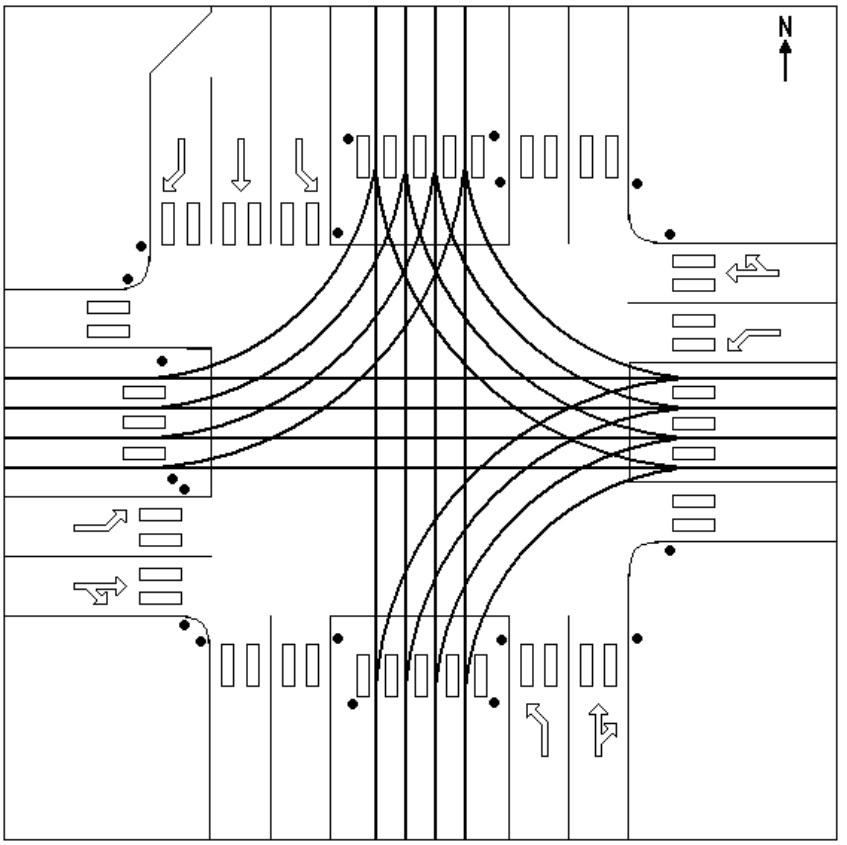
CHALMERS



Traffic Control with Standard Genetic Algorithm

A simulated optimization control of a Traffic Intersection

Master of Science Thesis/ Thesis work in Intelligent Systems Design

GUSTAF JANSSON

Department of Applied Information Technology
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden, 2010
Report No. 2010:127
ISSN: 1651-4769

REPORT NO. 2010/127

Traffic Control with Standard Genetic Algorithm

A simulated optimization control of a Traffic Intersection

GUSTAF JANSSON

Department of Applied Information Technology
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2010

Genetic Algorithm controlled Traffic Intersection
A practical use of Standard Genetic Algorithm for Traffic Intersection control
GUSTAF JANSSON

Master's Thesis report no 2010:127
ISSN: 1651-4769
Department of Applied Information Technology
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Cover: Figure of the traffic intersection, also found on page 12

Reproservice / Department of Applied Information Technology
Göteborg, Sweden 2010

Genetic Algorithm controlled Traffic Intersection
A practical use of Standard Genetic Algorithm for Traffic Intersection control
GUSTAF JANSSON
Department of Applied Information Technology
Chalmers University of Technology

Abstract

In this master's thesis, the possibility to use genetic algorithms to solve real world problem is tested and evaluated. The type of genetic algorithm considered in this thesis is the standard genetic algorithm, and the chosen problem involves traffic control of an intersection with road vehicle, tram and pedestrian traffic. Genetic algorithms are stochastic and biology inspired search techniques mostly used in science related work to find optimal solutions. They are usually very resource taking in terms of CPU time and memory size. A hardware issue, relating to how to implement the genetic algorithms into an embedded system, is also covered.

The major part in this work is concerned with applying genetic algorithms to find optimized scheduling solutions for efficient traffic flow. The traffic intersection is used to illustrate problems where optimal scheduling for drive order is needed, thus the results produced by the genetic algorithm depend on the present situation in the intersection. An additional search method called Depth First Search (DFS) is used to verify the results from the genetic algorithm. The problems with constraint rules affecting this work are also covered.

FPGA and microcontroller are both suitable hardware for genetic algorithm implementation, this work will only cover implementation of the microcontroller. The control of the intersection is assumed to be directly from this hardware implementation, and being totally independent from any outside control system. The traffic intersection, which is not a real existing intersection, is independent from adjacent intersections. All simulation programs acts as controller programs for the intersection. There are three different simulation programs in total, all coded in programming language C.

This work will not present a complete final result, for example a hardware implementation ready to use. It stays on simulation programs only.

Keywords: Standard Genetic Algorithm, Traffic Intersection, Embedded Systems, FPGA, Microcontroller, Depth First Search, Constraint Rules.

Content

Acknowledgement

This master's thesis is carried out in Chalmers and IT University in Gothenburg during 2010.

From my point of view it has been very inspiring to do this work. Several new ideas and perspectives have coming up along this work. More than a dozen millions artificial individuals have passing by on developments and simulations.

Special thanks to my examiner and supervisor to this work Claes Strannegård. I would also thank class and family members for additional supports.

1 Introduction

The aim of this master's thesis work is twofold. The first is to find a way to implement a genetic algorithm (GA), into an embedded system. The second and the major part, is to solve a real world problem with this GA. This problem is to find good traffic flow in the traffic intersection crossed with road vehicles, trams and pedestrians.

The specific GA used in this work is a Standard GA. GAs is an adaptive and efficient heuristics that are able to solve optimization problems. This is a stochastic search technique to look for optimal solution. Most GA is used in research and science related work to look for optimal solutions. They usually run on powerful computers as GAs generally are resources taking in term of CPU time and memory size. Some methods GA uses are selection, crossover and mutation inspired from evolution in the real nature. A GA produced solutions comes out from one of many artificial individuals that contain the highest fitness value. Out from this individual a hopefully good to perfect answer of the specified problem can be found.

The main purpose of this work is to test if a GA can be used out in the field. To do this, it will be applied and tested on a reasonably simple problem. This work also includes finding simpler hardware for implementation. This traffic intersection comes into illustrate a real world problem to work with. Optimal scheduling will be the work method to produce drive order sequence depending up on the present situation.

There are three simulation programs in total; computed Standard GA, microcontrolled Standard GA and a computed DFS version. The last one use depth first search (DFS) algorithm. The DFS will be used to find the real answer in the search space. This answer will then be compared with the results from both the other Standard GAs simulations. All simulation programs are coded in programming language C. The embedded system used during this work is a microcontroller that must use C language.

In terms of controlling the intersection with GA, only independent control from hardware implementation will be considered in this work. No other outside control system will be considered. The intersection will also be assessed in isolation, meaning, not considering adjacent intersections.

2 Description of Standard GA

The specific kind of GA used throughout this work is a Standard GA [Wahde, 2008]. A Standard GA is one type of different GAs. In a big view GAs is one under group to the term of evolutionary algorithms. In general GAs is a search algorithm based on the natural selection and genetics [Goldberg, 1989]. It uses a number of artificial individuals looking through a complex search space by using functions of selection, crossover and mutation. The purpose to use GA is searching and finding optimal or good enough solution. This solution will hide in a big search space to look through. Is no guaranty to find any exact solutions when using a GA. Some result can even be far from optimal when GA gets stuck in so called local optimum in the search space.

2.1 Standard GA description

This general description includes how GA works and special the Standard GA. Some detailed functionality for the simulation programs is also included. The Standard GA starts by initial a population with certain number of individuals to work on. Each individual consist a number of chromosomes depending on the problem to solve. Each chromosome consists of certain number of genes (see Figure 2.1). The genes are binary represented in the chromosome and they are decoded to get out a special parameter value to working on.

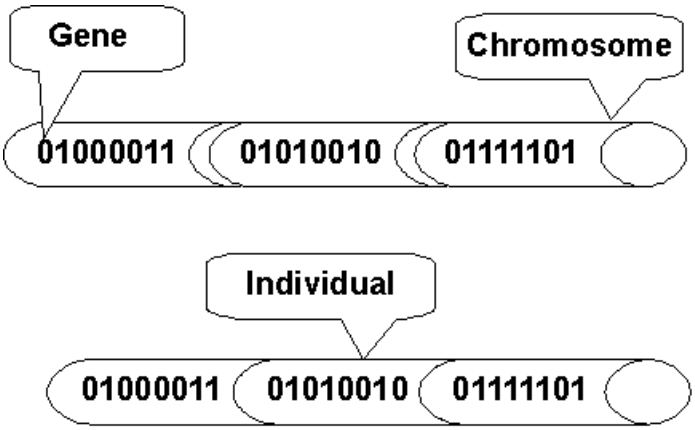


Figure 2.1: One individual with 24 genes and three chromosomes

When entire individuals are decoded, all results from each chromosome are evaluated for fitness calculation. After decoding and evaluating all individuals from entire population, the best individual is compared to the best one so far. If this new individual is better than the previous best individual, it will be saved to the next generation and entitled as the new best one so far. What is considered the fittest one depends on the problem defined by user. One example can be a two variable function when the best result is the minimum value of the function. In this case the best result is the smallest and (the result need to be inverted by *fitness=1/y*). In this work, only maximum values are considered. Before the next generation is produced, three procedures will take place

to create a new population including selection, crossover and mutation. These are described in more detail further down and Figure 2.2 shows the Standard GA work flow. If a best or good enough individual is found, the Standard GA can terminate. In case of when the maximum number of a generation run out without finding a best or good enough solution, the best individual that is found will be presented as the solution.

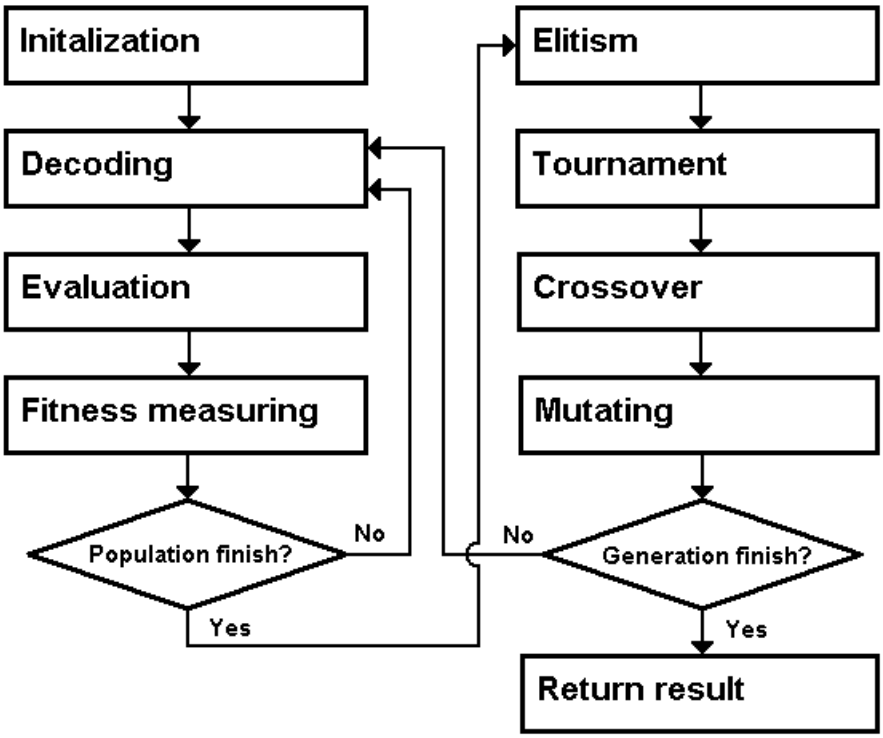


Figure 2.2: Work flow over Standard GA

The program starts with initialization and then beginning the first generation in a loop by decoding, evaluation and fitness measure. After that loop is finish a new population are created by first do elitism and there after selection by tournament. Last following two procedures is crossover and mutation before next generation begins. The following descriptions for used Standard GA have source code in Appendix G.

2.1.1 Initialization

Initialization involves settings the parameters for the algorithm, set up input values for the simulation, and creating population of individuals to the first generation. The most basic set up are the number of genes and chromosomes, the number of generations as well as the number of individuals per population. The number of individuals is preferably even numbered. Other adjustable settings for any GA are the probabilities of crossover, mutation and tournament. These floating variable values need smaller changes when improving the performance of the GA. Other variables are to be described in each part where they belong. Array *population* is given random values 0 and 1 to each gene and that makes binary values to each chromosome. A gene can for an example have other values for example natural number representing something as cities

in the Travelling Salesman Problem (TSP) [Stuart & Norvig, 2003].

A special vector in this program called *inputValue* containing "real world sensor inputs" from the intersection to simulate on. Array *inputValues* have up to eight stored inputs for eight future time periods for simulations. All simulations do ten time periods, where the last two time periods are input free.

2.1.2 Decoding

After initialization, the program enters the main loop of Standard GA, commencing the decoding of the chromosomes. The decoding is user definable. For example, one chromosome can represent many different values, as demonstrated in Figure 2.3.

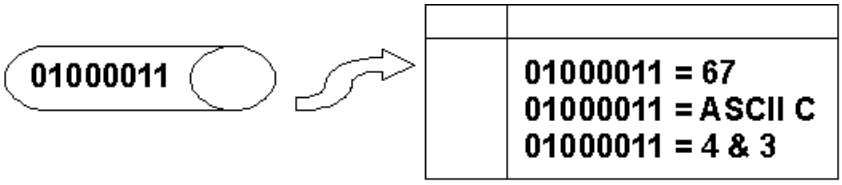


Figure 2.3: Three different ways to decode one binary chromosome

In this program each chromosome have four binary genes representing binary number from 0 to 15, providing 16 separate solutions representing drive orders for the intersection (see Appendix A). The arguments *population* and *i*, identifies certain individuals to read from. The number of genes and chromosomes, and actual chromosome order, tells where to read on the actual individual. The first gene on the chromosome is the Least Significant Bit (LSB) and the resulting number is returned and stored in variable called *parameterValue(chromosome number)*.

2.1.3 Evaluation

The purpose of evaluation is to determine the fitness of each individual in a generation. The evaluation function applies a mathematical function to calculate the fitness values (see example in Figure 2.4). Each calculated fitness value will be returned from the function called EvaluateIndividual. The arguments to the function contain chromosome values, input vector and the previous drive order.

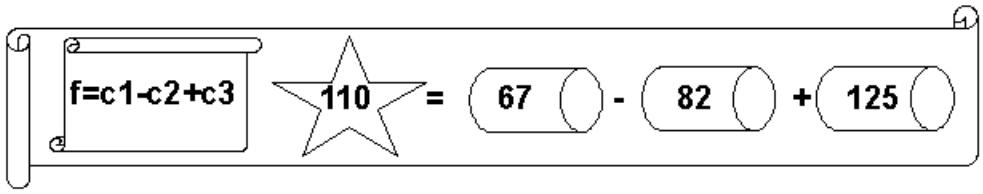


Figure 2.4: Fitness calculation example from individual with three chromosomes

The fitness calculation process is the same for all programs, and uses three calculation stages. Evaluation of present situation at the intersection are calculated according to inputs sensors from the site, older inputs with added priority and the previous drive order that occurred.

The first calculation stage, are summary points $c_{nInputs}$ from new and old inputs by mathematical functions (see Appendix B). These values are retrieved from *inputValue* containing input values from the intersection. The values range from 0 to 6, where 0 is non-requesting sensor, 1 is new input and other values are older remaining requests. The input is set to zero in the main code when the equal output, of that input, is known to be executed. In simulation view will be a road vehicle that have driven away and then left the sensor. Also in main code is the priority boost for non-executed requests, is done by adding value one until next time period.

The second calculation stage is dependent on the previous drive order, variable *previousExcecution* and a set of rules, listed in Appendix D. The propositional rules only tell what previous drive orders that will give extra credits of two points or not by s_n. The purpose is to favour certain sequence of drive orders that are good for the traffic flow. On every simulation start are *previousExcecution* equal to value 16 that is a non-valid zero credit drive order. In case of same drive order repeats itself, a penalty calculation will be used by subtract minus five points to avoid repetition.

The last calculation stage summarise all chromosome points by following equation:

$$f = \sum_{n=1}^{6} (7-n)(c_{nInputs} + s_n)$$

This function is the actual fitness calculation returned by the evaluation function. Each chromosome c_n is a summary from the previous two stage calculations. Example of chromosome two is $c_2 = 5(c_{2Inputs}+s_2)$. The aim for the solution is to execute many requests earliest as possible. So each chromosome is multiplied by a number from six and down, to favour higher points on the early time periods rather the last ones.

2.1.4 Fitness function

After an individual has been decoded and evaluated it has been given a fitness value. If this value is more (not equal) than the present maximum fitness value, this fitness value will become the new maximum fitness value. In addition to updating the maximum fitness value in variable *maxFitness*, each parameter value from this individual is also stored under the six variables *bestParametervalue(chromosome number)*.

2.1.5 Elitism

When all individuals have been evaluated, the one with the highest fitness value is stored unchanged in the first row in the next population array. This is done because the following procedures for the other individuals are going through tournament selection,

crossover and mutation. The last two will destroy or hopefully improve current individuals in the next generation.

2.1.6 Tournament selection

At the beginning of generating the next population, all individuals are taken through a selection procedure called tournament selection. In this procedure, individuals are compared against each other in a tournament where individuals with the highest fitness value have a higher probability to be selected. The tournament probability is a fixed value in variable *tournamentProbability*. In this program 30 individuals are taken through the selection, two individuals per tournament, resulting in 15 tournaments between randomly selected individuals.

2.1.7 Crossover

Crossover and mutation are the two procedures that bring new evolutionary material for the GA to work on. In the crossover procedure two individuals are chosen from the population to create new offspring. This is achieved by choosing one or two randomly chosen crossover point (one in this program) along the bit strings. The crossover points indicate where exchange of values between the individual are to occur to create the new individuals. In this case, the first section (before the crossover point) of the bit string of a new offspring of an individual remains unchanged while the end section (after the crossover point) exchanges values with the other individual. When a second crossover point is used the remaining third part is from same individual again (see Figure 2.5). The probability to do crossovers is defined by variable *crossoverProbability* which let most of all individuals to undertake the crossover procedure.

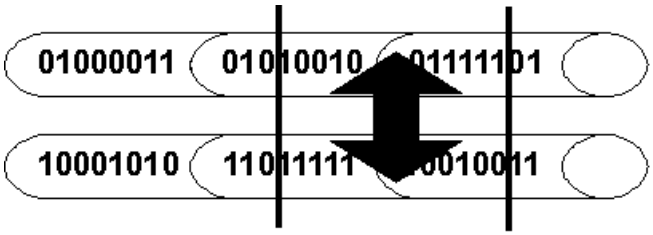


Figure 2.5: Example of crossover by two crossover points

2.1.8 Mutation

The mutation operation makes small changes to an individual and is used to maintain genetic diversity from one generation to next. In this work, the mutation operation involves generating a random variable for each bit in a sequence. This random variable *mutationProbability* tells whether or not a particular bit will be modified. Therefore, mutations might or might not have an effect on an individual (as shown in Figure 2.6). When a very low mutation probability is used, then this operation becomes rare. In the long-run, mutation brings new and more different individuals that hopefully give better fitness value.

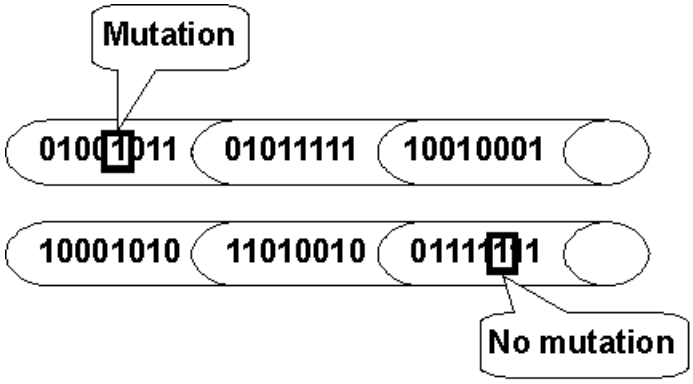


Figure 2.6: Mutation on first individual and no mutation on the other one

2.2 Program details for Standard GA

This section will outline a general description how this Standard GA works in the simulation program. The settings are the same in all Standard GA simulations except for the microcontroller (see Chapter 6.2).

Number of genes in one individual are 24 divided on six chromosomes gives binary value to decoding on each from 0 to 15.

Each population have 30 individuals and new population is created to every generation that is maximum 100 for one time period simulation. So every generation do 30 evaluations multiplied with 100 generations will be 3000 evaluated individuals. But because of elitism is used approx 29 unique individuals are expected per population. Settings for crossover probability are 85%, mutation probability 4% and tournament probability 90%.

The Standard GA is a main part inside the loop that re-run every time period. Before the loop starts, an initiation will take place and the *inputValue*, for the present time period to be simulated, brought in. Following this, the actual Standard GA loop will start running, working according to the description in Chapter 2.1. When finished, the best individual is to be stored in a specific array called *goodIndividuals* for future use. This is a kind of elitism, when saving individuals unchanged after each complete Standard GA run. In order to store an individual to *goodIndividuals* array, the best individual's *maxFitness* needs to be higher than *totalMaxFitness* from the entire ongoing simulation. This array will then contain some of the best previous individuals which are copied to array *population* in every new Standard GA loop. Reusing previously identified good solution for future needs allows for faster and improved results. In that way the simulation program is self-learning.

In the simulations there are up to eight sets of inputs and the program runs the Standard GA ten times in total. After some tests of worst case scenarios, it was concluded that at least six chromosomes are needed per individual to easily read enough predicted drive orders. Predicted drive orders can be represented as a forecast over possible upcoming drive orders. This forecast is based upon the present situation. When the actual time

period has been executed, the Standard GA will start searching for a solution for the following time period. Depending on the inputs to the new time period, this new solution may or may not be equal to the previous predicted one for this actual time period. The result is presented in a symbolic way by printout on the screen only (see Appendix E and Appendix F). The final code for this description, and belonging simulation inputs of *Worst case one*, are included in Appendix G.

2.3 Search space

Both the Standard GA and the DFS version (described in Chapter 4.2) are working on a search space containing large numbers of possible solutions. Due to rule problems (Chapter 5), a limited search space is use in this work. This search space is the same for all simulations. The number of possible solutions for this search space, when looking at a six time period into the future, is up to $1.7 \cdot 10^7$ solutions. This equals the number of bottom nodes on the search tree (see Figure 2.7) resembling an upside down tree, where each node grooving with 16 branches. This figure illustrates one simulation by DFS with previous executed drive order 12. Because the previous executed drive order needs to be included in the evaluation calculation, the total search space is actually 16 times bigger, giving $2.7 \cdot 10^8$ solutions. Only one of these 16 is used for each time period simulation, depending on the previous executed one.

Note that all simulations starts from non-valid drive order 16. This simulated search space O is calculated by six levels deep d and each of all nodes have 16 branches b. That makes a six levels search tree on following equation:

$O(b^d)$, $b = 16$, $d = 6$, $16^6 \approx 1.7 \cdot 10^7$

This concludes to 17 millions of possible solutions to look through. The Standard GA will only conduct up to 3000 evaluations out of this search space.

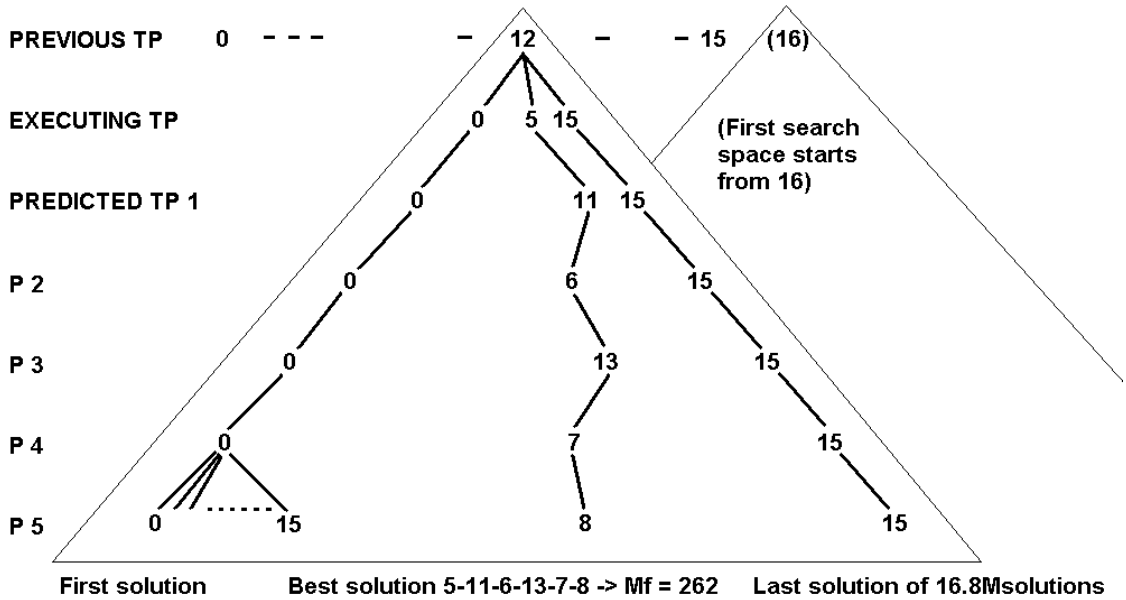


Figure 2.7: Search space for all simulations

3 Traffic intersection

The purpose of this work is to assess the potential of applying GA to solve a real world problem. In this work, it is represented by a traffic intersection. This choice is based on the basis the function and purpose of an intersection is easily understood and still provides an advanced enough problem to work on. Typically intersections equal to this, with limited space and tramways, are found in large cities.

The purpose is to find an optimal schedule for optimal drive order sequence. The time duration of the traffic lights is supposed to be fixed. The assumption maid in this work, is the use of a controller unit with implemented hardware in form of an embedded system, contains the Standard GA program controlling the intersection in the reality. Inputs to the controller are all sensors and push button around the intersection and the output is the actual light poles. The actual traffic lights are not printed on Figure 3.1 but the output variables are mentioned in Appendix A and listed in Appendix C. The location of the input sensors are, for pedestrians the push buttons (black dots), for road vehicles the sensors under the road lane edge where they stops for red lights (under the arrows), and for trams just before the turn out on tram line where they usually stop for there red lights.

There are several works that have provided inspiration to this work with GA and traffic controlling. One example is [Guan, et al., 2008], a paper discussing signal time optimization tested on 30 intersections in Changchun city. Dynamic signal control is also used in [Yang, et al., 2006], a paper about one isolated intersections only. A different but inspiriting paper is traffic flow for lane closures to minimize travel delay [Ma, et al., 2004]. In another paper with dynamic traffic lights time control for pedestrian and passing road vehicles on one intersection [Turky, et al., 2009]. But almost all of these are outside the intention to do scheduling by drive orders. Additional inspiration for scheduling has been drawn from paper [Fissgus].

