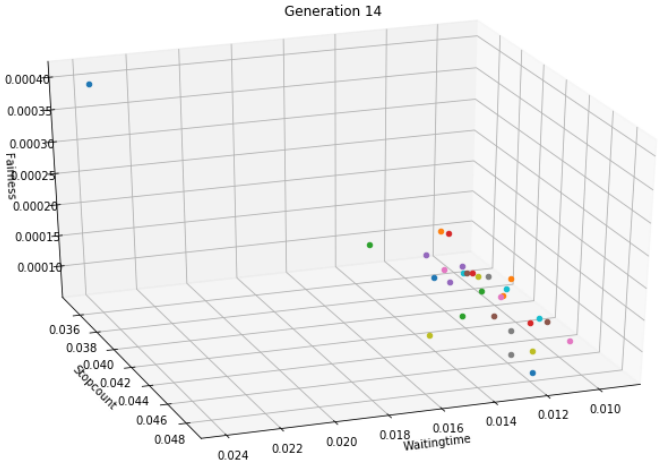

Figure 5.2: sample result of the NSGA-II on the night scenario after 14 generation

Figure 5.3 shows the distribution of the optimised cycle time in the population. It can clearly be seen that the NSGA-II prefers smaller cycle times in the bin from 30 to 35. This is especially remarkable since the population was initialised uniformly random between 5 and 70. In a practical application the usefulness of such a short time is debatable: This only works if the cars drive nearly perfectly and start immediately after the light goes green without much of a reaction time.

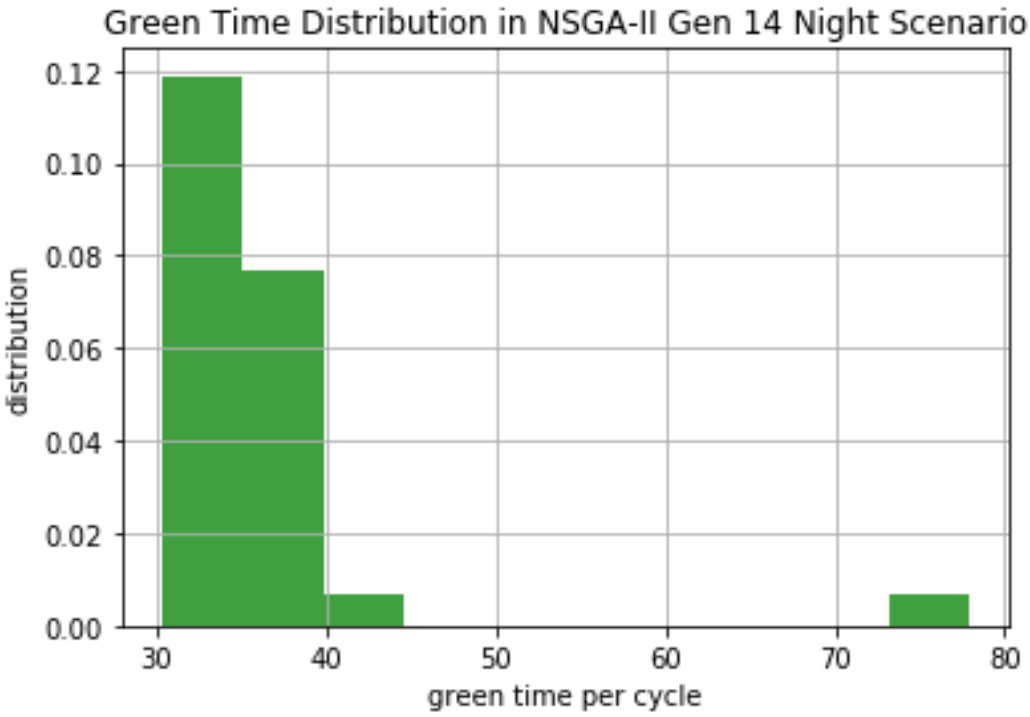

Figure 5.3: Histogram of the cycle time distribution in the last population of the NSGA-II on the night scenarion.

Looking at the offset times throughout the population is not very productive. These values are distributed between -78 and 161 and there is no way to draw any sensible conclusions from them.

5.2 NSGA-II day

The table 5.1 below shows the setup for the experiment using an NSGA-II on the daytime scenario. In conjunction with the experiment on the night scenario, only the input stream and the maximum number of generations changed.

Parameter	Value
Population Size	30
Initialisation min	$\vec{x}^T = [5 \cdots 5]$
Initialisation max	$\vec{x}^T = [70 \cdots 70]$
Generation max	20
Simulation time	3600 steps = 1h
Scenario layout	symmetric
Replications	10
Input Stream λ	0.41625

Table 5.2: parameter table of experiment NSGA-II during the day

Again, the following figures show the distribution of the population in the function space. Figure 5.4 shows the population at generation 0. It is remarkable, that some solution cluster together and form groups at the stop count of 0, 5, 10 and 20. The population at generation 19 seen in figure 5.5 clearly traces out a pareto front.

