Part II

Search Algorithms for Architectural design

5

Search problems and algorithms

Computer scientists over the years have developed an immense body of work in the field of search, creating a large number of algorithms and studying their efficiency. Selecting the right algorithm is a task that is very much related to the search problem it needs to solve. Computer scientists have also developed a big number of search problems and studied their complexity in terms of their how efficiently algorithms can solve them. A good example is the Traveling Salesman Problem (TSP). This problem involves a salesman that needs to visit n number of cities in his sales itinerary, and this cities are not all equidistant, travel between cities takes different amounts of time and have different cost (see figure 5.1). The problem requires the algorithm to search for sales routes that minimize the travel time and/or costs while taking him to through all of the cities in his itinerary and bring him back home.

This problem is not very related to the architectural search problems that are described in this thesis. If we are to select an appropriate algorithm for our problems, a good understanding of search problems in general and of the search problems we face is required.

A good definition of a search problem from a computational point of view is provided in(Schooler et al. 2012). Schooler et al. simply state that a search problem is represented by three elements (S, f, W) where S represents the search space of the problem, $f : S \to R$ is a function that assigns objective values to the solutions contained in S, and W is a set of constraints. This is a very general definition of a search problem, that applies to many situations. It certainly applies to the search problems discussed in this PhD thesis:

- Search spaces S are defined by the user by setting up a parametric

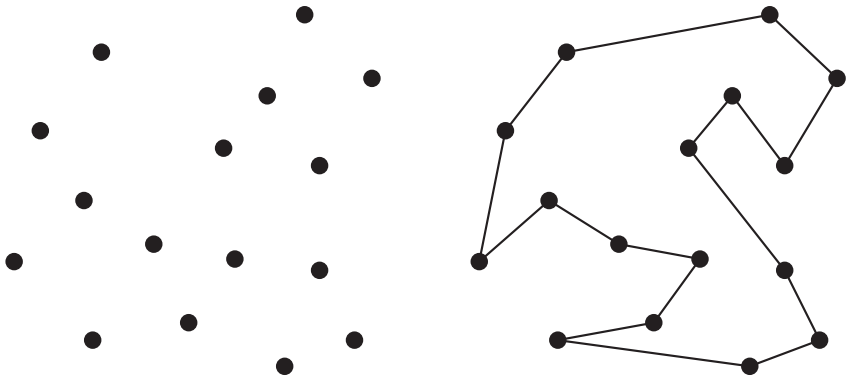


Figure 5.1: Diagram of he traveling salesman problem and one solution - Image from (Edelkamp & Schrödl 2012).

models. The number of parameters in the model define the dimensionality of S. The extension of S is defined by the parameter domains.

- Objective values are assigned by performance evaluation via computational simulation. Functions f are therefore defined by the performances we chose to use during the search process.

- Constraints are rare in our search problems, most of the limits to the search problem are set with parameter domains and invariants. Including in infeasible solutions (or black areas in the search space) is done by means of constrains W. This is not present in this thesis, but it is not excluded as a possibility in search methods.

Clearly this definition applies to our problems, and it gives us a starting point, but a more specific definition of search problems that we face is required.

Search algorithms in this PhD research are used to explore parametric models for high-performing solutions to multiple and contrasting performance functions. Parametric models may contain a large number of solutions, all identified by combinations of parameters.

A more detailed characterization of the problems in this thesis can be done by studying the search spaces involved. The search space S in architectural search problems is defined in the creation of the parametric model, it is confined by parameter domains. Perhaps the most important characteristic of our search space is that it is continuous. The number of possible values that a parameter can take is infinite, as is the number of subdivisions inside a given domain. If for example we have one parameter that defines sphere radii, and it has a domain between 1 and 10 meters, there are infinite spheres in this model.

Continuous search spaces are problematic for search problems, the number of possible solutions to search is infinite, as opposed to other problems like the traveling salesman. The number of possible routes in the TSP is directly related to the number of cities in his itinerary. The more cities there are, the more possible routes the salesman can take. But however high the number of cities the number of routes is always finite. Only if infinite cities are present does this problem have infinite possible solutions.

All search algorithms inevitably need to transform continuous search spaces into discrete ones, in other words it is impossible for algorithms to consider infinite solutions, only a finite number is studied. But some algorithms are better equipped to select deal with continuous search spaces and infinite possibilities. Even if we discretize the search space, thus dramatically reducing the number of solutions, if a high number of parameters are present, the number of possible solutions are can still be very high. This makes exhaustive search processes only feasible in very simple problems with a very coarse discretization of the search space.

Another important way of characterizing search problems is by means of the so called objective space*. The objective space is the counterpart to the search space, it represents not the solutions or their parameters, but instead it shows their objective values, the result of the f function. If we take for example a problem involving a single f function and a set of solutions contained in S, we can represent the objective space for such a problem with a curve. Figure 5.2 shows two objective spaces for single-objective minimization problems. We can see that solutions in S contain varying objective values for the same f function.

The problem presented on the left in figure 5.2 shows what is called an unimodal problem, meaning that the problem has a single minimum (optimal) value. The problem shown on the left is a multimodal problem, it contains multiple minima, meaning that the curve has various valleys. Some

*A detailed definition of objective space with examples is given in section 7.4.

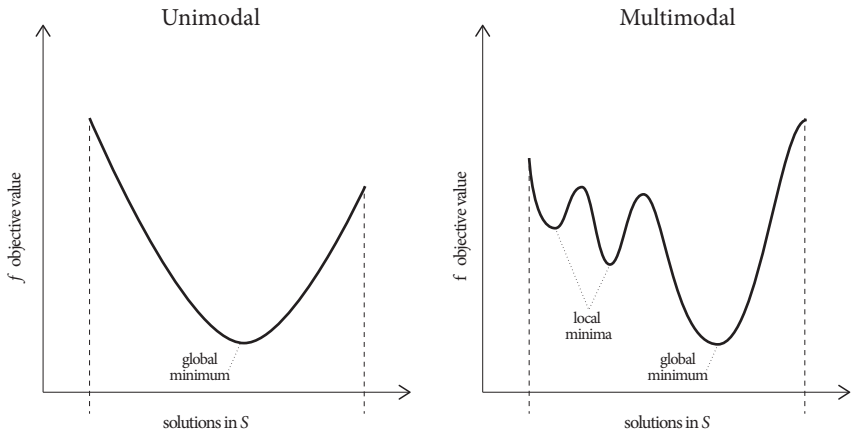


Figure 5.2: Objective Spaces showing Unimodal and Multimodal problems.

of the minima are only minimum when compared to the solutions adjacent to them. these are called local minima. The minimum value for the entire solution set is called global minimum. The distinction between unimodal and multimodal problems also apply to problems with a larger number of objective functions.

Multimodality represents a mayor challenge for search algorithms as they tend to confuse local minima with global minima, meaning that they do not converge into the solution the user has been looking for. Objective spaces might also be discontinuous or highly irregular.

The characteristics of the objective space are determined by the objective functions contained in the problem. In architectural search problems the objective spaces are impossible to know beforehand, we are blind to their shape and complexity when we perform a search process. More importantly, they are very different depending on the fact that we are studying structural shapes, acoustic quality in rooms or energy efficiency. All of this issues point to the fact that the search algorithms that we select for this PhD thesis needs to be able to deal with complex and unknown objective spaces, most likely multimodal. The algorithms need to be *robust*.

Another important characteristic of the search algorithms to be used in this PhD research can be signaled out by establishing what we are searching for, what distinguishes sought after solutions from the rest. As it has been

explained above, we are looking for high-performing solutions given our objective functions, calculated via performance evaluation software.

The identification of what the algorithm is looking for allows us to exclude a great number of search algorithms and focus on one category. The category of search algorithms that is best suited to find high-performing solutions out of an entire set is that of optimization algorithms.

In the previous chapters, a very important distinction was made between search and optimization. This distinction is not intrinsic to the inner workings of the algorithms, it regards the purpose and the moment in the design process it is carried out. This distinction does not refer to the mathematical definition of optimization, or different algorithm types. All search algorithms tend to minimize some objective function, but there is a group of algorithms that is more deliberate in this purpose, and that works in continuous search spaces. They are most commonly known as optimization algorithms. Optimization algorithms are search algorithms that minimize objective functions.

5.1 Algorithm classification

As it was declared above, we will focus our attention on optimization algorithms. As such, a classification of optimization algorithms is presented bellow. The most important way of classifying optimization algorithms is to divide them according to their optimization strategy. In this case we get deterministic and stochastic algorithms.

Deterministic algorithms are also referred to as classical because they represent the first efforts in optimization algorithms. Many deterministic algorithms also called gradient-based, because they look for optimal solutions by climbing or descending (for maximization or minimization problems respectively) in the direction of the highest gradient, the steepest hill. This means that they depend on a certain knowledge of the search problem. Algorithms in this category include Newton's method, pattern search and gradient descent. They can be defined by the following characteristics:

- Deterministic algorithms follow a strict formulation and produce the same result every time they are used on the same problem and from the same starting point.
- Deterministic algorithms consider one solution at a time, and they modify their position following the information obtained by studying only this solution.

- Deterministic algorithms' success is very much dependent on their starting point. In multimodal problems, they will arrive at the global minimum only if their starting point is in the basin of the global minimum and not of any local minima.

- Deterministic algorithms usually converge at a minimum (global or otherwise) at a very fast pace, especially when compared to stochastic algorithms.

- Exploration is not a big part of deterministic algorithms, they tend to converge into a minimum rather quickly without exploring other possibilities. This is why they have problems with multimodal problems.

Stochastic algorithms are characterized by the use of random or casual operations during their search process. This is done to improve exploration and guarantee that as big a part of the search space as possible is considered. An important part of their success and widespread use is the fact that they require no information on the problem to be solved, they are blind to the shape or complexity of the objective space. Algorithms in this category include Simulated Annealing, Particle Swarm Optimization, Evolutionary strategies and Genetic Algorithms. Most of these algorithms have their inspiration in natural phenomena or animal behavior. They can be defined by the following characteristics:

- Random events during the search process means that stochastic algorithms follow produce different results when they are used in the same problem.

- Stochastic algorithms consider one many at a time, and they modify their position following the information obtained by studying all of these solutions. This group of solutions is often called a population, and these algorithms are sometimes referred to as population based algorithms.

- Stochastic algorithms' have many starting points. For this reason, in multimodal problems they will arrive have a greater chance of finding the global optima when compared to deterministic algorithms.

- Stochastic algorithms usually converge at a minimum (global or otherwise) at a much slower pace, especially when compared to deterministic algorithms.

- Exploration is an important part of Stochastic algorithms, they have not only one position but many in order to better consider the entire search space.

5.2 Algorithm Selection

Search algorithms have been used by researchers and practitioners in the architecture and engineering fields for some time. The problems addressed by these architects and engineers are very varied in nature, disciplines and complexity, but an important trend towards the use of stochastic algorithms and genetic algorithms in particular is present in the literature. An investigation into optimization methods used in Building Performance Simulation (BPO) and Building Performance Optimization (BPO) was carried out by Attia et al. and published in their 2013 article "Assessing gaps and needs for integrating building performance optimization tools in net zero energy buildings design":

> "Through the 2000s, the development of mathematical and algorithmic techniques and the advancement of BPS tools gave way to BPO tools that could solve multi-objective optimization problems of a design. Mechanical and structural engineers working on complex buildings have been among the early adopters of BPO techniques, but architects and other engineers now start using these techniques as well. Today, there is a strong trend towards population-based search algorithms such as evolutionary algorithms[†] and particle swarms. These algorithms have been proven to be very successful in optimizing one or many performance criteria while handling search constraints for large design problems."

(Attia et al. 2013)

This trend is also visible in other architecture related disciplines such as structure and acoustics. Some of this examples are to be considered as search processes in the way they are presented in this PhD thesis, and others are more in line with the traditional optimization process. But in all cases the use of stochastic population based algorithms is significant.

[†]Evolutionary algorithms is a term that is used to group all stochastic algorithms that are based on natural evolution. These are mainly Genetic algorithms and evolutionary strategies.

This PhD thesis does not present a comprehensive study of many search algorithms, nor a comparison of them as to which is best suited for some problems, or which one is more efficient. This PhD thesis focuses mainly on the use of genetic algorithms. Exhaustive comparisons of different algorithms would only serve the purposes of improving efficiency of the search process, and that is no small matter. Future efforts in comparing search algorithms as to their adaptation to architectural search could prove to be very useful in reducing calculation times or usability. But it is unlikely that the use of other algorithms (Apart from genetic algorithms) would provide additional architectural knowledge, or shed further light into the subjects discussed in this research. This thesis focuses on the inclusion of search algorithms and performance information in the early design phase, therefore algorithm comparisons lie outside of it's scope.

The selection of genetic algorithms for this thesis is not casual. They have been employed by many researchers and practitioners in the architecture and construction field.

GA's have been around longer than some of the other algorithms described above, and for this reason, they have been used and studied more in depth that other. Their efficiency has been discussed and testes in many occasions, and their use has been suggested by researchers in related fields. Furthermore, many implementations of multi-objective GA's are in existence, and their performance has been evaluated and proven successful (Zitzler et al. 2000, Deb 2001).