3.1 Description of the intersection

The structure of the intersection is based on general intersection that can be found in any city around the world. In general this is a four-way intersection from north to south and west to east (see Figure 3.1).

Some traffic rules applied on this intersection are as follow. Road vehicles can turn in any direction except back to where they come from, so U-turns are not allowed in this test. Road vehicles turning left or right have to give way for pedestrians at pedestrian crossings. Both road vehicles and pedestrians are to go northbound and southbound at the same time, or in the same manner, westbound and eastbound traffic at the same time. The tram-way travels in all directions except turning between west and south, assuming this curve is not required in the intersection. The north and south going street is considered to be the main road, carrying the majority of the traffic load. The street to the west has moderate traffic and slightly more tram traffic. Finally, the small street to the east, has a low traffic load but trams are more frequent. All four streets have pedestrian crossings, linking the pedestrian paths between the pavements on either side of each

street. Due to heavy traffic and the width of the streets, three of the crossings are divided up into three separate sections (road vehicle lane, tram lane, road vehicle lane). Two push buttons are provided for each section.

The intersection will not be dependent on other adjacent intersections nearby. This scenario could be assessed in a potential extension of this work. The Standard GA is only to solve optimal drive order sequences for this intersection.

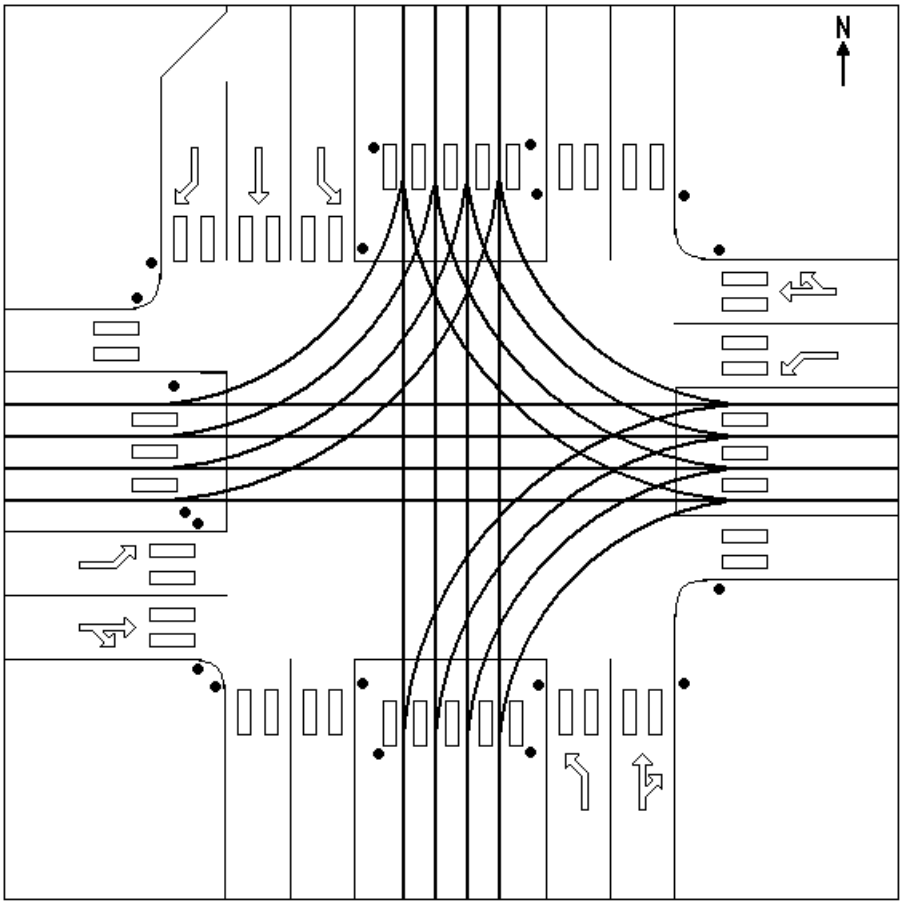


Figure 3.1: Image over the intersection

3.2 General functionality

The source code contains all the inputs and outputs variables represented by the intersection. The inputs are retrieved from *inputValues* arrays and the outputs are printed on the screen. All drive orders allow certain pedestrian, trams and road vehicles to go safely. Most importantly, to avoid accidents, 16 prepared drive orders have been assigned as safe output solutions (see Appendix A).

Each sensor input can be seen as a request from the activator that gives a numerical value one. A request needs to be executed, by its equal output, through an appropriate drive order. Outputs have only a symbolic function in the code, although it can be seen as Boolean representations *true* for execute otherwise *false*. Note that red and green lights are dependent on the executing drive order.

Sets of mathematical functions and logic rules evaluate each drive order to know how valuable they are to be executed. Drive order that executing many requests is better than less executing ones. Executed requests get there input value set to zero. For each new time period, outputs for executing the requests are always coming from chromosome number one c_1. Non-executed requests will get their value added by one to get higher priority. This process provides a better chance to get higher fitness value in the next time period. The mathematical functions are listed in Appendix B, additional logic rules in Appendix D and outputs according to certain drive order in Appendix C. Further down is a list of letters that form the variable names of inputs requests and their equal outputs. An example, abbreviations meaning road vehicles going from south to north and east would be called *RSNE*. Output variable name of this will then be O_{RSNE}, and corresponding input request variable I_{RSNE}.

First letter:	I for input
	O for output
Second letter:	P for pedestrians
	T for trams
	R for road vehicles
Third letter:	N from northbound direction
	S from southbound direction
	W from westbound direction
	E from eastbound direction
Forth letter:	N to northbound direction
	S to southbound direction
	W to westbound direction
	E to eastbound direction
Fifth letter:	W also to westbound direction
	E also to eastbound direction

3.3 Easy working example

The following example scenario has been created to further describe the functionality. This case has one of each pedestrian, tram and road vehicle request. Entire simulation takes three time periods to complete.

The pedestrian request is *PW1* on the north-west pavement. A push button acts as sensor input and inputValue I_{PW1} equals *true*. By then is this pedestrian lane requested. This pedestrian lane will cause interference to the tram eventually. Pedestrian lane *PW2* might be activated there after.

Assume a tram standing from east activating sensor I_{TEW}, tempting to go west. Also assuming that the tram driver somehow can give a direction order, depending on where the tram route going. Only one direction is possible to request.

On road lane *RSNE*, a car coming to stop for red light and sensor I_{RSNE} is activated. How many other road vehicles standing behind on this lane doesn't matter. There is nothing that tells where this road vehicles going north, east or both, they are allowed for both. Although the *RSNE* requests do interfering the trams request anyway.

This beginning situation is shown on Figure 3.2. The car and the pedestrian do not interfering each others, the tram do. Only the tram or both pedestrian and the car can go on first time period. Two solutions are possible by resulting sequences 8-10-15 and 10-8-15. Most fitness do 8-10-15 have where the tram go first on drive order 8. Output variable O_{TEW} sets to *true*, equals to execute the tram request. The second predicted drive order 10 will be next executable, where both O_{PW1} and O_{RSNE} are set to *true*. In theory the pedestrian needs to activate I_{PW2} although is not necessary. The logical sequence rule *R15* in Appendix D giving credits when drive order 15 comes after 10. No request for *PW2* is needed due this becomes executed by drive order 15 anyway.

The second solution 10-8-15 is an example of a result not as good as what could be expected. The tram needs to wait and there are no extra credits for previous drive order. For this simple example it doesn't matter who goes first. Otherwise, it's more crucial in heavier traffic situations. The real simulation result of this *Example case* is shown in Chapter 7.2.

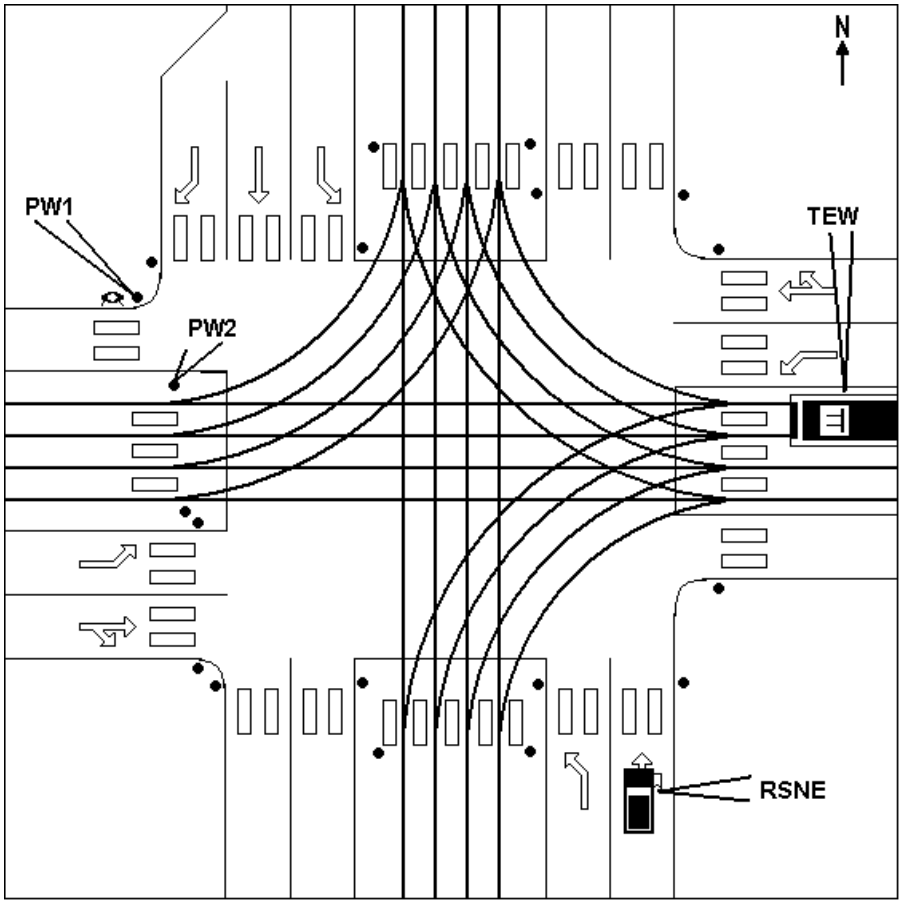


Figure 3.2: Beginning of *Example case*

Table 3.3: Two different fitness calculations on each drive order sequence

Tram first	E	P1	P2	fitness
T0	**8**	**10**	**15**	
f=c1+c2+c3	6	40	8	54
T1	**10**	**15**	**8**	
f=c1+c2+c3	48	10	0	58

P & R first	E	P1	P2	fitness
T0	**10**	**8**	**15**	
f=c1+c2+c3	18	10	0	28

A close look in Table 3.3 and a calculation example following. This table show the second solution sequence 10-8-15. Why it's second best is simple proven by verifying the fitness results 28 against 54. Note that, in both solutions, drive order 10 have higher value than 8. This does not matter when the total fitness value counts for the final result.

In the tables, on *T1* is executable *E* drive order 10. Chromosome one c_1 have a total value 48, this value is calculated further down. Note the previously drive order on *T0* is 8. First calculation is to summarise inputs belonging drive order 10, in following equation (see Appendix B):

$$c_{1Inputs} = I_{PN2} + I_{PW1} + I_{PW2} + I_{PW3} + I_{PS2} + I_{PE} + I_{RNW} + I_{RNS} + 2I_{RSNE}$$

Note that the *RSNE* is weighted with multiplication by two. Both *PW1* and *RSNE* have value one from previous time period. Until this time period their input values are added by priority with one more point as follow:

$$c_{1Inputs} = 0 + 2 + 0 + 0 + 0 + 0 + 0 + 0 + 2 \cdot 2 = 6$$

Second calculation may give a credit of two points. Is depends on logic rules for drive order sequence from Appendix D. The drive order 10 has following sequence rule:

$$R10 \quad P_{10} \wedge (E_{P8} \vee E_{P9}) \rightarrow A_{TwoPoints}$$

This give in Boolean representation due is *true* that previously drive order was 8:

$$R10 \quad true \wedge (true \vee false) \rightarrow true$$

As the result is Boolean $A_{TwoPoints} = true$, that is equal to $s_1 = 2$ in the evaluation function. This give the first chromosome $c_1 = 6(c_{1Inputs}+s_1) = 6(6+2) = 48$. The final fitness functions summarise all values by equation:

$$f = \sum_{n=1}^{6} (7-n)(c_{nInputs}+s_n)$$

The second chromosome have no more requests left and $c_{2Inputs} = 0$, previous drive order was 10 gives $s_2 = 2$. Total chromosome value is $c_2 = 5(c_{2Inputs}+s_2) = 10$. Then the total

fitness value for *T1* is $f = 48+10 = 58$.

The outputs for drive order 10 when it executes are:

$$E_{10} \rightarrow O_{PN2} \wedge O_{PW1} \wedge O_{PW2} \wedge O_{PW3} \wedge O_{PS2} \wedge O_{PE} \wedge O_{RNW} \wedge O_{RNS} \wedge O_{RSNE}$$

As all of these output variables are set to *true*, the pedestrian from *PW1* can continue walking directly on to *PW2* as this too, has green light.

3.4 Details and modifications

Throughout the testing process, a few adjustments have been made to improve the program. These adjustments involve weighting of inputs variables in the evaluation function. The reason for the weighting is to favour some of the rarely used drive orders. The most used drive orders include 7, 8, 12 and 13, who are often presented by the Standard GA, and therefore the most strategic drive order used. One reason for this is due to the fact that some trams and road vehicles requests are harder to execute. For example, tram O_{TEW} (two cases possible) and road vehicles O_{RSW} (only one case possible). Another reason is the amount of pedestrian inputs, as these accumulate many points by being superior in numbers.

To adjust for these rarely used drive orders, some variables get weighted by multiplying with a certain number. They get higher probability to be used as smarter options in low traffic situations. Further down is an example of more heavy multiplied inputs values to get a chance to be selected. Note that tram I_{TEN} and I_{TNE} are multiplied with four which is the highest multiplied value used. Following equation is an example for drive order 6:

$$c_{nInputs} = 4I_{TNE} + 4I_{TEN} + I_{PN3} + I_{PW1} + I_{PW2} + I_{PW3} + I_{PS2} + I_{PS3} + I_{RNW} + I_{RNS} + I_{RNE}$$

4 Programs

The programs that simulate the intersection (computed and microcontroller implemented), both uses Standard GA to find the optimal drive order to execute. In this work, an additional program with another search algorithm, Depth First Search (DFS) [Stewart & Norvig, 2003], is used to verify the performance of both Standard GAs. The specific simulation program used is called "DFS version". The results from the DFS version will be compared to the results derived from each Standard GA program.

The DFS only looks for one solution at the time and goes down all levels in the search tree. The DFS can be programmed to terminate the search if a good solution is found. In this work, the DFS version will search trough the entire search space to find the best answer (a complete path to bottom level six). This path, representing the drive order sequence, is illustrated in Figure 2.7.

In summary the main characteristics of each search method are:

- The Standard GA (computed and microcontroller implemented) is fast, although cannot guarantee a best solution is identified. Otherwise, a good enough solution can be found.

- The DFS version will find the best solution to the cost of extensive use of time and computation.

In addition to these three simulation programs there are four more versions of simulations to evaluate. All simulations are explained in Chapter 7 and they are called *Example case* (from Chapter 3.3), *Normal case*, *Worst case one* and *Worst case two*. Listed source codes are Standard GA version of *Worst case one* on Appendix G and DFS version of *Normal case* on Appendix H. All programs are coded in program language C.

4.1 Standard GA versions

The following description of the Standard GA version is for both computed and microcontrolled one. Functions used in the program include DecodeChromosome, EvaluateIndividual, ReturnResult, TournamentSelect and RandFunktion.

- DecodeChromosome returning the numerical value of a chromosome.

- EvaluateIndividual calculates fitness value for incoming parameters values.

- ReturnResult is called after each completed GA loop to list the best parameters value results. As this is a simulation program, the outputs are only represented in a symbolic expression.

- TournamentSelect doing the tournaments.

- RandFunktion is called from several places in the main code whenever a random value is needed. A good random function is essential for all GAs to work properly.

In main, the initializations are done in the beginning. For simulation reasons, up to eight sets of input values for each time period are used. The main loop runs ten times, starting with reading and updating present input values for that time period. Then the Standard GA loop goes through all generations up to the numbered value of maxGeneration. Inside the Standard GA loop is the traditional work flow with decoding, evaluation, fitness measuring, elitism, tournament select, crossover and mutation. In the end is a special array list goodIndividuals that store the best individual for future time periods. The next new population will get the content from this list. In the long run, this list will contain better and more improved individuals. This list is a part of the self-learning process in this program. When the loop is completed, all best parameter values are send to function ReturnResult for print out.

In the simulations, the program is run through ten time periods in a so called “for loop”. In reality, where the program is supposed to run continuously a “while loop” will be used.

4.2 DFS version

This DFS version is smaller than the Standard GA version as many of the GA procedures are then not required. The code is listed in Appendix H. The only functions used are EvaluateIndividual and ReturnResult equal to the same functions in the Standard GA version. In main the usual initialization, input values and main loop are almost the same as for the Standard GA version. Instead of a GA loop there is a DFS search with “nested for loops” running through the entire search space. This program goes through search down seven levels in the search space, where the last six represent the solution. The first level only contains one start node, that value is the previous drive order. The second level will hold the executable drive order. Finally the third and lower levels are predicted drive orders (see Figure 2.7). The size of the search space is calculated in Chapter 2.3.

When the first search is completed the result is printed through ReturnResult. Following this, the second simulation loop commences to search for second executable drive orders until the next time period. There is no guarantee that the previous predicted drive order will be the next executable one. The program is finished when all ten time periods are simulated. This program will find the optimal solution with cost of time. And time consumption is a major drawback in certain application.

5 Rule problems

This chapter discuss experiences encountered during this work related to rules and how they can work against the GA. The rules considered in this work are ”fitness measuring rules” and “constraint rules”. They are represented with propositional logic. Both are supposed to be applied in the evaluation function to grade individuals according to these rules. This chapter explains the constraint rules and the problem they cause.

Constraint rules are strict and must be fulfilled. Situation may occur where individuals break constrained rules. When this happens these individuals may not be allowed to survive and, to discard them, their fitness values are simply set to zero.

To exclude or forbid individuals and their solutions could be considered wasted, both of computing time and other system resources, as these solutions are not needed. Constrained relations are mentioned in [Goldberg, 1989] where it is explained that constrains can be fine, although finding a feasible solution can be almost as difficult as finding the best one. More reading and details in this area is found on [Yoon]. In science research constrains are generally not causing any major problems. They generally have sufficient computation as the importance of finding good results are essential.

Although constraint rules do not appear to be a big issue, they do cause a certain problem in this work. Significant time loss and memory usage are to be expected on the small, computerised system used in this work. Theoretically constraint rules may not be needed in a well-defined problem. More often than not they cannot be avoided. Although a few constraint rules will have little effect, they may cause a real problem in finding decent solutions if they become more in numbers.

5.1 Example of rule problems

Imagine an individual with one chromosome *c* containing three binary represented genes. That gives a search space of eight possible solutions. The binary genes are called g_1, g_2 and g_3. Assume that if the combination of g_2 and g_3 is *true* and the g_1 value doesn't matter, then (x11) will result in high fitness value. Also assume a constraint rule declaring that if g_1 is *true* (1xx) then that individual must be forbidden. The outcome will be a half population being forbidden. If GA finds an individual with content (111), then that produced result will be wasted. Otherwise content (011) is not and can be used.

Assume the change of the constrained rule to only forbid when g_1 is *true* and g_2 is *false* (10x). By this rule two solutions are gained back from the search space (110) and (111). Left is (100) and (101) which represent 25% of the search space. The loss of possible solutions caused by constrained rules depends on their amount and formulation.

A Standard GA, working with both fitness rules and constrained rules, was used in early attempt in this work. Fitness and constrained rules were both propositional logic, representing *true* or *false* values. They also shared and worked with the same input variables. These variables were representing all inputs from the intersection and represented by one single chromosome containing 29 genes. The search space had about

$2^{29} \approx 5.4 \cdot 10^7$ different solutions as a maximum. The outcome from this attempt resulted in a serious loss of solutions even were only one set of constrained rules is used. Following example demonstrates how rule *R1* forbid a solution when it becomes satisfied by *true* value:

$R1 = I_{PN1} \wedge (I_{RNW} \vee I_{RNE} \vee I_{RNS})$

To pass a non forbidden solution the rule *R1* must be unsatisfied by *false*. This rule is satisfied when gene one I_{PN1} is *true* and at least one of the other genes 20 (I_{RNW}), 21 (I_{RNE}) and 22 (I_{RNS}) are *true*. All of the three last genes must be *false* to ensure a safe traffic situation. The rule *R1* is likely to be satisfied when I_{PN1} is *true* and at least one or more of the other genes are *true* (or holding value 1) in the following calculation:

$$O_{ok} = \frac{I_{PW1}}{2} \cdot \left(\frac{\bar{I}_{RNW}}{2} + \frac{\bar{I}_{RNS}}{4} + \frac{\bar{I}_{RNE}}{8}\right) = \frac{\frac{1}{2} \cdot 7}{8} \approx 0.44$$

Approximately 44% of the search space *O* will be satisfied and therefore forbidden. If the last three genes are not included in this rule, the chance will be just 50%. The last genes actually give back 6% of the search space, increasing possible solution to 56%. To have in mind is that this example contains the first rule of 29 in total. Together they drop the chances by finding any descent solution to almost nothing.

The following work will use another strategy to avoid all forms of constrained rules, with the one exception of a single “if statement” that acts as constrained rule. This statement avoids chromosome c_1 to get value zero. If that happens, the first drive order will not execute any requests and that would be a pointless solution.

5.2 Solutions for this problem

A preferable method to avoid problems is to ensure that all the possible solutions that the GA brings up are without any conflict, or at least keep the amount of constrained rules to a minimum. The only rules that should exist are fitness measuring rules that increase the fitness value. A drawback with removing constrained rules is less optimal formulation of the problem to solve. That is, the freedom to really get into the problem might be limited and the optimal solution may become unreachable. Although a good enough solution can still be found. In this work constraints are removed with 16 prepared solutions. This appear not so many to work with, otherwise in a sequence of six they instead become vast in number of combinations.

6 Embedded solutions

One part of this work involves working out how GA can be implemented into an embedded system. Two main types of embedded systems, FPGA and microcontrollers, have been considered. This hardware implementation is one fundamental part of this work. In reality it would be unrealistic for a control unit or relay box, used to control the traffic intersection, to contain an ordinary computer. It would be more reasonable to have an electronic circuit board with one of the two mentioned embedded systems. The two main reasons for this include component costs and more secure functionality.

The aim in this work is to test the potential to use GA in this type of alternative application. Though the ambition is to make it work, it might not be achievable due to CPU and memory limitations of the GA. Due to time constraints and necessary resources not being available, FPGAs will not be tested in this work, instead some earlier work will be presented. On other hand, a microcontroller and belonging develop kit [Atmel STK500] is available. A drawback with the microcontroller is that its memory size potentially is too small for what is required for this application. Otherwise, with a few modifications, it can still achieve some success. The microcontroller uses programming language C, thus all simulation programs will be coded in C from the beginning.

6.1 FPGA

FPGA (Field Programmable Gate Array) is a versatile integrated circuit that can be designed for specific functionalities. The most commonly used program language is VHDL (Very high speed integrated circuit Hardware Description Language), for more readings see [VHDL].

Some earlier work has been done to implement GA into FPGA [Aporntewan & Chongstitvatana, 2001] and [Shackleford, et al., 2001]. In [Aporntewan & Chongstitvatana, 2001] a speed comparison was done using Compact GA running as software on 200 MHz computer and the 20 MHz FPGA hardware. The result in this study was that the hardware version was 1000 times faster than software version. A similar comparison [Shackleford, et al., 2001] used 1 MHz FPGA and 100 MHz computer. The FPGA was more than 2200 times faster. Both studies show faster and more effective results by the hardware. A drawback is when reconfiguration of the fitness function is required, as that can take hours to do.

Though the resulting effectiveness appears promising, it is a different story when comes to computers. The work stations used in these studies, around 2000, was up to 200 MHz and this work is done in 2010 using a computer with 2 GHz. Although the clock speed on FPGAs has also improved during the years. For more reading about FPGA see manufactures [Xilinx] and [Altera].

6.2 Microcontroller

A microcontroller can be explained as a small computer inside one single integrated circuit. The main components include processor, memory and programmable input and output peripherals. The microcontroller used in this work is an ATmega16 [Atmel Atmega16], programmed through a starter development kit system, called STK500 [Atmel STK500] from Atmel [Atmel]. With a program memory of 16 Kbytes it becomes difficult to fit in a reasonable simulation program. The simulation program DFS version (described in Chapter 4) will not be implemented in a microcontroller, as it is not needed. It would be possible to do so, although the result would be the same as the computed one. In Figure 6.1 is a picture of the entire set up for the simulations. Figure 6.2 illustrates a close up picture of the presented results.

The Standard GA for the microcontroller will require some smaller program changes in order to work properly. Small modifications are made in random and print out functions. Especially in the print out function as there is no computer screen to show the results on, only a LCD display. The population is reduced from 30 to four individuals as a direct consequence of the smaller memory size. Both the computed and the microcontrolled Standard GA will perform 3000 individuals per simulated time period. The microcontroller will compensate with 750 generations rather than 100. One drawback is the number of performed elitism in each simulation. There is one elitism performed in each generation, now 750 rather that only 100, in the microcontrolled Standard GA. Elitism itself does not contribute with new material in the evolutionary process, with the consequence of potential limitation in performance.

