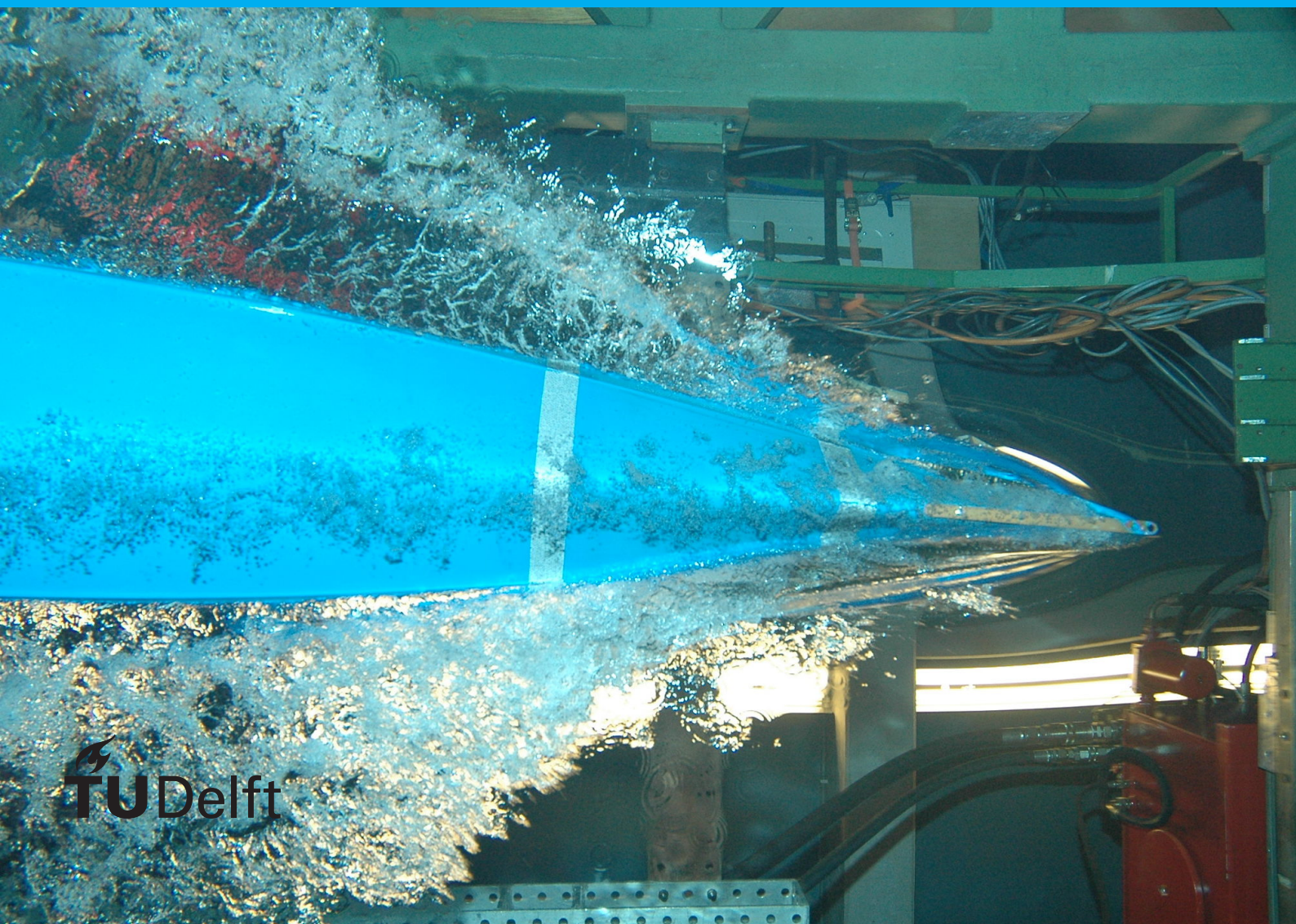
Automatically designing diverse golf course routings

W. R. Morssink



Acknowledgements

W. R. Morssink
Hazerswoude-Dorp, July 2022

I would like to thank dr. Neil Yorke-Smith, my supervisor from Delft University of Technology, who provided critical and constructive feedback throughout my thesis project.