6

Genetic Algorithms

6.1 Intruduction

Genetic Algorithms (GAs) were first proposed by John Holland in the mid 1970's in the University of Michigan, hist most important publication being "Adaptation in Natural and Artificial Systems" (Holland 1975). They have been successfully employed in varied fields of study, more important for this work, they have been widely employed in the architecture and construction field, see for example (Bogar et al. 2013, Méndez Echenagucia, Pugnale & Sassone 2013, Miles et al. 2001, Turrin et al. 2011).

> "Genetic Algorithms are search algorithms based on the mechanics of natural selection and natural genetics. They combine survival of the fittest among string structures with a structured yet randomized information exchange to form a search algorithm with some of the innovative flair of human search. In every generation, a new set of artificial creatures (strings) is created using bits and pieces of the fittest of the old; an occasional new part is tried for good measure. While randomized, genetic algorithms are no simple random walk. They efficiently exploit historical information to speculate on new search points with expected improved performance."

(Goldberg 1989)

GAs follow a darwinian model of search in which the concept of fitness is defined by the user and it can be any such function that can be expressed

numerically. In nature, survival of the fittest has, through the process of evolution, generated animal species that are very well equipped to life in their particular habitat. We can look at evolution as a search process intended to find the attributes that can best guarantee an animal's survival and reproduction under particular environmental conditions. In this analogy, survival in a specific environment is the problem set, the animal population is the set of proposed solutions, and natural selection and reproduction is the search process that generates the best solutions. While in nature its the environment to determine what problems the search process needs to solve, in computational genetic search its the user who determines what the population of solutions must achieve. This is one of the reasons why this technique has been employed in such varied fields.

As Goldberg said in the definition above, new solutions are created in GAs by the use of bits and pieces taken from the best of the previously considered solutions. These bits and pieces are taken from the problem variables. Like in all search algorithms, in GAs solutions to the given problem are characterized by a set of variables, and these variables are typically represented by numbers. These numbers are then coded, there are many ways of coding the variables, including real numbers coding, but perhaps the most common code used is binary code. Following the biological analogies used to describe genetic algorithms, this code then becomes the *chromosome* or *genome* of the individuals in the population. Like in nature, the offspring or new generation of individuals are made up from the chromosomes of their parents*. The success of the GA is related to the selection of the right parents, and the correct combination of their chromosomes.

Genetic algorithms differ from more traditional search methods in a few key points:

- GAs are population based, search is made in several points at each iteration.
- GAs normally work from a coded version of the parameters or variables, not often from the parameters themselves.
- GAs use objective or fitness functions to drive the search process, they do not use any other auxiliary or additional information.
- GAs use probabilistic transition or movement rules, they do not use deterministic calculations to determine the next step, such as gradient based search.

*A more detailed explanation of this operation is given in section 6.3 bellow.

(Goldberg 1989)

Genetic algorithms perform very well in multi modal problems. While more traditional search algorithms are vulnerable to getting stuck in local minima, GAs can surpass this issue by relying on the fact that they are population based and that their transition rules are probabilistic and not deterministic. Individuals often get stuck in local minima, but this does not mean that the GA is stuck, because there are other individuals searching the solution space. Also, the stochastic elements present in the GA can steer the search process to new directions, often out from local minima and towards global minima.

The population approach to search also facilitates the use of GAs in multi dimensional problems (problems with multiple design variables). The fact that GAs perform in Multi-dimensional and multimodal problems is associated with what is called search *robustness* (Goldberg 1989).

6.2 Exploration vs. Exploitation

In any search method, either computational or human search, there are two distinct processes at work: Exploration and Exploitation[†]. Exploration is the process responsible for covering the entire search space, include the vast majority of solutions in the search, and in the case of population based search to guarantee the preservation of diversity in the solution population. Exploitation is the process responsible for signaling out the best performing solutions, to direct the search process towards promising areas and generally reduce the search space by focusing on the best solutions.

These two processes can be considered opposite processes in that one increases the search area while the other reduces it. One preserves diversity, the other tries to focus on a small number of solutions that most often are quite similar to each other. It is precisely the contrast between these two processes that gives GAs their robustness. Most search algorithms with a deterministic transition rule, especially gradient based search algorithms, have very little exploratory power and concentrate on exploitation. This is one of the reasons why they have do not perform well in multimodal problems.

Exploratory and Exploitation power of GAs is significantly determined by its operators, and the parameters needed to control these operators. This

[†]a cognitive approach to exploration and exploitation is discussed in section 1.2

is why it is very important to get to know the operators and select the right ones.

6.3 A Genetic Algorithm Run

Genetic algorithms are made up of a series of operations performed to and with the population of candidate solutions. Coding and Scaling for example are necessary operations in a coded GA, but the most characteristic operations in GAs are the Selection or Reproduction operator, the Crossover operator and the Mutation operator. Some authors single out the mutation operator as being optional and not fundamental to the functioning of the GA, but others contend that they significantly improve the GAs efficiency.

Let's briefly look at the pseudocode of a simple GA, we will be using a binary coded GA:

```
1: Generate a random and coded population
2: for i ← 1, number of Generations do
3:     Decode and Scale the population
4:     Calculate fitness values for the population
5:     Run Selection operation
6:     Run Crossover operation
7:     Run Mutation operation
8:     Perform Exit condition test
9:     if Exit condition test = True then
10:        Exit GA
11:    else if Exit condition test = False then
12:        Continue to the next Generation
13:    end if
14: end for
```

In this section we will go through the Genetic Algorithm operations listed in the above code, and describe in detail its operators. For this purpose we will use a simple problem we will call Test problem A:

$$Test\ Problem\ A: \begin{cases} Maximize & f(x) = \frac{x_2}{x_1+0.1}, \\ subject\ to & 0 \leq x_1 \leq 1, \\ & 3 \leq x_2 \leq 5. \end{cases} \tag{6.1}$$

where x_1 and x_2 are the two problem variables, we can think of them for example as geometrical variables in a form search process, for now let's just think of them as variable numerical values. The object of this problem

is to find x_1 and x_2 values that maximize $f(x)$. We can also see that x_1 is confined to values between 0 and 1 and x_2 between 3 and 5, these are the domains of our two variables.

We will also be using a few GA specific elements that we need to properly run the GA, we can think of them as GA inputs. We will list them here and they will be explained further down. Some of these we already mentioned in the problem description:

Population Size (N)	6	
Number of Variables	2	
Number of binary digits	8 for x_1	4 for x_2
Variable Domains	$x_1 \in [0, 1]$	$x_2 \in [3, 5]$
Mutation Probability (p_m)	0.2	

6.3.1 Initial Population

In order for us create the initial population we need to use the number of binary digits in each variable. Binary digits define how the variable domain will be discretized. In this example we chose 8 binary digits for x_1, this means we will divide the x_1 domain (0 to 1) into 256 equal parts. This is because, a binary code 8 digits long contains 256 numbers. We chose 4 binary digits for x_2, so we divide its domain into 16 equal parts. Using this example we can compute how the discretization of the problem results.

We first calculate the domain length for each variable:

$$D_{len} = |D_{max} - D_{min}| \tag{6.2}$$

where D_{max} is the highest member of the domain and D_{min} is the lowest. In our example D_{len} is 1 for x_1 and 2 for x_2. We can now calculate the length of the discretized element for each variable:

$$L = \frac{D_{len}}{n} \tag{6.3}$$

where L is the length of the discretized element and n is the number of divisions or the amount of numbers in the binary digits we selected. In our example we get $L = 0.004$ for x_1 and $L = 0.125$ for x_2. We discretized x_1 in a much smaller element than we did x_2, because the domain was smaller for x_1, and most importantly we chose a higher number of digits for its discretization.

Search Complexity It is important to notice from this example that the number of possible solutions the GA will be considering is determined in these two numbers, the number of variables and their discretization. The higher number of variables or the more we discretize them, the higher number of possible solutions. We can relate the complexity of the *search* problem we pose to the GA to the number of possible solutions. In our example we have 16×256 possible solutions. This complexity can also serve us as a way to judge how many iterations of the GA are necessary to obtain an acceptable result.

Returning to the initial population, we can see that we need a binary number of $8+4$ digits for each individual (the chromosome of each solution). We generate the population using 12 random values (either 0 or 1). This is repeated this for the number of individuals in our population, in our example 6. The resulting population is the following:

Individual												
1	0	0	1	1	0	0	0	1	1	0	1	1
2	1	1	1	0	1	1	0	0	0	0	0	0
3	1	1	1	1	0	0	1	1	0	1	1	0
4	1	1	1	1	1	1	1	1	1	1	1	1
5	0	0	0	0	1	1	1	1	1	1	1	1
6	0	0	0	0	0	0	0	0	0	0	0	0

With this initial population the main loop of the GA can start.

6.3.2 Decoding and Scaling

We now need to decode and scale the binary numbers into values that we can use to calculate fitnesses. The decoding is done using the number of binary digits for each variable. In our example, the first 8 digits belong to the first variable and the other 4 belong to the second one. So let's take the chromosome of individual 1 and decode it. The chromosome is:

$$0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 1$$

If we divide it according to our coding scheme we get:

$$00110001 \quad 1011$$

We now decode these values into integers:

$$00110001 = 140 \quad 1011 = 13$$

Next we need to scale these values into the variable domains. This is done with the following equation:

$$S = D_{min} + (D_{max} - D_{min}) \times \frac{1}{M_{bin}} \times d \tag{6.4}$$

where S is the scaled value, M_{bin} is the maximum value obtainable with the number of binary digits of the value in study and d is the decoded value for the variable (140 and 13 in our first individual). In our example M_{bin} is 255 for the first variable and 15 for the second. This results in scaled values of 0.54 for x_1 and 4.73 for x_2. If we repeat this operation for the entire population we get these results:

Individual	x_1	x_2
1	0.54	4.73
2	0.21	3.00
3	0.81	3.80
4	1.00	5.00
5	0.94	5.00
6	0.00	3.00

This is the decoded and scaled population. We can now proceed to the next step in the GA which is fitness calculation.

6.3.3 Fitness Calculation

This is probably the step that need less explanation in the whole GA process. We simply follow the formula detailed in the problem definition (equation 6.1 in page 82). Of course in our example the fitness function is a very simple mathematical formula but we can think of this also as a building performance simulation or any other fitness function we can use to describe our search process. Following our formula and using our decoded and scaled values, we get a fitness for all of the individuals in our population:

Individual	Fitness
1	7.30
2	9.60
3	4.17
4	5.54
5	4.80
6	30.00

If we study the fitness formula and the results, we can see that in order to maximize the fitness we need a low x_1 and a high x_2 value. We can also see that the fitness is mostly sensitive to x_1, thus we made a good choice in investing 8 binary digits (and the added search complexity) to this variable, since we will be able to study it more in depth than x_2. In short we can already deduce that the best individual for this problem is:

x_2	x_2	code
0	5	000000001111

We will keep this code in mind when we study the rest of the GA operators and how they improve the fitness of the population.

6.3.4 Selection or Reproduction Operator

The Selection operator has three main functions:

- Identify good solutions from the population.
- Multiply those good solutions.
- Delete bad solutions from the population.

(Deb 2001)

The selection operator has the objective of choosing which of the individuals in the population will be used for reproduction. This group is called the mating pool. It is a very simple operation but it has a significant impact in the success of the GA.

From an exploration and exploitation point of view, the selection operator is most related to exploitation, but depending on the particular selection operator, they can have an influence on both. A selection operator that employs some randomness on the selection scheme is related to exploration, while selection operators that only focus on fitness to select individuals for reproduction are more related to exploitation. Different authors have proposed different selection operators that can accomplish these three tasks. Blickle and Thiele report the following list of selection operators and compare their functionality:

- Tournament Selection
- Truncation Selection

- Linear Ranking Selection
- Exponential Ranking Selection
- Proportionate Selection

(Blickle & Thiele 1995)

The Tournament selection operator uses randomly selected couples from the population to compete for a place in the mating pool. All of the members in the population are selected for two competitions with different members of the population. The competition in the couple is settled by means of their fitness values. This means that the best individual in the population will win both of its contests, and thus he will be copied twice in the mating pool. This process guarantees that there is a higher chance of good performing individuals to be selected for reproduction, increasing the exploitative power of the GA. However, since the couples are selected randomly, there is a little exploratory aspect to this particular brand of selection operator. For example, if all of the best performing individuals are coupled together, some of them will eliminate each other, leaving places open in the mating pool for not so well performing individuals. In a more exploitative selection operator, these low performing individuals would be cut out, but leaving them insures a certain level of diversity in the population, increasing exploration.

In the Proportionate selection operator the mating pool is filled in proportion to the fitness values of the individuals. If the average fitness value of the population is f_{avg} and the fitness value for the i_{th} individual is f_i, then the i_{th} individual would be expected to have a f_i/f_{avg} number of copies in the mating pool. This method is comparable to a tricked roulette wheel.

The Truncation Selection on the other hand employs no randomness in its selection. It simply sorts the population according to fitness, and selects the first individuals in the list using a user defined fraction of the population, for example 1/2. In this case there is no exploration added to the GA, exploration is left to the remaining operators.

Some selection operators require user defined parameters (for example the fraction of the population used in the truncation selection). As with other kinds of operators, the correct selection of this parameters is important for a correct GA run.