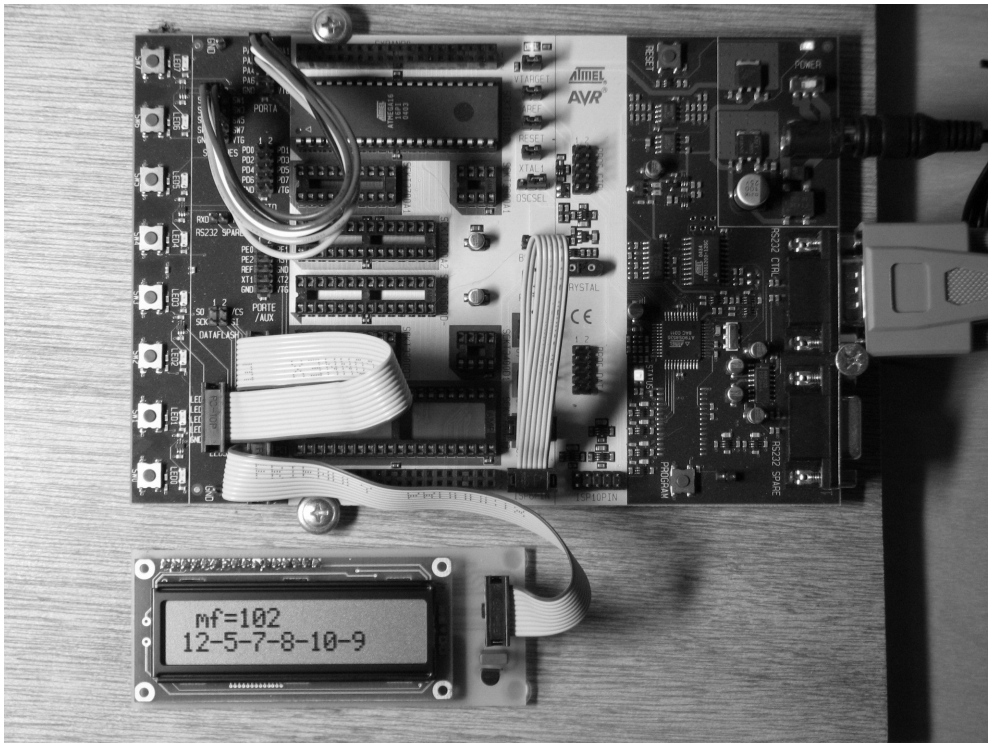


Figure 6.1: Entire simulation set up

Figure 6.2: Result print out on display

7 Simulations

It is important to remember that a complete or final result is not expected. The aim is to test how well the GA can perform and further improvements will still be required.

This work will compare and assess the simulations of the three programs; computed Standard GA, microcontrolled Standard GA and DFS version. Each program has four prepared simulation cases that will give 12 different results. All results and their behaviours will be explained briefly.

The simulation outputs actually looks in two different ways as follow. The first and last simulation programs are computed ones, doing their printout windows on a computer screen. The second simulation program is implemented inside a microcontroller, performing printouts on a LCD-display. These microcontrolled printouts look rather different and only have two printed lines (see Figure 6.2). Example of result from printout windows from the computed Standard GA and the DFS version can be found in Appendix E and Appendix F.

7.1 About the simulations

The four simulation cases are:

- *Example case* Easy case with only three requests to handle.
- *Normal case* An expected daily life situation or normal condition.
- *Worst case one* All inputs requesting including easier tram requests.
- *Worst case two* All inputs requesting including harder tram requests.

The results from the simulations are listed in pairs, in the result tables further down. The first table contains the *Example case* and the *Normal case*. The second table contains both *Worst case one* and *Worst case two*.

The *Example case*, with one of each pedestrian, tram and road vehicle request, has some interference, which is further described in Chapter 3.3. In the program code there are three sets of input variables, which is enough for the entire simulation. The results are listed in Table 7.2, Table 7.4 and Table 7.6.

The *Normal case* is supposed to illustrate a real simulation from the daily life. Normally the sensor inputs are in a moderate amount most of the time (see table 7.1). Some lanes have more traffic and other less. A consistent and continuous pedestrian traffic is assumed, with an even distribution over the corners of the intersection. Trams tend to come and go from northbound direction. The set of input values are chosen in a way so no duplicated tram requests can occur in any simulations, that is, a tram can only request one direction. The input values for time periods 8 and 9 all have inputs set to zero. The reason for this is to verify the ending of the simulation without any new inputs

affecting the result. The results are listed in Table 7.2, Table 7.4 and Table 7.6.

Table 7.1: Input values for *Normal case* empty cells have value 0

Inputs from pedestrians

	iPN1	iPN2	iPN3	iPW1	iPW2	iPW3	iPE	iPS1	iPS2	iPS3
T0	1		1			1				1
T1		1		1	1		1	1	1	
T2			1	1	1			1	1	
T3	1							1	1	1
T4				1			1			
T5			1							1
T6				1				1		
T7	1		1			1	1			
T8										
T9										

Inputs from trams

	iTNW	iTNE	iTNS	iTWN	iTWE	iTEN	iTEW	iTES	iTSN	iTSE
T0	1			1					1	
T1										
T2			1		1					
T3										
T4								1		1
T5		1								
T6										
T7						1				1
T8										
T9										

Inputs from road sensors

	iRNW	iRNE	iRNS	iRWN	iRWSE	iRENW	iRES	iRSW	iRSNE
T0		1	1					1	1
T1	1		1	1					1
T2	1		1	1					1
T3		1		1	1	1		1	
T4		1		1				1	
T5							1		1
T6	1		1			1			1
T7		1			1				
T8									
T9									

The following two simulations are less authentic the real world, they can occur although not often. What *Worst case one* and *Worst case two* have in common is that they both only have input values in the first time period. The following time periods are free from any requests to avoid simulation interference. Ten time periods are used to get a complete view of the solved problem. Both cases have all pedestrian and road vehicle requests set in the beginning. The difference between the cases is the type of requests coming from the trams. In *Worst case one* there is one tram on either side of the intersection waiting to go straight. These are TNS, TWE, TEW and TSN. In *Worst case two*, there are trams that intend to turn, resulting in more traffic being blocked. These trams are TNE, TWN, TES and TSE. The best predicted drive order from *Worst case one* would be 7, 8, 12 and 13. This is for the first four time periods, followed by two random drive orders. *Worst case two* requires six time periods with drive orders of 5, 6, 7 or 10, 8 or 11, 12 and 13. Note that some drive orders can complement each other to some extend. The results are listed in Table 7.3, Table 7.5 and Table 7.7.

Note that all simulations for each time period are dependent on the previous executed drive order. Present inputs with new and older prioritised request values will change the outcome in every new time period. Also unpredictable results can occur along longer simulation time.

7.2 Performed simulations

All tables contain two simulations each, and there are 12 simulations in total. Each completed time period from both the Standard GA and the DFS version, results in a solution with six drive orders. The first drive order becomes the executable one. The following one is the first predicted drive order, out of five in total. Notes for table labels is the first time period called *T0* and counting upwards. The first executing drive order is denoted *E* and the following first predicted drive order is *P1* and counting upwards. Resulting fitness value is each time periods maximum fitness value denoted *Mf*. Underlined numbers simply tells that drive orders do execute on or more requests. Time periods without any waiting requests can hold any random drive order as a result.

7.2.1 Computed Standard GA simulations

The following is a summary of the results from the computed Standard GA simulations. The results of the *Example Case* and *Normal case* are outlined in Table 7.2 and the results from *Worst case one* and *Worst case two* in Table 7.3.

The resulting drive order sequence in the *Example case* is 8-10-15, the rest are non-valid outputs. In the *Normal case* there is a long sequence where 5 and 13 are repeated. These two drive orders are repeatedly predicted and so is 7, 8 and 12. Compared to the *Example case*, the *Normal case* has higher fitness values as a result of the higher number of request involved. Time taken to run the simulation with 2 GHz was about one second.

Tables 7.2: Standard GA on computer for *Example case* and *Normal case*

Standard GA Example case

	E	P1	P2	P3	P4	P5	Mf
T0	8	12	8	7	11	0	56
T1	10	4	5	12	14	9	70
T2	15	7	13	7	11	0	38
T3	15	7	13	7	13	15	0
T4	15	7	13	7	13	15	0
T5	15	7	13	7	13	15	0
T6	15	7	13	7	13	15	0
T7	15	7	13	7	13	15	0
T8	15	7	13	7	13	15	0
T9	15	7	13	7	13	15	0

Standard GA Normal case

	E	P1	P2	P3	P4	P5	Mf
T0	5	12	7	8	12	8	103
T1	13	12	7	8	12	8	159
T2	7	12	5	13	7	8	236
T3	8	12	5	13	3	8	228
T4	12	13	5	13	7	8	216
T5	5	13	6	10	9	10	188
T6	13	6	9	10	15	11	202
T7	6	10	5	11	3	6	240
T8	11	10	5	11	0	4	156
T9	10	15	5	12	8	7	136

Worst case one is almost as predictable, resulting in sequence 7-8-12 and 13. In *Worst case two* it was predicted in the first time period *T0* that four more drive orders were needed to solve the problem. A closer look in Table 7.3 reveals that the problem only becomes solved in time period *T5*, rather than *T4*. In other words, one additional time period was needed to complete the simulation. Notice that drive order 13 is first predicted in *P1* for several time periods. The reason for this is that the condition in each time period is affected by previous drive orders and new incoming requests.

Tables 7.3: Standard GA on computer for both worst cases

Standard GA Worst case 1

	E	P1	P2	P3	P4	P5	Mf
T0	7	8	12	14	9	1	187
T1	8	12	13	7	11	6	274
T2	12	13	5	12	8	0	162
T3	13	5	12	5	13	3	126
T4	13	5	12	5	13	3	0
T5	13	5	12	5	13	3	0
T6	13	5	12	5	13	3	0
T7	13	5	12	5	13	3	0
T8	13	5	12	5	13	3	0
T9	13	5	12	5	13	3	0

Standard GA Worst case 2

	E	P1	P2	P3	P4	P5	Mf
T0	5	13	7	11	6	11	180
T1	12	13	6	11	10	15	240
T2	11	13	6	10	15	8	240
T3	10	13	6	8	12	8	230
T4	6	13	7	11	0	8	238
T5	13	0	5	13	5	12	166
T6	13	0	5	13	5	12	0
T7	13	0	5	13	5	12	0
T8	13	0	5	13	5	12	0
T9	13	0	5	13	5	12	0

7.2.2 Microcontrolled Standard GA simulations

Due to the limitations of the Standard GA in this microcontroller implementation, some differences in the results are to be expected. The clock speed for the microcontroller is 8 MHz. The time taken to complete the simulation was approximately 24 seconds. There are no underlined drive orders to indicate if that drive order will execute a request, in Table 7.4 and Table 7.5. Only the most essential numbers, drive order sequence and maximum fitness value are presented.

Tables 7.4: Standard GA on microcontroller for *Example case* and *Normal case*

Standard GA Example case

	E	P1	P2	P3	P4	P5	Mf
T0	10	8	7	13	1	3	51
T1	15	8	12	8	12	14	53
T2	8	12	14	9	10	5	46
T3	16	0	0	0	0	0	0
T4	16	0	0	0	0	0	0
T5	16	0	0	0	0	0	0
T6	16	0	0	0	0	0	0
T7	16	0	0	0	0	0	0
T8	16	0	0	0	0	0	0
T9	16	0	0	0	0	0	0

Standard GA Normal case

	E	P1	P2	P3	P4	P5	Mf
T0	12	5	7	8	10	9	102
T1	5	10	9	14	11	3	139
T2	8	10	9	7	13	2	199
T3	7	13	12	5	11	4	274
T4	11	5	13	3	12	9	203
T5	5	12	13	6	4	13	230
T6	12	13	6	1	13	14	266
T7	13	6	11	10	9	5	304
T8	6	11	10	5	12	5	330
T9	5	11	10	9	1	3	192

Tables 7.5: Standard GA on microcontroller for both worst cases

Standard GA Worst case 1

	E	P1	P2	P3	P4	P5	Mf
T0	8	12	7	13	7	8	186
T1	12	13	7	8	10	9	224
T2	7	13	7	11	2	13	214
T3	13	7	8	7	8	1	136
T4	16	0	0	0	0	0	0
T5	16	0	0	0	0	0	0
T6	16	0	0	0	0	0	0
T7	16	0	0	0	0	0	0
T8	16	0	0	0	0	0	0
T9	16	0	0	0	0	0	0

Standard GA Worst case 2

	E	P1	P2	P3	P4	P5	Mf
T0	12	5	13	6	8	2	180
T1	13	5	6	11	10	0	234
T2	6	13	11	3	13	2	242
T3	13	11	8	10	9	15	234
T4	11	10	9	10	1	9	184
T5	10	9	8	10	4	13	92
T6	16	0	0	0	0	0	0
T7	16	0	0	0	0	0	0
T8	16	0	0	0	0	0	0
T9	16	0	0	0	0	0	0

The general impression is that the microcontrolled Standard GA have slightly lower performance than the computed one. This is expected as it is affected by many more elitisms. The drive order sequences for the microcontrolled Standard GA are different to both computed Standard GA and DFS version. The fitness values are also lower than in the other versions, but not far behind the computed ones.

7.2.3 DFS version simulations

The advantage with the DFS version is that it always searches through the entire search space to find the best results. For that reason all resulting solutions from each separate time period, can be considered as the right answer. On the other hand, it is less certain it will get the best outcome in long term simulations with several time periods. The DFS version program performs the DFS process in all the ten time periods, until it is completely finished. Time taken to complete the simulation with 2 GHz from start to

finish was about 1 minute and 50 seconds, or about 11 seconds per time period.

Table 7.6: DFS version for *Example case* and *Normal case*

DFS version Example case

	E	P1	P2	P3	P4	P5	Mf
T0	8	10	9	10	9	10	63
T1	10	9	10	9	10	4	78
T2	15	5	12	5	11	0	38
T3	15	5	12	5	11	0	0
T4	15	5	12	5	11	0	0
T5	15	5	12	5	11	0	0
T6	15	5	12	5	11	0	0
T7	15	5	12	5	11	0	0
T8	15	5	12	5	11	0	0
T9	15	5	12	5	11	0	0

DFS version Normal case

	E	P1	P2	P3	P4	P5	Mf
T0	5	12	8	7	8	7	108
T1	12	8	7	13	5	11	178
T2	8	7	13	5	11	0	202
T3	7	13	5	11	12	4	270
T4	11	12	5	13	7	8	224
T5	12	5	13	6	10	4	253
T6	5	13	6	10	4	0	292
T7	13	6	10	5	11	0	319
T8	6	5	11	10	9	10	324
T9	5	11	10	9	10	4	194

In the *Example case* outlined in Table 7.6 the sequence 8-10-15 is expected. All other numbers have no effect even though they happen to be in repeating order all the way. This is an effect in all DFS simulations. After *T2* no time period will have any effectiveness. The *Normal case* uses almost the same drive orders as the computed Standard GA, only in a slightly different order.

Table 7.7: DFS version for both worst cases

DFS version Worst case 1

	E	P1	P2	P3	P4	P5	Mf
T0	8	7	13	5	12	4	193
T1	7	13	5	12	5	11	238
T2	13	12	5	12	5	11	164
T3	12	5	12	5	11	0	126
T4	12	5	12	5	11	0	0
T5	12	5	12	5	11	0	0
T6	12	5	12	5	11	0	0
T7	12	5	12	5	11	0	0
T8	12	5	12	5	11	0	0
T9	12	5	12	5	11	0	0

DFS version Worst case 2

	E	P1	P2	P3	P4	P5	Mf
T0	12	5	11	6	13	3	187
T1	5	11	6	13	7	8	262
T2	11	6	13	10	9	10	259
T3	6	13	10	9	10	4	256
T4	13	10	9	10	9	10	190
T5	10	9	10	9	10	4	102
T6	10	9	10	9	10	4	0
T7	10	9	10	9	10	4	0
T8	10	9	10	9	10	4	0
T9	10	9	10	9	10	4	0

The results for *Worst case one* in Table 7.7 show excellent result with executed sequence 8-7-13-12. *Worst case two* has the same drive orders as the computed Standard GA, only in a different sequence order.

7.3 Simulation results

To come to a final conclusion, the results from all previous simulations with computed Standard GA, microcontrolled Standard GA and DFS version need to be compared and verified. The essential data to verify are the maximum fitness values, as this is the measure of their performances in each simulation. The simulations from the *Example case* and the *Normal case* are shown in Figure 7.8. And the simulations from *Worst case one* and *Worst case two* in Figure 7.9.

The DFS version is used as guidance of assumed best result in each simulation case. Thus, the ideal solution from both computed and microcontrolled Standard GA would be an exact copy of the DFS version's drive order sequence. No Standard GA is expected to perform the same solutions as the DFS version under the same conditions. To have in mind that both have the exact same simulation cases and evaluation function to work with. The large amount of good solutions will enable the Standard GA to bring up its own solutions, which differ from the DFS version. These solutions can present different drive orders as early as in the first or second time period. This means the DFS version and the Standard GA start working on different solutions, which are not directly comparable. This means a strict comparison between the two versions is not possible. The comparison will instead come to rely on indirect comparison or a general overview from the maximum fitness values *Mf*. The key factor in assessing the results is comparing the two version's final maximum fitness values and not matching the drive order sequences.

The best guideline to compare all simulations is to look for high fitness in the beginning and lower fitness in the end. A higher fitness indicates that more requests are executed. Consequently, opposite results indicates less good performance.

The graph in Figure 7.8 shows the simulation from *Example case* and *Normal case*. In the *Example case* both computed simulations perform the exact same sequence. The microcontroller on the other hand finds a less good solution and gets lower fitness in the beginning.

In the *Normal case* something different occurs in the long run. On the first two time periods the computed Standard GA gets sequence 5-13 and the DFS version has higher fitness with 5-12. It has a small difference, although in *T2* the Standard GA performs much better than DFS version. Lower fitness values, following after higher ones, means many requests have been executed and less requests remaining. All results from *T3* to *T9* show that the computed Standard GA has less work to do. This is an interesting result, demonstrating that the DFS version do not impress in the long run in this case.

In the *Worst cases* in Figure 7.9 there is ones again interesting results from the computed Standard GA and the DFS version. In *Worst case one* the results gives a clear view of good and less good performance between the three programs. They have almost equal fitness in *T0*, then the computed Standard GA execute drive order 7 instead of the best one 8, as this is a result from DFS version. In *T1* the computed Standard GA has a significant higher fitness than the DFS version, and then they become almost equals again in *T2* and *T3*. The microcontrolled Standard GA do not performing that well which results in higher fitness value in *T2* and *T3*.

Finally in *Worst case two* a very equal performance is shown with only slightly higher fitness value in the DFS version. The microcontrolled Standard GA performs very well through the entire simulation, even slightly better than the DFS version in the ending. In *T2* and *T3* the fitness value of the computed Standard GA is slightly lower then the others. These results in much higher fitness value in *T4* and *T5* compared to both the microcontrolled Standard GA and the DFS version.

Both computed and microcontrolled Standard GA manage to avoid getting stuck in "local optimum". A local optimum is a "believed" good solution or result that is far from the really good one. It is proven by the maximum fitness value, which will be significant lower than the DFS version fitness value. If a search space could be presented in a graph, it would appear as a mountain chain in the horizon with peaks and valleys. Where the y-axle is the fitness, represented by the height of the mountains and horizon is the x-axle. Although the highest peak holds the best solution, lower peaks can mislead a GA to believe they actually are the highest peak. A GA is more likely to continue climbing on the peek it believes is being the best one.

All simulations here do show what drive orders are best for this particular intersection. It is possible to figure out good solutions directly from the rules. Drive orders 5, 6, 7, 8, 12, 13 and 15 are most usable and the best sequence to use in theory is 7-13-5-12-8 according to the sequence rules. Of all 16 drive orders some of them are rarely or almost never used as 0, 1, 2, 3, 4 and 14. This is because their configuration easily can be replaced by the others.

A brief conclusion of these simulations is summarised in Chapter 8.1.

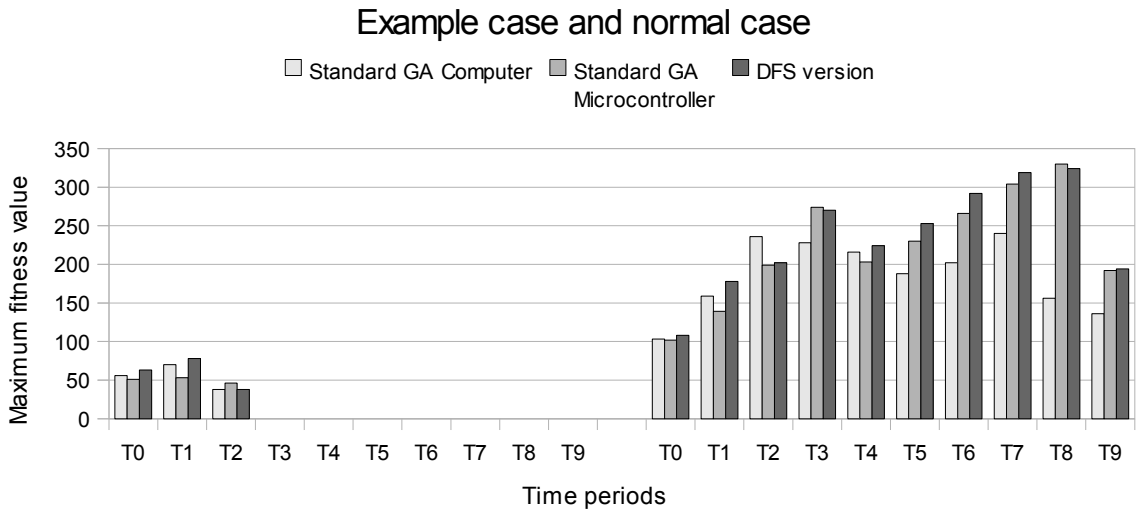


Figure 7.8: *Example case* and *Normal case* with comparing both Standard GA and DFS version

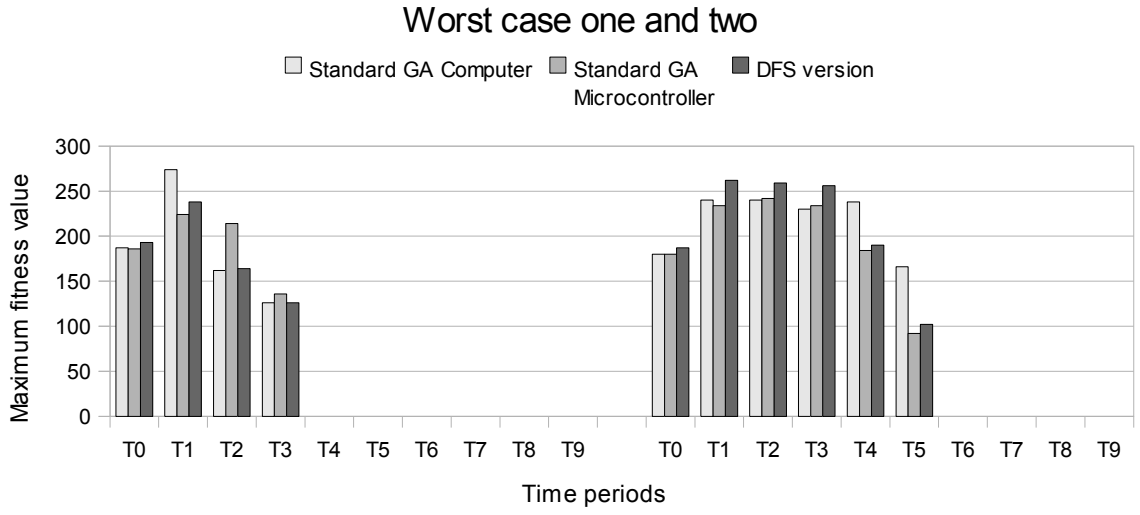


Figure 7.9: *Worst case one* and two with comparing both Standard GA and DFS version

8 Conclusion

The nature of this traffic intersection becomes rather simple due to safety reason. Although, it turn out to gain enough complexity when a sequence of time periods are considered. Important to point out that more work is required for further improvements, the results from this thesis establishes that Standard GA do works as expected. The test of implementing a microcontroller also turns out well.

8.1 Simulation conclusion

The outcome of the comparison, between results from the four different cases in Chapter 7.3, is quite surprising. Despite the fact that only four cases were used, performance differences between the cases could be detected. The results also demonstrate that a solution that might be the best one in a particular moment in time might not prove to be the best solution in the long run. A good example is the microcontrolled Standard GA, with lower performance can perform quite well in some circumstances. For both Standard GA programs, a good performance is achieved when their fitness is very close to that of the DFS version. This applies both for each time period comparison as well as for the general outcome in a long run simulation. To get a better understanding of their performances more simulations will be required. Although these simulations provide an indication of how good results they can perform.

A final conclusion of the simulation results is that a GA is capable of finding good solutions with little effort. The question is not what simulation program is the best. In fact that they do work at all matters. The aim of this work was to realise all three simulation programs and prove their functionalities. In particular the microcontrolled Standard GA implemented inside a simpler hardware system. The limitation of the microcontrolled Standard GA was the over numbered amount of elitisms which in the simulations resulted in lower performance and fitness values as compared to the computed simulation programs.

8.2 Future improvements

They are room for more adjustments and there are still many possibilities for further refinements and improvements. One example is to enhance the program to enable control of multiple intersections. Testing how the random function differs between computers and microcontrollers, is another potential area for improvements, as different random functions might give different performances.

A few things which were considered to do from the outset of this work included FPGA implementation, more self learning and robust functionality for direct control.

- A complete FPGA implementation would be considered as a final result for this work. Due to time limits and no hardware access the microcontroller implementation where realised instead. Microcontrollers are slower than FPGAs, although the speed does not matter in the simulations.

- Self learning is somewhat used in a simplified way in this work by storing good individuals for future use. This method makes it faster to look for better solutions. It could be extended in this work. Otherwise it works well despite being very simple.

- More robust functionality is needed for direct control in the reality. Things to improve include rules settings in the evaluation function, and convert the hardware version of this simulation program into a real controller program.

8.3 Summary

The result from this thesis may not contain a final solution, for example a hardware implementation ready to use, its only stays on simulation programs. Otherwise, it could be seen as a tool or an optional idea for further development into similar problems in the future. By their nature, GAs has a somewhat unpredicted behaviour that may not fit in safety critical environments. Thus, whether or not to recommend direct control for the intersection is not straight forward. In guidance for developments GA are really good.

This is not a common area where not many attempts are done. Usages of GAs are almost occurring in scientific area as research or optimization. About putting GA into a microcontroller is not very common. The papers with scientific implementation of GAs on FPGAs are today old. They where successful by then and should be even today. One aim in this work is how to move out the GAs to the real outside world for different needs. A conclusion so far is still not many reasons to do that. Traffic intersection is better to be developed by GA rather than controlled from it after all. In other hand this need of GA may grooving in the future. For example a hand held devices in car navigator might need to solve Travelling Salesmen Problem (TSP), which is commonly solved by GA.

The most interesting discovery is the constrained rules. It was hard to predict from the beginning and a surprise when it was discovered. Otherwise the lesson from this is good and important to have in mind when defining GA for certain problems.

Environmental benefits behind intersection control are better traffic flow and saving time and pollution. Most scientific work with GA do traffic flow improvements through several intersections, in some real cities around the world. Adjacent intersections where not considered in this work.

Is not that much work on this area in general, and not many other works to reference. Almost all scientific works referring to one single book [Goldberg, 1989], that is the first real description about GAs. Most of these simulation programs and rules developing is made from the beginning.

At last the work of this master's thesis has being very interesting to work with. Several new ideas and improvements to this work have coming up along the way. Otherwise, this work will be considered as good enough.

9 References

9.1 Books

[Goldberg, 1989]
Goldberg, D. E., 1989. *Genetic algorithm in search, optimization and machine learning.* Publisher: Addison-Wesley.

[Stuart & Norvig, 2003]
Stuart, R. & Norvig, P., 2003. *Artificial Intelligence A Modern Approach.* 2nd ed. Publisher: Prentice Hall.