Furthermore I would like to thank Frank Pont, my supervisor at Infinite Variety Golf Design, who provided a me with this opportunity for this thesis project.

Finally I would like to thank my family and friends, who supported me throughout the project.

Abstract

Designing golf course routings is a challenging problem as a golf course should obey the golf course regulation, be safe to play by having enough room between the holes and be diverse and challenging to the golf players. A potential exact solution can come in the form of a constraint programming model, in which solvers take care of the generation of the golf course routing. However, there are some issues when working with a constraint programming model as exponential calculation and scalability are tough to deal with. A random search algorithm can deal better with the golf course routing problem. By dividing the problem into two smaller sub-problems. The first step consists of generating holes 1 to 9, while the second part takes care of the remaining holes. A genetic approach can also work by mutating parts of the initial population and using crossover to swap holes around. These three approaches are tested and compared against one another.

Contents

1 Introduction 1
1.1 Problem description . . . 1
1.2 Thesis research questions . . . 2
1.3 Thesis structure . . . 3

2 Related Work and Background 5
2.1 Related Work . . . 5
2.1.1 General adversary networks . . . 5
2.1.2 Race track design . . . 6
2.1.3 City design . . . 6
2.1.4 Building design . . . 6
2.2 Background . . . 7
2.2.1 Constraint Programming . . . 7
2.2.2 Genetic algorithm . . . 7
2.2.3 Random search . . . 7
2.2.4 Training data . . . 8
2.2.5 Computational geometry . . . 8

3 Problem statement 9
3.1 Basic terminology . . . 9
3.2 Hard constraints . . . 9
3.3 Soft constraints . . . 10
3.4 Subjective constraints . . . 10
3.5 Physical programming . . . 10
3.6 Mathematical Model . . . 11
3.6.1 Notation . . . 11
3.6.2 Helper functions . . . 11
3.6.3 Parameters . . . 11
3.6.4 Decision variables . . . 11
3.6.5 Implied variables . . . 11
3.6.6 Objective functions . . . 11
3.6.7 Hard constraints . . . 11
3.6.8 Soft constraints . . . 12
3.6.9 Ranges soft constraints . . . 12
3.6.10 Subjective constraints . . . 12

4 Constraint Programming 13
4.1 Transforming the model . . . 13
4.1.1 Constraints . . . 13
4.1.2 Helper functions and predicates . . . 13
4.2 Challenges . . . 14
4.2.1 Variables domain . . . 14
4.2.2 Functions . . . 14
4.3 Split generation . . . 15
4.4 Diversity . . . 15
4.5 Scalability . . . 16
4.6 Conclusion of constraint programming . . . 18

5 Heuristics approaches 19
5.1 New approach . . . 19
5.2 Building blocks . . . 19
5.3 Random search . . . 20
5.3.1 Designing holes . . . 21
5.3.2 Backtracking . . . 22
5.3.3 Selecting the best solutions . . . 22
5.4 Genetic programming. . . . 23

6 Results 25
6.1 Test setup . . . 25
6.2 Constraint programming model . . . 25
6.3 Random search algorithm. . . . 27
6.3.1 Problems halfway results. . . . 27
6.3.2 Results random search algorithm . . . 28
6.3.3 Genetic algorithm . . . 30
6.3.4 Comparison between the three approaches . . . 32

7 Conclusion & Recommendations 33
7.1 Recommendations . . . 34

A Appendix A 35
A.1 Problem desciption . . . 36

B Appendix B 39
B.1 First Model . . . 40
B.2 Second Model generator 1 . . . 45

C Appendix C 49
C.1 Halfway results L-shaped run . . . 50

D Appendix D 51
D.1 Diversity on 6 hole courses . . . 52

Bibliography 57

1 Introduction

The sport of golf originated in Scotland in the 16th century [16]. While the exact origins may not be clear, it rapidly grew into a big sport, and nowadays it still is. As of 2019, there were almost 40000 different golf courses in over 200 different countries [15], and as of 2021 there are over 66 million golfers worldwide [20].