Let's use the tournament selection operator for our example GA and compute the mating pool. First we need to randomly select couples to compete with each other, twice. For example:

Tournament	Individual A		Individual B
1	3	vs.	4
2	1	vs.	6
3	2	vs.	5
4	4	vs.	5
5	1	vs.	2
6	3	vs.	6

If we run these tournaments, selecting as the winer the individual with the highest fitness (highest for a maximization problem like our example, lowest for a minimization problem), we get the following results:

Tournament	Winner
1	4
2	6
3	2
4	4
5	2
6	6

As we expected, the best individual in the population (individual 6) won both of its contests and has 2 copies in the mating pool. This was also the case for individuals 4 and 2, so we can say that individuals 1, 3 and 5 were eliminated from the GA. It's interesting to note that even though individual 1 had a higher fitness value than individual 4, it was eliminated and 4 went on. This is a product of the stochastic nature of the tournament selection operator. The resulting mating pool is the following vector:

Mating Pool	4	6	2	4	2	6

6.3.5 Crossover Operator

The crossover operator is responsible for creating a new population from the individuals present in the mating pool, it is the reproduction of the individuals in the mating pool or parent individuals and the generation of the offspring individuals or new generation. This operation is done by using the problem variables, the characteristics of the individuals. As we have seen, these variables are normally coded in some way, most commonly in binary numbers.

As it was previously mentioned in the introduction of this chapter, solutions are described in terms of their chromosomes. The crossover operator

simply copies part of the genes of one parent and the other part from the second parent, thus generating the chromosome for the child. This operation can be summarized in this way. We assume two parents A and B with the following chromosomes:

Parent A △△△△△△△△△△△△
Parent B □□□□□□□□□□□□

Now we need to specify a crossover point, the point in until which one string of genes will be used, the rest of the string will be taken from the second parent. This number is usually selected at random. Let's say we select the 8_{th} gene to be the last one in parent A. This means that the crossover operation will proceed as follows

	1 2 3 4 5 6 7 8	9 10 11 12		
Parent A	△△△△△△△△	△△△△	⟶	△△△△△△△△□□□□
Parent B	□□□□□□□□	□□□□		□□□□□□□□△△△△

The crossover generated two completely new individuals that we could call AB and BA:

Offspring AB △△△△△△△△□□□□
Offspring BA □□□□□□□□△△△△

The crossover operation supports any kind of coding, using binary code, using a larger alphabet (or rather an alphabet with a higher cardinality) to formulate the strings [‡] or even the use of real coded variables, meaning that the variables are represented numerically as real numbers.

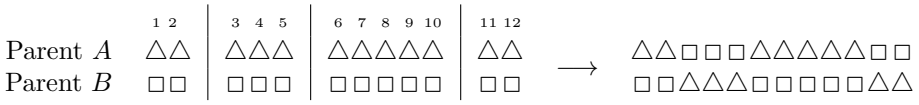
The above example employed the Simple Crossover operator. It is called the simple crossover operator because it employes only one crossover point. As we saw, this crossover point is normally selected at random, but the possible crossover points to select from need not be all gene positions. Some Crossover operators choose to only allow crossover points to be in between variables. If we for example have a coding scheme that attributes 2 genes for each variable, and there are 4 variables in the problem, then the chromosome of any individual would look like this:

	$variable1$	$variable2$	$variable3$	$variable4$
Individual	□ □	□ □	□ □	□ □

[‡]for a discussion on the effects of the cardinality of coding alphabets see (Deb 2001) pages 108-109.

In this example the only acceptable crossover point would be in between the variables, where the | lines are shown. This is especially useful in the case of real coded GAs. Otherwise there is the risk of allowing variables to go outside of their respective domains.

The other important type of crossover is the Multiple point crossover. As its name suggests it allows for the chromosome to be split at multiple positions, requiring multiple crossover points. Using the same two parents A and B we can show an example of this kind of crossover. In this case we will use 3 crossover points after the 2_{nd}, 5_{th} and 10_{th} genes:



Multiple point operators include a higher randomness to the GAs procedure, however, Goldberg (Goldberg 1989) explains that in the case of the Crossover, this increased randomness is not necessarily a good thing. He argues that increasing the braking points significantly improves the chances of breaking significant pieces of the chromosome called "schema". This makes the crossover more like a random shuffle of genes, and less like a planned creation of schemata.

Let us return to our example GA (Test Problem A). We will be using the simple crossover operator with variable crossing points. We will take the first two individuals in the mating pool and combine them to create two offspring individuals, then we will take the next two, and so on until we go trough the entire mating pool and we create a complete offspring population. We will begin with parents 1 and 2 from the mating pool (individuals 4 and 6):

Individual 4	11111111	1111	$\longrightarrow$	111111110000
Individual 6	00000000	0000		000000001111

The results of this crossover operation are quite interesting for our example. If we look at the second offspring in this operation, it is precisely the best possible individual we signaled out above (see page 86). Although there is some randomness involved in the operation, the fact that we ended up with this optimal individual is not completely by chance. It is the product of the genetic operator used until this point. The fact that the selection operator chose individuals that had 0s and 1s in the correct spaces to generate the correct variable values is not by chance. The selection operator chose them

because they had good fitness values, and then their combination yielded an even better result. On the other hand, if we look at the first result in this crossover, we notice that is exactly the opposite, it has highest x_1 value in the domain and lowest x_2 value in its domain. Hence we get the lowest possible fitness value. This is also a possible outcome in the GA. However this kind of bad result will get eliminated in the next generation and the population fitness will grow.

From an exploration vs. exploitation point of view, the crossover operator is involved in both processes. The recombination of genes from two individuals into a third one implies big changes in focus of the GA in the search space. The previous crossover example shows that the parents positions in the search space were replaced by completely new positions far way. This suggests that the exploration is at work in this operation. But as we just explained, this exploration is not entirely random, so we can see exploitation taking part in this operation as well.

If we finish the crossover for the rest of the population we obtain the following coded population:

offspring												
1	1	1	1	1	1	1	1	1	0	0	0	0
2	0	0	0	0	0	0	0	0	1	1	1	1
3	1	1	1	0	1	1	0	0	0	0	0	0
4	1	1	1	1	1	1	1	1	1	1	1	1
5	1	1	1	0	1	1	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0

Keep in mind that the crossover point is random, so not all crossover operations have the same result, even when the same individuals are involved.

6.3.6 Mutation Operator

The Mutation operator can be defined as a diversity preservation operator. It's role is mainly to increase the exploratory power of the GA by slightly altering a few genes in the chromosomes of the population. The number of genes that get altered is determined by the Mutation probability. This parameter is selected by the user as another GA input, and has different meanings in different mutation methods. the correct mutation operator for a GA must be selected depending on the coding scheme, we will first examine the simple mutation operator for binary coded GAs using offspring individual 3 from our example. For each gene in the individuals chromosome a random value $r \in [0, 1]$ is generated. The simple mutation operator

modifies a gene if this random value r is lower that the mutation probability p_m selected by the user. In our example the mutation probability p_m was set at 0.2. We will now run the simple mutation on the offspring individual 3:

Chromosome	1	1	1	0	1	1	0	0	0	0	0	0
Random Value r	.43	.56	.11	.95	.62	.22	.33	.80	.73	.59	.25	.44
Mutate	×	×	√	×	×	×	×	×	×	×	×	×
Result	1	1	0	0	1	1	0	0	0	0	0	0

We can see that of all of the genes in the chromosome, only one was modified because only one of the random values generated was lower than our selected mutation probability p_m (0.2). In the case of binary code, the modification of the gene is obvious, it its a 0 it gets turned into a 1 and vice versa.

Goldberg (Goldberg 1989) introduced a particular mutation operator that was computationally less expensive than the simple crossover, because it does not require the generation of a random number for each gene in the GA. This operator, called Mutation clock operator works by establishing a counter that reduces in number by each gene processed, and the next mutation occurs when the counter reaches 0. The length of the counter is different each time, its determined by a random value r, and it is inversely proportional to the mutation probability p_m. The higher p_m the shorter the counters will be. In this case r is only generated every time a new counter is needed, not at every gene.

When the coding scheme is not binary or even real coded, the modification of a gene is different. There are many different mutation operators for this coding schemes [§]. An example is the Non-Uniform Mutation in which the mutated gene is switched to a new value, but the value is selected by means of a probability curve. This curve is in the shape of a tent, and its highest at the previous gene value (the parent gene). In this way, the gene is likely to be modified by a value that is not that different from the parent gene.

Global and Local exploration There is a difference in exploration between the mutation and crossover operators. We saw earlier that the modifications involved in the crossover operation entailed big movements in the search space, we can call this *global* exploration. Mutation on the other hand tends to modify the chromosomes in small ways. The impact the mutation of

[§]for a review of mutation operators for real coded GAs see (Deb 2001) pages 122-124.

a single gene has on the individual depends on the point in which the change is made. For example, in binary coding, the first digits are reserved for high values, so changes in these digits have a higher impact than changes in the final digits. But even so, they would only make changes in one variable. While crossover operations may change quite radically, sometimes changing more than half of the genes. So we can say that the mutation operator is responsible for small movements in the search space or *local* exploration. The mixture of local and global exploration is also a reason for the GAs strength and robustness.

Another important aspect of the mutation operator is avoiding homogeneous populations. If we let the GA gain too much exploitative power, we run the risk of having a completely homogeneous population before the GA converges into a global optimal solution. The extreme case of this being a population made up of individuals who are identical. In this case, neither selection or crossover operators will help us regain diversity, and we will not see an increase in fitness, the GA is in fact stuck. This does not happen when we introduce mutation into the algorithm, and in particular, when a good mutation probability p_m value is selected. However, we must also be carful not to select a p_m value that is too high, because its might significantly reduce exploitation power. With a low p_m, schemata have very little probability of being affected by mutation (Goldberg 1989) so we can say that the effect of mutation is mostly beneficial if the correct p_m value is selected.

6.3.7 Elitism

Elitism can be though more as a concept than an actual operator, for it can be introduced into the GA in different ways. The point of elitism is to preserve a fraction of the best individuals in the population, in order to not let the maximum fitness value in each generation decrease. While the fitness tends to always increase generation by generation, in some kinds of problems, some stagnation can be found, and in this cases, genetic operators may even produce individuals with a lower fitness than their parents. Elitism is the process of keeping the parents in case they have better fitness that their children, thus keeping the fitness value constant in the next generation. This is a guarantee that a good solution is never lost in the GA operations unless a better solution is found. This is possible because GAs work with populations, we can afford to reserve a few "elite" spaces in the population to make sure fitness values don't decrease.

Elitism dramatically increases the efficiency and convergence speed of

GAs, especially un multi-objective problems (Deb 2001). A common way of introducing elitism into GAs is to directly copy the best α individuals directly into the next generations, while still having them participate into the selection process. Like all other GA input parameters, a good balance needs to be achieved in the selection of a α value. If we select an α value that is too high, we run the risk of loosing diversity in the population very quickly, and if we select a very low α we do not take advantage of having an elite population. Low α means higher exploration and high α means high exploitation. Values of $\alpha = 1$ and $\alpha = 0.1N$ (10% of the population) are common (Deb 2001).

6.3.8 End Conditions for GAs

Depending on the problem, there is no way of knowing when a GA has reached an optimal result, or if the GA can improve the fitness of the population by continuing to calculate. There are also no concrete rules of thumb for most of the GAs parameters such as population size or mutation probability. So deciding when to stop a GA is also a choice that is left to the judgment of the user. Never the less, at least one ending condition must be set, but multiple end conditions are also possible and in some cases desirable. There are three basic types of end conditions for GAs:

- Fitness related end conditions
- Number of calculations end conditions
- Time limit end conditions

When the problem is formulated in such a way that allows us to know a minimum acceptable fitness value, then it is very useful to have an end condition relating to fitness. For example the GA can end when the fitness value of the best individual in the population surpasses a user defined minimum fitness value. This avoids unnecessary further calculations and makes the GA more efficient. If the object of the search is not a single solution, the average of minimum fitness value in the entire population can be used in the fitness related end condition. Since there is no way of knowing whether the GA will ever reach the given minimum fitness, additional end conditions are usually given to complement this one (this is true of all fitness related end conditions). When there is no way of knowing a minimum acceptable fitness value, or such a value is not desired, other end conditions must be employed.

Another fitness related end condition is the maximum generations without improvement or maximum stagnant generations. This is useful when the user wants to achieve a certain result quickly and if the GA is not successful in a user defined number of generations, the user will reformulate the problem and restart the GA.

The maximum number of calculations can be estimated by setting a maximum number of generations plus the number of individuals in the population. It is clear that the more calculations the GA performs, the better its result will be (especially when there is an elitism component in the operators). It is then obvious that setting a number of calculations to perform is a valid end condition. This value can be set using the complexity of the problem as a guide, the more variables there are in the problem the more calculations we are going to need. But like it was said before, there is no exact way of knowing a perfect number of calculations.

When the problem at hand needs to be solved within a certain time window, then setting a maximum calculation time as an end condition is an obvious solution.

In our GA example, if we would have set a fitness related end condition the GA would have been over fairly quickly, but this is due to the simplicity of the problem we chose. If we would have set a maximum number of generations as an end condition, then the GA would not have been as quick. As we saw in the pseudocode, if the end condition would not have been met, then the loop would continue. In more complex problems than our example, the GA would continue to search both locally and globally for high performing solutions and storing valuable information in the process. This information can be accessed later on by the user as a sort of data mining of the GA progress, and in so doing, get a good understanding of the problem that was formulated.