[Wahde, 2008]
Wahde, M., 2008. *Biologically inspired optimization methods an introduction.* Southampton: WIT Press.

9.2 Documents

[Aporntewan & Chongstitvatana, 2001]
Aporntewan, C. and Chongstitvatana, P., 2001. A hardware Implementation Of The Compact Genetic Algorithm, 1, pp. 624–629

[Fissgus]
Fissgus, U., Scheduling Using Genetic Algorithms.

[Guan, et al., 2008]
Guan, Q., Yang, Z., Wang, Y., Hu, J., Qin, J., 2008. Research on the Coordination Optimization Method between Traffic Control and Traffic Guidance based on Genetic Algorithm, pp. 320-325.

[Ma, et al., 2004]
Ma, W., Cheu, R. L., Lee, D-H., 2004. Scheduling of Lane Closures Using Genetic Algorithms with Traffic Assignments and Distributed Simulations, *Journal of Transportation Engineering*, May 1, 130 (3), pp. 322-329.

[Shackleford, et al., 2001]
Shackleford, B., Snider, G., Carter, R., Okushi, E., Yasuda, M., Seo, K., Yasuura, H., 2001. A High-Performance, Pipelined, FPGA-Based Genetic Algorithm Machine, *Genetic Programming And Evolvable Machines,* 2, pp. 33-60

[Turky, et al., 2009]
Turky, A. M., Ahmad, M.S., Yusoff, M.Z.M., Hammad, B. T., 2009. Using Genetic Algorithm for Traffic Light Control System with a Pedestrian Crossing, pp. 512–519.

[Yang, et al., 2006]
Yang, Z., Huang, X., Liu, H., Xiang, C., 2006. Multi-phase Traffic Signal Control for Isolated Intersections Based on Genetic Fuzzy Logic, pp. 3391-3395.

[Yoon]
Yoon, J. P., Techniques for Data and Rule Validation in Knowledge Based Systems, *Department of Electrical Engineering University of Florida, Gainesville, FL 3261 I*, pp. 62-70.

9.3 Internet

[Altera]
Available at: http://www.altera.com/
[Accessed 19 May 2010].

[Atmel]
Available at: http://www.atmel.com/
[Accessed 19 May 2010].

[Atmel ATmega16]
Atmel Corporation, 2009. *8-bit Microcontroller with 16K Bytes In-System Programmable Flash,* [internet] Atmel Corporation, Available at: http://www.atmel.com/dyn/resources/prod_documents/doc2466.pdf
[Accessed 19 May 2010].

[Atmel STK500]
Available at: http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2735
[Accessed 19 May 2010].

[VHDL]
Available at: http://www.eda.org/vasg/
[Accessed 20 June 2010].

[Xilinx]
Available at: http://www.xilinx.com/
[Accessed 19 May 2010].

10 Appendix

Appendix A. **Traffic intersection drive orders chart**

Appendix B. **Mathematical fitness functions**

Appendix C. **Propositional logic for outputs**

Appendix D. **Propositional logic for drive order sequence**

Appendix E. **Program windows of Standard GA**

Appendix F. **Program windows of DFS version**

Appendix G. **Program code of Standard GA**

Appendix H. **Program code of DFS version**

Appendix A Traffic intersection drive orders chart

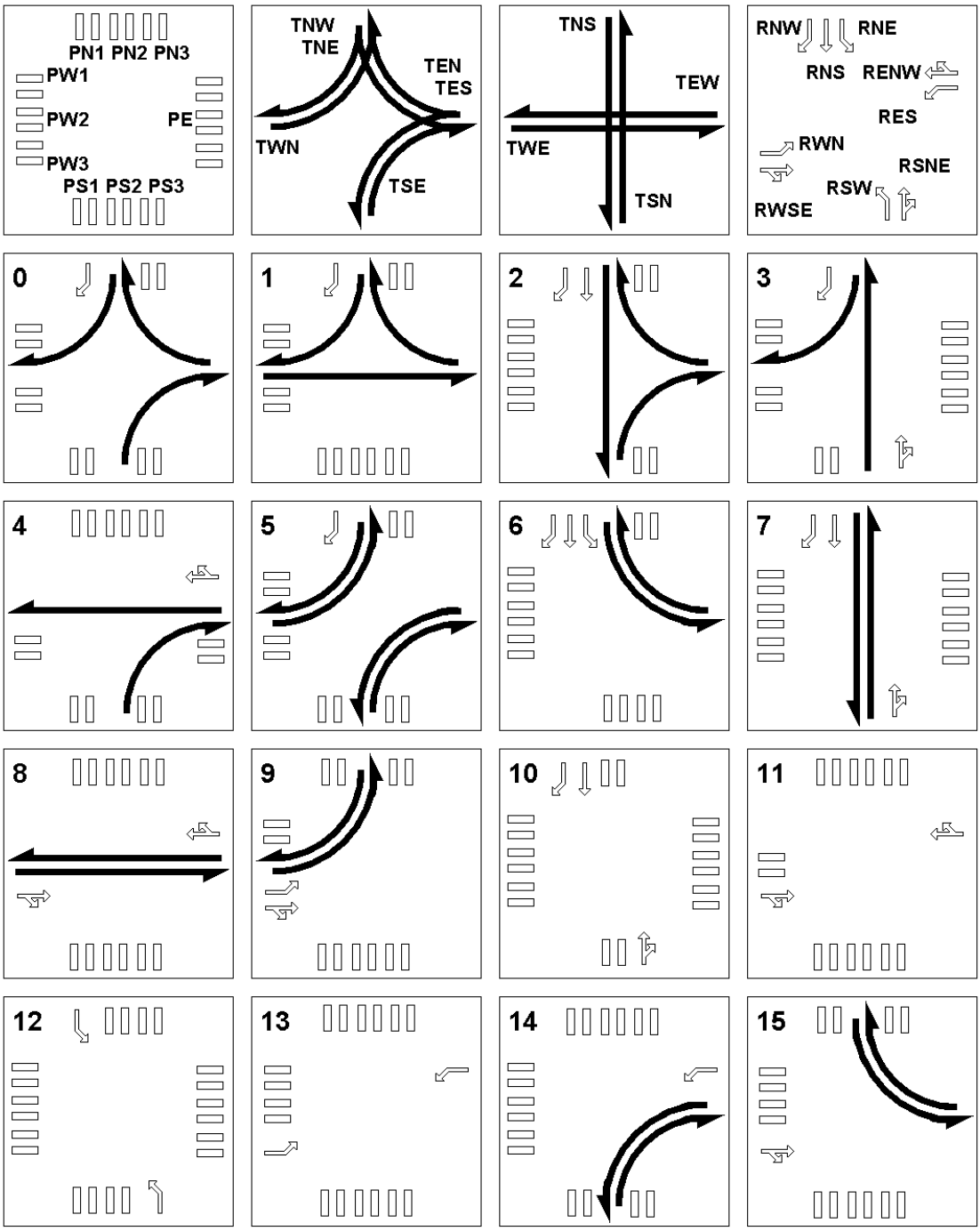


Figure A: Chart contain all 16 prepared drive order outputs

Appendix B Mathematical fitness functions

List of fitness calculation for variable $c_{nInputs}$ each drive order.

Drive order 0:

$$c_{nInputs} = I_{TNW} + I_{TEN} + I_{TSE} + I_{PN1} + I_{PN2} + I_{PN3} + I_{PS1} + I_{PS3} + I_{RNW}$$

Drive order 1:

$$c_{nInputs} = I_{TNW} + I_{TEN} + I_{TWE} + I_{PN3} + I_{PW1} + I_{PS1} + I_{PS2} + I_{PS3} + I_{RWSE}$$

Drive order 2:

$$c_{nInputs} = I_{TNS} + I_{TEN} + I_{TSE} + I_{PW1} + I_{PW2} + I_{PW3} + I_{PN3} + I_{PS3} + I_{RNW} + I_{RNS}$$

Drive order 3:

$$c_{nInputs} = I_{TNW} + I_{TSN} + I_{PW1} + I_{PW3} + I_{PE} + I_{PS1} + I_{RNW} + I_{RSNE}$$

Drive order 4:

$$c_{nInputs} = I_{TEW} + I_{TSE} + I_{PN1} + I_{PN2} + I_{PN3} + I_{PW3} + I_{PS1} + I_{PS3} + I_{RENW}$$

Drive order 5:

$$c_{nInputs} = I_{TNW} + I_{TWN} + I_{TES} + I_{TSE} + I_{PN3} + I_{PW1} + I_{PW3} + I_{PS1} + I_{PS3} + I_{RNW}$$

Drive order 6:

$$c_{nInputs} = I_{TNE} + I_{TEN} + I_{PN3} + I_{PW1} + I_{PW2} + I_{PW3} + I_{PS2} + I_{PS3} + I_{RNW} + I_{RNS} + I_{RNE}$$

Drive order 7:

$$c_{nInputs} = I_{TNS} + I_{TSN} + I_{PW1} + I_{PW2} + I_{PW3} + I_{PE} + I_{RNW} + I_{RNS} + I_{RSNE}$$

Drive order 8:

$$c_{nInputs} = I_{TWE} + I_{TEW} + I_{PN1} + I_{PN2} + I_{PN3} + I_{PS1} + I_{PS2} + I_{PS3} + I_{RENW} + I_{RWSE}$$

Drive order 9:

$$c_{nInputs} = I_{TNW} + I_{TWN} + I_{PN1} + I_{PN3} + I_{PW1} + I_{PS1} + I_{PS2} + I_{PS3} + 2I_{RWN} + I_{RWSE}$$

Drive order 10:

$$c_{nInputs} = I_{PN2} + I_{PW1} + I_{PW2} + I_{PW3} + I_{PS2} + I_{PE} + I_{RNW} + I_{RNS} + 2I_{RSNE}$$

Drive order 11:

$$c_{nInputs} = I_{PN1} + I_{PN2} + I_{PN3} + I_{PW2} + I_{PS1} + I_{PS2} + I_{PS3} + 2I_{RENW} + 2I_{RWSE}$$

Drive order 12:

$$c_{nInputs} = I_{PN2} + I_{PN3} + I_{PW1} + I_{PW2} + I_{PW3} + I_{PE} + I_{PS1} + I_{PS2} + 2I_{RNE} + 2I_{RSW}$$

Drive order 13:

$$c_{nInputs} = I_{PN1} + I_{PN2} + I_{PN3} + I_{PW1} + I_{PW2} + I_{PS1} + I_{PS2} + I_{PS3} + 2I_{RWN} + 2I_{RES}$$

Drive order 14:

$$c_{nInputs} = I_{TSE} + I_{TES} + I_{PN1} + I_{PN2} + I_{PN3} + I_{PW1} + I_{PW2} + I_{PW3} + I_{PS1} + I_{PS3} + I_{RES}$$

Drive order 15:

$$c_{nInputs} = I_{TNE} + I_{TEN} + I_{PN1} + I_{PN3} + I_{PW1} + I_{PW2} + I_{PS1} + I_{PS2} + I_{PS3} + 2I_{RWSE}$$

Appendix C Propositional logic for outputs

List of logical outputs for each drive order that executes by E_n.

Drive order 0:

$E_0 \rightarrow O_{TNW} \wedge O_{TEN} \wedge O_{TSE} \wedge O_{PN1} \wedge O_{PN2} \wedge O_{PN3} \wedge O_{PS1} \wedge O_{PS3} \wedge O_{RNW}$

Drive order 1:

$E_1 \rightarrow O_{TNW} \wedge O_{TEN} \wedge O_{TWE} \wedge O_{PN3} \wedge O_{PW1} \wedge O_{PS1} \wedge O_{PS2} \wedge O_{PS3} \wedge O_{RWSE}$

Drive order 2:

$E_2 \rightarrow O_{TNS} \wedge O_{TEN} \wedge O_{TSE} \wedge O_{PW1} \wedge O_{PW2} \wedge O_{PW3} \wedge O_{PN3} \wedge O_{PS3} \wedge O_{RNW} \wedge O_{RNS}$

Drive order 3:

$E_3 \rightarrow O_{TNW} \wedge O_{TSN} \wedge O_{PW1} \wedge O_{PW3} \wedge O_{PE} \wedge O_{PS1} \wedge O_{RNW} \wedge O_{RSNE}$

Drive order 4:

$E_4 \rightarrow O_{TEW} \wedge O_{TSE} \wedge O_{PN1} \wedge O_{PN2} \wedge O_{PN3} \wedge O_{PW3} \wedge O_{PS1} \wedge O_{PS3} \wedge O_{RENW}$

Drive order 5:

$E_5 \rightarrow O_{TNW} \wedge O_{TWN} \wedge O_{TES} \wedge O_{TSE} \wedge O_{PN3} \wedge O_{PW1} \wedge O_{PW3} \wedge O_{PS1} \wedge O_{PS3} \wedge O_{RNW}$

Drive order 6:

$E_6 \rightarrow O_{TNE} \wedge O_{TEN} \wedge O_{PN3} \wedge O_{PW1} \wedge O_{PW2} \wedge O_{PW3} \wedge O_{PS2} \wedge O_{PS3} \wedge O_{RNW} \wedge O_{RNS} \wedge O_{RNE}$

Drive order 7:

$E_7 \rightarrow O_{TNS} \wedge O_{TSN} \wedge O_{PW1} \wedge O_{PW2} \wedge O_{PW3} \wedge O_{PE} \wedge O_{RNW} \wedge O_{RNS} \wedge O_{RSNE}$

Drive order 8:

$E_8 \rightarrow O_{TWE} \wedge O_{TEW} \wedge O_{PN1} \wedge O_{PN2} \wedge O_{PN3} \wedge O_{PS1} \wedge O_{PS2} \wedge O_{PS3} \wedge O_{RENW} \wedge O_{RWSE}$

Drive order 9:

$E_9 \rightarrow O_{TNW} \wedge O_{TWN} \wedge O_{PN1} \wedge O_{PN3} \wedge O_{PW1} \wedge O_{PS1} \wedge O_{PS2} \wedge O_{PS3} \wedge O_{RWN} \wedge O_{RWSE}$

Drive order 10:

$E_{10} \rightarrow O_{PN2} \wedge O_{PW1} \wedge O_{PW2} \wedge O_{PW3} \wedge O_{PS2} \wedge O_{PE} \wedge O_{RNW} \wedge O_{RNS} \wedge O_{RSNE}$

Drive order 11:

$E_{11} \rightarrow O_{PN1} \wedge O_{PN2} \wedge O_{PN3} \wedge O_{PW2} \wedge O_{PS1} \wedge O_{PS2} \wedge O_{PS3} \wedge O_{RENW} \wedge O_{RWSE}$

Drive order 12:

$E_{12} \rightarrow O_{PN2} \wedge O_{PN3} \wedge O_{PW1} \wedge O_{PW2} \wedge O_{PW3} \wedge O_{PE} \wedge O_{PS1} \wedge O_{PS2} \wedge O_{RNE} \wedge O_{RSW}$

Drive order 13:

$E_{13} \rightarrow O_{PN1} \wedge O_{PN2} \wedge O_{PN3} \wedge O_{PW1} \wedge O_{PW2} \wedge O_{PS1} \wedge O_{PS2} \wedge O_{PS3} \wedge O_{RWN} \wedge O_{RES}$

Drive order 14:

$E_{14} \rightarrow O_{TSE} \wedge O_{TES} \wedge O_{PN1} \wedge O_{PN2} \wedge O_{PN3} \wedge O_{PW1} \wedge O_{PW2} \wedge O_{PW3} \wedge O_{PS1} \wedge O_{PS3} \wedge O_{RES}$

Drive order 15:

$E_{15} \rightarrow O_{TNE} \wedge O_{TEN} \wedge O_{PN1} \wedge O_{PN3} \wedge O_{PW1} \wedge O_{PW2} \wedge O_{PS1} \wedge O_{PS2} \wedge O_{PS3} \wedge O_{RWSE}$

Appendix D Propositional logic for drive order sequence

Propositional logic describes what drive order sequence that gives addition of two points to the fitness value calculation by variable s_n. Example rule *R1*, if P_1 and E_{P13} are *true*, $A_{TwoPoints}$ will also be *true* and that is equal to variable $s_n = 2$ in the evaluation function.

Previous executed drive order = E_{Pn}
Present drive order = P_n
Addition of two points *true* or *false* = $A_{TwoPoints}$

R1 $P_1 \wedge (E_{P11} \vee E_{P13}) \rightarrow A_{TwoPoints}$
R2 $P_2 \wedge (E_{P11} \vee E_{P13}) \rightarrow A_{TwoPoints}$
R3 $P_3 \wedge (E_{P11} \vee E_{P13}) \rightarrow A_{TwoPoints}$
R4 $P_4 \wedge (E_{P10} \vee E_{P12}) \rightarrow A_{TwoPoints}$
R5 $P_5 \wedge (E_{P12} \vee E_{P13}) \rightarrow A_{TwoPoints}$
R6 $P_6 \wedge (E_{P11} \vee E_{P13}) \rightarrow A_{TwoPoints}$
R7 $P_7 \wedge (E_{P8} \vee E_{P13}) \rightarrow A_{TwoPoints}$
R8 $P_8 \wedge (E_{P7} \vee E_{P12}) \rightarrow A_{TwoPoints}$
R9 $P_9 \wedge (E_{P10} \vee E_{P14}) \rightarrow A_{TwoPoints}$
R10 $P_{10} \wedge (E_{P8} \vee E_{P9}) \rightarrow A_{TwoPoints}$
R11 $P_{11} \wedge (E_{P5} \vee E_{P7}) \rightarrow A_{TwoPoints}$
R12 $P_{12} \wedge (E_{P5} \vee E_{P8}) \rightarrow A_{TwoPoints}$
R13 $P_{13} \wedge (E_{P5} \vee E_{P7}) \rightarrow A_{TwoPoints}$
R14 $P_{14} \wedge (E_{P9} \vee E_{P12}) \rightarrow A_{TwoPoints}$
R15 $P_{15} \wedge (E_{P10} \vee E_{P13}) \rightarrow A_{TwoPoints}$

Appendix E Program windows of Standard GA

Two printout windows and the first window are Figure E.A and last window Figure E.B.

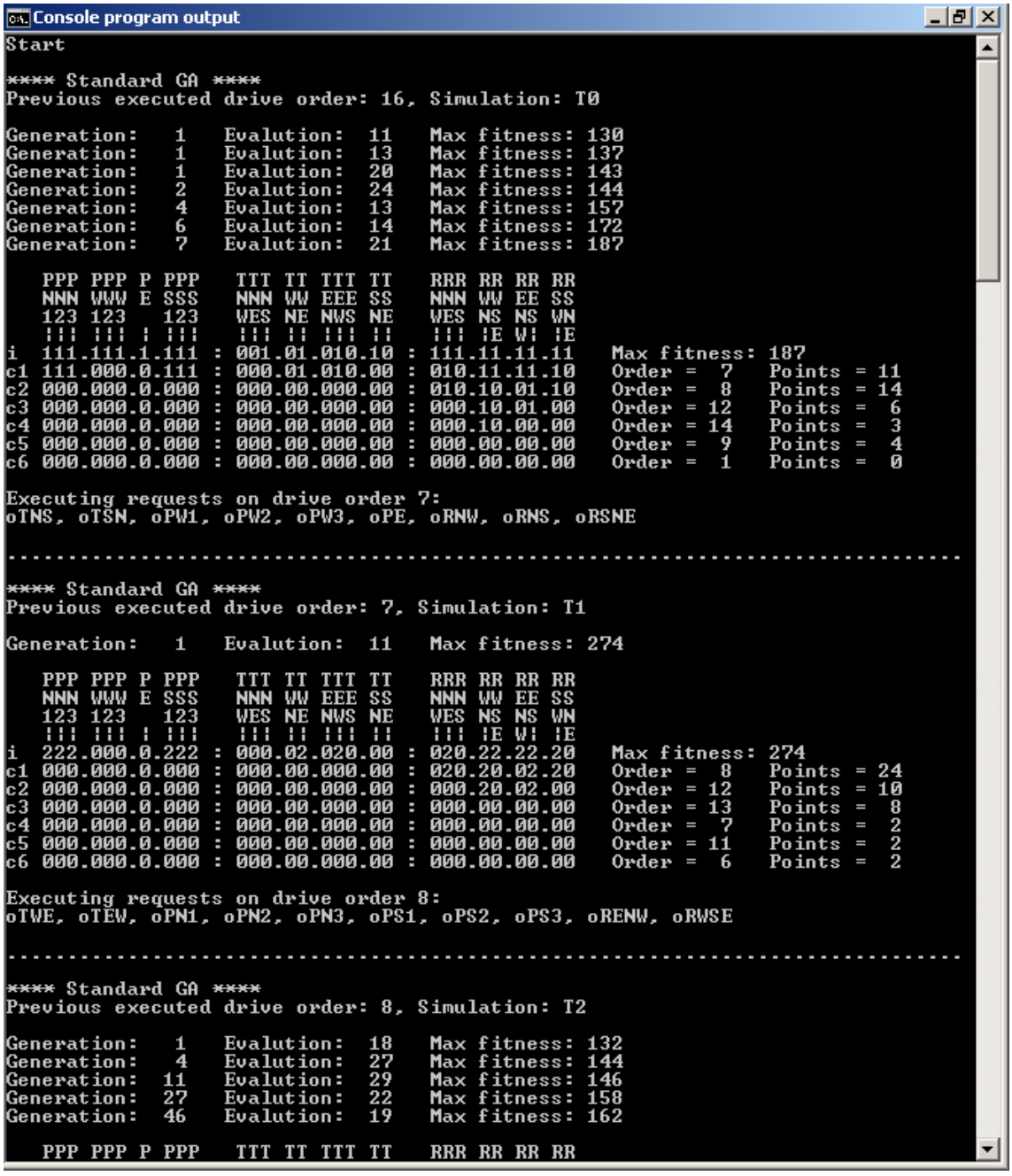


Figure E.A: Print out image one with *Worst case one* results

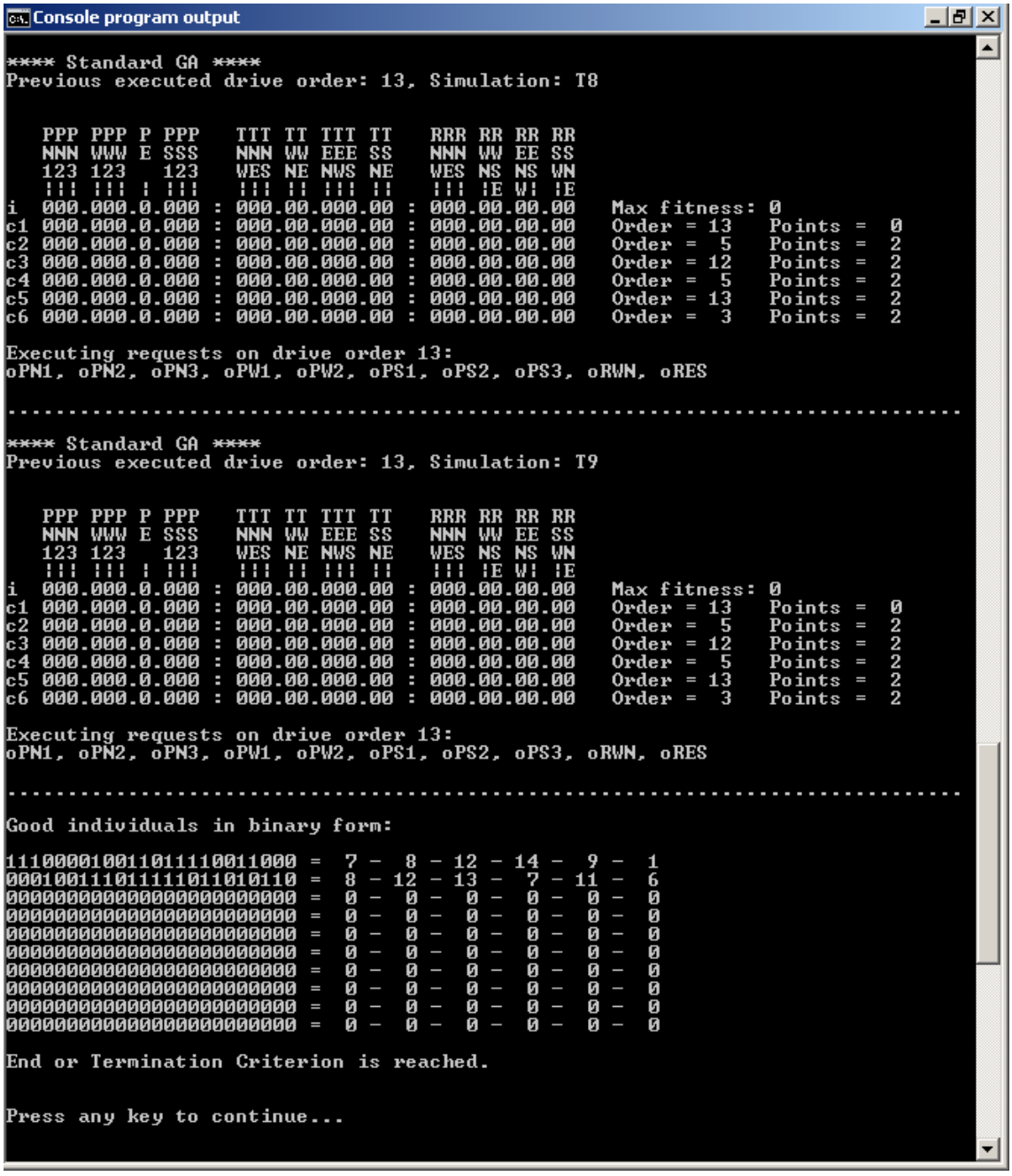


Figure Appendix E.B: Print out image two with *Worst case one* results

Appendix F Program windows of DFS version

Two printout windows and the first window are Figure F.A and last window Figure F.B.