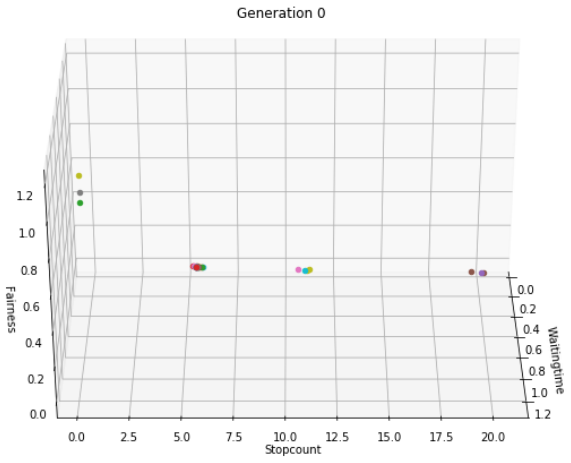

Figure 5.4: sample result of the NSGA-II on the day scenario after initialisation

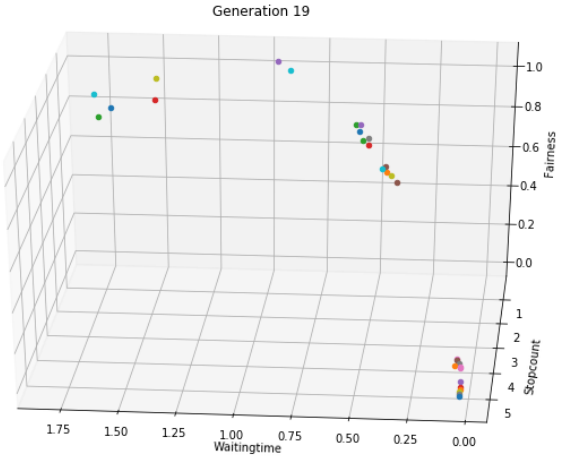

Figure 5.5: sample result of the NSGA-II on the day scenario after 19 generation

Figure 5.6 shows the distribution of the resulting cycle times in the last population. Most of the cycle times end up in the bin between 5 and 20. This again shows that the NSGA-II tends to grow towards shorter cycle times.

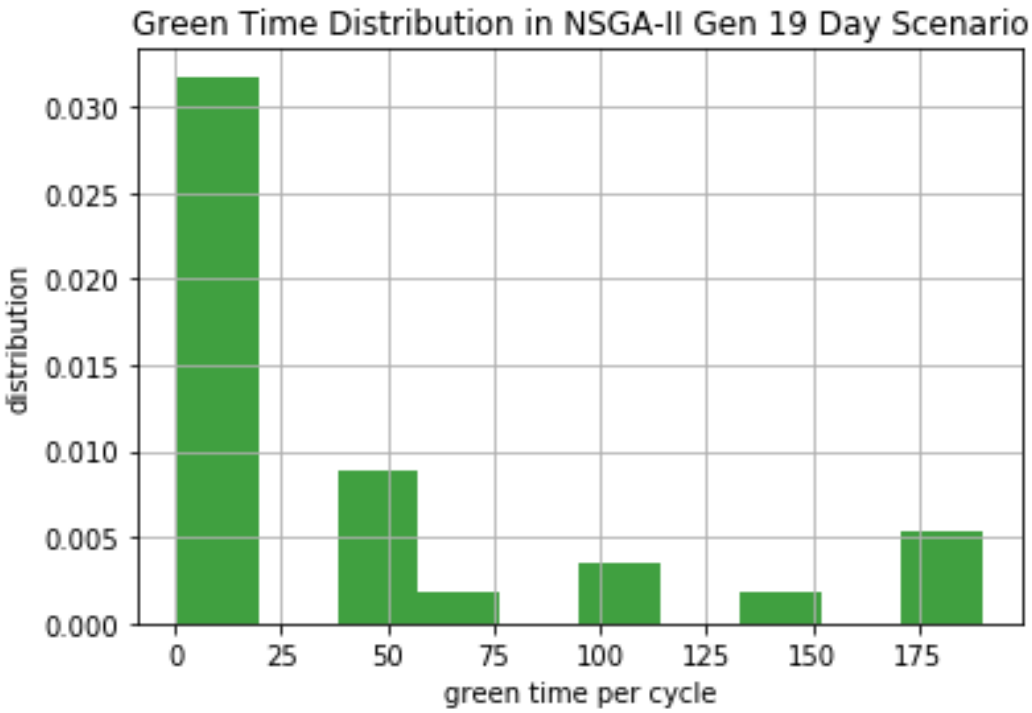

Figure 5.6: Histogram of the cycle time distribution in the last population of the NSGA-II on the day scenarion.

These values for the offset times are again very widely distributed between -107 and 189.

5.3 HC night

The table below shows the parameter setup for running the HC algorithm on the night scenario. The starting point is initialised to a vector $\vec{x}^T$ shown below. This point was chosen for two different reasons:

- it is already a pretty good solution; from there the algorithm should do the optimisation
- it is in middle of the last population found by the NSGA-II

This point servers as a the starting point for all other single-criteria algorithms.

Parameter	Value
Initialisation	$\vec{x}^T = [90, 5, 10, 15, 20, 25, 30, 35, 40]$
Iteration max	60
Step Size	1
Simulation time	3600 steps = 1h
Scenario layout	symmetric
Replications	10
Weights	$[1, 1, 1]$
Input Stream λ	0.0832499999999292

Table 5.3: parameter table of experiment HC at night

The following plot in figure 5.7 shows how the function value decreases over time. The blue line denotes the explored function value in every direction, the orange line shows only the function values that have been improved.