7

Multi-Objective Search

7.1 Introduction

Now that we have seen the mechanics of the genetic algorithm (a single objective search method) we can look into the main issues of multi-objective search. In this chapter we will look into the theoretical aspects of multi-objective search. We will try to do so in a way that is independent from any specific search algorithm, and try to look more into the issues, possible outcomes and what to expect from a multi-objective search. The issues relating to specific multi-objective search algorithms will be dealt with in chapter 8.

7.2 Difference between single and multiple objectives

The obvious and most important difference between single objective, and multiple objective search is of course the number of objectives. But the implications of this difference do require some attention, especially when these multiple objectives are in contrast with one another. In order to describe this difference, let's make an example of a multi-objective problem with contrasting objectives. A typical case for is the contrast between the cost and the quality of an object. We can associate quality with the materials and manufacturing techniques of an object, but the better they are the more they tend to cost. In this chapter we will be using another mathematical test problem, in this case having two simple functions, described as follows:

$$Test\ Problem\ B: \begin{cases} Minimize & f_1(x) = x_1, \\ Minimize & f_2(x) = \frac{1+x_2}{x_1}, \\ subject\ to & 0.1 \leq x_1 \leq 1, \\ & 0 \leq x_2 \leq 1. \end{cases} \tag{7.1}$$

Test problem B states that we should minimize f_1 and f_2 which are both dependent of x_1 and x_2, and they both have a particular domain. A quick study of f_1 and f_2 reveals that the more we increase x_1 the higher the value for f_1 and the lower the value of f_2. This two functions are clearly contrasting each other. Moreover, we can say that the optimal value of f_1 is reached when $x_1 = 0.1$ and the optimal for f_2 is reached when $x_1 = 1$ and $x_2 = 1$. So if we had to minimize either f_1 or f_1 we would know what to do, but if we need to find minimal values for *both* f_1 and f_1, we do not know which x_1 and x_2 values to take. We cannot say that $x_1 = 1$ is optimal for both functions.

One way of solving this problem that is very common is to use a weighted sum of both functions. To a assign a weight factor for each function according to their relative importance. This kind of solution would in fact translate problem B into the following problem:

$$B\ (Weighted\ sum): \begin{cases} Minimize & f_w(x) = x_1 \cdot w_1 + (\frac{1+x_2}{x_1}) \cdot w_2, \\ subject\ to & 0.1 \leq x_1 \leq 1, \\ & 0 \leq x_2 \leq 1. \end{cases} \tag{7.2}$$

where w_1 and w_2 are the weight factors for each function. We could for example decide that f_1 and f_2 are equally important to us and so we give a value of 0.5 to both w_1 and w_2. In this way we are simply translating f_1 and f_2 into a single function f_w, translating the multiple objective problem into a single objective problem. We would then proceed to use a single objective search algorithm and find a single optimal solution to the problem. This approach is perfectly valid when we have information that can give us the correct weights for each function. However, in most real world problems this is not the case.

Let's discuss some of the real world objectives that are addressed in this research, like the multi-disciplinary problems discussed in chapter 19. Can we say that the structural efficiency in a building envelope is more important that its environmental quality and energy efficiency? Can we say this for all buildings? Can we say this for any building? And perhaps more importantly, can we say in which measure one is more important that the

other? Is using a 50-50 value a valid approach? The acoustic performance of a room compared to the structural capacity is also no easy matter to solve. Keeping in mind that the weighted sum approach only works if we assign correct and exact weights to all functions, in such problems using this approach would lead to arbitrary results that are of little interest.

7.3 The concept of Dominance

Going back to test problem B, we previously stated that we could not say which values of x_1 and x_2 are optimal for both f_1 and f_2. But there is something we can say about some values of x_1 and x_2 with respect to others, and it is the concept of domination. Let's start by comparing a set of solutions:

Solution	x_1	x_2	$f_1(x)$	$f_2(x)$	color
1	0.1	0	0.1	10	●
2	1	0	1	1	●
3	1	1	1	2	●

Solution 1 is the minimal(best) solution for f_1 and the maximum(worst) for f_2. Solution 2 is the exact opposite, the maximum for f_1 and the minimal for f_2. If we compare this two functions we cannot say that one is better that the other if we consider both functions, we can say that neither of these solutions dominates the other.

If we compare solutions 2 and 3 however, we can see they have equal results for f_1, but solution 2 outperforms solution 3 in f_2. In this case we can say that solution 2 *dominates* solution 3.

If we generalize this concept, we can say:

- In order for solution A to dominate solution B, solution A has to outperform, or equal B in all functions, as well as outperform B in at least one function.
- If solution A outperforms or equals solution B in all objective functions except in one in which solution B outperforms A, then A and B do not dominate each other.

To continue studying these relationships, we will introduce the search and objective spaces, as a way of visualizing these solutions in their variable domains and their function results.

7.4 Search Space and Objective Space

Multi-objective search problems are often analyzed by looking into two separate domains, called the search or decision variable space and the objective or function space.

The *search space* is a representation of the proposed solutions described by their variable values. Their variable values are mapped in such a way as to showcase their values and the relative distances and similarities between them. The search space has as many dimensions as the problem has variables, and this dimensions are confined to their respective variable domains.

The *objective space* on the other hand, is a representation of the solutions from the point of view of their objective function values. They are mapped out according to the values obtained in each objective function present in the multi-objective problem. Hence, the objective space has as many dimensions as the problem has objective functions. The confines of the objective space are determined by the objective functions, they depend on the values obtained by them.

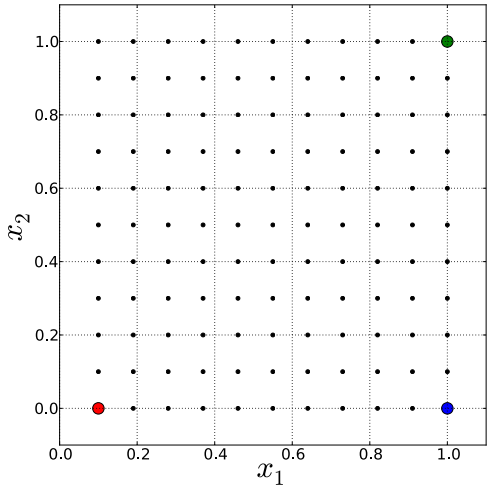
Figure 7.1a shows the search spaces for our test problem B and 7.1b shows the objective space for the same problem. The search space shows x_1 and x_2 values selected in a grid, this is done so that we can get a good idea of how the entire set of possible x_1 and x_2 values end up in the objective space. In both spaces solution 1 is represented with a red dot, solution 2 is in blue, and solution 3 in green. The rest of the values are represented with small black dots.

This way of representing the problem shows very clearly the relationship between the two functions. One very important feature to focus on is the contrast between the two functions *. This contrast can be seen in the objective space (figure 7.1b) by the fact that when solutions approach a minimal value for f_1 they loose optimality for f_2. Another important feature that can be seen in the objective space is the Pareto Front.

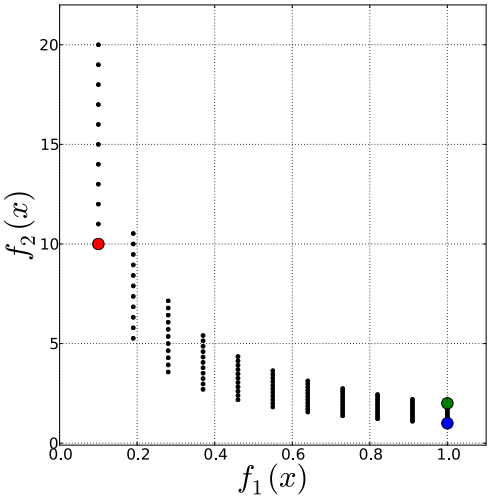
7.5 The Pareto Front

In the previous section we explained the concept of domination, we will now explain what is the Pareto Front, starting by the concept of a non-dominated individual. A non-dominated individual is one that is not dominated by any other individual in the population. A non-dominated individual typically

*A more detailed study of contrasting objectives is given in section 7.6.



(a) Search Space



(b) Objective Space

Figure 7.1: Search and Objective Space for Test problem B

dominates many of the other individuals in the population, and it is never dominated by others. A non-dominated individual may have many individuals which he does not dominate, but none that dominate him.

If we look back at individuals 1,2,3 in test problem B, we can see the following relationships:

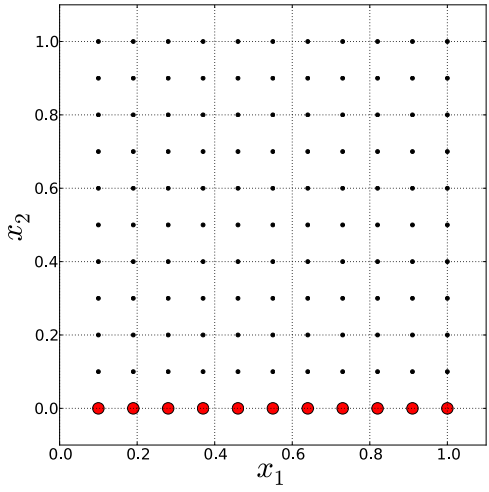
- Individuals 1 and 2 do not dominate each other
- Individual 2 dominates individual 3
- Individuals 1 and 3 do not dominate each other

Therefore we can say that only individuals 1 and 2 are non-dominated. Although individual 1 does not dominate individual 3 (as does individual 2) individual 1 is never dominated by any other solution, therefore it is non-dominated.

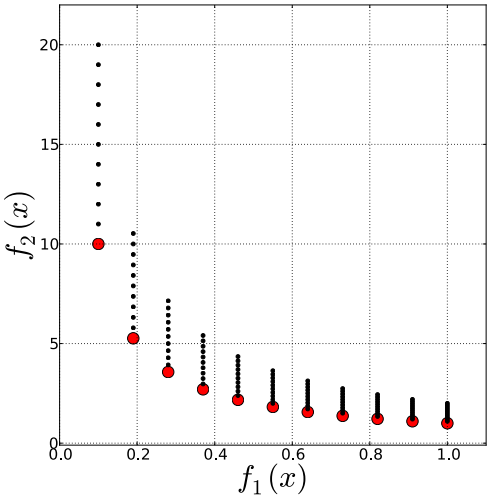
The *Pareto Front*, also called trade-off set or non-dominated set, is the set of all non-dominated solutions in a given group. They represent the set of solutions that cannot be said to be better from each other if we consider all objective functions in the problem. In our example, if we only considered solutions 1,2,3, then solutions 1 and 2 would comprise the Pareto front. But if the consider any other possible combination of x_1 and x_2 that we see in figure 7.1a, then the Pareto set would be made up of a few more individuals.

Figure 7.2 shows the entire Pareto front for test problem B, in the search and objective spaces. We can see that individuals 1 and 2 are still present in the Pareto Front and that individual 3 is not. Moreover we see a series of individuals in the objective space forming a curve that is convex towards the ideal point (in the case of a problem with two minimization functions this ideal point would have coordinates 0,0). We discretized the search space into 10 equal segments in x_1 (11 points) and 10 equal segments in x_2 (also 11 points) for a total of 121 points analyzed. If we had used an infinite number of points, the Pareto front would be a continuous curve representing all of the solutions that are non-dominated. However, since we only studied 121 solutions, we only obtained a part of the entire Pareto front. We can say that the individuals we found "lie" in the Pareto Front, and they are certainly part of it, but we cannot say that they comprise the entire non-dominated front.

The red dots in figure 7.2 represent the non-dominated set in the entire population, so we can say that the population could be divided into two groups, the non-dominated and the dominated. But if we wanted to divide the population (from a Pareto dominance point of view) into more



(a) Search Space



(b) Objective Space

Figure 7.2: Search and Objective Space for Test problem B with Pareto front in Red.

groups than just two groups, we can introduce more domination related sets. The Red dots can be considered as the first in a series of Pareto fronts. The second Pareto front would be made up of those individuals that are only dominated by individuals in the first front. If we eliminated the individuals of the first Pareto front from the population and recalculated the non-dominated set, we would in fact obtain the individuals of the second front. We can continue this subdivision into sequential fronts until all of the individuals of the population are part of one front, thus obtaining a series of sets that are "non-dominated sorted"[†].

Pareto fronts can have many different shapes, depending of course on the objective functions analyzed. As was previously mentioned, the objective space has as many dimensions as the problem it studies has objective functions, so this means we can only graphically represent objective spaces of up to 3 objective functions. But this does not mean that we cannot calculate of find Pareto fronts that have 4 dimensions or more, we simply need to represent them in a different way, numerically or through $2D$ or $3D$ "sections" in a multi dimensional space. This representational difficulty aside, the study of the shape of the Pareto front is an interesting way of studying the problem. Among the interesting features of such analysis, we can mention the discontinuity of a Pareto set, the degree of contrast (or lack there of) between objective functions and the relative dimension of the front with respect to the entire domain of feasible solutions.

Another important study is done in the position of the Pareto individuals in the search space. Figure 7.2a shows that the Pareto set in our test problem B lies entirely at the bottom of the search space and that it forms a continuous area of the space. But this is not necessarily the case, as we will see later on, the position of Pareto optimal individuals are often non continuous. This means that finding the entire set would be a multi-modal problem.