The goal of the sport of golf is to hit the golf ball from the start of a hole called the tee towards the end of the hole called the green. At the end of the hole on the green, there is a hole in which the players should putt the golf ball. Golfers hit the golf ball with their clubs, which come in different forms and sizes and try to use a minimum amount of strokes to complete a single golf hole. A game of golf usually consists of playing a series of nine or eighteen holes in series.

With the active sport golf is, golf courses need to be appealing to all of their players. Golf courses themselves need to be entertaining for the average player while still being challenging for the better players. Designing golf courses with enough variety and difficulty is therefore essential to keep appealing to the golf players. Furthermore, golf courses should also comply with the safety requirements as there should be a minimum distance between the holes and between obstacles in the form of surroundings of the golf course like a neighbourhood or woodlands.

The design of a golf course is created and worked out by a golf course architect. The first golf course architects came from the 19th century. As the name implies, golf course architects are responsible for designing safe and entertaining golf courses. The process of designing golf courses is mostly done by hand with the assistance of a computer in the form of visualization. Golf course architects create not only new golf courses but also re-design old ones as many of these approximately 40000 courses have poor routings as described in the problem statement in Appendix A.1.

1.1. Problem description

There are several factors that the golf course architect should consider when designing a golf course. All of the holes should fit into the piece of land that is available for the golf course while keeping enough distance from potential hazards on the outer edges of that piece of land. The clubhouse's location is also vital as it is the starting point for the golfers. This means that the clubhouse is a crucial location in the design process, and therefore the start of a series of holes and its end should be near the clubhouse to reduce the walking distance for the players. A golf course consists of 18 holes, and these 18 holes are often split into two individual loops of 9 holes. This means that a golfer can choose to play the whole golf course or one of the two loops, holes 1 to 9 or holes 10 to 18. The starting points, the tee of holes 1 and 10 and the greens of holes 9 and 18 need to be close to the clubhouse. A golf course routing consists of the playing lines of each of the individual holes.

Figure 1.1 shows a golf course routing, the first step in designing a golf course. A golf course routing consists of the playing lines of each of the individual holes. Each hole is represented as a red line, with the green being located at the green circle, while the tee is at the other end of the line. The blue outside border shows the outer edges of the area and the box located in the top middle is the clubhouse.

Figure 1.1: Example of a golf course routing

After the clubhouse's location, the golf course architect is tasked with designing an interesting golf course routing. This is an overview of the playing lines of all 18 holes together. After the playing lines are created, the golf course architect can design the individual holes themselves: placing the tee and green, and adding obstacles and hazards. Golf architects are good at designing the individual holes but have difficulty with creating the routings as mentioned in Appendix A.1. This is mostly done by trial and error based on their expertise and experience, but this takes a lot of time. This thesis will look into the possibility of the automatic generation of golf course routings.

While there are many different papers discussing golf course architecture or golf course design [5] [7], none of the literature I found contained any type of automatic generation for golf course design. Related work to automatic generation can be found in the areas of city planning, building design and track generation, but the research done on these topics does not translate completely to the golf course routing problem. When looking at possible techniques that can be used, evolutionary algorithms, random search, and constraint programming seem promising to use for generating golf course routings.

Constraint programming is an exact approach to the golf course routing problem, which uses a solver to solve a model consisting of all the constraints regarding golf course routings. These constraints ensure that the properties, such as the length of a hole, are secured and that the safety requirements are met. Due to constraint programming being an exact approach, scalability and speed can be limiting factors to the effectiveness of constraint programming.

Both a random search and genetic algorithm do not guarantee optimal results but can create generated routings of good quality within reasonable time. The random search algorithm would generate a golf course routing starting from the first hole and finishing at the final hole. The genetic algorithm would swap holes between already created golf course routings and mutate a selection of holes.

These three approaches have been implemented to find the best solution to the golf course routing problem.

1.2. Thesis research questions

The goal of this thesis is to find a solution to automatically generating golf course routings. The golf course problem first needs to be divided into smaller parts, which can be solved in order to create a final solution. This will be done by formulating a research question and sub-questions for the golf course routing problem:

> How can one automatically design a diverse set of golf course routings considering the architect's preferences?

1. Which techniques can be used to effectively design golf course routings?
2. What are the constraints for a golf course routing?
3. How to provide the architect with a diverse set of routings?