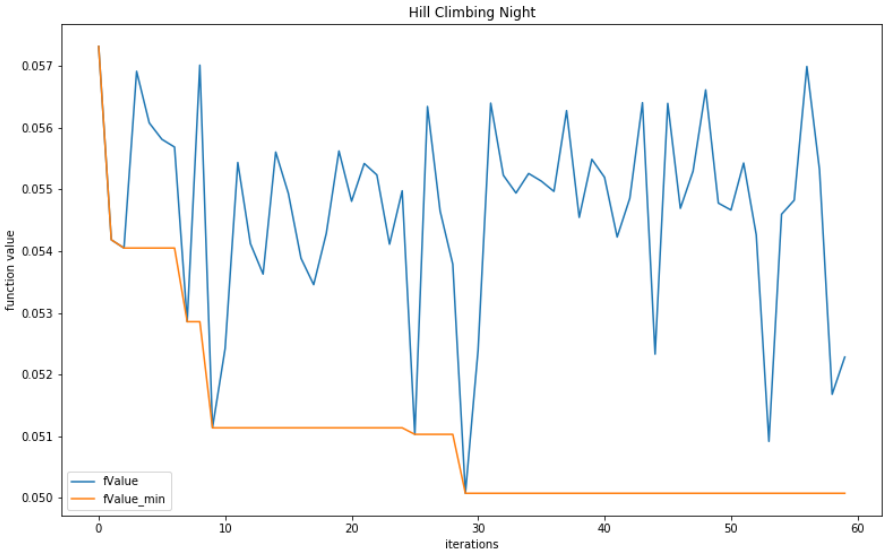

Figure 5.7: sample result of the HC Algorithm during the night scenario

The found optimum is located at vector seen in equation 5.1. There is only very little change from the original starting point. This might depend on a various reasons:

- starting point was already near the optimum
- not enough iteration
- trapped in local optimum

$$f\left(\begin{bmatrix} 94.31268064 \\ -2.49787839 \\ 0.22650818 \\ 20.68603951 \\ 10.84759343 \\ 17.31755371 \\ 28.43479486 \\ 35.95338739 \\ 36.52319049 \end{bmatrix}\right) = 0.056852236013176254 \tag{5.1}$$

5.4 HC day

Similar to the night scenario, the day scenario begins at the same starting point. Again the table 5.4 shows the experiment parameter.

Parameter	Value
Initialisation	$\vec{x}^T = [90, 5, 10, 15, 20, 25, 30, 35, 40]$
Iteration max	30
Step Size	1
Simulation time	3600 steps = 1h
Scenario layout	symmetric
Replications	10
Weights	$[1, 1, 1]$
Input Stream λ	0.41625

Table 5.4: parameter table of experiment HC during the day

The plot in figure 5.8 shows how the function value decreases over time. It is important to note, that the same starting point results in a greater function value compared to the night scenario. This must be correct since there are more cars in the system as compared to the night scenario.

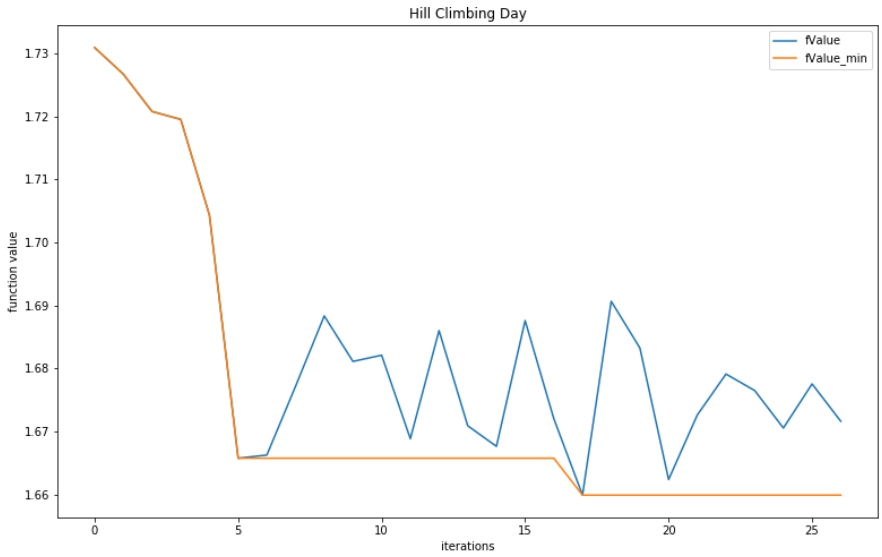

Figure 5.8: sample result of the HC Algorithm during the day scenario

The result is shown in the equation 5.2. The cycle time of this optimum is smaller than the cycle time of the night scenario. This also checks out with the NSGA-II experiment where the cycle of the night scenario is greater than the one of the day scenario.

$$f\left(\begin{bmatrix} 85.6798195 \\ 13.0753775 \\ 17.4374849 \\ 6.08137965 \\ 37.1726901 \\ 37.5230003 \\ 39.3042561 \\ 42.1506951 \\ 48.8756786 \end{bmatrix}\right) = 1.6746488062350436 \tag{5.2}$$

5.5 CGD on night and day

As described above in 3.3 the CGD is especially helpful for quadratic functions. Since we have no idea of the form of this function, it is save to assume that the CGD is not very well suited for this problem. Further, the simulation is stochastic and thus the linesearch does employed along the gradient is results in a wrong stepsize. The following tables 5.5 and 5.6 show the parameter settings for both experiments.