The extreme points in the Pareto set are interesting individuals, as they represent the optimal solutions for one of the objective functions. In our example, these extremes are individuals 1 and 2. We could expect the two extremes of a Pareto front to be in quite different positions in the search space, and this is commonly the case, but as we mentioned, there is not always continuity between search space and objective space. We could find individuals that are next to each other in the Pareto front, and that are very far apart in the search space. For this reason, it is considered an important

[†]for a more in depth analysis of the non-dominated sorting procedure see section 8.3.1 bellow.

element of a Pareto analysis to find a uniform distribution of solutions in the Pareto front, so that we can then get a good idea of where the entire Pareto set is in the search space. To get a good idea of the *diversity* of the solutions that cannot be said to dominate each other.

Pareto fronts can be studied independently of whether a problem has minimization or maximization functions or even mixed functions. The only difference in this case is the position of the ideal or optimal point in the objective space. As we saw above, if the problem is has both minimization functions, the optimal corner is the lower-left corner, max-max problems have it in the upper-right corner, and so on.

7.6 Contrasting Objectives

In this section we will study the contrast between objective functions from a theoretical point of view. To help us understand these issues we will be looking at the objective space of a series of problems, and most importantly the shape of the Pareto fronts in these cases.

We should perhaps start by talking about non contrasting objectives. Not all multi-objective problems have contrasting objectives, this of course depends on the objective functions we study. Sometimes we do not know exactly how these functions relate to one another before we analyze them. It is for this reason that is quite useful to study Pareto front shapes. Let's start by seeing what the objective space of a couple of non contrasting functions looks like. First we describe the problem, in this case we will call it Test Problem C:

$$Test\ Problem\ C \begin{cases} Maximize & f_1(x) = x_1 + x_2^2, \\ Maximize & f_2(x) = x_1 \cdot x_2, \\ subject\ to & 0 \leq x_1 \leq 5, \\ & 0 \leq x_2 \leq 1. \end{cases} \tag{7.3}$$

In this case we see a problem in which both functions need to be maximized. A quick view of the objective functions of problem C reveals that the more we increase the values for x_1 and x_2 we optimize both functions. So we have a set of non contrasting functions.

Figure 7.3 shows the objective space for problem C. We see that the Pareto front is comprised of a single solution (represented by the red dot). This is due to the fact that there is no contrast in the two functions. The combination of x_1 and x_2 that maximizes f_1 also maximizes x_2. Hence, this solution dominates all others in the population, and it is the only one in the

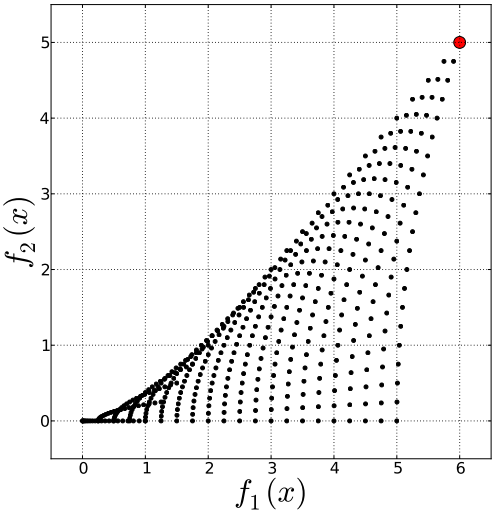


Figure 7.3: Objective Space for Test problem C.

front. The extreme case of a non contrasting problem would be one in which all functions result in equal values. If we draw the objective space for a two function problem in which $f_1 = f_2$ we would get individuals that all lie in a 45° line with a one solution Pareto front at the end of the line. In cases like this there is no need for a multi-objective approach to the problem, because we could just optimize one of the functions and find the same individual that we would find if we optimized the other. In such cases the multi-objective approach has no additional information to offer than the one provided by a single-objective search process.

Let's now move on to more contrasted problems. To do so we will study two more test problems, D and E. Test problem D is defined as follows:

$$Test\ Problem\ D \begin{cases} Maximize & f_1(x) = x_1 - (x_2)^a, \\ Maximize & f_2(x) = x_2 - (x_1)^a, \\ subject\ to & 0 \leq x_1 \leq 1, \\ & 0 \leq x_2 \leq 1. \end{cases} \tag{7.4}$$

where a is a constant that we will use to transform the level of contrast

of the problem. Looking at f_1 and f_2 for problem D we can already say that they are contrasting equations. Furthermore we can say that the end result of the search process in problem D depends greatly on the value given to a.

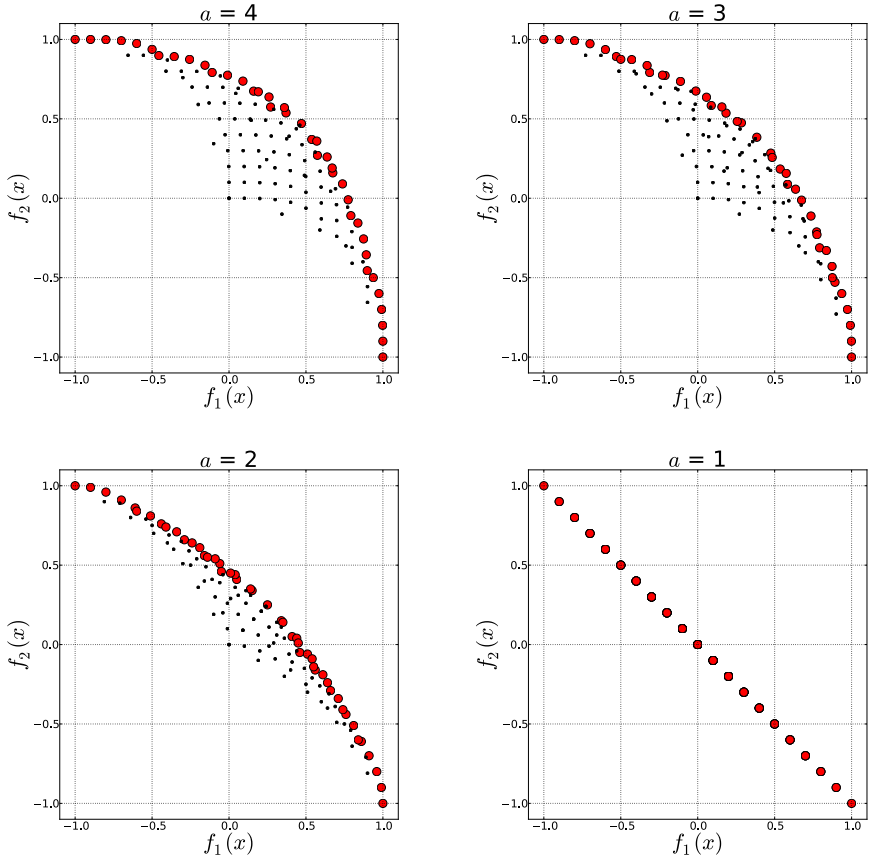


Figure 7.4: Objective Spaces for Test problem D with varying values for a.

Figure 7.4 shows the objective space for four versions of problem D, where $a = 4, a = 3, a = 2$ and $a = 1$. When $a = 1$ the Pareto front forms a 45° line that is perpendicular to the ideal point (in this case upper right corner). We can also see that as the value of a increases, the Pareto front becomes an

increasingly convex curve (convex towards the ideal point). A convex curved Pareto front denotes contrast in the objective functions, because increasing the value of f_1 necessarily means decreasing f_2. But we can study the rate in which f_2 decreases as we increase f_1, and use this rate to compare the contrast between these four Pareto fronts. In this sense we can say that the higher the rate of decrease, the higher the contrast. Of course the rate of decrease in these curves is variable, but we will look at the rate of decrease at the midpoint.

Looking at figure 7.4 we can see that in all cases the Pareto Front goes from a f_1 value of -1 (the worst value) to an optimal value of +1. The same is true for f_2. Let's start by looking at the rate in which f_2 decreases as we increase f_1 in the case of $a = 4$. When f_1 is at -1 we get $f_2 = 1$, but when we increase f_1 to 0, $f_2 = 0.75$. So from -1 to 0, we only lost 0.25, we have a rate of descent of 25%.

Now, let's make the same observation for $a = 2$. We can see that the point where $f_1 = 0$ corresponds to an $f_2 = 0.50$, A rate of descent of 50%. So we can clearly see that the rate in descent is higher in $a = 2$. Following the same logic, we can say that among these four curves, the highest rate is that of $a = 1$, in which, the rate of descent is 100%. We can conclude that convex Pareto fronts are a sign of contrasted problems, but also that they are more and more contrasted as they become less convex, and more linear.

But what happens when the Pareto front becomes concave? To study concave Pareto fronts we will use test problem E described as follows:

$$Test\ Problem\ E \begin{cases} Maximize & f_1(x) = \frac{x_1}{x_2}, \\ Maximize & f_2(x) = \frac{x_2}{x_1}, \\ subject\ to & 0.1 \leq x_1 \leq 1, \\ & 0.1 \leq x_2 \leq 1. \end{cases} \tag{7.5}$$

The resulting Pareto front is shown in figure 7.5. The rate of descent at the midpoint in this case is close to 198%, which is higher than the one we found in the linear Pareto front (100%). In the case of concave fronts, the more concave they are, the higher the rate and the higher the contrast.

To summarize this section, non contrasted objectives are identifiable by a Pareto front made up of a single solution. Contrasting objectives are shown by fronts with more than one solution. The degree of contrast of the functions can be determined by the shape of the front, convex is a low contrast, linear is a constant contrast, concave is a high contrast. Pareto fronts may be a combination of shapes, they may contain linear, convex and concave parts. They might also be discontinuous. But we can say that

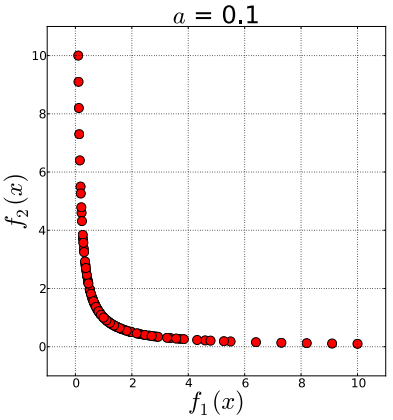


Figure 7.5: Objective Space for Test problem E.

the behavior that we have studied will be true for the parts of the front corresponding to the shapes we have just mentioned.

7.7 Final Selection Criteria

So far we have learned that in a multi-objective problem, there is no way of determining which is the single individual that can best satisfy *all* objective functions. But in real world multi-objective problems, we will have to select a single solution for the problem. If for example, the problem we face regards the structural vs. acoustical properties of a building, we cannot build all of the solutions in the Pareto front. The Pareto front represents a very useful tool in reducing the number of alternatives in the search space, to a limited number, but we still need to choose a single solution out of the front. This section tries to address this issue and talk about criteria for making a final selection.

Pareto fronts describe only the functions we introduce into the problem, so the first alternative to select a solution from the Pareto front is to use information that was not introduced in the problem. This is what we have described in the first part of this thesis as implicit search goals and what Deb calls higher-level information (Deb 2001). We can use functions that were not present in the original problem, or we can use information that is

possible to introduce in the problem.

Not all functions are introduced in multi-objective problems for a series of reasons. Sometimes the phenomenon these functions describe are not as relevant as the other functions, or sometimes the functions are computationally too expensive to be used in a search process. The convective thermal exchange in complex geometry for example is computationally very expensive to calculate, sometimes requiring more than a day. Ideally we would not need to run this calculation more that a few times.

There are other factors about the building that we usually do not introduce into search processes that can help us select from the Pareto front. The one that comes first in mind is an aesthetic judgement of the solutions. Search processes are quite useful in gathering information about the design object that can be measured, calculated or simulated. Typically, we use it to inform us on physical phenomena occurring in the building, its structural capabilities, acoustical or lighting quality of the spaces or thermal exchanges. Also economical and construction issues can be measured or simulated, but aesthetic issues are not. Search algorithms can mostly help us with the "tame" parts of the design process, and provide us information that can help us deal with the "wicked" parts[‡]. The importance of designer interaction in search processes was discussed in section 1.7, and the selection of a final solution is an important moment of interaction.

There is also the alternative of further refining the Pareto individuals with the use of Data Mining techniques. If we have Pareto sets made up of too many alternatives for a designer to consider, we can also use clustering and classification algorithms that can further reduce the number of options. We can look at the distance in the Search space (the variable differences) as a way of differencing individuals in the front. We know that a good multi-objective search process will produce a set of well distributed individuals in the Pareto front, so there is already a bit of data mining involved. But the solutions in the set may come from any part of the search space, and this information is not present in the Pareto front. A useful refinement of the Pareto set would be the separation of solutions by search space distance, to signal out solutions that are most different form a variable point of view. As is the case in the studies shown in this research, this variables represent geometrical features of the solutions. So we would be sorting solutions that have the shapes that differ the most.

In this frame of mind, diversity is an important issue. We want to keep diverse population from a variable (search space) point of view. Following

[‡]For an explanation of "tame" and "wicked" problems see section 1.3.

this logic it makes good sense to keep a record of all of the Pareto fronts found during the process. A search history that can help the designer interact and consider near-optimal solutions, that are perhaps very meaningful from a geometry point of view. We can think of this as a different kind of exploitation, a designer involved directly in the exploitation process can yield important results.

8

NSGA-II

8.1 Introduction

NSGA stands for Non-dominated Sorting Genetic Algorithm. After a first non elitist Multi-Objective GA, NSGA-II was developed by Kalyanmoy Deb and his students in 2000 as an elitist version of NSGA (Deb 2001). NSGA-II has been employed successfully in many architecture and construction related problems.*

Being a genetic algorithm, NSGA-II shares the same overall GA dynamic that was explained above. There is a main loop that iterates generation by generation, there is fitness evaluation (in this case we have multiple fitnesses) and there are selection, crossover and mutation operators. These operators however are specially designed to work in multi-objective problems. In addition to these modifications, NSGA-II has two special operators that will be described bellow.