1.3. Thesis structure

This thesis contains seven chapters in order to find an excellent solution to the golf course routing problem and answer the research questions from section 1.2. In chapter 2 related work to the golf course routing problem will be discussed, such as race track generation and building design. This chapter will also look at solving techniques, which could be used, and go over the computational geometry that is needed to satisfy the safety constraints. Chapter 3 dives deeper into the problem description by going over all the rules and regulations regarding the golf course routing and will provide a mathematical model containing all these requirements. Chapter 4 will look into the use of constraint programming for the golf course routing problem by creating a constraint programming that can generate golf course routings. In the next chapter, chapter 5, a different approach to the constraint programming solution will be used to create both a random search algorithm and a genetic algorithm. In the chapter of the results, chapter 6, the developed solutions to the golf course routing problem will be tested, analyzed and compared against one another. Finally, in the last chapter, chapter 7, the thesis will be concluded by providing the answers to the research questions and giving several recommendations for future work.

2

Related Work and Background

2.1. Related Work

There are several papers regarding golf course design and golf course architecture, such as Hurzan [5] who discusses the history and evolution of golf course architecture or Johnson [7] diving into what he explains are the five principles of golf design. However, non of the work on golf course design and architecture I found had any research on the topic of automated design for golf course routings. As the problem description from appendix A.1 is the reason for this thesis, it will lead in terms of golf course routing constraints over the related work on golf course architecture.

2.1.1. General adversary networks

A generative adversarial network (GAN) is a machine learning framework proposed by Goodfellow. The concept of a GAN is to use two neural networks instead of only one generative neural network. The first network is called the generative model, which tries to produce new samples that resemble the training data, but only gets feedback from the second model. The second model, which is called the discriminative model, tries to guess which samples are from the training data and which samples are created by the first model. The two models of the GAN are competing against one another in order to improve.

Oh [13] makes use of GANs in the field of engineering designs. Oh uses the Boundary Equilibrium Generative Adversarial Networks (BEGAN) framework to generate two-dimensional wheel designs. Besides using the BEGAN framework, Oh worked with topology optimization to create new 2D wheel topologies. With this combination, the training data of 658 topologies was turned into 2004 newly generated topologies with two generations of the BEGAN framework and the topology optimization.

In a completely different profession, Vanhaelen discusses the use of GANs in the field of biomedical sciences [19]. GANs can be used in the field of chemistry to discover new molecular designs for drug design. Several GAN systems have been experimented with and show promising results. However, the area of biomedical sciences brings some additional challenges to using GANs for drug discovery. While the results from several different approaches seem to be alright, verifying the results is still a challenge. The workflow of design-make-test-and-evaluate in drug design is an expensive and time-consuming cycle, in which the GANs can only help with the first part. Furthermore, a lot of manual input is required for creating the training data from chemical building blocks and turning it into footprints that can be used by the AI.

GANs could be a viable option for generating golf courses. The generative model can generate golf course routings, while the discriminative model is trained with existing routings and judges the generated routings. Based on the two examples from the literature above, there are still several challenges with using GANs. In both instances, the GAN was only a part of the pipeline as additional verification and optimization was used in order to improve the results. Another challenge comes with the formulating and gathering of the training data in order to be used by the GAN.

2.1.2. Race track design

An unexpected similarity can be found between race track generation and golf course routings. A race track consists of a series of turns and starts and begins at the same point. This is similar to the clubhouse on a golf course, as the first hole starts close to the clubhouse and the final hole ends near the clubhouse as well. Loiacono [8] discusses the use of evolutionary computation to generate racing tracks for racing games, focusing on creating a diverse set of racing tracks. Loiacono uses search-based procedural content generation in combination with an evolutionary algorithm for The Open Racing Car Simulator (TORCS). Two entropy functions were used regarding speed and curvature diversity in both single- and multi-objective genetic algorithms to focus on maximizing either one of the two entropy functions or maximizing both simultaneously.

While there are similarities between a racing track and a golf course, the way Loiacono generates tracks for TORCS cannot be applied to golf course routing as the relation between parts of the race track does not translate to the relation between holes. However, the part concerning the entropy functions could be useful as golf courses have several different components that are used to measure their quality. The tactic Loiacono applied by optimizing one of the entropy functions or both simultaneously of the generation process could translate to designing golf course routings.