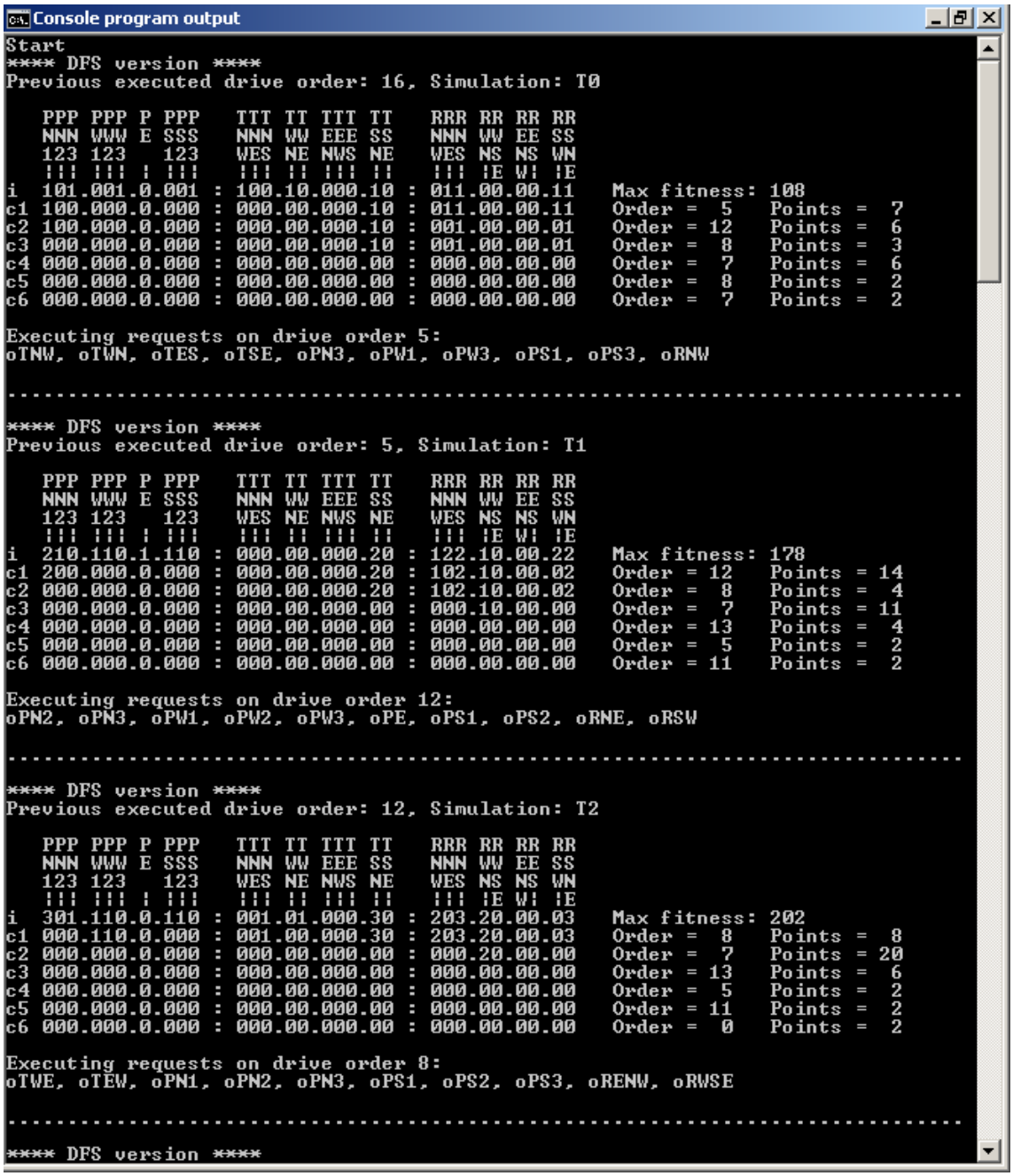


Figure F.A: Print out image one with *Normal case* results

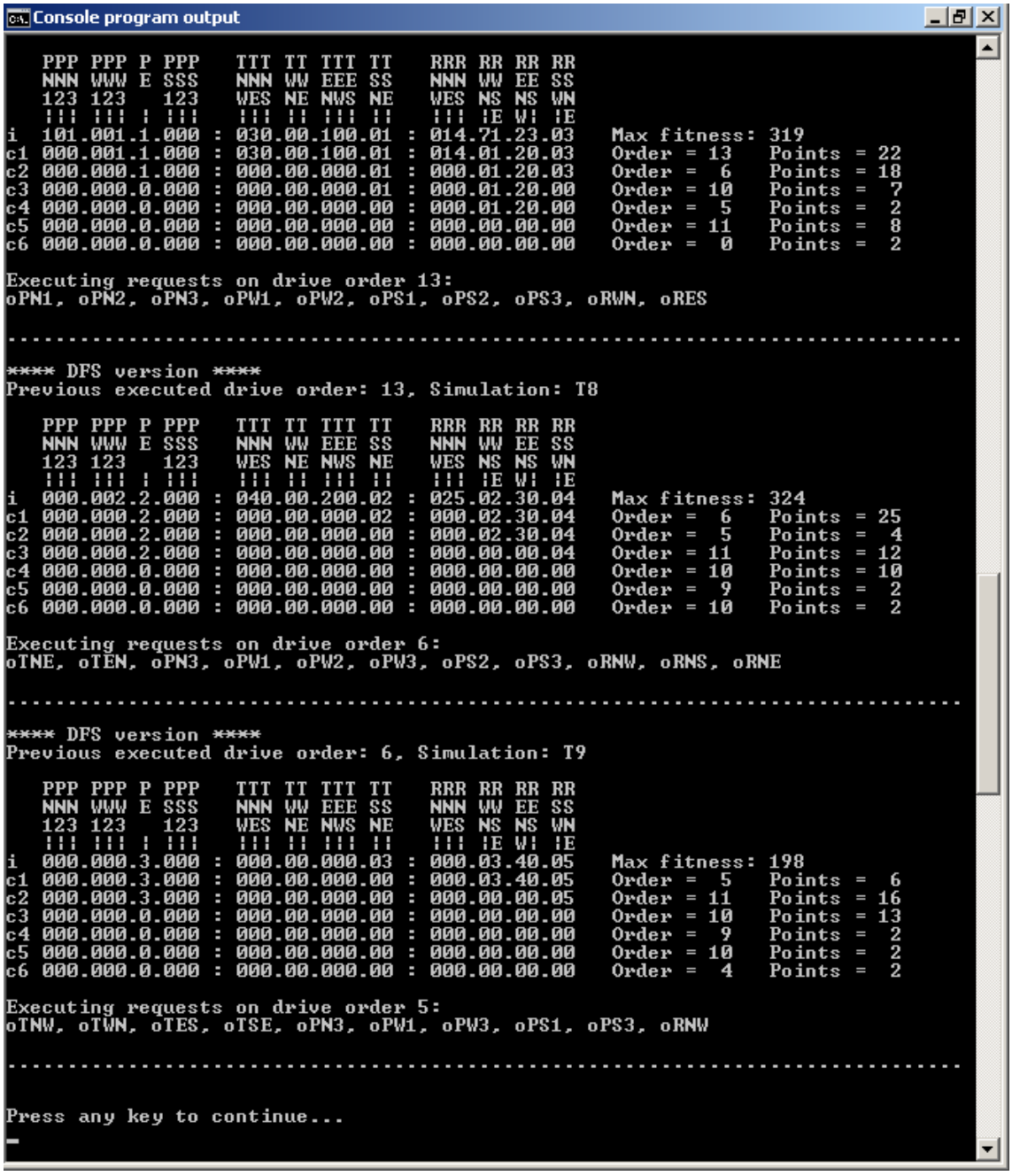


Figure F.B: Print out image two with *Normal case* results

Appendix G Program code of Standard GA

Source code for Standard GA with *Worst case one* inputs implemented.

```
/* Standard GA worst case one */

#include <stdio.h>
#include <conio.h>
#include <math.h>
#include <stdlib.h>

int DecodeChromosome(int population[30][24], int i, int numberOfGenes, int
chromosomeOrder, int numberOfChromosomes)
{
	int x = 0;
	int nCdiv = numberOfGenes/numberOfChromosomes;//Set lengh of genes
in one chromosome
	chromosomeOrder--;

	x = x + population[i][0 + nCdiv*chromosomeOrder]*1;
	x = x + population[i][1 + nCdiv*chromosomeOrder]*2;
	x = x + population[i][2 + nCdiv*chromosomeOrder]*4;
	x = x + population[i][3 + nCdiv*chromosomeOrder]*8;

	return x;
}

int EvaluateIndividual(int c1, int c2, int c3, int c4, int c5, int c6, int inputValue[29],
int previousExcecution)
{
	int f = 0, i = 0, x = 0, p = 0, points = 0, previousOrder;

	int iPN1 = inputValue[0];
	int iPN2 = inputValue[1];
	int iPN3 = inputValue[2];
	int iPW1 = inputValue[3];
	int iPW2 = inputValue[4];
	int iPW3 = inputValue[5];
	int iPE  = inputValue[6];
	int iPS1 = inputValue[7];
	int iPS2 = inputValue[8];
	int iPS3 = inputValue[9];
	int iTNW = inputValue[10];
	int iTNE = inputValue[11];
	int iTNS = inputValue[12];
	int iTWN = inputValue[13];
	int iTWE = inputValue[14];
	int iTEN = inputValue[15];
	int iTEW = inputValue[16];
	int iTES = inputValue[17];
	int iTSN = inputValue[18];
	int iTSE = inputValue[19];
	int iRNW = inputValue[20];
	int iRNE = inputValue[21];
	int iRNS = inputValue[22];
	int iRWN = inputValue[23];
	int iRWSE = inputValue[24];
	int iRENW = inputValue[25];
	int iRES = inputValue[26];
	int iRSW = inputValue[27];
	int iRSNE = inputValue[28];

	x = previousExcecution;
	for(i = 0; i <= 6 - 1; i++)
	{
		previousOrder = x;
		if(i == 0) {x = c1;}
		if(i == 1) {x = c2;}
		if(i == 2) {x = c3;}
		if(i == 3) {x = c4;}
		if(i == 4) {x = c5;}
		if(i == 5) {x = c6;}
		points = 0;

		//Inputs functions calculations
		switch(x)
		{
			case 0:
				points = iTNW + iTEN + iTSE + iPN1 + iPN2 +
iPN3 + iPS1 + iPS3 + iRNW;
				iTNW = 0; iTEN = 0; iTSE = 0; iPN1 = 0; iPN2 = 0;
iPN3 = 0; iPS1 = 0; iPS3 = 0; iRNW = 0;
				break;
			case 1:
				points = iTNW + iTEN + iTWE + iPN3 + iPW1 +
iPS1 + iPS2 + iPS3 + iRWSE;
				iTNW = 0; iTEN = 0; iTWE = 0; iPN3 = 0; iPW1 =
0; iPS1 = 0; iPS2 = 0; iPS3 = 0; iRWSE = 0;
				break;
			case 2:
				points = iTNS + iTEN + iTSE + iPW1 + iPW2 +
iPW3 + iPN3 + iPS3 + iRNW + iRNS;
				iTNS = 0; iTEN = 0; iTSE = 0; iPW1 = 0; iPW2 = 0;
iPW3 = 0; iPN3 = 0; iPS3 = 0; iRNW = 0; iRNS = 0;
				break;
			case 3:
				points = iTNW + iTSN + iPW1 + iPW3 + iPE + iPS1
+ iRNW + iRSNE;
				iTNW = 0; iTSN = 0; iPW1 = 0; iPW3 = 0; iPE = 0;
iPS1 = 0; iRNW = 0; iRSNE = 0;
				break;
			case 4:
				points = iTEW + iTSE + iPN1 + iPN2 + iPN3 +
iPW3 + iPS1 + iPS3 + iRENW;
				iTEW = 0; iTSE = 0; iPN1 = 0; iPN2 = 0; iPN3= 0;
iPW3 = 0; iPS1 = 0; iPS3 = 0; iRENW = 0;
				break;
			case 5:
				points = 2*iTNW + 2*iTWN + 2*iTES + 3*iTSE +
iPN3 + iPW1 + iPW3 + iPS1 + iPS3 + iRNW;
				iTNW = 0; iTWN = 0; iTES = 0; iTSE = 0; iPN3 = 0;
iPW1 = 0; iPW3 = 0; iPS1 = 0; iPS3 = 0; iRNW = 0;
				break;
			case 6:
				points = 4*iTNE + 4*iTEN + iPN3 + iPW1 + iPW2
+ iPW3 + iPS2 + iPS3 + iRNW + iRNS + iRNE;
				iTNE = 0; iTEN = 0; iPN3 = 0; iPW1 = 0; iPW2 = 0;
iPW3 = 0; iPS2 = 0; iPS3 = 0; iRNW = 0; iRNS = 0; iRNE = 0;
				break;
			case 7:
				points = 2*iTNS + 2*iTSN + iPW1 + iPW2 + iPW3
+ iPE + iRNW + iRNS + iRSNE;
				iTNS= 0; iTSN = 0; iPW1 = 0; iPW2 = 0; iPW3 = 0;
iPE = 0; iRNW = 0; iRNS = 0; iRSNE = 0;
				break;
			case 8:
				points = 3*iTWE + 3*iTEW + iPN1 + iPN2 + iPN3
+ iPS1 + iPS2 + iPS3 + iRENW + iRWSE;
				iTWE = 0; iTEW = 0; iPN1 = 0; iPN2 = 0; iPN3 = 0;
iPS1 = 0; iPS2 = 0; iPS3 = 0; iRENW = 0; iRWSE = 0;
				break;
			case 9:
				points = iTNW + iTWN + iPN1 + iPN3 + iPW1 +
iPS1 + iPS2 + iPS3 + 2*iRWN + iRWSE;
				iTNW = 0; iTWN = 0; iPN1 = 0; iPN3 = 0; iPW1 =
0; iPS1 = 0; iPS2 = 0; iPS3 = 0; iRWN = 0; iRWSE = 0;
				break;
			case 10:
				points = iPN2 + iPW1 + iPW2 + iPW3 + iPS2 + iPE
+ iRNW + iRNS + 2*iRSNE;
				iPN2 = 0; iPW1 = 0; iPW2 = 0; iPW3 = 0; iPS2 = 0;
iPE = 0; iRNW = 0; iRNS = 0; iRSNE = 0;
				break;
			case 11:
				points = iPN1 +  iPN2 +  iPN3 + iPW2 + iPS1 +
iPS2 + iPS3 + 2*iRENW + 2*iRWSE;
				iPN1 = 0; iPN2 = 0; iPN3 = 0; iPW2 = 0; iPS1 = 0;
iPS2 = 0; iPS3 = 0; iRENW = 0; iRWSE = 0;
				break;
			case 12:
				points = iPN2 +  iPN3 +  iPW1 + iPW2 + iPW3 +
iPE + iPS1 + iPS2 + 2*iRNE + 2*iRSW;
				iPN2 = 0; iPN3 = 0; iPW1 = 0; iPW2 = 0; iPW3 = 0;
iPE = 0; iPS1 = 0; iPS2 = 0; iRNE = 0; iRSW = 0;
				break;
			case 13:
				points = iPN1 +  iPN2 +  iPN3 + iPW1 + iPW2 +
iPS1 + iPS2 + iPS3 + 2*iRWN + 2*iRES;
				iPN1 = 0; iPN2 = 0; iPN3 = 0; iPW1 = 0; iPW2 = 0;
iPS1 = 0; iPS2 = 0; iPS3 = 0; iRWN = 0; iRES = 0;
				break;
			case 14:
				points = iTSE + iTES + iPN1 + iPN2 + iPN3 + iPW1
+ iPW2 + iPW3 + iPS1 + iPS3 + iRES;
				iTSE = 0; iTES = 0; iPN1 = 0; iPN2 = 0; iPN3 = 0;
iPW1 = 0; iPW2 = 0; iPW3 = 0; iPS1 = 0; iPS3 = 0; iRES = 0;
				break;
			case 15:
				points = iTNE + iTEN + iPN1 + iPN3 + iPW1 +
iPW2 + iPS1 + iPS2 + iPS3 + 2*iRWSE;
				iTNE = 0; iTEN = 0; iPN1 = 0; iPN3 = 0; iPW1 = 0;
iPW2 = 0; iPS1 = 0; iPS2 = 0; iPS3 = 0; iRWSE = 0;
				break;
			case 16:
				points = 0;
				break;
		}
		//Previous order calculations
		switch(x)
		{
			case 0:
				if((previousOrder == 11) || (previousOrder == 13))
{points = points + 2;}
				if(previousOrder == 0) {points = points - 5;}
```

```
                                break;
                        case 1:
                                if((previousOrder == 11) || (previousOrder == 13))
{points = points + 2;}
                                if(previousOrder == 1) {points = points - 5;}
                                break;
                        case 2:
                                if((previousOrder == 11) || (previousOrder == 13))
{points = points + 2;}
                                if(previousOrder == 2) {points = points - 5;}
                                break;
                        case 3:
                                if((previousOrder == 11) || (previousOrder == 13))
{points = points + 2;}
                                if(previousOrder == 3) {points = points - 5;}
                                break;
                        case 4:
                                if((previousOrder == 10) || (previousOrder == 12))
{points = points + 2;}
                                if(previousOrder == 4) {points = points - 5;}
                                break;
                        case 5:
                                if((previousOrder == 12) || (previousOrder == 13))
{points = points + 2;}
                                if(previousOrder == 5) {points = points - 5;}
                                break;
                        case 6:
                                if((previousOrder == 11) || (previousOrder == 13))
{points = points + 2;}
                                if(previousOrder == 6) {points = points - 5;}
                                break;
                        case 7:
                                if((previousOrder == 8) || (previousOrder == 13))
{points = points + 2;}
                                if(previousOrder == 7) {points = points - 5;}
                                break;
                        case 8:
                                if((previousOrder == 7) || (previousOrder == 12))
{points = points + 2;}
                                if(previousOrder == 8) {points = points - 5;}
                                break;
                        case 9:
                                if((previousOrder == 10) || (previousOrder == 14))
{points = points + 2;}
                                if(previousOrder == 9) {points = points - 5;}
                                break;
                        case 10:
                                if((previousOrder == 8) || (previousOrder == 9))
{points = points + 2;}
                                if(previousOrder == 10) {points = points - 5;}
                                break;
                        case 11:
                                if((previousOrder == 5) || (previousOrder == 7))
{points = points + 2;}
                                if(previousOrder == 11) {points = points - 5;}
                                break;
                        case 12:
                                if((previousOrder == 5) || (previousOrder == 8))
{points = points + 2;}
                                if(previousOrder == 12) {points = points - 5;}
                                break;
                        case 13:
                                if((previousOrder == 5) || (previousOrder == 7))
{points = points + 2;}
                                if(previousOrder == 13) {points = points - 5;}
                                break;
                        case 14:
                                if((previousOrder == 9) || (previousOrder == 12))
{points = points + 2;}
                                if(previousOrder == 14) {points = points - 5;}
                                break;
                        case 15:
                                if((previousOrder == 10) || (previousOrder == 13))
{points = points + 2;}
                                if(previousOrder == 15) {points = points - 5;}
                                break;
                }
                if(i == 0) {f = 6*points;}
                if(i == 1) {f = f + 5*points;}
                if(i == 2) {f = f + 4*points;}
                if(i == 3) {f = f + 3*points;}
                if(i == 4) {f = f + 2*points;}
                if(i == 5) {f = f + points;}
                if(i == 0 && points <= 2) {p = 1;}
        }
        if(p == 1) {f = 0;}//if p1 = 0 -> f = 0
        return f;
}

void ReturnResult(int bestParameterValue1, int bestParameterValue2, int
bestParameterValue3, int bestParameterValue4, int bestParameterValue5, int
bestParameterValue6, int inputValue[29], int maxFitness)
{
        int f = 0, x = 0, points = 0, previousOrder;

        int iPN1 = inputValue[0];
        int iPN2 = inputValue[1];
        int iPN3 = inputValue[2];
        int iPW1 = inputValue[3];
        int iPW2 = inputValue[4];
        int iPW3 = inputValue[5];
        int iPE  = inputValue[6];
        int iPS1 = inputValue[7];
        int iPS2 = inputValue[8];
        int iPS3 = inputValue[9];
        int iTNW = inputValue[10];
        int iTNE = inputValue[11];
        int iTNS = inputValue[12];
        int iTWN = inputValue[13];
        int iTWE = inputValue[14];
        int iTEN = inputValue[15];
        int iTEW = inputValue[16];
        int iTES = inputValue[17];
        int iTSN = inputValue[18];
        int iTSE = inputValue[19];
        int iRNW = inputValue[20];
        int iRNE = inputValue[21];
        int iRNS = inputValue[22];
        int iRWN = inputValue[23];
        int iRWSE = inputValue[24];
        int iRENW = inputValue[25];
        int iRES = inputValue[26];
        int iRSW = inputValue[27];
        int iRSNE = inputValue[28];

        printf("\n   PPP PPP P PPP   TTT TT TTT TT   RRR RR RR RR\n");
        printf("   NNN WWW E SSS   NNN WW EEE SS   NNN WW EE SS\n");
        printf("   123 123   123   WES NE NWS NE   WES NS NS WN\n");
        printf("   ||| ||| | |||   ||| || ||| ||   ||| |E W| |E\n");
        printf("i  %d%d%d.%d%d%d.%d.%d%d%d : %d%d%d.%d%d%d.%d%d%d.
%d%d : %d%d%d.%d%d.%d%d.%d%d%d   Max fitness: %d\n", iPN1, iPN2, iPN3,
iPW1, iPW2, iPW3, iPE, iPS1, iPS2, iPS3, iTNW, iTNE, iTNS, iTWN, iTWE,
iTEN, iTEW, iTES, iTSN, iTSE, iRNW, iRNE, iRNS, iRWN, iRWSE, iRENW,
iRES, iRSW, iRSNE, maxFitness);

        for(int i = 0; i <= 6 - 1; i++)
        {
                previousOrder = x;
                if(i == 0) {x = bestParameterValue1;}
                if(i == 1) {x = bestParameterValue2;}
                if(i == 2) {x = bestParameterValue3;}
                if(i == 3) {x = bestParameterValue4;}
                if(i == 4) {x = bestParameterValue5;}
                if(i == 5) {x = bestParameterValue6;}
                points = 0;

                //Inputs functions calculations
                switch(x)
                {
                        case 0:
                                points = iTNW + iTEN + iTSE + iPN1 + iPN2 +
iPN3 + iPS1 + iPS3 + iRNW;
                                iTNW = 0; iTEN = 0; iTSE = 0; iPN1 = 0; iPN2 = 0;
iPN3 = 0; iPS1 = 0; iPS3 = 0; iRNW = 0;
                                break;
                        case 1:
                                points = iTNW + iTEN + iTWE + iPN3 + iPW1 +
iPS1 + iPS2 + iPS3 + iRWSE;
                                iTNW = 0; iTEN = 0; iTWE = 0; iPN3 = 0; iPW1 =
0; iPS1 = 0; iPS2 = 0; iPS3 = 0; iRWSE = 0;
                                break;
                        case 2:
                                points = iTNS + iTEN + iTSE + iPW1 + iPW2 +
iPW3 + iPN3 + iPS3 + iRNW + iRNS;
                                iTNS = 0; iTEN = 0; iTSE = 0; iPW1 = 0; iPW2 = 0;
iPW3 = 0; iPN3 = 0; iPS3 = 0; iRNW = 0; iRNS = 0;
                                break;
                        case 3:
                                points = iTNW + iTSN + iPW1 + iPW3 + iPE + iPS1
+ iRNW + iRSNE;
                                iTNW = 0; iTSN = 0; iPW1 = 0; iPW3 = 0; iPE = 0;
iPS1 = 0; iRNW = 0; iRSNE = 0;
                                break;
                        case 4:
                                points = iTEW + iTSE + iPN1 + iPN2 + iPN3 +
iPW3 + iPS1 + iPS3 + iRENW;
                                iTEW = 0; iTSE = 0; iPN1 = 0; iPN2 = 0; iPN3= 0;
iPW3 = 0; iPS1 = 0; iPS3 = 0; iRENW = 0;
                                break;
                        case 5:
                                points = 2*iTNW + 2*iTWN + 2*iTES + 2*iTSE +
iPN3 + iPW1 + iPW3 + iPS1 + iPS3 + iRNW;
                                iTNW = 0; iTWN = 0; iTES = 0; iTSE = 0; iPN3 = 0;
iPW1 = 0; iPW3 = 0; iPS1 = 0; iPS3 = 0; iRNW = 0;
                                break;
                        case 6:
                                points = 2*iTNE + 4*iTEN + iPN3 + iPW1 + iPW2
+ iPW3 + iPS2 + iPS3 + iRNW + iRNS + iRNE;
                                iTNE = 0; iTEN = 0; iPN3 = 0; iPW1 = 0; iPW2 = 0;
iPW3 = 0; iPS2 = 0; iPS3 = 0; iRNW = 0; iRNS = 0; iRNE = 0;
                                break;
                        case 7:
                                points = 2*iTNS + 2*iTSN + iPW1 + iPW2 + iPW3
+ iPE + iRNW + iRNS + iRSNE;
                                iTNS= 0; iTSN = 0; iPW1 = 0; iPW2 = 0; iPW3 = 0;
iPE = 0; iRNW = 0; iRNS = 0; iRSNE = 0;
                                break;
                        case 8:
                                points = 2*iTWE + 2*iTEW + iPN1 + iPN2 + iPN3
+ iPS1 + iPS2 + iPS3 + iRENW + iRWSE;
                                iTWE = 0; iTEW = 0; iPN1 = 0; iPN2 = 0; iPN3 = 0;
iPS1 = 0; iPS2 = 0; iPS3 = 0; iRENW = 0; iRWSE = 0;
```

```
                        break;
                    case 9:
                        points = iTNW + iTWN + iPN1 + iPN3 + iPW1 +
iPS1 + iPS2 + iPS3 + 2*iRWN + iRWSE;
                        iTNW = 0; iTWN = 0; iPN1 = 0; iPN3 = 0; iPW1 =
0; iPS1 = 0; iPS2 = 0; iPS3 = 0; iRWN = 0; iRWSE = 0;
                        break;
                    case 10:
                        points = iPN2 + iPW1 + iPW2 + iPW3 + iPS2 + iPE
+ iRNW + iRNS + 2*iRSNE;
                        iPN2 = 0; iPW1 = 0; iPW2 = 0; iPW3 = 0; iPS2 = 0;
iPE = 0; iRNW = 0; iRNS = 0; iRSNE = 0;
                        break;
                    case 11:
                        points = iPN1 +  iPN2 +  iPN3 + iPW2 + iPS1 +
iPS2 + iPS3 + 2*iRENW + 2*iRWSE;
                        iPN1 = 0; iPN2 = 0; iPN3 = 0; iPW2 = 0; iPS1 = 0;
iPS2 = 0; iPS3 = 0; iRENW = 0; iRWSE = 0;
                        break;
                    case 12:
                        points = iPN2 +  iPN3 +  iPW1 + iPW2 + iPW3 +
iPE + iPS1 + iPS2 + 2*iRNE + 2*iRSW;
                        iPN2 = 0; iPN3 = 0; iPW1 = 0; iPW2 = 0; iPW3 = 0;
iPE = 0; iPS1 = 0; iPS2 = 0; iRNE = 0; iRSW = 0;
                        break;
                    case 13:
                        points = iPN1 +  iPN2 +  iPN3 + iPW1 + iPW2 +
iPS1 + iPS2 + iPS3 + 2*iRWN + 2*iRES;
                        iPN1 = 0; iPN2 = 0; iPN3 = 0; iPW1 = 0; iPW2 = 0;
iPS1 = 0; iPS2 = 0; iPS3 = 0; iRWN = 0; iRES = 0;
                        break;
                    case 14:
                        points = iTSE + iTES + iPN1 + iPN2 + iPN3 + iPW1
+ iPW2 + iPW3 + iPS1 + iPS3 + iRES;
                        iTSE = 0; iTES = 0; iPN1 = 0; iPN2 = 0; iPN3 = 0;
iPW1 = 0; iPW2 = 0; iPW3 = 0; iPS1 = 0; iPS3 = 0; iRES = 0;
                        break;
                    case 15:
                        points = iTNE + iTEN + iPN1 + iPN3 + iPW1 +
iPW2 + iPS1 + iPS2 + iPS3 + 2*iRWSE;
                        iTNE = 0; iTEN = 0; iPN1 = 0; iPN3 = 0; iPW1 = 0;
iPW2 = 0; iPS1 = 0; iPS2 = 0; iPS3 = 0; iRWSE = 0;
                        break;
                    case 16:
                        points = 0;
                        break;
                }
                //Previous order calculations
                switch(x)
                {
                    case 0:
                        if((previousOrder == 11) || (previousOrder == 13))
{points = points + 2;}
                        break;
                    case 1:
                        if((previousOrder == 11) || (previousOrder == 13))
{points = points + 2;}
                        break;
                    case 2:
                        if((previousOrder == 11) || (previousOrder == 13))
{points = points + 2;}
                        break;
                    case 3:
                        if((previousOrder == 11) || (previousOrder == 13))
{points = points + 2;}
                        break;
                    case 4:
                        if((previousOrder == 10) || (previousOrder == 12))
{points = points + 2;}
                        break;
                    case 5:
                        if((previousOrder == 12) || (previousOrder == 13))
{points = points + 2;}
                        break;
                    case 6:
                        if((previousOrder == 11) || (previousOrder == 13))
{points = points + 2;}
                        break;
                    case 7:
                        if((previousOrder == 8) || (previousOrder == 13))
{points = points + 2;}
                        break;
                    case 8:
                        if((previousOrder == 7) || (previousOrder == 12))
{points = points + 2;}
                        break;
                    case 9:
                        if((previousOrder == 10) || (previousOrder == 14))
{points = points + 2;}
                        break;
                    case 10:
                        if((previousOrder == 8) || (previousOrder == 9))
{points = points + 2;}
                        break;
                    case 11:
                        if((previousOrder == 5) || (previousOrder == 7))
{points = points + 2;}
                        break;
                    case 12:
                        if((previousOrder == 5) || (previousOrder == 8))
{points = points + 2;}
                        break;
                    case 13:
                        if((previousOrder == 5) || (previousOrder == 7))
{points = points + 2;}
                        break;
                    case 14:
                        if((previousOrder == 9) || (previousOrder == 12))
{points = points + 2;}
                        break;
                    case 15:
                        if((previousOrder == 10) || (previousOrder == 13))
{points = points + 2;}
                        break;
                    case 16:
                        //
                        break;
                }
                if(i == 0) {f = 6*points;}
                if(i == 1) {f = f + 5*points;}
                if(i == 2) {f = f + 4*points;}
                if(i == 3) {f = f + 3*points;}
                if(i == 4) {f = f + 2*points;}
                if(i == 5) {f = f + points;}
                printf("c%d %d%d%d.%d%d%d.%d.%d%d%d : %d%d%d.%d%d.
%d%d%d.%d%d : %d%d%d.%d%d.%d%d.%d%d%d   Order = %2d   Points =
%2d\n", i+1, iPN1, iPN2, iPN3, iPW1, iPW2, iPW3, iPE, iPS1, iPS2, iPS3, iTNW,
iTNE, iTNS, iTWN, iTWE, iTEN, iTEW, iTES, iTSN, iTSE, iRNW, iRNE, iRNS,
iRWN, iRWSE, iRENW, iRES, iRSW, iRSNE, x ,points);
            }
            printf("\n");
            //Output print out
            switch(bestParameterValue1)
            {
                case 0:
                    printf("Executing requests on drive order %d:\noTNW,
oTEN, oTSE, oPN1, oPN2, oPN3, oPS1, oPS3, oRNW\n", bestParameterValue1);
                    break;
                case 1:
                    printf("Executing requests on drive order %d:\noTNW,
oTEN, oTWE, oPN3, oPW1, oPS1, oPS2, oPS3, oRWSE\n", bestParameterValue1);
                    break;
                case 2:
                    printf("Executing requests on drive order %d:\noTNS,
oTEN, oTSE, oPW1, oPW2, oPW3, oPN3, oPS3, oRNW, oRNS\n",
bestParameterValue1);
                    break;
                case 3:
                    printf("Executing requests on drive order %d:\noTNW,
oTSN, oPW1, oPW3, oPE, oPS1, oRNW, oRSNE\n", bestParameterValue1);
                    break;
                case 4:
                    printf("Executing requests on drive order %d:\noTEW,
oTSE, oPN1, oPN2, oPN3, oPW3, oPS1, oPS3, oRENW\n", bestParameterValue1);
                    break;
                case 5:
                    printf("Executing requests on drive order %d:\noTNW,
oTWN, oTES, oTSE, oPN3, oPW1, oPW3, oPS1, oPS3, oRNW\n",
bestParameterValue1);
                    break;
                case 6:
                    printf("Executing requests on drive order %d:\noTNE,
oTEN, oPN3, oPW1, oPW2, oPW3, oPS2, oPS3, oRNW, oRNS, oRNE\n",
bestParameterValue1);
                    break;
                case 7:
                    printf("Executing requests on drive order %d:\noTNS,
oTSN, oPW1, oPW2, oPW3, oPE, oRNW, oRNS, oRSNE\n",
bestParameterValue1);
                    break;
                case 8:
                    printf("Executing requests on drive order %d:\noTWE,
oTEW, oPN1, oPN2, oPN3, oPS1, oPS2, oPS3, oRENW, oRWSE\n",
bestParameterValue1);
                    break;
                case 9:
                    printf("Executing requests on drive order %d:\noTNW,
oTWN, oPN1, oPN3, oPW1, oPS1, oPS2, oPS3, oRWN, oRWSE\n",
bestParameterValue1);
                    break;
                case 10:
                    printf("Executing requests on drive order %d:\noPN2,
oPW1, oPW2, oPW3, oPS2, oPE, oRNW, oRNS, oRSNE\n", bestParameterValue1);
                    break;
                case 11:
                    printf("Executing requests on drive order %d:\noPN1, oPN2,
oPN3, oPW2, oPS1, oPS2, oPS3, oRENW, oRWSE\n", bestParameterValue1);
                    break;
                case 12:
                    printf("Executing requests on drive order %d:\noPN2, oPN3,
oPW1, oPW2, oPW3, oPE, oPS1, oPS2, oRNE, oRSW\n", bestParameterValue1);
                    break;
                case 13:
                    printf("Executing requests on drive order %d:\noPN1, oPN2,
oPN3, oPW1, oPW2, oPS1, oPS2, oPS3, oRWN, oRES\n", bestParameterValue1);
                    break;
                case 14:
                    printf("Executing requests on drive order %d:\noTSE, oTES,
oPN1, oPN2, oPN3, oPW1, oPW2, oPW3, oPS1, oPS3, oRES\n",
bestParameterValue1);
                    break;
                case 15:
```

```
                        printf("Executing requests on drive order %d:\noTNE,
oTEN, oPN1, oPN3, oPW1, oPW2, oPS1, oPS2, oPS3, oRWSE\n",
bestParameterValue1);
                        break;
                case 16:
                        printf("Do not executing any drive order.\n");
                        break;
        }
        printf("\n");
}

int TournamentSelect(int fitness[30], int populationSize, float
tournamentProbability)
{
        int indv;
        int itemp1 = 1 + rand() %(populationSize - 1);
        int itemp2 = 1 + rand() %(populationSize - 1);

        float r = 0;
        int ri = 0;
        ri = rand() %99;
        r = ((float)ri/100);

        if(r <= tournamentProbability)
        {