As it was outlined in chapter 7, the final output of the multi-objective search process is not a single solution. NSGA-II was designed to obtain a set of solutions evenly distributed in the Pareto front, taking full advantage of the fact that GAs work with populations of solutions. This solutions will be evaluated for their dominance within the population, and they will also be evaluated for their distribution along the Pareto front.

An important characteristic of NSGA-II is that it works with a population that will change in size during the procedure. If we select an initial

*see for example (Attia et al. 2013), in this article there is a complete review of optimization tools used for building performance optimization. A part of the article is devoted to the use of NSGA-II in this field.

population of size N, during the course of the main loop, this population will be doubled to $2N$ and then taken back to N. In this way NSGA-II considers a *parent* population and a *current* population in the same loop, thus providing the possibility of elitism.

8.2 The NSGA-II procedure

Figure 8.1 is the flow chart for NSGA- II. We can see the general procedure is not very different ftom that of the simple GA. The main loop is enclosed in the segmented rectangle, the population size at changing points is signaled in red, and the operations that most differ from the GA are signaled in yellow.

While in the single-objective GA the fitness values were one for each individual in the population, in NSGA-II we have a matrix of fitness values. The matrix has a dimension that is equal to the number of individuals in the population times the number of fitness functions in the problem. The fitness calculation is basically done in a double loop, as follows:

1: **for** $i \leftarrow 1, numPopulation$ **do**
2: **for** $j \leftarrow 1, numFitnessFunctions$ **do**
3: $FitnessValues\ [i, j] = f_j(x)$
4: **end for**
5: **end for**

In this way the entire population is calculated for $f_1, f_2, f_3, ..., f_n$, and the fitness values are stored in the matrix $FitnessValues$.

We can also describe the NSGA-II algorithm with the following Pseudocode:

1: Generate a random and coded *Initial* population with n number of individuals
2: Decode and Scale *Initial* population
3: Calculate fitness values for *Initial* population
4: Copy resulting fitness values to *Parent* population vector
5: Generate a random and coded *Current* population with n number of individuals
6: **for** $i \leftarrow 1$, number of Generations **do**
7: Decode and Scale *Current* population
8: Calculate fitness values for *Current* population
9: Combine *Parent* and *Current* creating a vector of size $2n$

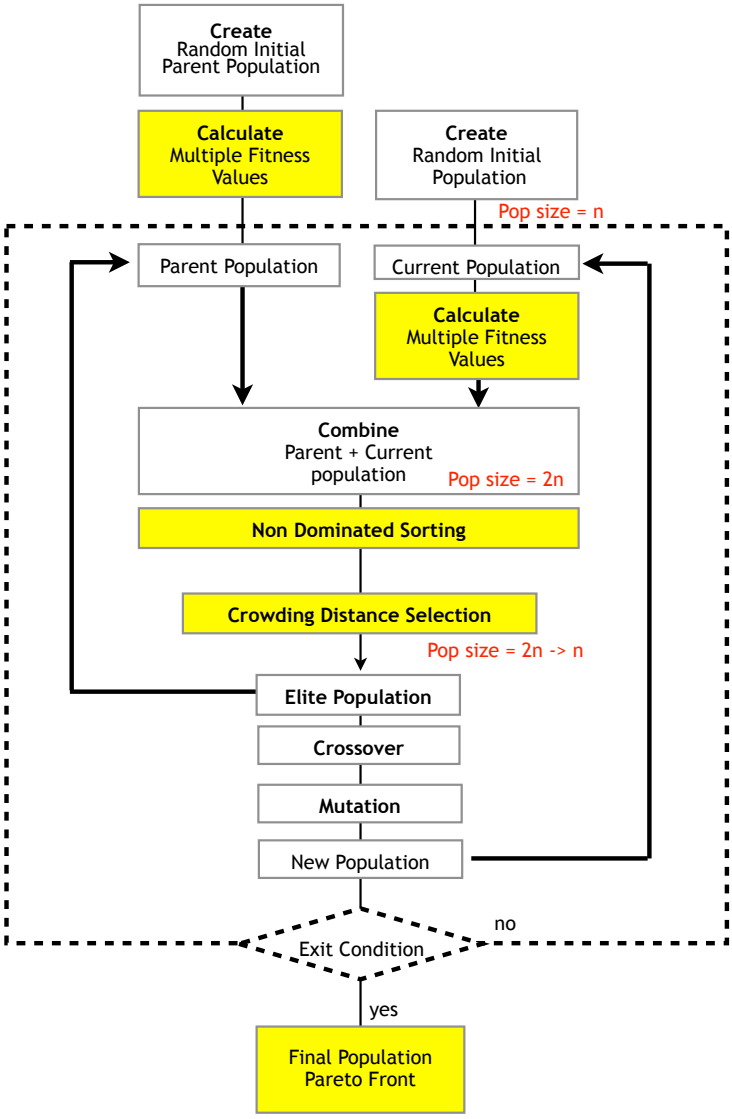


Figure 8.1: NSGA-II Flow chart

10: Run Non Dominated Sorting Algorithm for the combined population
11: Run Crowding distance Algorithm for the combined population
12: Replace *Parent* population with first n individuals from the combined population according to the NDS and CD results
13: Run Tournament Selection operation
14: Run Crossover operation
15: Run Mutation operation
16: Use the resulting population to replace *Current* population
17: Perform Exit condition test
18: **if** Exit condition test = `True` **then**
19: `Exit` NSGA-II
20: **else if** Exit condition test = `False` **then**
21: Continue to the next Generation
22: **end if**
23: **end for**

In steps 1 trough 4 of the above pseudocode we generate, decode, scale and calculate fitnesses for the initial population that will serve as the first *parent* population. Since we are only at the beginning of the Multi-Objective Genetic Algorithm (MOGA) we do not yet have a parent population so we need to create one randomly and have ready its fitness values before entering the main loop. Also necessary before entering the main loop is the generation of the random *current* population, seen in step 4.

Step 6 marks the beginning of the main loop, and the first thing we do in it is to decode and scale the randomly generated current population (step 7). We calculate fitness values for the current population, and we combine it with the parent population, thus forming the *combined* and changing the number of individuals in the population to $2N$. The combined population is further studied by the non-Dominated sorting and crowding distance operators. These special operators will be described in section 8.3. Their main function is to provide information necessary to use the selection operator. Since we have many fitness functions, selection cannot be done by simply selecting the individuals with the highest fitness (as was explained in chapter 7). Selection in NSGA-II is done by using the sequential Pareto fronts, the non-dominated individuals are preferred to the dominated ones. Further refinement is done in the crowding distance to signal out the best distributed individuals on these fronts.

During selection, we take the best half of these individuals, they will replace the parent population such as they are, thus preventing the elite individuals from being lost in further operations. We will use these same elite

individuals for crossover and mutation operators. During selection we return to a population size of N. We perform crossover and mutation operations to create the new population, the offspring population. We replace the current population with this new population and we conclude the operations in the loop.

8.3 Special Operators

All Multi-objective search algorithms (genetic or otherwise) need to incorporate special operators that allow them to work with multiple fitnesses. In comparison with the normal GA, NSGA-II has a series of modifications to its operators, most importantly its selection operator. In particular NSGA-II does not use the fitness values directly to select the best individuals and consequently the individuals who will be used for reproduction. As its name suggests, NSGA-II uses a Non-dominated Sorting (NDS) algorithm to asses the position of all solutions in the objective space, to sort them according to Pareto fronts. Additionally, NSGA-II uses a special *diversity preservation* algorithm called Crowding Distance (CD). It is the combination of the values obtained with the NDS and CD algorithms that NSGA-II selects its best individuals.

In this section we will go trough these two algorithms in detail in order to better understand the way NSGA-II works.

8.3.1 Non Dominated Sorting

The Non-dominated Sorting algorithm starts simply by comparing all of the individuals in the population. The comparison is done in all of the different fitness values present in the problem. Each fitness value can come from different types of problems (maximization or minimization).

When comparing individuals in the population, the algorithm has to produce two sets of information:

- The first one is a *Domination Count*. For each individual in the population, the algorithm has to count how many other individuals dominate it. This information is stored in a single vector that is as long as the population of individuals, called *DominationCount*.
- The second is a *Dominated set* of each individual in the population. This means that for each individual in the population, the algorithm must keep a list of all those other individuals which are dominated

by it. This information is kept in two vectors. The first vector called *DominatedSet* contains all of the dominated individuals, the dominating individuals that correspond to the dominated set are kept in the second vector called *DominatingIndividuals*.

The following example can help to clarify the method. Figure 8.2 represents the objective space of a two objective problem f_1 and f_2. On the X axis we see fitness values for f_1 and on the Y axis the values for f_2. Both functions are minimization functions. The numbers in the figure (1 to 9) represent individual solutions in our population, and we need to sort these individuals into non-dominated sets (a sequence of Pareto Fronts).

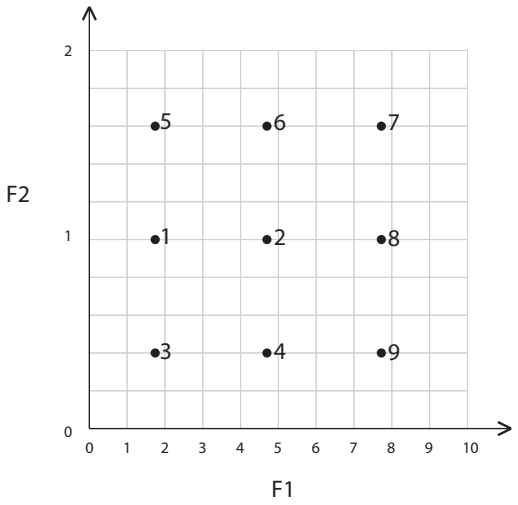


Figure 8.2: Example Objective space for f_1 and f_2

In this case the *DominationCount* vector would be the following:

Individual	1	2	3	4	5	6	7	8	9
DominationCount	1	3	0	1	3	5	8	5	2

The *DominatedSet* and the *DominatingIndividuals* vectors would be the following:

Dominated Set	1111122233333333444445568 99
Dominating Individuals	5672867812456789267896777 78

Once we have all of this information memorized, the NDS algorithm has to use these three vectors to define and memorize the Pareto fronts. We first define the way we will memorize the fronts. They will be copied in two vectors, one of them called *ParetoFrontIndividuals* and it contains all of the individuals of the populated, organized by Pareto fronts, the other one is called *ParetoFrontIndexes* and it contains the indexes in which a particular front ends. In this example:

ParetoFrontIndividuals	3	1	4	2	5	9
ParetoFrontIndexes	0	2	5			

The first front is comprised solely by the individual 3, because the index indicates that the first front ends at the 0 index, the second front goes from the 1 to the 2 index, so it comprises the 1 and the 4 individuals, and the third front goes from the 3 to the 5 index (the 2 the 5 and the 9 individuals). This result is shown in figure 8.3.

To create the Pareto fronts, the algorithm has to perform the following procedure explained in pseudocode:

```
1: ParetoFrontNumber = 0
2: while length(ParetoFrontIndividuals) < numPopulation do
3:     for i ← 1, numPopulation do
4:         if DominationCount = 0 then
5:             ParetoFrontIndividuals.append(i)
6:         end if
7:         ParetoFrontIndexes.append(length(ParetoFrontIndividuals)−
   1)
8:     end for
9:     ParetoFrontNumber = ParetoFrontNumber + 1
10:    for all k ∈ ParetoFrontNumber do
11:        for h ← 1, length(DominatedSet do
12:            if DominatingIndividuals[k] = ParetoFrontIndividual[h]
   then
13:                if DominationCount[ParetoFrontIndividuals[k]] => 0
   then
14:                    DominationCount = DominationCount − 1
```

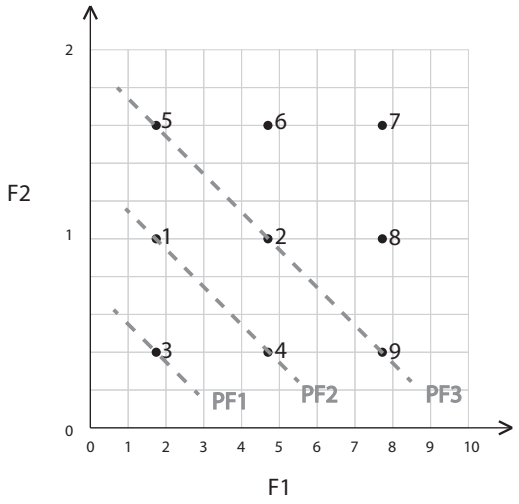


Figure 8.3: Example Objective space for f_1 and f_2 - Solved for the first 3 Pareto Fronts

15: **end if**
16: **end if**
17: **end for**
18: **end for**
19: **end while**

After this algorithm we have the whole population divided into sequential Pareto fronts. These fronts are then going to be used by the selection operator, along with the results of the crowding distance algorithm.

8.3.2 Crowding Distance

The crowding distance algorithm has the responsibility of determining which solutions are most dissimilar from each other from the point of view of all of their fitness values. This guarantees that the end result of NSGA-II will be a set of solutions well distributed along the entire Pareto front. In genetic terms, this operation is often described as a niching operation. It bears resemblance to the clustering operations common in data mining techniques, in that the Euclidean distance between individuals is used to determine their

similarity.