2.1.3. City design

Designing golf courses is not the only design process which is done by hand. The design of city layouts is also mostly done by hand. Stojanovski [18] describes the workflow of how cities can be designed using City Information Modeling software (CIM). This is a process that is done by hand inside the CIM. In the current generative design process, the role of using AI is limited to only validating the designs. Stojanovski compares a flowchart of a typical generative algorithm consisting of a problem, framing the design problem, and generating and evaluating the solutions. In this model, there is only one feedback loop between the generating part and the evaluating part. By increasing the role of the AI in the design process of cities by including additional feedback loops of the developed solutions and incorporating the preferences of the city architect into the framing part of the design process, CIM and the AI could work better together, making the tasks of the architect easier.

A more concrete example of using generative design for city design is provided by Daher [3], in the form of planning a refugee container city. Daher's process consists of two parts. The first part of the framework consists of preparing the container city on a high level, while the second part of the framework takes care of designing the smaller subpart of the city generated by the first part. The first part consists of planning possible access to the site where the city will be built and placing the clusters of facilities, such as the reception for the whole city and the living area for the different groups of refugees. These clusters have restrictions on their placement as some clusters need to be easily accessible from each other. The second part of the framework designs the clusters by generating a layout for the clusters and planning the locations of the containers.

A generative approach could be a good way to generate golf course routings by having some kind of custom framework that can generate good quality golf course routings. While both generative approaches above are about city design, they both work very differently with their own framework. The main takeaways are from the higher level, such as the splitting into smaller subparts and a feedback loop, which could be applied to generating golf courses.

2.1.4. Building design

Similarly to city design, building design can be done by making use of a generative algorithm. Zhang [23] proposes a way building design can be done by making use of a generative algorithm in order to improve building design in order to make residential buildings energy efficient. Zhang makes use of four steps in order to design a building. Before the first step begins, first all the features of the building need to be inputted. These features range from the orientation of the building and floor plan to the size of single units and heating facilities. The first step consists of a generative module which generates possible solutions for the floors. These candidate solutions are verified in the next part of the model in the evaluation module. If all the constraints are satisfied, then the candidate solutions pass on to the third part, but if the constraints are violated, the feedback is provided back to the generative model. The third part consists of energy simulation in the generated building and the fourth consists of visualizing the designed building for the user.

2.2. Background

2.2.1. Constraint Programming

Constraint programming is an effective paradigm for solving search problems with an ample solution space, also known as combinatorial search problems [17]. The programmer creates a constraint programming model by providing decision variables, constraints for the model and a target goal. The decision variables are the variables that are set by the constraint programming solver, which solves the model. The constraints consist of all restrictions the decision variables need to satisfy in order to have a correct solution. The goal of the model can be either finding a satisfactory solution or finding an optimized result.

Constraint programming can be a feasible solution to the golf course routing problem. By formulating the decision variables to consists of the individual holes of a golf course, the golf course routing problem can be written as a constraint programming problem. The constraints consist of ensuring the golf course regulations, such as the properties of a hole and the safety margins. The objective of the constraint programming model would be to optimize the overall rating of the generated golf course routing consisting of items like the overall walking distance and length of the course.

MiniZinc [11] is a constraint modelling language in which constraint programming problems can formulate on a high-level [21] and be solved by a wide range of solvers. Chapter 4 will look into constraint programming as a solution for the routing problem.

2.2.2. Genetic algorithm

A genetic algorithm is a type of evolutionary algorithm which can be used for optimization problems [10]. A genetic algorithm starts with an initial population consisting of the candidate solutions, which can be seen as chromosomes. In contrast, the individual parameters of each chromosome are described as a gene. The goal of a genetic algorithm is to improve the initial population over several generations in order to find good solutions to the problem.