Night Scenario	
Parameter	Value
Initialisation	$\vec{x}^T = [90, 5, 10, 15, 20, 25, 30, 35, 40]$
Stopping Criteria ϵ	0.1
Safety Angle α	0.1
Line Search η	10
Numeric Gradient h	machine epsilon
Simulation time	3600 steps = 1h
Scenario layout	symmetric
Replications	10
Input Stream λ	0.0832499999999292

Table 5.5: parameter table of experiment CGD at night

Day Scenario	
Parameter	Value
Initialisation	$\vec{x}^T = [90, 5, 10, 15, 20, 25, 30, 35, 40]$
Stopping Criteria ϵ	0.1
Safety Angle α	0.1
Line Search η	10
Numeric Gradient h	machine epsilon
Simulation time	3600 steps = 1h
Scenario layout	symmetric
Replications	10
Input Stream λ	0.41625

Table 5.6: parameter table of experiment CGD during the day

In the resulting plots of the figures 5.9 and 5.9 it is clearly depicted that the CGD does not work for such a problem. Not only does the algorithm fails to optimise the function, it is also very computationally expensive: the numeric gradient must be calculated by taking the difference quotient in every direction. This already results in 18 function evaluation. Then the linesearch must be done which further increases the function evaluation count.

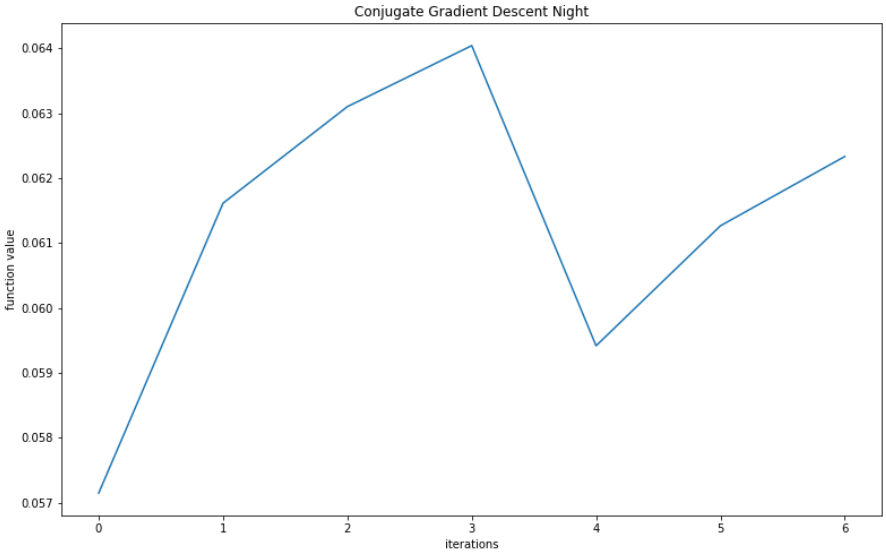

Figure 5.9: sample result of the CGD Algorithm during the night scenario

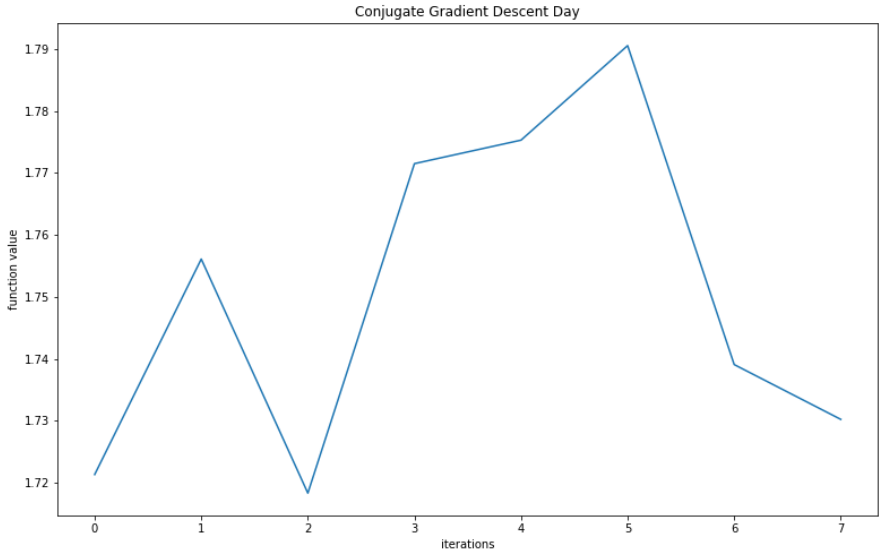

Figure 5.10: sample result of the CGD Algorithm during the day scenario

5.6 DE on night and day

The tables below 5.7 and 5.8 below fully describe the parameter setting for the experiments. However, there have been a few issues with these experiments.