For a given set of solutions F, and a given set of fitness values m the crowding distance for all individuals $f_{i,m}$ in the set can be calculated by the use of the following algorithm:

1: **for** $m \leftarrow 1$, number of Fitness Functions **do**
2: $f_m^{max} = max_m(F)$ ▷ we find the maximum fitness m in F
3: $f_m^{max} = min_m(F)$ ▷ we find the minimum fitness m in F
4: create vector I_m so that $I_m = sort(f_m, >)$
5: $d_{I_{first}^m} = \infty$ ▷ we assign an infinite CD for the first individual in I_m
6: $d_{I_{last}^m} = \infty$ ▷ we assign an infinite CD for the last individual in I_m
7: $d_{I_j^m} = 0$ ▷ we first assign a CD of 0 to all individuals
8: **for** $j \leftarrow 1$, number of individuals in I_m **do**
9: calculate and assign $d_{I_j^m}$ using equation 8.1
10: **end for**
11: **end for**

$$d_{I_j^m} = d_{I_j^m} + \frac{f_m^{(I_{j+1}^m)} - f_m^{(I_{j-1}^m)}}{f_m^{max} - f_m^{min}} \tag{8.1}$$

As we can see in steps 2 and 3 the maximum and minimum values in each particular fitnesses are signaled out. They are very important in that they represent the maximum distance between solutions in that fitness function (in that dimension). This maximum distance serves for a sort of normalization operation that is done in equation 8.1 as we can see by their presence in the denominator. All other distances are compared after being normalized with the maximum distance.

Extreme solutions in each function are protected. The extreme cases serve to mark the end point of the Pareto front, hence they are given an infinite distance to allow them to pass the subsequent genetic selection operator. In that selection operator the individuals with the highest CD will be considered superior to those with lower CD. The total crowding distance of each individual is equal to the sum of all of its distances in all dimensions.

Once we have both the crowding distance and the Pareto fronts, we have all we need to employ our NSGA-II selection operator.

8.3.3 NSGA-II End Conditions

The exit conditions for NSGA-II, as for the simple GA, can be many combined and they can be varied in nature. End conditions relating to number

of calculations such as number of generations, and end conditions relating to time are valid alternatives. Fitness related end conditions on the other hand make much less sense when compared to the simple GA. In the simple GA we can establish that the algorithm should stop when the fitness function reaches a minimum acceptable value. This is not a good solution in the case of multiple and contrasting objectives, because we might reach that value without obtaining a complete and well populated Pareto front. A maximum number of stagnant generations can be used to stop a MOGA that is not evolving properly, however, stagnation is far less likely when we have many different functions. Also in many cases the MOGA can remain stagnant but still work to better populate the Pareto front.

After these considerations we can see that perhaps the most common and sensible end condition for a MOGA is a maximum number of generations or a maximum calculation time.

8.4 NSGA-II Python Implementation

NSGA-II was implemented into Python for this research. In this section we briefly describe the programing details of the implementation.

Since the main objective of this research is to use MOGAs to search for high-performing architectural shapes in many disciplines, the MOGA had to remain completely independent of the fitness functions employed. A function neutral NSGA-II implementation was thus created, following the following structure:

The Main file contains all of the user related inputs, it is the only file that needs editing to make different MOGA runs. Fitness functions are selected in this main file, as well as all of the genetic variables such as population size, binary string length and mutation probability. In this main file, an instantiation of the NSGA-II class is created, and trough it we pass the all of the above mentioned input data to the MOGA.

The NSGA-II class contains all of the necessary functions to run the MOGA, all of the operators are separated into functions, thus allowing us to replace a given operator with another almost effortlessly. The main loop of the MOGA is contained in the most important function in the class. This function is called Multioptimize, and it follows the pseudocode described in page 112. It is considered the most important function because it is the only function that is called directly in the Main file, and because it is the one calling all of the genetic operator functions.

The Multioptimize function shares information directly with the fitness

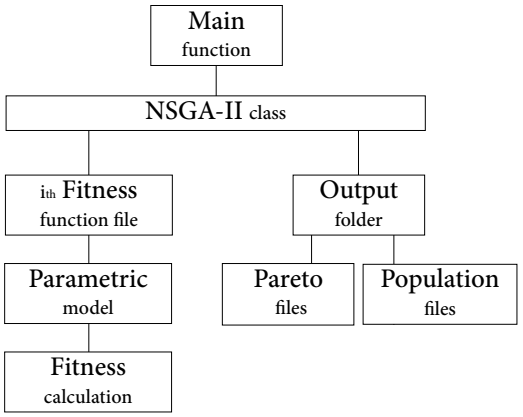


Figure 8.4: Diagram of the Data structure for the NSGA-II implementation.

functions, they are called by it, and they receive the decoded and scaled individuals to be studied. This fitness functions generally call many other functions, to calculate structural, acoustical and/or energy phenomena and attribute a fitness value accordingly. This functions will be studied in more depth in the chapters dedicated to each discipline. But perhaps more importantly, contained in the fitness function, is the parametric model. The individual solution variables in this research almost always represent geometric data to be fed to a parametric model. This model is often content specific, meaning that the model generated for structural analysis is quite different from the one used by energy simulations. They differ in their level of detail, type of computational geometry, their representation and commonly their discretization.

Fitness values are returned to the Multioptimize function in oder for it to pass it to the necessary genetic operator functions. At each generation of the MOGA, two output files are written and saved in their respective folders as shown in figure 8.4. One file contains all of the data necessary to describe the individuals in the combined population from the search space point of view. This is the population file. The other file expectedly contains the information to describe the objective space at each particular generation. It is called the Pareto file.

The particular implementation of NSGA-II used in this research uses an end condition that considers only the number of generations. When the

MOGA has completed a user defined number of generations the algorithm stops and generates another file containing the final results. This file is stored in the output folder.

8.5 Mathematical Benchmarks for NSGA-II

A series of simple mathematical functions were used to test the capabilities of the NSGA-II python implementation to find Pareto front individuals. By observing the resulting Pareto fronts and comparing them to those found in literature, we can asses the correctness of the implementation of NSGA-II.

8.5.1 Benchmark A

The first benchmark problem we presented to NSGA-II is found in "Multi-Objective Optimization using Evolutionary Algorithms"(Deb 2001) page 176. In this book the problem is called $MinEx$, it is used throughout the book to compare the performance of many different algorithms. $MinEx$ is described in equation 8.2:

$$MinEx \ : \begin{cases} Minimize & f_1(x) = x_1, \\ Minimize & f_2(x) = \frac{1+x_2}{x_1}, \\ subject\ to & 0.1 \leq x_1 \leq 1, \\ & 0 \leq x_2 \leq 5. \end{cases} \tag{8.2}$$

The implementation of NSGA-II was used to search for solutions for $MinEx$. The following GA input was given to NSGA-II:

Population Size (N)	100	
Number of Variables	2	
Number of binary digits	8 for x_1	8 for x_2
Variable Domains	$x_1 \in [0.1, 1]$	$x_2 \in [0, 5]$
Mutation Probability (p_m)	0.1	
End Condition	Number of Generations	50

Figure 8.5 shows the parameter and objective spaces for Benchmark A, $MinEx$, at the end of its run (after 50 generations).

We can see that the algorithm found a very good distribution of solutions along the Pareto front. The results compares fairly well with the results found by Deb in his book. We can interpret this as a sign that the Python

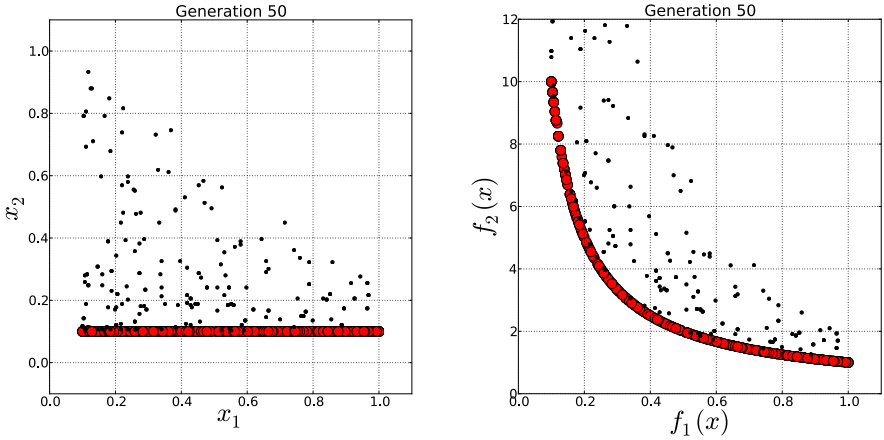


Figure 8.5: Parameter and Objective Spaces for Benchmark A.

implementation of NSGA-II done for this research was successful. Figure 8.6 shows the search and objective spaces found at generation 1, 5 and 10. We can see the different Pareto fronts found by NSGA-II in these generations and note how the evolutionary process improves the result at each iteration.

Generation 1 has a non-dominated set that is not yet close to the real Pareto front that we see in Generation 50. However, already at this early stage, we can see that x_2 values are all between 0 and 1, no solutions with x_2 between 1 and 5 are being considered because they give sub-optimal values. This is a very quick reduction of the search space. By generation 5 almost all $x_2 = 0.1$ (the correct value) but not perfectly. As a result, the Pareto curve is not yet perfectly drawn. We can also see that at generation 5 the distribution of individuals in the Pareto front is not very regular. When we reach generation 10 all $x_2 = 0.1$, and we have a fair distribution, but this distribution is further improved generation by generation. By the time we reach generation 50 the distribution has improved considerably.

8.5.2 Benchmark B

The second Benchmark we set for NSGA-II is described in (Zitzler et al. 2000). Zitzler et al. wrote a series of two function problems to test and compare the efficiency of multi-objective search algorithms. These problems

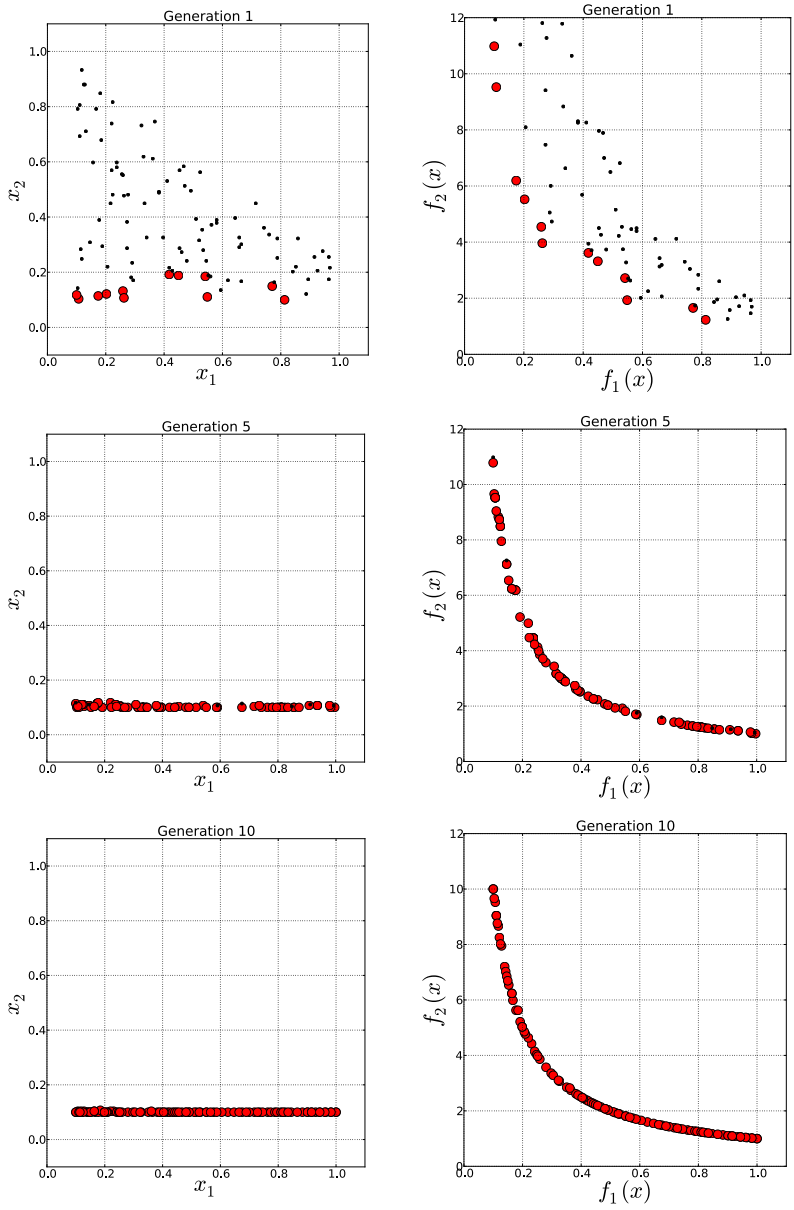


Figure 8.6: Search and Objective Spaces for Benchmark A at generation 1(top), generation 5 (middle) and generation 10 (bottom).