When the initial population is selected, the genetic algorithm modifies the candidate solutions with two actions: mutation and crossover. In the mutation part, one or multiple genes of the chromosomes are changed in order to add some randomness and diversity between the solutions. The crossover part consists of combining the genes of two chromosomes in order to design a new one. After executing multiple of these actions on the initial populations many new chromosomes are created. The final part of the genetic algorithm consists of the selection part. In this last part, all the newly created are judged by a fitness function which determines the change that they are selected for the new generation. Chromosomes with a higher score have a bigger chance of being selected for the next generation, while chromosomes with a low score only have a small change to be selected.

This process of creating a new generation is repeated several times with the goal of improving the score of each generation in order to find an optimal solution. The genetic algorithm can have different stopping conditions, for example, after a specific amount of generations or when the score of a generation reaches a predetermined threshold.

While genetic algorithms do not generate new solutions but require an initial population to start, it can still be used to design golf course routings. A golf course routing can be seen as a chromosome, with the individual holes being represented as genes. This would allow for swapping parts of two solutions and mutating a part of the current solution. When having an initial population of golf course routings, these can be improved and optimized via a genetic algorithm. Important to note here is that after the mutation and crossover actions, the newly created candidate solutions need to be verified that they are still good golf courses and comply with the safety constraints.

2.2.3. Random search

Random search algorithms are helpful in dealing with complex ill-structured optimization problems [22]. Zabinsky discusses how using random search deals well with problems that have an objective function that has no derivative. Instead of always finding the global optimum solution according to the objective function, random search algorithms can create a good solution relatively fast compared to other types of search algorithms.

Random search can also be applied to generating golf course routings. In the generation process, choices regarding placement of the holes are not clear cut and choosing partially random could help reach a solution. This solution could later be improved via a genetic algorithm if the quality is not high enough.

2.2.4. Training data

At this moment, there are over 40000 different golf courses around the world [15], which could function as training data. However, not all of these different golf courses are of good quality. According to Infinite Variety in appendix A.1, many of these 40000 courses contain a routing of the holes, which is not up to the standards. Another challenge is extracting all the routings from the courses and labelling them. The routings of around 3000 different courses can be found on ProVisualizer, an online tool for designing golf courses. For creating more training data, satellite images of the remaining courses would be extracted. Not just the routings but also the clubhouse and obstacles would need to be labelled, and the courses need to be rated by a golf course architect.

2.2.5. Computational geometry

Many of the related work above uses some kind of generative algorithm in order to generate candidate solutions. While many of these approaches check and validate the proposed solution to see if they satisfy all the constraints, this could also be done during the generation process. Either way, geometric equations are needed to ensure that the safety margins around the playing lines are of the correct size and measure the length of several components, like the length of the individual holes and the walking distance between them. Trigonometric functions are also required in order to check the dogleg angle and diversity in playing direction.

For measuring the length of the individual components of the golf course, a basic function to determine the distance between two points can be used, which is shown below in equation 2.1. In this equation, the distance between point a and point b is calculated, where a_x indicates the x-coordinate and a_y indicates the y-coordinate of point a.

$$d(a,b) = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2} \tag{2.1}$$

For measuring the distance between two line segments, a different function is required. Antoine describes a function that requires a line to be represented in a different way than just its starting point and end point [1]. Formula 2.2 shows a mathematical way to represent all the points on a line segment between points P1 and P2.

$$P = P1 + \alpha(P2 - P1) \tag{2.2}$$

With equation 2.2 Antoine can combine the formulas of the two line segments in order to check if these line segments intersect [1], see formula 3.3. If the lines intersect, that means that there is a point P, which is on both line segments. If there is a point on both line segments, the value of both α and β should be in the range of [0,1]. Otherwise, the line segments overlap.

There are situations in which simply checking if lines overlap does not suffice. In the case that there could be an obstacle completely inside the safety margins of a hole without intersecting with the playing line or outrebounds of the safety margin, one needs to check this in order to prevent it. Preparata and Shamos present a solution to this [14], by drawing a line from the centre of the object that could potentially be inside a hole to a point outside the safety margin. By keeping track of how often this newly created line intersects with the hole, one can determine if the point is inside the polygon, defining the hole or outside of it. If the amount of times the drawn line intersects with the hole is even, it means that the line entered and exited the hole an equal amount of time and thus is the obstacle on the outside of the hole. If the count is uneven, then the point is inside the safety margins of the hole.