Night Scenario	
Parameter	Value
Population Size	20
Initialisation min	$\vec{x}^T = [5 \cdots 5]$
Initialisation max	$\vec{x}^T = [70 \cdots 70]$
Function Evaluation max	10000
Mutation Operator	rand 1
Mutation Scale Factor F	0.5
Crossover Operator	binomial
Crossover Probability	0.1
Simulation time	3600 steps = 1h
Scenario layout	symmetric
Replications	10
Input Stream λ	0.0832499999999292

Table 5.7: parameter table of experiment DE at night

Scenario	
Parameter	Value
Population Size	20
Initialisation min	$\vec{x}^T = [5 \cdots 5]$
Initialisation max	$\vec{x}^T = [70 \cdots 70]$
Function Evaluation max	10000
Mutation Operator	rand 1
Mutation Scale Factor F	0.5
Crossover Operator	binomial
Crossover Probability	0.1
Simulation time	3600 steps = 1h
Scenario layout	symmetric
Replications	10
Input Stream λ	0.41625

Table 5.8: parameter table of experiment DE during the day

No results were gained in doing this experiment. The function evaluation (in this case the simulation) did not work correctly. Sumo experienced a lot of crashes and the DE algorithm could not even finish one generation. There might be a few reasons for that issue:

- The DE mutation operator ***rand 1*** is known for spreading widely across search space as Epitropakis, Plagianakos, and Vrahatis 2011 show in their paper. It is possible, that this operator generates a bad trail solutions that causes Sumo to crash.
- There might be an error in the simulation function that has gone under the radar. However this is rather unlikely because the same functions was used for all other algorithms.

6 Critical Discussion

A major issue that leads the CGD Algorithm to fail, might be the calculation of the numeric gradient. The tables 6.1 and 6.2 below show 10 different function evaluation for the day and night scenario where each result is already a mean over 10 replications. The stepsize for the difference quotient is the machine epsilon which is proven to be the best value. However, this number is so small, that it completly perishes in the stochastic effects of the simulation. Thus, the gradient does not necessarily point towards the highest increase.

This effect could be minimised by using more replications. But since the experiments already take very long to calculate, this is not a productive measure. Another solution would be to take a bigger step and thus only look at the average change over a wider space.

A different issue arises from the single-criteria function by calculating the weighted sum. The separate values for waiting time and fairness are relatively small compared to the big value of the stop count (depending on the solution this might be a factor between 100 and 1000). This means that the optimisation algorithm is more sensitive towards the stop count. If the algorithm should minimise all parameters equally, the weights must be adapted.

Run	Result
1	0.05845304094188188
2	0.058203791618237855
3	0.06447016694180632
4	0.058089395318890756
5	0.05607968915837128
6	0.06045055736765529
7	0.059554536881858855
8	0.06064499410810244
9	0.061072665992490185
10	0.06496445913659102
Mean	0.060198329746588584
Variance	7.084166340545999e-06
$f(x+h)$	0.05425510359003312

Table 6.1: results of 10 different runs on the night scenario

Run	Result
1	1.7470391724996315
2	1.7903318700728446
3	1.734687539778108
4	1.7597643271768404
5	1.7526521713795522
6	1.7783639367329034
7	1.7603225664014694
8	1.7497671306402292
9	1.7328548281632024
10	1.7292531384605312
Mean	1.7535036681305312
Variance	0.0003485028568951494
$f(x+h)$	1.7446520153831648

Table 6.2: results of 10 different runs on the day scenario

7 Conclusion and Further Work

There are a few conclusions that could be drawn from the experiments:

- At first this might seem counterintuitive, but the experiments show that for the night scenario a greater cycle time is preferred as compared to the day scenario.
- The NSGA-II works quite well for this problem. The experiments show that it optimises correctly, and it even traces out a (rudimentary) pareto front.
- The Hill Climbing is the simplest of the tested algorithms. Although it takes longer for deterministic functions, it is very reliable on this kind of stochastic function.
- The Conjugate Gradient Descent Algorithm does not optimise this function. This might be a consequence of 3 different issues: the small stepsize, the stochastic function and the need of a deterministic descent direction.

During the project, several problems were encountered:

- Sumo is not very well documented, making the start not very gentle. The tutorials do not cover all the necessary parts for this project. This left us with a scrappy understanding of Sumo and its capabilities.
- There is no possibility to evaluate the found results, except some self-constructed heuristic ideas.
- Depending on the optimisation algorithm and the size of the problem, the computation time is very long (ranging from several hours to even days). Since Sumo is not installed on the school computers, this means that the students laptops are occupied for that time. This strictly limits the number of experiments that could be done.
- Sometimes Sumo stops the simulation without any reasonable error. This means, that any running experiment must be started again with the last state. This further contributes to a temporal overhead.
- During the simulation it is possible, that cars get teleported when the streets get crammed to much. This means that such a simulation is not accurate and must be disregarded.

Further work could contain a renewed and optimised version of the simulation environment that takes less time to evaluate the function. This could be done by starting Sumo by a command line call, without going through the slower API communication which is done over TCP sockets. Further, the replications could be run in parallel to save even more time.
Although the parameters used are the ones that are typically recommended for these algorithms, the experiments could be redone by using different parameters. This might give a greater insight to which parameter are successful and how this particular problem is structured.

Bibliography

Deb, K. et al. (Apr. 2002): "A fast and elitist multiobjective genetic algorithm: NSGA-II". en. In: *IEEE Transactions on Evolutionary Computation* 6.2, pp. 182–197. ISSN: 1089778X. DOI: 10.1109/4235.996017. URL: http://ieeexplore.ieee.org/document/996017/ (visited on 11/13/2019) (cit. on p. 8).