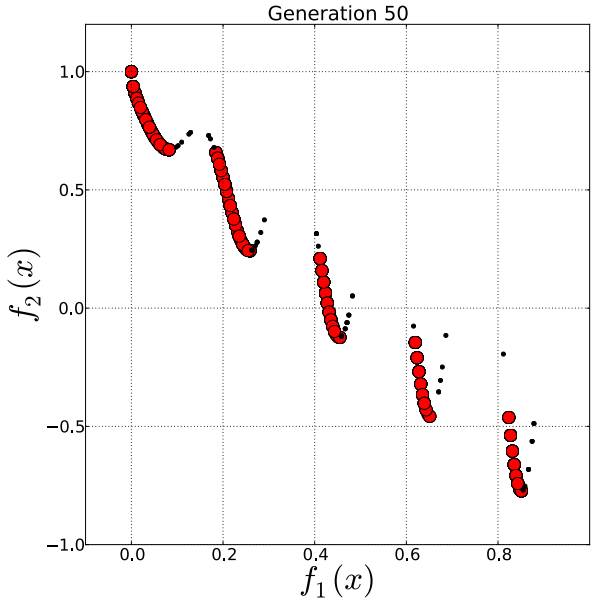


Figure 8.7: Objective Space for Benchmark B.

are called ZDT1 all the way to ZDT6. They all posses a high number of variables n and they vary in complexity. In particular, for this research we will be using ZDT3 as Benchmark B. The most important characteristic of ZDT3 is that it results in a very discontinuous Pareto front, composed of 5 different curves. Discontinuous Pareto fronts are of particular difficulty for search algorithms as they require global and local search methods to work simultaneously. Also the high number of variables (in the case of ZDT3 $n = 30$) means that the problem complexity is high. ZDT3 is described in the following equation:

$$ZDT3 : \begin{cases} Minimize & f_1(x) = x_1, \\ Minimize & f_2(x) = g(x) \cdot h(f_1(x), g(x)), \\ where & g(x) = 1 + \frac{9}{n-1} \cdot \sum_{i=2}^{n} x_i, \\ & h(f_1, g) = 1 - \sqrt{f_1/g} - (f_1/g) \cdot sin(10\pi f_1), \\ subject\ to & 0 \leq x_i \leq 1 \\ & n = 30. \end{cases} \tag{8.3}$$

Genetic variables for Benhckmark B were as follows:

Population Size (N)	100
Number of Variables	30
Number of binary digits	8 for x_i
Variable Domains	$x_i \in [0, 1]$
Mutation Probability (p_m)	0.1
End Condition	Number of Generations 50

Figure 8.7 shows the Objective space for Benchmark B. In it we can see that out implementation of NSGA-II was capable of finding an even distribution of individuals in all 5 curves of the Pareto front. This result also compares very well with the results found by Zitzler et al.

9

Parametric Models

In part I of this thesis we discussed parametric models from a architectural point of view, in this chapter we will go trough them in a more mathematical approach, starting from a mathematical definition. Daniel Davis provides such a definition in his PhD dissertation:

> "Returning to the Concise Encyclopedia of Mathematics, a parametric equation is defined as a "set of equations that express a set of quantities as explicit functions of a number of independent variables known as parameters". The mathematical definition can be refined by recognizing that the "set of quantities" in the context of design representation is typically geometry (although not always). Thus, a parametric model can be defined as: *a set of equations that express a geometric model as explicit functions of a number of parameters*."

(Davis 2013)

In our previous example problems we have not dealt with geometry, we have been using simple equations. The MinEx problem we saw on page 8.2 had two simple functions $f_1(x) = x_1$ and $f_2(x) = \frac{1+x_2}{x_1}$, in which x_1 and x_2 are the parameters. This set of equations conforms well to the first part of the definition we just saw, the definition for a parametric equation.

A parametric model would use its parameters to generate a geometrical object. We can make an example of a parametric model of a surface with two parameters. In this example the surface will be parametrized by means of a couple of Parabolas. These Parabolas will be determined by their heights

h_1 and h_2, which will be our parameters. Using These two parameters we will be able to generate the complete surface, calculate the position of any point in the surface, for any combination of h_1 and h_2 values.

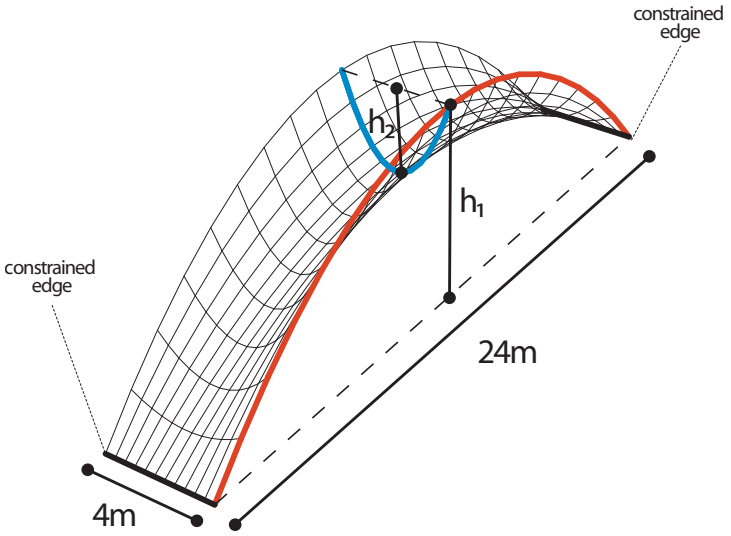


Figure 9.1: Example Parametric Model - Parametric Surface

Figure 9.1 shows a diagram of the parametric model, the two parabolas and the use of h_1 and h_2. The surface has a fixed rectangular shape in plan, the rectangle is 24×4 meters. The short borders of this rectangle are constrained, they will not change position in the Z coordinate, no matter which values h_1 and h_2 take. All of the other points in the surface are subject to changes in their Z coordinate.

The first parabola is anchored at the short edges of the rectangle (hence it is 24 meters long in plan), and it's height is determined by h_1. This parabola is the longitudinal one shown in red. We can see that the surface is symmetric, and that the first parabola is repeated at both ends of the surface. The second parabola (the transversal one shown in blue) is anchored at the midpoint of the first parabola and its height is determined by h_2.

The parameters on their own will not generate the whole geometry, they are merely the variables in the set of equations that will. In the parametric modeling techniques employed in architectural research and practice, and also the ones used in this thesis, these set of equations are computationally solved with the help of powerful CAD software that can generate even complex geometry with simple commands. Our current example is simple enough for us to use the equations of the parabolas to determine the geometry, this will allow us to see how a simple parametric model is computed. The parabola is usually expressed as a function of x and y cartesian coordinates:

$$\begin{aligned} y = f_{(x)} &= ax^2 + bx + c \\ &or \\ x = f_{(y)} &= ay^2 + by + c \end{aligned} \tag{9.1}$$

where a, b and c are coefficients that determine the shape and position of the parabola. But a more parametric equation, and an equation that is more directly related to our problem constraints is:

$$y = f_{(x)} = \frac{(bx - x^2) \cdot 4h}{b^2} \tag{9.2}$$

where h is the height of the parabola and b is it's base. In this case, given h and b we can get y values for all points along the x axis. Since our parametric surface is a tridimensional object, we will need to take this into account in our set of equations. They will be set up in such a way as to obtain the Z coordinate of a point determined by its X and Y coordinates, and of course our parameters h_1 and h_2. Since in our example, the plan of the surface is fixed to a rectangle measuring 24×4 meters, we can say that $0 \leq X \leq 24$ and $0 \leq Y \leq 4$. We can also say that b in the parametric parabola equation (equation 9.2) will be $b_1 = 24$ for the longitudinal parabola, and $b_2 = 4$ for the transversal one. The following set of equations defines our parametric surface:

$$\begin{aligned} Z &= \tfrac{(b_1 X - X^2) \cdot 4 \cdot (h_1 + H)}{b_1^2} \\ where \qquad H &= \tfrac{(b_2 Y - Y^2) \cdot 4 h_2}{b_2^2} \end{aligned} \tag{9.3}$$

where H is the height of a longitudinal parabola that has its midpoint at the transversal parabola at Y. In this sense we can see the set of equations as giving us a series of parabolas at X that have a height determined by h_1, h_2 and the transversal parabola.

So if for example we wanted to draw one surface using our parametric model, we would have to select value for our parameters h_1 and h_2, and then

could compute all values of Z for a grid of X and Y points, thus creating point cloud of our surface. If we then connect these points with lines we would obtain a 3D mesh representation of our surface like the one shown in figure 9.1.

9.1 Parametric models and Search Space

The most important thing to consider when creating a parametric model for the purposes of computational search is the possible outcomes present in the model, the kind of geometry that we will be including in the model (and thus the search) and the kind of geometry that would not be included. When we create the parametric model we are effectively defining the *search space* of our search problem, defining which set of geometry we will study. Designers generate parametric models in many different ways, all having advantages and disadvantages, but all parametric models have their limits, they can generate a wide range of geometry, but not all geometry. The limits of the model are defined by the parametric equations and the domains we give to our parameters.

In order to better understand this we will use the parametric surface example described above and see what kind of geometry we can get out of it, and what kind we cannot.

Figure 9.2 shows a series of surfaces that were all generated with our parametric model. 9.2(a) shows the resulting surface when both h_1 and $h_2 = 0$, we get a flat surface. If we fix $h_2 = 0$ but we assign a $h_1 \neq 0$ we obtain a single curvature surface as shown in 9.2(b). Double curvature surfaces are also possible, if we set both h_1 and $h_2 > 0$ or both h_1 and $h_2 < 0$ we get sinclastic surfaces or positive double curvature surfaces as shown in 9.2(c). On the other hand if we set $h_1 > 0$ and $h_2 < 0$ or $h_1 < 0$ and $h_2 > 0$ we obtain anticlastic surfaces or negative double curvature surfaces, as shown in 9.2(d).

Limiting the geometric possibilities that a model has is a good choice when using computational search methods. This allows the designer to only consider a very defined set of geometric possibilities at a time, if he so wishes. Not only is it possible to limit the geometry within dimensional values (for example limiting the surface from reaching heights above 10 meters), but also is possible to limit the geometry in a more qualitative way. Using boundary domains in the models parameters we can easily limit the possibilities of the model, we can prevent double curvatures, concave surfaces from the top, convex surfaces, flat surfaces, etc. The following table shows a few possible

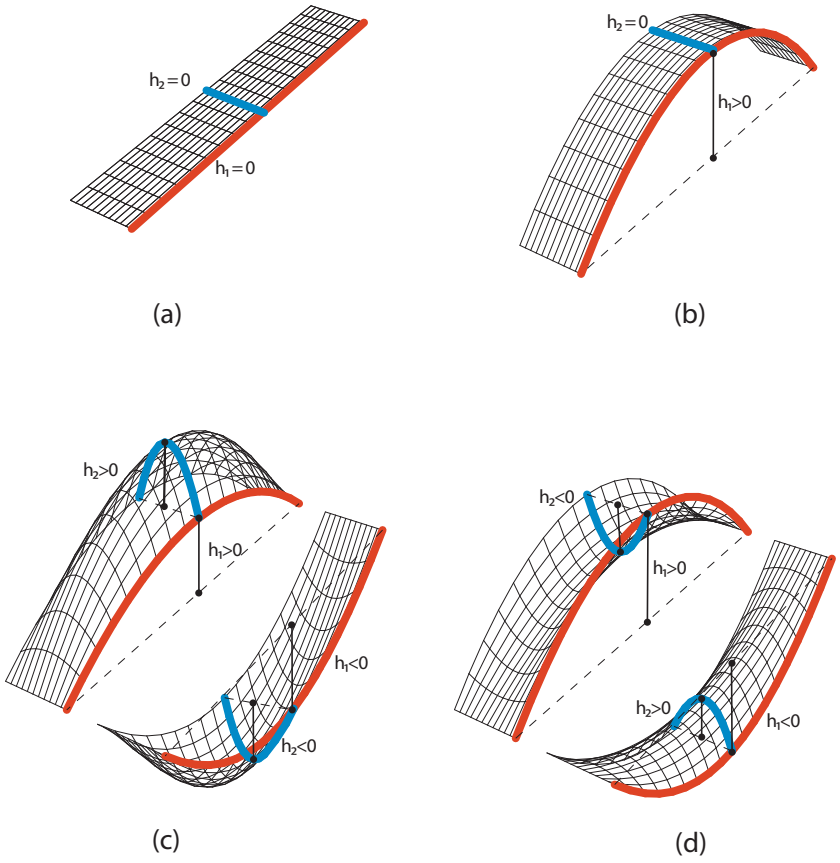


Figure 9.2: Parametric Surface possible Outcomes

domain combinations of h_1 and h_2 and the included and excluded geometry:

h_1 domain	h_2 domain	Included	Excluded
$-\infty < h_1 < \infty$	$h_2 = 0$	single curvature concave convex	sinclastic anticlastic
$h_1 > 0$	$h_2 > 0$	sinclastic convex	single curvature anticlastic concave
$h_1 < 0$	$h_2 < 0$	sinclastic concave	single curvature anticlastic convex
$h_1 > 0$	$h_2 < 0$	anticlastic convex	single curvature sinclastic concave

We can se that the parameter domains are a powerful way of controlling the search space. But of course the vast majority of the geometrical options are defined with the parametric equations themselves. No matter what limits or values we give to the parameters, in our example we will not obtain folded surfaces, multiple vertices or periodical surfaces. If we wanted to include one or all of those possibilities in the search space we would need to define a new model.

.