Trigonometric functions are needed to calculate the angle of the doglegs and rate the diversity of direction of the golf course. The direction of a hole for measuring the diversity can be calculated by using the sin or cosine functions. However, for calculating the angle between two line segments and determining the balance between doglegs played to the right and left, another function is needed. Function 2.4 allows for measuring the angle between two lines: the result is the angle between the two line segments, and a and b represent the angle of the line segments individually calculated by the cosine function. The difference between the two angles is always a number between 0 and 180.

$$diff = min(a - b, 360 - (a - b) \tag{2.3}$$

3

Problem statement

The sport of golf originated in Scotland in the 16th century. While its origins may not be clear, it rapidly grew into a big sport. The first golf course architects came from the 19th century [4]. As the name implies, golf course architects are responsible for designing safe and entertaining golf courses.

3.1. Basic terminology

Golf courses consist of several holes linked together, most commonly 18 holes, which the golfer plays. Each hole starts with the tee and ends with the green. The tee is the place where the golfer tees off and hits the golf ball from the tee in the direction of the green. The green is at the end of the hole, where the golfer tries to putt the golf ball into the cup. The playing line is the line from the start of a hole to the end.

Not every hole consists of a just straight line. The par of a hole indicates the difficulty and length of that hole and serves as an indication of the number of strokes a golfer would need to complete that hole. The par of a hole can be either 3, 4 or 5. Par 4 and par 5 contain 1 and 2 doglegs, respectively. A dogleg is bent in the playing line of at most 80 degrees.

There are more components of an individual hole, namely the rough, the grass next to the fairway and hazards in the form of bunkers or water. These components are not part of the first step of golf course design nor of this research. After the initial holes are designed when the playing lines are finished, then most of these components are added.

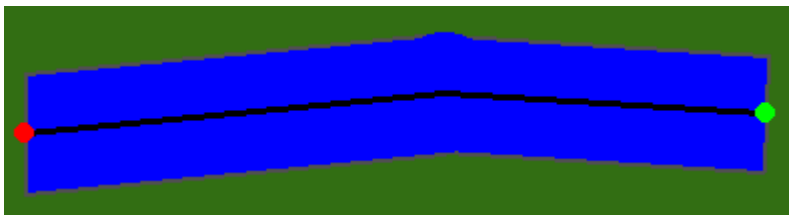


Figure 3.1: Example of a par 4 hole. Played from the tee (red dot) to the green site (green dot) with a dogleg.

3.2. Hard constraints

There are some hard constraints for the golf course in the form of safety and design. Looking at an individual hole, there should not be another hole at a distance of 60 meters to each side of the playing line, as this could cause dangerous situations if both holes are being played. At the tee and green of a hole, this margin is only 30 meters. Figure 3.1 illustrates a basic hole where the blue area surrounding the black playing line represents the safety margin. There are also safety constraints regarding obstacles and outer boundaries. The minimum distance is regarding a potential hazard. For an outer edge or obstacle consisting of a body of water or woodlands, the minimum length is 15 or 30 meters, respectively. In comparison, for houses and roads, the distance is 60 meters, as a wide golf ball can cause severe damage.

The design constraints are considered to be for the individual holes. There is a relation between the length of a hole, its par and doglegs and there are rules regarding the length to the next dogleg or the end of the hole. A par 4 hole is longer than a par 5 hole but shorter than a par 3. The par 4 and 5 holes have one and two doglegs, respectively, while the par 3 holes do not have any. These doglegs can form an angle of at most 80 degrees, but it is common that the doglegs form an angle between 10 to 40 degrees.

3.3. Soft constraints

There are also several soft constraints to help design a golf course. Some soft constraints are hole specific while others apply to the whole routing. The position of the sun at the beginning and end of the day influences the direction of the first and last holes of a golf course routing. Also, the position of the clubhouse in regards to the first, ninth, tenth and eighteenth hole matter for the distance a golfer should walk before starting or after finishing a round.

Soft constraints that apply to the whole course regard the total walking distance, variety in holes, balance in doglegs and the par count. The aim is to walk as little as possible between holes. Furthermore, there should be variety in a course, so different courses.