Epitropakis, Michael G.; Plagianakos, Vassilis P., and Vrahatis, Michael N. (Apr. 2011): "Finding multiple global optima exploiting differential evolution's niching capability". In: *2011 IEEE Symposium on Differential Evolution (SDE)*. 2011 IEEE Symposium on Differential Evolution (SDE), pp. 1–8. DOI: 10.1109/SDE.2011.5952058 (cit. on p. 30).

Hestenes, Magnus R and Stiefel, Eduard (1952): "Methods of Conjugate Gradients for Solving Linear Systems". en. In: p. 28 (cit. on p. 10).

Storn, Rainer and Price, Kenneth (Dec. 1997): "Differential Evolution – A Simple and Efficient Heuristic for global Optimization over Continuous Spaces". en. In: *Journal of Global Optimization* 11.4, pp. 341–359. ISSN: 1573-2916. DOI: 10.1023/A:1008202821328. URL: https://doi.org/10.1023/A:1008202821328 (visited on 04/05/2019) (cit. on p. 6).

Appendices

A Task Description (*german*)

Projektbeschreibung

„multikriterielle Optimierung eines umlaufbasierten, ampelgesteuerten Strassenverkehrsnetzes mit Hilfe evolutionärer Algorithmen“

In diesem Projekt soll untersucht werden, wie mit Hilfe evolutionärer Algorithmen eine umlaufbasierte Ampelsteuerung optimiert werden kann. Zu optimierende Kriterien sollen sein:

- Wartezeit
- Anzahl Stopps
- Fairness

Damit handelt es sich also um ein multikriterielles Optimierungsproblem. Zur Optimierung soll dabei (Vorgabe) einmal der bekannte evolutionäre Algorithmus NSGA-II [1], und im Vergleich dazu ein „unikriterieller“ evolutionärer oder genetischer Algorithmus Ihrer Wahl gewählt werden. Letzterer soll die drei obigen Kriterien linear in einer eindimensionalen Zielfunktion optimieren.

Projektziel ist dabei grundsätzlich die Analyse des Optimierungsproblemes zum vertieften Verständnis von umlaufbasierten Ampelregelungen und deren Optimierung durch evolutionäre Algorithmen. Es sollen also synthetische Verkehrsszenarien, nicht reale, definiert und untersucht werden, unter Verfolgung dieses Projektzieles. Zur Bewertung der Leistung der untersuchten beiden Algorithmen soll deshalb auch immer eine manuell (nach „gesundem Menschenverstand“) definierte Vergleichsampelregelung erstellt und bewertet werden.

Folgende Einschränkungen grenzen den Untersuchungsgegenstand ein:

- nur (einheitliche) PKWs, keine Lastwagen, Busse, Anhänger, Motoräder oder Fahrräder
- kein Linksabbiegen bei Gegenverkehr
- nur beampelte Verkehrszusammenflüsse
- nur umlaufbasierte Ampelregelungen
- nur stabile Verkehrsflüsse
- Quellverkehr in das betrachtete System hinein ohne Platoons, reine Exponentialverteilungen der Zwischenankunftszeiten
- kein Fussgängerverkehr
- kein Überholen
- kein adaptives Routing der Verkehrsteilnehmer

Technologische Vorgaben:

- Simulation mit SUMO

Vorgehen:

Grundsätzlich sollten Sie von einfachen zu komplexeren Verkehrsszenarien fortschreiten. Beginnen Sie zum Beispiel mit einer einzigen Kreuzung. Dort können Sie manuell recht einfach eine ordentliche Ampelsteuerung implementieren, da man sich die Auswirkungen verschiedener Umlaufzeiten und Phasen unter verschiedenen Lastszenarien auf die drei obigen Optimierungskriterien noch relativ leicht überlegen kann, unter Berücksichtigung der in der LV vorgestellten Verkehrstheorie. Die evolutionären Algorithmen kann man dann auch entsprechend einfach analysieren.

Dann könnten Sie, mit den gewonnenen Erkenntnissen und einer ersten Implementierung der IT-Infrastruktur (Simulator, Optimierungsalgorithmus, Schnittstelle zwischen beiden, Datensammlung

und Auswertung), zu einer etwas komplexeren Struktur mit mehreren Kreuzungen übergehen (z.B. drei Kreuzungen in Serie). Hier könnte das Hauptziel die Realisierung und die Analyse der Auswirkung einer „grünen Welle“ auf die Optimierungskriterien sein. Zunächst etwa ein Fluss nur in eine Richtung, dann in beide, mit verschiedenen Lastszenarien des die Hauptrichtung querenden Verkehrs etc.

Schliesslich könnten Sie zu einem komplexen Netzwerk (vereinfacht können wir reguläre Gitter wählen) fortschreiten, mit mehreren Verkehrsachsen (mehrere „grüne Wellen“?). Mindestens ein 2x3 (3x3?) Kreuzungsnetz. Auch hier zunächst einmal einen einfachen Fluss (nur geradeaus über die Kreuzungen fahren, Einbahnstrassen) untersuchen, dann Flüsse mit Gegenverkehr, mit Abbiegewahrscheinlichkeiten. Verschiedene Lastszenarien (Haupt- und Nebenverkehrsflüsse) könnten folgen.