                if(fitness[itemp1] >= fitness[itemp2])
                {
                        indv = itemp1;
                }
                else
                {
                        indv = itemp2;
                }
        }
        else
        {
                if(fitness[itemp1] >= fitness[itemp2])
                {
                        indv = itemp2;
                }
                else
                {
                        indv = itemp1;
                }
        }
        return indv;
}

float RandFunktion(void)
{
        float r = 0;
        int ri = rand() %99;
        return r = ((float)ri/100);
}

void PrintOutFunktion(int msgOrder, int numberEvalutaionsPerformed, int
generationNbr, int maxFitness, int bestParameterValue1, int bestParameterValue2,
int bestParameterValue3, int bestParameterValue4, int bestParameterValue5, int
bestParameterValue6)
{
    if(msgOrder == 0)
    {
        printf("Start \n");
    }
    else if(msgOrder == 1)
    {
        printf("Generation: %3d   Evaluation: %3d   Max fitness: %3d \n",
generationNbr, numberEvalutaionsPerformed, maxFitness);
    }
    else if(msgOrder == 2)
    {
        printf("Result generation %3d: c1 = %d c2 = %d c3 = %d c4 = %d c5 =
%d c6 = %d Fitness = %d\n", generationNbr, bestParameterValue1,
bestParameterValue2, bestParameterValue3, bestParameterValue4,
bestParameterValue5, bestParameterValue6, maxFitness);
    }
    else if(msgOrder == 3)
    {
        printf("\nEnd or Termination Criterion is reached. \n\n");
    }
    else
    {
        printf("Error!\n\n");
    }
}

int main()
{
        int populationSize = 30;
        int numberOfGenes = 24;
        int numberOfChromosomes = 6;
        float crossoverProbability = 0.85;
        float mutationProbability = 0.03;
        float tournamentProbability = 0.90;
        int inputValue[29];
        int inputValues[10][29];// [=sim][29]
        int maxGeneration = 100;
        int numberEvalutaionsPerformed = 0;
        int maxFitness = 0;
        int totalMaxFitness = 0;
        int fitness[populationSize];
        int population[populationSize][numberOfGenes];
        int tempPopulation[populationSize][numberOfGenes];
        int goodIndividuals[populationSize][numberOfGenes];
        int newIndividual1[numberOfGenes];
        int newIndividual2[numberOfGenes];
        int tempIndividual[numberOfGenes];
        int parameterValue1 = 0;
        int parameterValue2 = 0;
        int parameterValue3 = 0;
        int parameterValue4 = 0;
        int parameterValue5 = 0;
        int parameterValue6 = 0;
        int bestInduvidual = 0;
        int bestParameterValue1 = 0;
        int bestParameterValue2 = 0;
        int bestParameterValue3 = 0;
        int bestParameterValue4 = 0;
        int bestParameterValue5 = 0;
        int bestParameterValue6 = 0;
        int previousExcecution = 16;//Store c1 (E) from previous time period
        int sim = 10;//Number of simulations
        // Other variables
        int generationNbr, i = 0, j = 0, h = 0, indv1, indv2; float r = 0; int msgOrder
= 0, mEInt = 0; float mEfloat = 0;

        for(j = 0; j <= 28; j++)
        {
                inputValue[j] = 0;
        }
        for(i = 0; i <= sim - 1; i++)
        {
                for(j = 0; j <= 28; j++)
                {
                        inputValues[i][j] = 0;
                }
        }
        //Only 5 first is filled with inputs

        //Init inputValue 0
        inputValues[0][0] = 1; /* PN1 */                inputValues[0][10] = 0; /*
TNW */                inputValues[0][20] = 1; /* RNW */
        inputValues[0][1] = 1; /* PN2 */                inputValues[0][11] = 0; /*
TNE */                inputValues[0][21] = 1; /* RNE */
        inputValues[0][2] = 1; /* PN3 */                inputValues[0][12] = 1; /*
TNS */                inputValues[0][22] = 1; /* RNS */
        inputValues[0][3] = 1; /* PW1 */                inputValues[0][13] = 0; /*
TWN */                inputValues[0][23] = 1; /* RWN */
        inputValues[0][4] = 1; /* PW2 */                inputValues[0][14] = 1; /*
TWE */                inputValues[0][24] = 1; /* RWSE */
        inputValues[0][5] = 1; /* PW3 */                inputValues[0][15] = 0; /*
TEN */                inputValues[0][25] = 1; /* RENW */
        inputValues[0][6] = 1; /* PE  */                inputValues[0][16] = 1; /*
TEW */                inputValues[0][26] = 1; /* RES */
        inputValues[0][7] = 1; /* PS1 */                inputValues[0][17] = 0; /*
TES */                inputValues[0][27] = 1; /* RSW */
        inputValues[0][8] = 1; /* PS2 */                inputValues[0][18] = 1; /*
TSN */                inputValues[0][28] = 1; /* RSNE */
        inputValues[0][9] = 1; /* PS3 */                inputValues[0][19] = 0; /*
TSE */

        //Init inputValue 1
        inputValues[1][0] = 0; /* PN1 */                inputValues[1][10] = 0; /*
TNW */                inputValues[1][20] = 0; /* RNW */
        inputValues[1][1] = 0; /* PN2 */                inputValues[1][11] = 0; /*
TNE */                inputValues[1][21] = 0; /* RNE */
        inputValues[1][2] = 0; /* PN3 */                inputValues[1][12] = 0; /*
TNS */                inputValues[1][22] = 0; /* RNS */
        inputValues[1][3] = 0; /* PW1 */                inputValues[1][13] = 0; /*
TWN */                inputValues[1][23] = 0; /* RWN */
        inputValues[1][4] = 0; /* PW2 */                inputValues[1][14] = 0; /*
TWE */                inputValues[1][24] = 0; /* RWSE */
        inputValues[1][5] = 0; /* PW3 */                inputValues[1][15] = 0; /*
TEN */                inputValues[1][25] = 0; /* RENW */
        inputValues[1][6] = 0; /* PE  */                inputValues[1][16] = 0; /*
TEW */                inputValues[1][26] = 0; /* RES */
        inputValues[1][7] = 0; /* PS1 */                inputValues[1][17] = 0; /*
TES */                inputValues[1][27] = 0; /* RSW */
        inputValues[1][8] = 0; /* PS2 */                inputValues[1][18] = 0; /*
TSN */                inputValues[1][28] = 0; /* RSNE */
        inputValues[1][9] = 0; /* PS3 */                inputValues[1][19] = 0; /*
TSE */

        //Init inputValue 2
        inputValues[2][0] = 0; /* PN1 */                inputValues[2][10] = 0; /*
TNW */                inputValues[2][20] = 0; /* RNW */
        inputValues[2][1] = 0; /* PN2 */                inputValues[2][11] = 0; /*
TNE */                inputValues[2][21] = 0; /* RNE */
        inputValues[2][2] = 0; /* PN3 */                inputValues[2][12] = 0; /*
TNS */                inputValues[2][22] = 0; /* RNS */
        inputValues[2][3] = 0; /* PW1 */                inputValues[2][13] = 0; /*
TWN */                inputValues[2][23] = 0; /* RWN */
        inputValues[2][4] = 0; /* PW2 */                inputValues[2][14] = 0; /*
TWE */                inputValues[2][24] = 0; /* RWSE */
        inputValues[2][5] = 0; /* PW3 */                inputValues[2][15] = 0; /*
TEN */                inputValues[2][25] = 0; /* RENW */
        inputValues[2][6] = 0; /* PE  */                inputValues[2][16] = 0; /*
TEW */                inputValues[2][26] = 0; /* RES */
        inputValues[2][7] = 0; /* PS1 */                inputValues[2][17] = 0; /*
TES */                inputValues[2][27] = 0; /* RSW */
        inputValues[2][8] = 0; /* PS2 */                inputValues[2][18] = 0; /*
TSN */                inputValues[2][28] = 0; /* RSNE */
```

```
	inputValues[2][9] = 0; /* PS3 */			inputValues[2][19] = 0; /*
TSE */

	//Init inputValue 3
	inputValues[3][0] = 0; /* PN1 */			inputValues[3][10] = 0; /*
TNW */			inputValues[3][20] = 0; /* RNW */
	inputValues[3][1] = 0; /* PN2 */			inputValues[3][11] = 0; /*
TNE */			inputValues[3][21] = 0; /* RNE */
	inputValues[3][2] = 0; /* PN3 */			inputValues[3][12] = 0; /*
TNS */			inputValues[3][22] = 0; /* RNS */
	inputValues[3][3] = 0; /* PW1 */			inputValues[3][13] = 0; /*
TWN */			inputValues[3][23] = 0; /* RWN */
	inputValues[3][4] = 0; /* PW2 */			inputValues[3][14] = 0; /*
TWE */			inputValues[3][24] = 0; /* RWSE */
	inputValues[3][5] = 0; /* PW3 */			inputValues[3][15] = 0; /*
TEN */			inputValues[3][25] = 0; /* RENW */
	inputValues[3][6] = 0; /* PE  */			inputValues[3][16] = 0; /*
TEW */			inputValues[3][26] = 0; /* RES */
	inputValues[3][7] = 0; /* PS1 */			inputValues[3][17] = 0; /*
TES */			inputValues[3][27] = 0; /* RSW */
	inputValues[3][8] = 0; /* PS2 */			inputValues[3][18] = 0; /*
TSN */			inputValues[3][28] = 0; /* RSNE */
	inputValues[3][9] = 0; /* PS3 */			inputValues[3][19] = 0; /*
TSE */

	//Init inputValue 4
	inputValues[4][0] = 0; /* PN1 */			inputValues[4][10] = 0; /*
TNW */			inputValues[4][20] = 0; /* RNW */
	inputValues[4][1] = 0; /* PN2 */			inputValues[4][11] = 0; /*
TNE */			inputValues[4][21] = 0; /* RNE */
	inputValues[4][2] = 0; /* PN3 */			inputValues[4][12] = 0; /*
TNS */			inputValues[4][22] = 0; /* RNS */
	inputValues[4][3] = 0; /* PW1 */			inputValues[4][13] = 0; /*
TWN */			inputValues[4][23] = 0; /* RWN */
	inputValues[4][4] = 0; /* PW2 */			inputValues[4][14] = 0; /*
TWE */			inputValues[4][24] = 0; /* RWSE */
	inputValues[4][5] = 0; /* PW3 */			inputValues[4][15] = 0; /*
TEN */			inputValues[4][25] = 0; /* RENW */
	inputValues[4][6] = 0; /* PE  */			inputValues[4][16] = 0; /*
TEW */			inputValues[4][26] = 0; /* RES */
	inputValues[4][7] = 0; /* PS1 */			inputValues[4][17] = 0; /*
TES */			inputValues[4][27] = 0; /* RSW */
	inputValues[4][8] = 0; /* PS2 */			inputValues[4][18] = 0; /*
TSN */			inputValues[4][28] = 0; /* RSNE */
	inputValues[4][9] = 0; /* PS3 */			inputValues[4][19] = 0; /*
TSE */

	//Set values to population
	for(i = 0; i <= populationSize - 1;  i++)
	{
		for(j = 0; j <= numberOfGenes - 1; j++)
		{
			if((rand() % 6) >= 3)
			{
				population[i][j] = 1;
			}
			else
			{
				population[i][j] = 0;
			}
		}
	}
	//Set values to tempPopulation
	for(i = 0; i <= populationSize - 1;  i++)
	{
		for(j = 0; j <= numberOfGenes - 1; j++)
		{
			tempPopulation[i][j] = 0;
		}
	}
	for(j = 0; j <= 28; j++)
	{
		inputValue[j] = 0;
	}

	//Start main program
	PrintOutFunktion(msgOrder, mEInt, mEInt, mEInt, mEInt, mEInt, mEInt,
mEInt, mEInt, mEInt);

	for(h = 0; h <= sim - 1;  h++)
	{
		printf("\n**** Standard GA ****\n");
		printf("Previous executed drive order: %d, Simulation: T%d\n\n",
previousExcecution, h);

		for(i = 0; i <= sim - 1;  i++)
		{
			for(j = 0; j <= numberOfGenes - 1; j++)
			{
				population[i][j] = goodIndividuals[i][j];
			}
		}

		//UpdateInputValue(inputValue, bestParameterValue1, h);
		if(1)
		{
			int iPN1 = inputValue[0];
			int iPN2 = inputValue[1];
			int iPN3 = inputValue[2];
			int iPW1 = inputValue[3];
			int iPW2 = inputValue[4];
			int iPW3 = inputValue[5];
			int iPE  = inputValue[6];
			int iPS1 = inputValue[7];
			int iPS2 = inputValue[8];
			int iPS3 = inputValue[9];
			int iTNW = inputValue[10];
			int iTNE = inputValue[11];
			int iTNS = inputValue[12];
			int iTWN = inputValue[13];
			int iTWE = inputValue[14];
			int iTEN = inputValue[15];
			int iTEW = inputValue[16];
			int iTES = inputValue[17];
			int iTSN = inputValue[18];
			int iTSE = inputValue[19];
			int iRNW = inputValue[20];
			int iRNE = inputValue[21];
			int iRNS = inputValue[22];
			int iRWN = inputValue[23];
			int iRWSE = inputValue[24];
			int iRENW = inputValue[25];
			int iRES = inputValue[26];
			int iRSW = inputValue[27];
			int iRSNE = inputValue[28];

			switch(bestParameterValue1)
			{
				case 0:
					iTNW = 0; iTEN = 0; iTSE = 0; iPN1 = 0;
iPN2 = 0; iPN3 = 0; iPS1 = 0; iPS3 = 0; iRNW = 0;
					break;
				case 1:
					iTNW = 0; iTEN = 0; iTWE = 0; iPN3 = 0;
iPW1 = 0; iPS1 = 0; iPS2 = 0; iPS3 = 0; iRWSE = 0;
					break;
				case 2:
					iTNS = 0; iTEN = 0; iTSE = 0; iPW1 = 0;
iPW2 = 0; iPW3 = 0; iPN3 = 0; iPS3 = 0; iRNW = 0; iRNS = 0;
					break;
				case 3:
					iTNW = 0; iTSN = 0; iPW1 = 0; iPW3 = 0;
iPE = 0; iPS1 = 0; iRNW = 0; iRSNE = 0;
					break;
				case 4:
					iTEW = 0; iTSE = 0; iPN1 = 0; iPN2 = 0;
iPN3= 0; iPW3 = 0; iPS1 = 0; iPS3 = 0; iRENW = 0;
					break;
				case 5:
					iTNW = 0; iTWN = 0; iTES = 0; iTSE = 0;
iPN3 = 0; iPW1 = 0; iPW3 = 0; iPS1 = 0; iPS3 = 0; iRNW = 0;
					break;
				case 6:
					iTNE = 0; iTEN = 0; iPN3 = 0; iPW1 = 0;
iPW2 = 0; iPW3 = 0; iPS2 = 0; iPS3 = 0; iRNW = 0; iRNS = 0; iRNE = 0;
					break;
				case 7:
					iTNS= 0; iTSN = 0; iPW1 = 0; iPW2 = 0;
iPW3 = 0; iPE = 0; iRNW = 0; iRNS = 0; iRSNE = 0;
					break;
				case 8:
					iTWE = 0; iTEW = 0; iPN1 = 0; iPN2 = 0;
iPN3 = 0; iPS1 = 0; iPS2 = 0; iPS3 = 0; iRENW = 0; iRWSE = 0;
					break;
				case 9:
					iTNW = 0; iTWN = 0; iPN1 = 0; iPN3 = 0;
iPW1 = 0; iPS1 = 0; iPS2 = 0; iPS3 = 0; iRWN = 0; iRWSE = 0;
					break;
				case 10:
					iPN2 = 0; iPW1 = 0; iPW2 = 0; iPW3 = 0;
iPS2 = 0; iPE = 0; iRNW = 0; iRNS = 0; iRSNE = 0;
					break;
				case 11:
					iPN1 = 0; iPN2 = 0; iPN3 = 0; iPW2 = 0;
iPS1 = 0; iPS2 = 0; iPS3 = 0; iRENW = 0; iRWSE = 0;
					break;
				case 12:
					iPN2 = 0; iPN3 = 0; iPW1 = 0; iPW2 = 0;
iPW3 = 0; iPE = 0; iPS1 = 0; iPS2 = 0; iRNE = 0; iRSW = 0;
					break;
				case 13:
					iPN1 = 0; iPN2 = 0; iPN3 = 0; iPW1 = 0;
iPW2 = 0; iPS1 = 0; iPS2 = 0; iPS3 = 0; iRWN = 0; iRES = 0;
					break;
				case 14:
					iTSE = 0; iTES = 0; iPN1 = 0; iPN2 = 0;
iPN3 = 0; iPW1 = 0; iPW2 = 0; iPW3 = 0; iPS1 = 0; iPS3 = 0; iRES = 0;
					break;
				case 15:
					iTNE = 0; iTEN = 0; iPN1 = 0; iPN3 = 0;
iPW1 = 0; iPW2 = 0; iPS1 = 0; iPS2 = 0; iPS3 = 0; iRWSE = 0;
					break;
				case 16:
					break;
			}
			// Rewrite inputValue
			inputValue[0] = iPN1;
			inputValue[1] = iPN2;
			inputValue[2] = iPN3;
			inputValue[3] = iPW1;
			inputValue[4] = iPW2;
			inputValue[5] = iPW3;
```

```
                        inputValue[6] = iPE;
                        inputValue[7] = iPS1;
                        inputValue[8] = iPS2;
                        inputValue[9] = iPS3;
                        inputValue[10] = iTNW;
                        inputValue[11] = iTNE;
                        inputValue[12] = iTNS;
                        inputValue[13] = iTWN;
                        inputValue[14] = iTWE;
                        inputValue[15] = iTEN;
                        inputValue[16] = iTEW;
                        inputValue[17] = iTES;
                        inputValue[18] = iTSN;
                        inputValue[19] = iTSE;
                        inputValue[20] = iRNW;
                        inputValue[21] = iRNE;
                        inputValue[22] = iRNS;
                        inputValue[23] = iRWN;
                        inputValue[24] = iRWSE;
                        inputValue[25] = iRENW;
                        inputValue[26] = iRES;
                        inputValue[27] = iRSW;
                        inputValue[28] = iRSNE;

                        for(j = 0; j <= 28; j++)
                        {
                                if(inputValue[j] != 0) {inputValue[j] = inputValue[j]
+ 1;}// Add priority to remaining inputs
                                else {inputValue[j] = inputValue[j] + inputValues[h]
[j];}// Add new inputs if there are any
                        }
                }


                //Start of standard GA loop
                for (generationNbr = 0; generationNbr <= maxGeneration - 1;
generationNbr++)
                {
                        for(i = 0; i <= populationSize - 1; i++)
                        {
                                parameterValue1 = DecodeChromosome(population,
i, numberOfGenes, 1, numberOfChromosomes);
                                parameterValue2 = DecodeChromosome(population,
i, numberOfGenes, 2, numberOfChromosomes);
                                parameterValue3 = DecodeChromosome(population,
i, numberOfGenes, 3, numberOfChromosomes);
                                parameterValue4 = DecodeChromosome(population,
i, numberOfGenes, 4, numberOfChromosomes);
                                parameterValue5 = DecodeChromosome(population,
i, numberOfGenes, 5, numberOfChromosomes);
                                parameterValue6 = DecodeChromosome(population,
i, numberOfGenes, 6, numberOfChromosomes);
                                fitness[i] = EvaluateIndividual(parameterValue1,
parameterValue2, parameterValue3, parameterValue4, parameterValue5,
parameterValue6, inputValue, previousExcecution);
                                numberEvalutaionsPerformed =
numberEvalutaionsPerformed + 1;
                                if ((fitness[i] >= maxFitness) && (fitness[i] !=
maxFitness))//fitness > maxFitness
                                {
                                        maxFitness = fitness[i];
                                        bestInduvidual = i;
                                        bestParameterValue1 = parameterValue1;
                                        bestParameterValue2 = parameterValue2;
                                        bestParameterValue3 = parameterValue3;
                                        bestParameterValue4 = parameterValue4;
                                        bestParameterValue5 = parameterValue5;
                                        bestParameterValue6 = parameterValue6;
                                        msgOrder = 1;
                                        PrintOutFunktion(msgOrder,
numberEvalutaionsPerformed, generationNbr + 1, maxFitness,
bestParameterValue1, bestParameterValue2, bestParameterValue3,
bestParameterValue4, bestParameterValue5, bestParameterValue6);
                                }
                        }
                        numberEvalutaionsPerformed = 0;

                        for(i = 0; i <= populationSize - 1; i++)//copy array:
tempPopulation = population;
                        {
                                for(j = 0; j <= numberOfGenes - 1; j++)
                                {
                                        tempPopulation[i][j] = population[i][j];
                                }
                        }

                        for(i = 0; i <= populationSize - 1; i = i + 2)
                        {
                                indv1 = TournamentSelect(fitness, populationSize,
tournamentProbability);
                                indv2 = TournamentSelect(fitness, populationSize,
tournamentProbability);
                                r = RandFunktion();
                                if(r <= crossoverProbability)
                                {
                                        int j, cp = rand() %numberOfGenes - 1; //Get
crossover position.

                                        for(j = 0; j <= numberOfGenes - 1;  j++)
                                        {
                                                if(j <= cp)
                                                {
                                                        newIndividual1[j] =
population[indv1][j];
                                                        newIndividual2[j] =
population[indv2][j];
                                                }
                                                else
                                                {
                                                        newIndividual1[j] =
population[indv2][j];
                                                        newIndividual2[j] =
population[indv1][j];
                                                }
                                        }//end of crossover

                                        for(j = 0; j <= numberOfGenes - 1; j++)
                                        {
                                                tempPopulation[i][j] =
newIndividual1[j];
                                                tempPopulation[i + 1][j] =
newIndividual2[j];
                                        }
                                }
                                else
                                {
                                        for(j = 0; j <= numberOfGenes - 1; j++)
                                        {
                                                tempPopulation[i][j] =
population[indv1][j];
                                                tempPopulation[i + 1][j] =
population[indv2][j];
                                        }
                                }
                        }

                        //Set best individual first into tempPopulation
                        for(j = 0; j <= numberOfGenes - 1; j++)
                        {
                                tempPopulation[0][j] = population[bestInduvidual]
[j];
                        }

                        for(i = 1; i <= populationSize - 1; i++)
                        {
                                for(j = 0; j <= numberOfGenes - 1; j++)
                                {
                                        tempIndividual[j] = tempPopulation[i][j];
                                        r = RandFunktion();
                                        if (r <= mutationProbability)
                                        {
                                                r = RandFunktion();
                                                if(r >= 0.5)
                                                {
                                                        tempIndividual[j] = 1;
                                                }
                                                else
                                                {
                                                        tempIndividual[j] = 0;
                                                }
                                        }
                                        tempPopulation[i][j] = tempIndividual[j];
                                }
                        }

                        for(i = 0; i <= populationSize - 1; i++)//copy array
population = tempPopulation
                        {
                                for(j = 0; j <= numberOfGenes - 1; j++)
                                {
                                        population[i][j] = tempPopulation[i][j];
                                }
                        }
                        //New population is made for next generation

                        bestInduvidual = 0;// Resets until next generation
                        if ((maxFitness >= totalMaxFitness) && (maxFitness !=
totalMaxFitness))//maxFitness > totalMaxFitness
                        {
                                for(j = 0; j <= numberOfGenes - 1; j++)
                                {
                                        goodIndividuals[h][j] = tempPopulation[0][j];
                                }
                                totalMaxFitness = maxFitness;
                        }
                }

                ReturnResult(bestParameterValue1, bestParameterValue2,
bestParameterValue3, bestParameterValue4, bestParameterValue5,
bestParameterValue6, inputValue, maxFitness);
                printf("..........\n");
                previousExcecution = bestParameterValue1;
                maxFitness = 0;// Resets until next generation
        }

        printf("\nGood individuals in binary form:\n\n");
        for(int h = 0; h <= sim - 1;  h++)
        {
                for(j = 0; j <= numberOfGenes - 1; j++)
                {
                        printf("%d", goodIndividuals[h][j]);
                }
                bestParameterValue1 = DecodeChromosome(goodIndividuals, h,
```

```
numberOfGenes, 1, numberOfChromosomes);
                bestParameterValue2 = DecodeChromosome(goodIndividuals, h,
numberOfGenes, 2, numberOfChromosomes);
                bestParameterValue3 = DecodeChromosome(goodIndividuals, h,
numberOfGenes, 3, numberOfChromosomes);
                bestParameterValue4 = DecodeChromosome(goodIndividuals, h,
numberOfGenes, 4, numberOfChromosomes);
                bestParameterValue5 = DecodeChromosome(goodIndividuals, h,
numberOfGenes, 5, numberOfChromosomes);
                bestParameterValue6 = DecodeChromosome(goodIndividuals, h,
numberOfGenes, 6, numberOfChromosomes);
                printf(" = %2d - %2d - %2d - %2d - %2d - %2d\n",
bestParameterValue1, bestParameterValue2, bestParameterValue3,
bestParameterValue4, bestParameterValue5, bestParameterValue6);
        }

        msgOrder = 3;
        PrintOutFunktion(msgOrder, mEInt, mEInt, mEInt, mEInt, mEInt, mEInt,
mEInt, mEInt, mEInt);

        printf("\n");
        //getchar(); To stop command window.
}
```