Sie sollten bei allen Komplexitätsstufen möglichst drei verschiedene Belastungsstufen wählen: eine leichte (nachts), eine mittlere, und eine hohe. Letztere etwa bei 80%, siehe LV zur Verkehrstheorie. Die Fahrzeugquellen des Verkehrsnetzes sollten mit Zufahrtsraten versehen werden, die nach dem MaxFlow-Algorithmus zu berechnen sind, wie in der LV noch erklärt wird.

Beim Fortschreiten in den Komplexitätsstufen sollte das Verhalten der untersuchten Algorithmen zunehmend klar werden, und damit eine zunehmend rationalere Einstellung der Parameter des jeweiligen Algorithmus.

Bei allem ist auf gute Visualisierung der Ergebnisse zu achten, und vor allem auf Reproduzierbarkeit.

B NSGA-II Description

A FAST ELITIST MULTIOBJECTIVE GENETIC ALGORITHM: NSGA-II

ARAVIND SESHADRI

1. Multi-Objective Optimization Using NSGA-II

NSGA ([5]) is a popular non-domination based genetic algorithm for multi-objective optimization. It is a very effective algorithm but has been generally criticized for its computational complexity, lack of elitism and for choosing the optimal parameter value for sharing parameter σ_{share}. A modified version, NSGA-II ([3]) was developed, which has a better sorting algorithm , incorporates elitism and no sharing parameter needs to be chosen *a priori*. NSGA-II is discussed in detail in this.

2. General Description of NSGA-II

The population is initialized as usual. Once the population in initialized the population is sorted based on non-domination into each front. The first front being completely non-dominant set in the current population and the second front being dominated by the individuals in the first front only and the front goes so on. Each individual in the each front are assigned rank (fitness) values or based on front in which they belong to. Individuals in first front are given a fitness value of 1 and individuals in second are assigned fitness value as 2 and so on.

In addition to fitness value a new parameter called ***crowding distance*** is calculated for each individual. The crowding distance is a measure of how close an individual is to its neighbors. Large average crowding distance will result in better diversity in the population.

Parents are selected from the population by using binary tournament selection based on the rank and crowding distance. An individual is selected in the rank is lesser than the other or if crowding distance is greater than the other [1]. The selected population generates offsprings from crossover and mutation operators, which will be discussed in detail in a later section.

The population with the current population and current offsprings is sorted again based on non-domination and only the best N individuals are selected, where N is the population size. The selection is based on rank and the on crowding distance on the last front.

3. Detailed Description of NSGA-II

3.1. **Population Initialization.** The population is initialized based on the problem range and constraints if any.

[1]Crowding distance is compared only if the rank for both individuals are same

3.2. **Non-Dominated sort.** The initialized population is sorted based on non-domination [2]. The fast sort algorithm [3] is described as below for each

- for each individual p in main population P do the following
  - Initialize $S_p = \emptyset$. This set would contain all the individuals that is being dominated by p.
  - Initialize $n_p = 0$. This would be the number of individuals that dominate p.
  - for each individual q in P
    - if p dominated q then
      - add q to the set S_p i.e. $S_p = S_p \cup \{q\}$
    - else if q dominates p then
      - increment the domination counter for p i.e. $n_p = n_p + 1$
  - if $n_p = 0$ i.e. no individuals dominate p then p belongs to the first front; Set rank of individual p to one i.e $p_{rank} = 1$. Update the first front set by adding p to front one i.e $F_1 = F_1 \cup \{p\}$
- This is carried out for all the individuals in main population P.
- Initialize the front counter to one. $i = 1$
- following is carried out while the i^{th} front is nonempty i.e. $F_i \neq \emptyset$
  - $Q = \emptyset$. The set for storing the individuals for $(i+1)^{th}$ front.
  - for each individual p in front F_i
    - for each individual q in S_p (S_p is the set of individuals dominated by p)
      - $n_q = n_q - 1$, decrement the domination count for individual q.
      - if $n_q = 0$ then none of the individuals in the subsequent fronts would dominate q. Hence set $q_{rank} = i + 1$. Update the set Q with individual q i.e. $Q = Q \cup q$.
  - Increment the front counter by one.
  - Now the set Q is the next front and hence $F_i = Q$.

This algorithm is better than the original NSGA ([5]) since it utilize the information about the set that an individual dominate (S_p) and number of individuals that dominate the individual (n_p).

3.3. **Crowding Distance.** Once the non-dominated sort is complete the crowding distance is assigned. Since the individuals are selected based on rank and crowding distance all the individuals in the population are assigned a crowding distance value. Crowding distance is assigned front wise and comparing the crowding distance between two individuals in different front is meaning less. The crowing distance is calculated as below

- For each front F_i, n is the number of individuals.
  - initialize the distance to be zero for all the individuals i.e. $F_i(d_j) = 0$, where j corresponds to the j^{th} individual in front F_i.
  - for each objective function m
    - Sort the individuals in front F_i based on objective m i.e. $I = sort(F_i, m)$.

[2] An individual is said to dominate another if the objective functions of it is no worse than the other and at least in one of its objective functions it is better than the other

- Assign infinite distance to boundary values for each individual in F_i i.e. $I(d_1) = \infty$ and $I(d_n) = \infty$
- for $k = 2$ to $(n-1)$
  - $I(d_k) = I(d_k) + \dfrac{I(k+1).m - I(k-1).m}{f_m^{max} - f_m^{min}}$
  - $I(k).m$ is the value of the m^{th} objective function of the k^{th} individual in I

The basic idea behind the crowing distance is finding the euclidian distance between each individual in a front based on their m objectives in the m dimensional hyper space. The individuals in the boundary are always selected since they have infinite distance assignment.

3.4. **Selection.** Once the individuals are sorted based on non-domination and with crowding distance assigned, the selection is carried out using a ***crowded-comparison-operator*** ($\prec_n$). The comparison is carried out as below based on

(1) non-domination rank p_{rank} i.e. individuals in front F_i will have their rank as $p_{rank} = i$.
(2) crowding distance $F_i(d_j)$

- $p \prec_n q$ if
  - $p_{rank} < q_{rank}$
  - or if p and q belong to the same front F_i then $F_i(d_p) > F_i(d_q)$ i.e. the crowing distance should be more.

The individuals are selected by using a binary tournament selection with crowed-comparison-operator.

3.5. **Genetic Operators.** Real-coded GA's use ***Simulated Binary Crossover (SBX)*** [2], [1] operator for crossover and ***polynomial mutation*** [2], [4].

3.5.1. *Simulated Binary Crossover.* Simulated binary crossover simulates the binary crossover observed in nature and is give as below.

$$c_{1,k} = \frac{1}{2}[(1-\beta_k)p_{1,k} + (1+\beta_k)p_{2,k}]$$
$$c_{2,k} = \frac{1}{2}[(1+\beta_k)p_{1,k} + (1-\beta_k)p_{2,k}]$$

where $c_{i,k}$ is the i^{th} child with k^{th} component, $p_{i,k}$ is the selected parent and β_k (≥ 0) is a sample from a random number generated having the density

$$p(\beta) = \frac{1}{2}(\eta_c + 1)\beta^{\eta_c}, \quad \text{if } 0 \leq \beta \leq 1$$
$$p(\beta) = \frac{1}{2}(\eta_c + 1)\frac{1}{\beta^{\eta_c+2}}, \quad \text{if } \beta > 1.$$

This distribution can be obtained from a uniformly sampled random number u between $(0, 1)$. η_c is the distribution index for crossover[3]. That is

$$\beta(u) = (2u)^{\frac{1}{(\eta+1)}}$$
$$\beta(u) = \frac{1}{[2(1-u)]^{\frac{1}{(\eta+1)}}}$$

[3]This determine how well spread the children will be from their parents.

3.5.2. *Polynomial Mutation.*

$$c_k = p_k + (p_k^u - p_k^l)\delta_k$$

where c_k is the child and p_k is the parent with p_k^u being the upper bound[4] on the parent component, p_k^l is the lower bound and δ_k is small variation which is calculated from a polynomial distribution by using

$$\delta_k = (2r_k)^{\frac{1}{\eta_m + 1}} - 1, \quad \text{if } r_k < 0.5$$

$$\delta_k = 1 - [2(1 - r_k)]^{\frac{1}{\eta_m + 1}} \quad \text{if } r_k \geq 0.5$$

r_k is an uniformly sampled random number between $(0, 1)$ and η_m is mutation distribution index.

3.6. **Recombination and Selection.** The offspring population is combined with the current generation population and selection is performed to set the individuals of the next generation. Since all the previous and current best individuals are added in the population, elitism is ensured. Population is now sorted based on non-domination. The new generation is filled by each front subsequently until the population size exceeds the current population size. If by adding all the individuals in front F_j the population exceeds N then individuals in front F_j are selected based on their crowding distance in the descending order until the population size is N. And hence the process repeats to generate the subsequent generations.

4. Using the function

Pretty much everything is explained while you execute the code but the main arguments to get the function running are population and number of generations. Once these arguments are entered the user would be prompted for number of objective functions and number of decision variables. Also the range for the decision variables will have to be entered. Once preliminary data is obtained, the user is prompted to modify the `objective function`.

Have fun and feel free to modify the code to suit your need!

References

[1] Hans-Georg Beyer and Kalyanmoy Deb. On Self-Adaptive Features in Real-Parameter Evolutionary Algorithm. *IEEE Trabsactions on Evolutionary Computation*, 5(3):250 – 270, June 2001.

[2] Kalyanmoy Deb and R. B. Agarwal. Simulated Binary Crossover for Continuous Search Space. *Complex Systems*, 9:115 – 148, April 1995.

[3] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A Fast Elitist Multiobjective Genetic Algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182 – 197, April 2002.

[4] M. M. Raghuwanshi and O. G. Kakde. Survey on multiobjective evolutionary and real coded genetic algorithms. In *Proceedings of the 8th Asia Pacific Symposium on Intelligent and Evolutionasy Systems*, pages 150 – 161, 2004.

[5] N. Srinivas and Kalyanmoy Deb. Multiobjective Optimization Using Nondominated Sorting in Genetic Algorithms. *Evolutionary Computation*, 2(3):221 – 248, 1994.

E-mail address: `aravind.seshadri@okstate.edu`

[4]The decision space upper bound and lower bound for that particular component.