Appendix H Program code of DFS version

Source code for DFS version with *Normal case* inputs implemented.

```
/* DFS version normal case */

#include <stdio.h>
#include <conio.h>

int EvaluateIndividual(int c1, int c2, int c3, int c4, int c5, int c6, int inputValue[29],
int previousExcecution)
{
	int f = 0, i = 0, x = 0, p = 0, points = 0, previousOrder;

	int iPN1 = inputValue[0];
	int iPN2 = inputValue[1];
	int iPN3 = inputValue[2];
	int iPW1 = inputValue[3];
	int iPW2 = inputValue[4];
	int iPW3 = inputValue[5];
	int iPE  = inputValue[6];
	int iPS1 = inputValue[7];
	int iPS2 = inputValue[8];
	int iPS3 = inputValue[9];
	int iTNW = inputValue[10];
	int iTNE = inputValue[11];
	int iTNS = inputValue[12];
	int iTWN = inputValue[13];
	int iTWE = inputValue[14];
	int iTEN = inputValue[15];
	int iTEW = inputValue[16];
	int iTES = inputValue[17];
	int iTSN = inputValue[18];
	int iTSE = inputValue[19];
	int iRNW = inputValue[20];
	int iRNE = inputValue[21];
	int iRNS = inputValue[22];
	int iRWN = inputValue[23];
	int iRWSE = inputValue[24];
	int iRENW = inputValue[25];
	int iRES = inputValue[26];
	int iRSW = inputValue[27];
	int iRSNE = inputValue[28];

	x = previousExcecution;
	for(i = 0; i <= 6 - 1; i++)
	{
		previousOrder = x;
		if(i == 0) {x = c1;}
		if(i == 1) {x = c2;}
		if(i == 2) {x = c3;}
		if(i == 3) {x = c4;}
		if(i == 4) {x = c5;}
		if(i == 5) {x = c6;}
		points = 0;

		//Inputs functions calculations
		switch(x)
		{
			case 0:
				points = iTNW + iTEN + iTSE + iPN1 + iPN2 +
iPN3 + iPS1 + iPS3 + iRNW;
				iTNW = 0; iTEN = 0; iTSE = 0; iPN1 = 0; iPN2 = 0;
iPN3 = 0; iPS1 = 0; iPS3 = 0; iRNW = 0;
				break;
			case 1:
				points = iTNW + iTEN + iTWE + iPN3 + iPW1 +
iPS1 + iPS2 + iPS3 + iRWSE;
				iTNW = 0; iTEN = 0; iTWE = 0; iPN3 = 0; iPW1 =
0; iPS1 = 0; iPS2 = 0; iPS3 = 0; iRWSE = 0;
				break;
			case 2:
				points = iTNS + iTEN + iTSE + iPW1 + iPW2 +
iPW3 + iPN3 + iPS3 + iRNW + iRNS;
				iTNS = 0; iTEN = 0; iTSE = 0; iPW1 = 0; iPW2 = 0;
iPW3 = 0; iPN3 = 0; iPS3 = 0; iRNW = 0; iRNS = 0;
				break;
			case 3:
				points = iTNW + iTSN + iPW1 + iPW3 + iPE + iPS1
+ iRNW + iRSNE;
				iTNW = 0; iTSN = 0; iPW1 = 0; iPW3 = 0; iPE = 0;
iPS1 = 0; iRNW = 0; iRSNE = 0;
				break;
			case 4:
				points = iTEW + iTSE + iPN1 + iPN2 + iPN3 +
iPW3 + iPS1 + iPS3 + iRENW;
				iTEW = 0; iTSE = 0; iPN1 = 0; iPN2 = 0; iPN3= 0;
iPW3 = 0; iPS1 = 0; iPS3 = 0; iRENW = 0;
				break;
			case 5:
				points = 2*iTNW + 2*iTWN + 2*iTES + 3*iTSE +
iPN3 + iPW1 + iPW3 + iPS1 + iPS3 + iRNW;
				iTNW = 0; iTWN = 0; iTES = 0; iTSE = 0; iPN3 = 0;
iPW1 = 0; iPW3 = 0; iPS1 = 0; iPS3 = 0; iRNW = 0;
				break;
			case 6:
				points = 4*iTNE + 4*iTEN + iPN3 + iPW1 + iPW2
+ iPW3 + iPS2 + iPS3 + iRNW + iRNS + iRNE;
				iTNE = 0; iTEN = 0; iPN3 = 0; iPW1 = 0; iPW2 = 0;
iPW3 = 0; iPS2 = 0; iPS3 = 0; iRNW = 0; iRNS = 0; iRNE = 0;
				break;
			case 7:
				points = 2*iTNS + 2*iTSN + iPW1 + iPW2 + iPW3
+ iPE + iRNW + iRNS + iRSNE;
				iTNS= 0; iTSN = 0; iPW1 = 0; iPW2 = 0; iPW3 = 0;
iPE = 0; iRNW = 0; iRNS = 0; iRSNE = 0;
				break;
			case 8:
				points = 3*iTWE + 3*iTEW + iPN1 + iPN2 + iPN3
+ iPS1 + iPS2 + iPS3 + iRENW + iRWSE;
				iTWE = 0; iTEW = 0; iPN1 = 0; iPN2 = 0; iPN3 = 0;
iPS1 = 0; iPS2 = 0; iPS3 = 0; iRENW = 0; iRWSE = 0;
				break;
			case 9:
				points = iTNW + iTWN + iPN1 + iPN3 + iPW1 +
iPS1 + iPS2 + iPS3 + 2*iRWN + iRWSE;
				iTNW = 0; iTWN = 0; iPN1 = 0; iPN3 = 0; iPW1 =
0; iPS1 = 0; iPS2 = 0; iPS3 = 0; iRWN = 0; iRWSE = 0;
				break;
			case 10:
				points = iPN2 + iPW1 + iPW2 + iPW3 + iPS2 + iPE
+ iRNW + iRNS + 2*iRSNE;
				iPN2 = 0; iPW1 = 0; iPW2 = 0; iPW3 = 0; iPS2 = 0;
iPE = 0; iRNW = 0; iRNS = 0; iRSNE = 0;
				break;
			case 11:
				points = iPN1 +  iPN2 +  iPN3 + iPW2 + iPS1 +
iPS2 + iPS3 + 2*iRENW + 2*iRWSE;
				iPN1 = 0; iPN2 = 0; iPN3 = 0; iPW2 = 0; iPS1 = 0;
iPS2 = 0; iPS3 = 0; iRENW = 0; iRWSE = 0;
				break;
			case 12:
				points = iPN2 +  iPN3 +  iPW1 + iPW2 + iPW3 +
iPE + iPS1 + iPS2 + 2*iRNE + 2*iRSW;
				iPN2 = 0; iPN3 = 0; iPW1 = 0; iPW2 = 0; iPW3 = 0;
iPE = 0; iPS1 = 0; iPS2 = 0; iRNE = 0; iRSW = 0;
				break;
			case 13:
				points = iPN1 +  iPN2 +  iPN3 + iPW1 + iPW2 +
iPS1 + iPS2 + iPS3 + 2*iRWN + 2*iRES;
				iPN1 = 0; iPN2 = 0; iPN3 = 0; iPW1 = 0; iPW2 = 0;
iPS1 = 0; iPS2 = 0; iPS3 = 0; iRWN = 0; iRES = 0;
				break;
			case 14:
				points = iTSE + iTES + iPN1 + iPN2 + iPN3 + iPW1
+ iPW2 + iPW3 + iPS1 + iPS3 + iRES;
				iTSE = 0; iTES = 0; iPN1 = 0; iPN2 = 0; iPN3 = 0;
iPW1 = 0; iPW2 = 0; iPW3 = 0; iPS1 = 0; iPS3 = 0; iRES = 0;
				break;
			case 15:
				points = iTNE + iTEN + iPN1 + iPN3 + iPW1 +
iPW2 + iPS1 + iPS2 + iPS3 + 2*iRWSE;
				iTNE = 0; iTEN = 0; iPN1 = 0; iPN3 = 0; iPW1 = 0;
iPW2 = 0; iPS1 = 0; iPS2 = 0; iPS3 = 0; iRWSE = 0;
				break;
			case 16:
				points = 0;
				break;
		}
		//Previous order calculations
		switch(x)
		{
			case 0:
				if((previousOrder == 11) || (previousOrder == 13))
{points = points + 2;}
				if(previousOrder == 0) {points = points - 5;}
				break;
			case 1:
				if((previousOrder == 11) || (previousOrder == 13))
{points = points + 2;}
				if(previousOrder == 1) {points = points - 5;}
				break;
			case 2:
				if((previousOrder == 11) || (previousOrder == 13))
{points = points + 2;}
				if(previousOrder == 2) {points = points - 5;}
				break;
			case 3:
				if((previousOrder == 11) || (previousOrder == 13))
{points = points + 2;}
				if(previousOrder == 3) {points = points - 5;}
				break;
			case 4:
				if((previousOrder == 10) || (previousOrder == 12))
```

```
{points = points + 2;}
                                        if(previousOrder == 4) {points = points - 5;}
                                        break;
                                case 5:
                                        if((previousOrder == 12) || (previousOrder == 13))
{points = points + 2;}
                                        if(previousOrder == 5) {points = points - 5;}
                                        break;
                                case 6:
                                        if((previousOrder == 11) || (previousOrder == 13))
{points = points + 2;}
                                        if(previousOrder == 6) {points = points - 5;}
                                        break;
                                case 7:
                                        if((previousOrder == 8) || (previousOrder == 13))
{points = points + 2;}
                                        if(previousOrder == 7) {points = points - 5;}
                                        break;
                                case 8:
                                        if((previousOrder == 7) || (previousOrder == 12))
{points = points + 2;}
                                        if(previousOrder == 8) {points = points - 5;}
                                        break;
                                case 9:
                                        if((previousOrder == 10) || (previousOrder == 14))
{points = points + 2;}
                                        if(previousOrder == 9) {points = points - 5;}
                                        break;
                                case 10:
                                        if((previousOrder == 8) || (previousOrder == 9))
{points = points + 2;}
                                        if(previousOrder == 10) {points = points - 5;}
                                        break;
                                case 11:
                                        if((previousOrder == 5) || (previousOrder == 7))
{points = points + 2;}
                                        if(previousOrder == 11) {points = points - 5;}
                                        break;
                                case 12:
                                        if((previousOrder == 5) || (previousOrder == 8))
{points = points + 2;}
                                        if(previousOrder == 12) {points = points - 5;}
                                        break;
                                case 13:
                                        if((previousOrder == 5) || (previousOrder == 7))
{points = points + 2;}
                                        if(previousOrder == 13) {points = points - 5;}
                                        break;
                                case 14:
                                        if((previousOrder == 9) || (previousOrder == 12))
{points = points + 2;}
                                        if(previousOrder == 14) {points = points - 5;}
                                        break;
                                case 15:
                                        if((previousOrder == 10) || (previousOrder == 13))
{points = points + 2;}
                                        if(previousOrder == 15) {points = points - 5;}
                                        break;
                        }
                        if(i == 0) {f = 6*points;}
                        if(i == 1) {f = f + 5*points;}
                        if(i == 2) {f = f + 4*points;}
                        if(i == 3) {f = f + 3*points;}
                        if(i == 4) {f = f + 2*points;}
                        if(i == 5) {f = f + points;}
                        if(i == 0 && points <= 2) {p = 1;}
                }
                if(p == 1) {f = 0;}//if p1 = 0 -> f = 0
                return f;
}

void ReturnResult(int bestParameterValue1, int bestParameterValue2, int
bestParameterValue3, int bestParameterValue4, int bestParameterValue5, int
bestParameterValue6, int inputValue[29], int maxFitness)
{
        int f = 0, x = 0, points = 0, previousOrder;

        int iPN1 = inputValue[0];
        int iPN2 = inputValue[1];
        int iPN3 = inputValue[2];
        int iPW1 = inputValue[3];
        int iPW2 = inputValue[4];
        int iPW3 = inputValue[5];
        int iPE  = inputValue[6];
        int iPS1 = inputValue[7];
        int iPS2 = inputValue[8];
        int iPS3 = inputValue[9];
        int iTNW = inputValue[10];
        int iTNE = inputValue[11];
        int iTNS = inputValue[12];
        int iTWN = inputValue[13];
        int iTWE = inputValue[14];
        int iTEN = inputValue[15];
        int iTEW = inputValue[16];
        int iTES = inputValue[17];
        int iTSN = inputValue[18];
        int iTSE = inputValue[19];
        int iRNW = inputValue[20];
        int iRNE = inputValue[21];
        int iRNS = inputValue[22];
        int iRWN = inputValue[23];
        int iRWSE = inputValue[24];
        int iRENW = inputValue[25];
        int iRES = inputValue[26];
        int iRSW = inputValue[27];
        int iRSNE = inputValue[28];

        printf("\n   PPP PPP P PPP   TTT TT TTT TT   RRR RR RR RR\n");
        printf("   NNN WWW E SSS   NNN WW EEE SS   NNN WW EE SS\n");
        printf("   123 123   123   WES NE NWS NE   WES NS NS WN\n");
        printf("   ||| ||| | |||   ||| || ||| ||   ||| |E W| |E\n");
        printf("i  %d%d%d.%d%d%d%d.%d.%d%d%d : %d%d%d.%d%d%d.
%d%d : %d%d%d.%d%d.%d%d.%d%d%d   Max fitness: %d\n", iPN1, iPN2, iPN3,
iPW1, iPW2, iPW3, iPE, iPS1, iPS2, iPS3, iTNW, iTNE, iTNS, iTWN, iTWE,
iTEN, iTEW, iTES, iTSN, iTSE, iRNW, iRNE, iRNS, iRWN, iRWSE, iRENW,
iRES, iRSW, iRSNE, maxFitness);

        for(int i = 0; i <= 6 - 1; i++)
        {
                previousOrder = x;
                if(i == 0) {x = bestParameterValue1;}
                if(i == 1) {x = bestParameterValue2;}
                if(i == 2) {x = bestParameterValue3;}
                if(i == 3) {x = bestParameterValue4;}
                if(i == 4) {x = bestParameterValue5;}
                if(i == 5) {x = bestParameterValue6;}
                points = 0;

                //Inputs functions calculations
                switch(x)
                {
                        case 0:
                                points = iTNW + iTEN + iTSE + iPN1 + iPN2 +
iPN3 + iPS1 + iPS3 + iRNW;
                                iTNW = 0; iTEN = 0; iTSE = 0; iPN1 = 0; iPN2 = 0;
iPN3 = 0; iPS1 = 0; iPS3 = 0; iRNW = 0;
                                break;
                        case 1:
                                points = iTNW + iTEN + iTWE + iPN3 + iPW1 +
iPS1 + iPS2 + iPS3 + iRWSE;
                                iTNW = 0; iTEN = 0; iTWE = 0; iPN3 = 0; iPW1 =
0; iPS1 = 0; iPS2 = 0; iPS3 = 0; iRWSE = 0;
                                break;
                        case 2:
                                points = iTNS + iTEN + iTSE + iPW1 + iPW2 +
iPW3 + iPN3 + iPS3 + iRNW + iRNS;
                                iTNS = 0; iTEN = 0; iTSE = 0; iPW1 = 0; iPW2 = 0;
iPW3 = 0; iPN3 = 0; iPS3 = 0; iRNW = 0; iRNS = 0;
                                break;
                        case 3:
                                points = iTNW + iTSN + iPW1 + iPW3 + iPE + iPS1
+ iRNW + iRSNE;
                                iTNW = 0; iTSN = 0; iPW1 = 0; iPW3 = 0; iPE = 0;
iPS1 = 0; iRNW = 0; iRSNE = 0;
                                break;
                        case 4:
                                points = iTEW + iTSE + iPN1 + iPN2 + iPN3 +
iPW3 + iPS1 + iPS3 + iRENW;
                                iTEW = 0; iTSE = 0; iPN1 = 0; iPN2 = 0; iPN3= 0;
iPW3 = 0; iPS1 = 0; iPS3 = 0; iRENW = 0;
                                break;
                        case 5:
                                points = 2*iTNW + 2*iTWN + 2*iTES + 2*iTSE +
iPN3 + iPW1 + iPW3 + iPS1 + iPS3 + iRNW;
                                iTNW = 0; iTWN = 0; iTES = 0; iTSE = 0; iPN3 = 0;
iPW1 = 0; iPW3 = 0; iPS1 = 0; iPS3 = 0; iRNW = 0;
                                break;
                        case 6:
                                points = 2*iTNE + 4*iTEN + iPN3 + iPW1 + iPW2
+ iPW3 + iPS2 + iPS3 + iRNW + iRNS + iRNE;
                                iTNE = 0; iTEN = 0; iPN3 = 0; iPW1 = 0; iPW2 = 0;
iPW3 = 0; iPS2 = 0; iPS3 = 0; iRNW = 0; iRNS = 0; iRNE = 0;
                                break;
                        case 7:
                                points = 2*iTNS + 2*iTSN + iPW1 + iPW2 + iPW3
+ iPE + iRNW + iRNS + iRSNE;
                                iTNS= 0; iTSN = 0; iPW1 = 0; iPW2 = 0; iPW3 = 0;
iPE = 0; iRNW = 0; iRNS = 0; iRSNE = 0;
                                break;
                        case 8:
                                points = 2*iTWE + 2*iTEW + iPN1 + iPN2 + iPN3
+ iPS1 + iPS2 + iPS3 + iRENW + iRWSE;
                                iTWE = 0; iTEW = 0; iPN1 = 0; iPN2 = 0; iPN3 = 0;
iPS1 = 0; iPS2 = 0; iPS3 = 0; iRENW = 0; iRWSE = 0;
                                break;
                        case 9:
                                points = iTNW + iTWN + iPN1 + iPN3 + iPW1 +
iPS1 + iPS2 + iPS3 + 2*iRWN + iRWSE;
                                iTNW = 0; iTWN = 0; iPN1 = 0; iPN3 = 0; iPW1 =
0; iPS1 = 0; iPS2 = 0; iPS3 = 0; iRWN = 0; iRWSE = 0;
                                break;
                        case 10:
                                points = iPN2 + iPW1 + iPW2 + iPW3 + iPS2 + iPE
+ iRNW + iRNS + 2*iRSNE;
                                iPN2 = 0; iPW1 = 0; iPW2 = 0; iPW3 = 0; iPS2 = 0;
iPE = 0; iRNW = 0; iRNS = 0; iRSNE = 0;
                                break;
                        case 11:
                                points = iPN1 +  iPN2 +  iPN3 + iPW2 + iPS1 +
iPS2 + iPS3 + 2*iRENW + 2*iRWSE;
                                iPN1 = 0; iPN2 = 0; iPN3 = 0; iPW2 = 0; iPS1 = 0;
iPS2 = 0; iPS3 = 0; iRENW = 0; iRWSE = 0;
```

```
                        break;
                case 12:
                        points = iPN2 +  iPN3 +  iPW1 + iPW2 + iPW3 +
iPE + iPS1 + iPS2 + 2*iRNE + 2*iRSW;
                        iPN2 = 0; iPN3 = 0; iPW1 = 0; iPW2 = 0; iPW3 = 0;
iPE = 0; iPS1 = 0; iPS2 = 0; iRNE = 0; iRSW = 0;
                        break;
                case 13:
                        points = iPN1 +  iPN2 +  iPN3 + iPW1 + iPW2 +
iPS1 + iPS2 + iPS3 + 2*iRWN + 2*iRES;
                        iPN1 = 0; iPN2 = 0; iPN3 = 0; iPW1 = 0; iPW2 = 0;
iPS1 = 0; iPS2 = 0; iPS3 = 0; iRWN = 0; iRES = 0;
                        break;
                case 14:
                        points = iTSE + iTES + iPN1 + iPN2 + iPN3 + iPW1
+ iPW2 + iPW3 + iPS1 + iPS3 + iRES;
                        iTSE = 0; iTES = 0; iPN1 = 0; iPN2 = 0; iPN3 = 0;
iPW1 = 0; iPW2 = 0; iPW3 = 0; iPS1 = 0; iPS3 = 0; iRES = 0;
                        break;
                case 15:
                        points = iTNE + iTEN + iPN1 + iPN3 + iPW1 +
iPW2 + iPS1 + iPS2 + iPS3 + 2*iRWSE;
                        iTNE = 0; iTEN = 0; iPN1 = 0; iPN3 = 0; iPW1 = 0;
iPW2 = 0; iPS1 = 0; iPS2 = 0; iPS3 = 0; iRWSE = 0;
                        break;
                case 16:
                        points = 0;
                        break;
        }
        //Previous order calculations
        switch(x)
        {
                case 0:
                        if((previousOrder == 11) || (previousOrder == 13))
{points = points + 2;}
                        break;
                case 1:
                        if((previousOrder == 11) || (previousOrder == 13))
{points = points + 2;}
                        break;
                case 2:
                        if((previousOrder == 11) || (previousOrder == 13))
{points = points + 2;}
                        break;
                case 3:
                        if((previousOrder == 11) || (previousOrder == 13))
{points = points + 2;}
                        break;
                case 4:
                        if((previousOrder == 10) || (previousOrder == 12))
{points = points + 2;}
                        break;
                case 5:
                        if((previousOrder == 12) || (previousOrder == 13))
{points = points + 2;}
                        break;
                case 6:
                        if((previousOrder == 11) || (previousOrder == 13))
{points = points + 2;}
                        break;
                case 7:
                        if((previousOrder == 8) || (previousOrder == 13))
{points = points + 2;}
                        break;
                case 8:
                        if((previousOrder == 7) || (previousOrder == 12))
{points = points + 2;}
                        break;
                case 9:
                        if((previousOrder == 10) || (previousOrder == 14))
{points = points + 2;}
                        break;
                case 10:
                        if((previousOrder == 8) || (previousOrder == 9))
{points = points + 2;}
                        break;
                case 11:
                        if((previousOrder == 5) || (previousOrder == 7))
{points = points + 2;}
                        break;
                case 12:
                        if((previousOrder == 5) || (previousOrder == 8))
{points = points + 2;}
                        break;
                case 13:
                        if((previousOrder == 5) || (previousOrder == 7))
{points = points + 2;}
                        break;
                case 14:
                        if((previousOrder == 9) || (previousOrder == 12))
{points = points + 2;}
                        break;
                case 15:
                        if((previousOrder == 10) || (previousOrder == 13))
{points = points + 2;}
                        break;
                case 16:
                        //
                        break;
        }
        if(i == 0) {f = 6*points;}
        if(i == 1) {f = f + 5*points;}
        if(i == 2) {f = f + 4*points;}
        if(i == 3) {f = f + 3*points;}
        if(i == 4) {f = f + 2*points;}
        if(i == 5) {f = f + points;}
        printf("c%d %d%d%d.%d%d%d.%d.%d%d%d : %d%d%d.%d%d.
%d%d%d.%d%d : %d%d%d.%d%d.%d%d.%d%d   Order = %2d   Points =
%2d\n", i+1, iPN1, iPN2, iPN3, iPW1, iPW2, iPW3, iPE, iPS1, iPS2, iPS3, iTNW,
iTNE, iTNS, iTWN, iTWE, iTEN, iTEW, iTES, iTSN, iTSE, iRNW, iRNE, iRNS,
iRWN, iRWSE, iRENW, iRES, iRSW, iRSNE, x ,points);
    }
    printf("\n");
    //Output print out
    switch(bestParameterValue1)
    {
        case 0:
            printf("Executing requests on drive order %d:\noTNW,
oTEN, oTSE, oPN1, oPN2, oPN3, oPS1, oPS3, oRNW\n", bestParameterValue1);
            break;
        case 1:
            printf("Executing requests on drive order %d:\noTNW,
oTEN, oTWE, oPN3, oPW1, oPS1, oPS2, oPS3, oRWSE\n", bestParameterValue1);
            break;
        case 2:
            printf("Executing requests on drive order %d:\noTNS,
oTEN, oTSE, oPW1, oPW2, oPW3, oPN3, oPS3, oRNW, oRNS\n",
bestParameterValue1);
            break;
        case 3:
            printf("Executing requests on drive order %d:\noTNW,
oTSN, oPW1, oPW3, oPE, oPS1, oRNW, oRSNE\n", bestParameterValue1);
            break;
        case 4:
            printf("Executing requests on drive order %d:\noTEW,
oTSE, oPN1, oPN2, oPN3, oPW3, oPS1, oPS3, oRENW\n", bestParameterValue1);
            break;
        case 5:
            printf("Executing requests on drive order %d:\noTNW,
oTWN, oTES, oTSE, oPN3, oPW1, oPW3, oPS1, oPS3, oRNW\n",
bestParameterValue1);
            break;
        case 6:
            printf("Executing requests on drive order %d:\noTNE,
oTEN, oPN3, oPW1, oPW2, oPW3, oPS2, oPS3, oRNW, oRNS, oRNE\n",
bestParameterValue1);
            break;
        case 7:
            printf("Executing requests on drive order %d:\noTNS,
oTSN, oPW1, oPW2, oPW3, oPE, oRNW, oRNS, oRSNE\n",
bestParameterValue1);
            break;
        case 8:
            printf("Executing requests on drive order %d:\noTWE,
oTEW, oPN1, oPN2, oPN3, oPS1, oPS2, oPS3, oRENW, oRWSE\n",
bestParameterValue1);
            break;
        case 9:
            printf("Executing requests on drive order %d:\noTNW,
oTWN, oPN1, oPN3, oPW1, oPS1, oPS2, oPS3, oRWN, oRWSE\n",
bestParameterValue1);
            break;
        case 10:
            printf("Executing requests on drive order %d:\noPN2,
oPW1, oPW2, oPW3, oPS2, oPE, oRNW, oRNS, oRSNE\n", bestParameterValue1);
            break;
        case 11:
            printf("Executing requests on drive order %d:\noPN1, oPN2,
oPN3, oPW2, oPS1, oPS2, oPS3, oRENW, oRWSE\n", bestParameterValue1);
            break;
        case 12:
            printf("Executing requests on drive order %d:\noPN2, oPN3,
oPW1, oPW2, oPW3, oPE, oPS1, oPS2, oRNE, oRSW\n", bestParameterValue1);
            break;
        case 13:
            printf("Executing requests on drive order %d:\noPN1, oPN2,
oPN3, oPW1, oPW2, oPS1, oPS2, oPS3, oRWN, oRES\n", bestParameterValue1);
            break;
        case 14:
            printf("Executing requests on drive order %d:\noTSE, oTES,
oPN1, oPN2, oPN3, oPW1, oPW2, oPW3, oPS1, oPS3, oRES\n",
bestParameterValue1);
            break;
        case 15:
            printf("Executing requests on drive order %d:\noTNE,
oTEN, oPN1, oPN3, oPW1, oPW2, oPS1, oPS2, oPS3, oRWSE\n",
bestParameterValue1);
            break;
        case 16:
            printf("Do not executing any drive order.\n");
            break;
    }
    printf("\n");
}

int main()
{
    int inputValue[29];//All inputs to work with
    int inputValues[10][29];//List of all time periods inputs [=sim][29]
    int maxFitness = 0;
    int fitness;//Store fitness value for each individual in present generation
    int parameterValue1 = 16;//Temporary parameter value (16 is for null result)
```

```
	int parameterValue2 = 16;//Temporary parameter value
	int parameterValue3 = 16;//Temporary parameter value
	int parameterValue4 = 16;//Temporary parameter value
	int parameterValue5 = 16;//Temporary parameter value
	int parameterValue6 = 16;//Temporary parameter value
	int bestParameterValue1 = 16;//Best chromosome value
	int bestParameterValue2 = 16;//Best chromosome value
	int bestParameterValue3 = 16;//Best chromosome value
	int bestParameterValue4 = 16;//Best chromosome value
	int bestParameterValue5 = 16;//Best chromosome value
	int bestParameterValue6 = 16;//Best chromosome value
	int previousExcecution = 16;//Store c1 (E) from previous time period
	int sim = 10;//Number of simulations

	int i, j, k, l, m, n, x = 0; int msgOrder = 0, mEInt = 0; float mEfloat = 0;//
Other variables

	for(j = 0; j <= 28; j++)
	{
		inputValue[j] = 0;
	}
	for(i = 0; i <= sim - 1; i++)
	{
		for(j = 0; j <= 28; j++)
		{
			inputValues[i][j] = 0;
		}
	}
	//Only 5 first is filled with inputs

	//Init inputValue 0
	inputValues[0][0] = 1; /* PN1 */		inputValues[0][10] = 1; /*
TNW */		inputValues[0][20] = 0; /* RNW */
	inputValues[0][1] = 0; /* PN2 */		inputValues[0][11] = 0; /*
TNE */		inputValues[0][21] = 1; /* RNE */
	inputValues[0][2] = 1; /* PN3 */		inputValues[0][12] = 0; /*
TNS */		inputValues[0][22] = 1; /* RNS */
	inputValues[0][3] = 0; /* PW1 */		inputValues[0][13] = 1; /*
TWN */		inputValues[0][23] = 0; /* RWN */
	inputValues[0][4] = 0; /* PW2 */		inputValues[0][14] = 0; /*
TWE */		inputValues[0][24] = 0; /* RWSE */
	inputValues[0][5] = 1; /* PW3 */		inputValues[0][15] = 0; /*
TEN */		inputValues[0][25] = 0; /* RENW */
	inputValues[0][6] = 0; /* PE  */		inputValues[0][16] = 0; /*
TEW */		inputValues[0][26] = 0; /* RES */
	inputValues[0][7] = 0; /* PS1 */		inputValues[0][17] = 0; /*
TES */		inputValues[0][27] = 1; /* RSW */
	inputValues[0][8] = 0; /* PS2 */		inputValues[0][18] = 1; /*
TSN */		inputValues[0][28] = 1; /* RSNE */
	inputValues[0][9] = 1; /* PS3 */		inputValues[0][19] = 0; /*
TSE */
	//Init inputValue 1
	inputValues[1][0] = 0; /* PN1 */		inputValues[1][10] = 0; /*
TNW */		inputValues[1][20] = 1; /* RNW */
	inputValues[1][1] = 1; /* PN2 */		inputValues[1][11] = 0; /*
TNE */		inputValues[1][21] = 1; /* RNE */
	inputValues[1][2] = 0; /* PN3 */		inputValues[1][12] = 0; /*
TNS */		inputValues[1][22] = 1; /* RNS */
	inputValues[1][3] = 1; /* PW1 */		inputValues[1][13] = 0; /*
TWN */		inputValues[1][23] = 1; /* RWN */
	inputValues[1][4] = 1; /* PW2 */		inputValues[1][14] = 0; /*
TWE */		inputValues[1][24] = 0; /* RWSE */
	inputValues[1][5] = 0; /* PW3 */		inputValues[1][15] = 0; /*
TEN */		inputValues[1][25] = 0; /* RENW */
	inputValues[1][6] = 1; /* PE  */		inputValues[1][16] = 0; /*
TEW */		inputValues[1][26] = 0; /* RES */
	inputValues[1][7] = 1; /* PS1 */		inputValues[1][17] = 0; /*
TES */		inputValues[1][27] = 1; /* RSW */
	inputValues[1][8] = 1; /* PS2 */		inputValues[1][18] = 0; /*
TSN */		inputValues[1][28] = 1; /* RSNE */
	inputValues[1][9] = 0; /* PS3 */		inputValues[1][19] = 0; /*
TSE */
	//Init inputValue 2
	inputValues[2][0] = 0; /* PN1 */		inputValues[2][10] = 0; /*
TNW */		inputValues[2][20] = 1; /* RNW */
	inputValues[2][1] = 0; /* PN2 */		inputValues[2][11] = 0; /*
TNE */		inputValues[2][21] = 0; /* RNE */
	inputValues[2][2] = 1; /* PN3 */		inputValues[2][12] = 1; /*
TNS */		inputValues[2][22] = 1; /* RNS */
	inputValues[2][3] = 1; /* PW1 */		inputValues[2][13] = 0; /*
TWN */		inputValues[2][23] = 1; /* RWN */
	inputValues[2][4] = 1; /* PW2 */		inputValues[2][14] = 1; /*
TWE */		inputValues[2][24] = 0; /* RWSE */
	inputValues[2][5] = 0; /* PW3 */		inputValues[2][15] = 0; /*
TEN */		inputValues[2][25] = 0; /* RENW */
	inputValues[2][6] = 0; /* PE  */		inputValues[2][16] = 0; /*
TEW */		inputValues[2][26] = 0; /* RES */
	inputValues[2][7] = 1; /* PS1 */		inputValues[2][17] = 0; /*
TES */		inputValues[2][27] = 0; /* RSW */
	inputValues[2][8] = 1; /* PS2 */		inputValues[2][18] = 0; /*
TSN */		inputValues[2][28] = 1; /* RSNE */
	inputValues[2][9] = 0; /* PS3 */		inputValues[2][19] = 0; /*
TSE */
	//Init inputValue 3
	inputValues[3][0] = 1; /* PN1 */		inputValues[3][10] = 0; /*
TNW */		inputValues[3][20] = 0; /* RNW */
	inputValues[3][1] = 0; /* PN2 */		inputValues[3][11] = 0; /*
TNE */		inputValues[3][21] = 1; /* RNE */
	inputValues[3][2] = 0; /* PN3 */		inputValues[3][12] = 0; /*
TNS */		inputValues[3][22] = 0; /* RNS */
	inputValues[3][3] = 0; /* PW1 */		inputValues[3][13] = 0; /*
TWN */		inputValues[3][23] = 1; /* RWN */
	inputValues[3][4] = 0; /* PW2 */		inputValues[3][14] = 0; /*
TWE */		inputValues[3][24] = 1; /* RWSE */
	inputValues[3][5] = 0; /* PW3 */		inputValues[3][15] = 0; /*
TEN */		inputValues[3][25] = 1; /* RENW */
	inputValues[3][6] = 0; /* PE  */		inputValues[3][16] = 0; /*
TEW */		inputValues[3][26] = 0; /* RES */
	inputValues[3][7] = 1; /* PS1 */		inputValues[3][17] = 0; /*
TES */		inputValues[3][27] = 0; /* RSW */
	inputValues[3][8] = 1; /* PS2 */		inputValues[3][18] = 0; /*
TSN */		inputValues[3][28] = 0; /* RSNE */
	inputValues[3][9] = 1; /* PS3 */		inputValues[3][19] = 0; /*
TSE */
	//Init inputValue 4
	inputValues[4][0] = 0; /* PN1 */		inputValues[4][10] = 0; /*
TNW */		inputValues[4][20] = 0; /* RNW */
	inputValues[4][1] = 0; /* PN2 */		inputValues[4][11] = 0; /*
TNE */		inputValues[4][21] = 1; /* RNE */
	inputValues[4][2] = 0; /* PN3 */		inputValues[4][12] = 0; /*
TNS */		inputValues[4][22] = 1; /* RNS */
	inputValues[4][3] = 1; /* PW1 */		inputValues[4][13] = 0; /*
TWN */		inputValues[4][23] = 1; /* RWN */
	inputValues[4][4] = 0; /* PW2 */		inputValues[4][14] = 0; /*
TWE */		inputValues[4][24] = 0; /* RWSE */
	inputValues[4][5] = 0; /* PW3 */		inputValues[4][15] = 0; /*
TEN */		inputValues[4][25] = 0; /* RENW */
	inputValues[4][6] = 1; /* PE  */		inputValues[4][16] = 0; /*
TEW */		inputValues[4][26] = 0; /* RES */
	inputValues[4][7] = 0; /* PS1 */		inputValues[4][17] = 1; /*
TES */		inputValues[4][27] = 1; /* RSW */
	inputValues[4][8] = 0; /* PS2 */		inputValues[4][18] = 0; /*
TSN */		inputValues[4][28] = 0; /* RSNE */
	inputValues[4][9] = 0; /* PS3 */		inputValues[4][19] = 1; /*
TSE */
	//Init inputValue 5
	inputValues[5][0] = 0; /* PN1 */		inputValues[5][10] = 0; /*
TNW */		inputValues[5][20] = 0; /* RNW */
	inputValues[5][1] = 0; /* PN2 */		inputValues[5][11] = 1; /*
TNE */		inputValues[5][21] = 0; /* RNE */
	inputValues[5][2] = 0; /* PN3 */		inputValues[5][12] = 0; /*
TNS */		inputValues[5][22] = 0; /* RNS */
	inputValues[5][3] = 0; /* PW1 */		inputValues[5][13] = 0; /*
TWN */		inputValues[5][23] = 0; /* RWN */
	inputValues[5][4] = 0; /* PW2 */		inputValues[5][14] = 0; /*
TWE */		inputValues[5][24] = 0; /* RWSE */
	inputValues[5][5] = 1; /* PW3 */		inputValues[5][15] = 0; /*
TEN */		inputValues[5][25] = 0; /* RENW */
	inputValues[5][6] = 0; /* PE  */		inputValues[5][16] = 0; /*
TEW */		inputValues[5][26] = 1; /* RES */
	inputValues[5][7] = 0; /* PS1 */		inputValues[5][17] = 0; /*
TES */		inputValues[5][27] = 0; /* RSW */
	inputValues[5][8] = 0; /* PS2 */		inputValues[5][18] = 0; /*
TSN */		inputValues[5][28] = 1; /* RSNE */
	inputValues[5][9] = 1; /* PS3 */		inputValues[5][19] = 0; /*
TSE */
	//Init inputValue 6
	inputValues[6][0] = 0; /* PN1 */		inputValues[6][10] = 0; /*
TNW */		inputValues[6][20] = 1; /* RNW */
	inputValues[6][1] = 0; /* PN2 */		inputValues[6][11] = 0; /*
TNE */		inputValues[6][21] = 0; /* RNE */
	inputValues[6][2] = 0; /* PN3 */		inputValues[6][12] = 0; /*
TNS */		inputValues[6][22] = 1; /* RNS */
	inputValues[6][3] = 1; /* PW1 */		inputValues[6][13] = 0; /*
TWN */		inputValues[6][23] = 0; /* RWN */
	inputValues[6][4] = 0; /* PW2 */		inputValues[6][14] = 0; /*
TWE */		inputValues[6][24] = 0; /* RWSE */
	inputValues[6][5] = 0; /* PW3 */		inputValues[6][15] = 0; /*
TEN */		inputValues[6][25] = 1; /* RENW */
	inputValues[6][6] = 0; /* PE  */		inputValues[6][16] = 0; /*
TEW */		inputValues[6][26] = 0; /* RES */
	inputValues[6][7] = 1; /* PS1 */		inputValues[6][17] = 0; /*
TES */		inputValues[6][27] = 0; /* RSW */
	inputValues[6][8] = 0; /* PS2 */		inputValues[6][18] = 0; /*
TSN */		inputValues[6][28] = 1; /* RSNE */
	inputValues[6][9] = 0; /* PS3 */		inputValues[6][19] = 0; /*
TSE */
	//Init inputValue 7
	inputValues[7][0] = 1; /* PN1 */		inputValues[7][10] = 0; /*
TNW */		inputValues[7][20] = 0; /* RNW */
	inputValues[7][1] = 0; /* PN2 */		inputValues[7][11] = 0; /*
TNE */		inputValues[7][21] = 1; /* RNE */
	inputValues[7][2] = 1; /* PN3 */		inputValues[7][12] = 0; /*
TNS */		inputValues[7][22] = 0; /* RNS */
	inputValues[7][3] = 0; /* PW1 */		inputValues[7][13] = 0; /*
TWN */		inputValues[7][23] = 0; /* RWN */
	inputValues[7][4] = 0; /* PW2 */		inputValues[7][14] = 0; /*
TWE */		inputValues[7][24] = 1; /* RWSE */
	inputValues[7][5] = 1; /* PW3 */		inputValues[7][15] = 1; /*
TEN */		inputValues[7][25] = 0; /* RENW */
	inputValues[7][6] = 1; /* PE  */		inputValues[7][16] = 0; /*
TEW */		inputValues[7][26] = 0; /* RES */
	inputValues[7][7] = 0; /* PS1 */		inputValues[7][17] = 0; /*
TES */		inputValues[7][27] = 0; /* RSW */
	inputValues[7][8] = 0; /* PS2 */		inputValues[7][18] = 0; /*
TSN */		inputValues[7][28] = 0; /* RSNE */
	inputValues[7][9] = 0; /* PS3 */		inputValues[7][19] = 1; /*
TSE */

	//Start main program
	printf("Start\n");
	for(int h = 0; h <= sim - 1;  h++)
```

```
{
	if(1)
	{
		int iPN1 = inputValue[0];
		int iPN2 = inputValue[1];
		int iPN3 = inputValue[2];
		int iPW1 = inputValue[3];
		int iPW2 = inputValue[4];
		int iPW3 = inputValue[5];
		int iPE  = inputValue[6];
		int iPS1 = inputValue[7];
		int iPS2 = inputValue[8];
		int iPS3 = inputValue[9];
		int iTNW = inputValue[10];
		int iTNE = inputValue[11];
		int iTNS = inputValue[12];
		int iTWN = inputValue[13];
		int iTWE = inputValue[14];
		int iTEN = inputValue[15];
		int iTEW = inputValue[16];
		int iTES = inputValue[17];
		int iTSN = inputValue[18];
		int iTSE = inputValue[19];
		int iRNW = inputValue[20];
		int iRNE = inputValue[21];
		int iRNS = inputValue[22];
		int iRWN = inputValue[23];
		int iRWSE = inputValue[24];
		int iRENW = inputValue[25];
		int iRES = inputValue[26];
		int iRSW = inputValue[27];
		int iRSNE = inputValue[28];
		switch(bestParameterValue1)
		{
			case 0:
				iTNW = 0; iTEN = 0; iTSE = 0; iPN1 = 0; iPN2 = 0; iPN3 = 0; iPS1 = 0; iPS3 = 0; iRNW = 0;
				break;
			case 1:
				iTNW = 0; iTEN = 0; iTWE = 0; iPN3 = 0; iPW1 = 0; iPS1 = 0; iPS2 = 0; iPS3 = 0; iRWSE = 0;
				break;
			case 2:
				iTNS = 0; iTEN = 0; iTSE = 0; iPW1 = 0; iPW2 = 0; iPW3 = 0; iPN3 = 0; iPS3 = 0; iRNW = 0; iRNS = 0;
				break;
			case 3:
				iTNW = 0; iTSN = 0; iPW1 = 0; iPW3 = 0; iPE = 0; iPS1 = 0; iRNW = 0; iRSNE = 0;
				break;
			case 4:
				iTEW = 0; iTSE = 0; iPN1 = 0; iPN2 = 0; iPN3= 0; iPW3 = 0; iPS1 = 0; iPS3 = 0; iRENW = 0;
				break;
			case 5:
				iTNW = 0; iTWN = 0; iTES = 0; iTSE = 0; iPN3 = 0; iPW1 = 0; iPW3 = 0; iPS1 = 0; iPS3 = 0; iRNW = 0;
				break;
			case 6:
				iTNE = 0; iTEN = 0; iPN3 = 0; iPW1 = 0; iPW2 = 0; iPW3 = 0; iPS2 = 0; iPS3 = 0; iRNW = 0; iRNS = 0; iRNE = 0;
				break;
			case 7:
				iTNS= 0; iTSN = 0; iPW1 = 0; iPW2 = 0; iPW3 = 0; iPE = 0; iRNW = 0; iRNS = 0; iRSNE = 0;
				break;
			case 8:
				iTWE = 0; iTEW = 0; iPN1 = 0; iPN2 = 0; iPN3 = 0; iPS1 = 0; iPS2 = 0; iPS3 = 0; iRENW = 0; iRWSE = 0;
				break;
			case 9:
				iTNW = 0; iTWN = 0; iPN1 = 0; iPN3 = 0; iPW1 = 0; iPS1 = 0; iPS2 = 0; iPS3 = 0; iRWN = 0; iRWSE = 0;
				break;
			case 10:
				iPN2 = 0; iPW1 = 0; iPW2 = 0; iPW3 = 0; iPS2 = 0; iPE = 0; iRNW = 0; iRNS = 0; iRSNE = 0;
				break;
			case 11:
				iPN1 = 0; iPN2 = 0; iPN3 = 0; iPW2 = 0; iPS1 = 0; iPS2 = 0; iPS3 = 0; iRENW = 0; iRWSE = 0;
				break;
			case 12:
				iPN2 = 0; iPN3 = 0; iPW1 = 0; iPW2 = 0; iPW3 = 0; iPE = 0; iPS1 = 0; iPS2 = 0; iRNE = 0; iRSW = 0;
				break;
			case 13:
				iPN1 = 0; iPN2 = 0; iPN3 = 0; iPW1 = 0; iPW2 = 0; iPS1 = 0; iPS2 = 0; iPS3 = 0; iRWN = 0; iRES = 0;
				break;
			case 14:
				iTSE = 0; iTES = 0; iPN1 = 0; iPN2 = 0; iPN3 = 0; iPW1 = 0; iPW2 = 0; iPW3 = 0; iPS1 = 0; iPS3 = 0; iRES = 0;
				break;
			case 15:
				iTNE = 0; iTEN = 0; iPN1 = 0; iPN3 = 0; iPW1 = 0; iPW2 = 0; iPS1 = 0; iPS2 = 0; iPS3 = 0; iRWSE = 0;
				break;
			case 16:
				break;
		}
		// Rewrite inputValue
		inputValue[0] = iPN1;
		inputValue[1] = iPN2;
		inputValue[2] = iPN3;
		inputValue[3] = iPW1;
		inputValue[4] = iPW2;
		inputValue[5] = iPW3;
		inputValue[6] = iPE;
		inputValue[7] = iPS1;
		inputValue[8] = iPS2;
		inputValue[9] = iPS3;
		inputValue[10] = iTNW;
		inputValue[11] = iTNE;
		inputValue[12] = iTNS;
		inputValue[13] = iTWN;
		inputValue[14] = iTWE;
		inputValue[15] = iTEN;
		inputValue[16] = iTEW;
		inputValue[17] = iTES;
		inputValue[18] = iTSN;
		inputValue[19] = iTSE;
		inputValue[20] = iRNW;
		inputValue[21] = iRNE;
		inputValue[22] = iRNS;
		inputValue[23] = iRWN;
		inputValue[24] = iRWSE;
		inputValue[25] = iRENW;
		inputValue[26] = iRES;
		inputValue[27] = iRSW;
		inputValue[28] = iRSNE;
		for(j = 0; j <= 28; j++)
		{
			if(inputValue[j] != 0) {inputValue[j] = inputValue[j] + 1;}// Add priority to remaining inputs
			else {inputValue[j] = inputValue[j] + inputValues[h][j];}// Add new inputs if there are any
		}
	}

	//Start loop
	maxFitness = 0;
	for(i = 0; i <= 16 - 1; i++)
	{
		for(j = 0; j <= 16 - 1; j++)
		{
			for(k = 0; k <= 16 - 1; k++)
			{
				for(l = 0; l <= 16 - 1; l++)
				{
					for(m = 0; m <= 16 - 1; m++)
					{
						for(n = 0; n <= 16 - 1; n++)
						{
							parameterValue1 = i, parameterValue2 = j, parameterValue3 = k, parameterValue4 = l, parameterValue5 = m, parameterValue6 = n;
							fitness = EvaluateIndividual(parameterValue1, parameterValue2, parameterValue3, parameterValue4, parameterValue5, parameterValue6, inputValue, previousExcecution);
							if ((fitness >= maxFitness) && (fitness != maxFitness))//fitness > maxFitness
							{
								maxFitness = fitness;

	bestParameterValue1 = parameterValue1;

	bestParameterValue2 = parameterValue2;

	bestParameterValue3 = parameterValue3;

	bestParameterValue4 = parameterValue4;

	bestParameterValue5 = parameterValue5;

	bestParameterValue6 = parameterValue6;
							}
						}
					}
				}
			}
		}
	}
	printf("**** DFS version ****\n");
	printf("Previous executed drive order: %d, Simulation: T%d\n", previousExcecution, h);
	ReturnResult(bestParameterValue1, bestParameterValue2, bestParameterValue3, bestParameterValue4, bestParameterValue5, bestParameterValue6, inputValue, maxFitness);
	printf("..........\n\n");
	previousExcecution = bestParameterValue1;

	}
	printf("\n");
	//getchar(); To stop command window.
}
```