3.4. Subjective constraints

Next to the hard and soft constraints, there are also subjective constraints, which contain the personal opinions of the golf course architect. While one architect would like to start with a par 4 course played to the north, another wants the fifth hole to be a par 5. While all these views on designing a golf course routing are not the same, they should be considered a different list of constraints.

3.5. Physical programming

For many of the soft constraints, exact values do not really matter. There is no significant difference between a golfer walking 30 or 35 meters to the next hole. However, there is a difference between having to walk 30 or 50 meters. A solution to this problem is physical programming. By expressing the level of satisfaction for a range of values, one does not consider the actual value nor add weights to the different constraints [12]. Values for the levels of satisfaction range from the ideal value to tolerable until the value becomes unacceptable. This principle can be applied to all the soft constraints. The four different types soft classes of physical programming: smaller-is-better, larger-is-better, value-is-better and range-is-better [6]. With smaller-is-better and larger-is-better, the ideal value is either the minimum or maximum value and the lower or higher levels decrease the level of satisfaction. The total walking distance soft constraint is an example of smaller-is-better. The other two, value-is-better and range-is-better, have the ideal value in the middle. The satisfaction level decreases further away from the ideal value, both higher and lower. This applies to the total par soft constraint, which is centred around 71/72.

The objective function is used to determine the value of a solution. The objective function consists of the score of all soft constraints and subjective constraints combined. All of these constraints have been put into a model in a mathematical model in section 3.6 where they are correctly formulated.

3.6. Mathematical Model

3.6.1. Notation

Capital letters indicate a set, while lowercase versions of the same letters indicate that they are part of that set. Tuples indicating locations (x,y) can be both positive and negative.
For the directions, each direction captures 90°. As an example: north indicates all angles between 315° and 45° with 0° pointing exactly north.

3.6.2. Helper functions

$A(a,b)$: Angle function between vector a and b. $\cos\theta = \frac{a\dot{b}}{|a|*|b|}$
$D(a,b)$: Distance function between point a and b. $\sqrt{(a.x-b.x)^2+(a.y-b.y)^2}$
$Dir(h)$: $(\text{atan2}(h.g.y-h.t.y, h.g.x-h.t.x)+North)\%360$: Function to determine the direction of a hole (North/East/South/West) $Dv(a,\overrightarrow{b})$: Distance function between a point and a vector[1]
$LorR((a,b),(m,n),(x,y))$: Function to determine if a dogleg is going to the left or right based on three tuples. Right returns 1, left returns -1 [2]
$Range(S)$: Function to return the range of set S
$Sd(S)$: Function to return the standard deviation of set S
$Sv(S,t) : Range(S)/t * (1 - \frac{|Sd(S)-(Range(S)/4|}{Range(S)/4})$ Function to rate the values in a set

3.6.3. Parameters

C: location of the clubhouse (tuple (x,y))
N: number of holes (integer), most common 9 or 18
$North$: the angle in degree (integer) from (0,0) pointing towards the north
PG: set of preferred greens. (set of tuples(x, y))
O: set of obstacles including water, nature, houses and the possible outer edge of the golf course. A tuple consisting of a set of points of the outer edges and a type

3.6.4. Decision variables

D: $|D| = N$, set containing the playing direction of each hole (set of integers)
Dl: set containing the dog legs of all the holes (set of tuples)
G: $|G| = N$, set containing the location of the green of each hole (set of tuples)
P: $|P| = N$, set containing the par of each hole. $p \in 3,4,5$ (set of tuples)
Pl: $|Pl| = N$, set containing the length of the playing line of each hole (set of integers)
T: $|T| = N$, set containing the location of the tees of each hole (set of tuples)

3.6.5. Implied variables

$$dl_i(p_i) = \begin{cases} 0 & \text{if } p_i = 3 \\ 1 & \text{if } p_i = 4 \\ 2 & \text{if } p_i = 5 \end{cases}$$

$$pl_i(p_i) = \begin{cases} 100-235 & \text{if } p_i = 3 \\ 220-450 & \text{if } p_i = 4 \\ 415-620 & \text{if } p_i = 5 \end{cases}$$

3.6.6. Objective functions

1. $\max\sum_{i=1}^{SC} SC_i$: Maximize the value of all soft constraints
2. $\max\sum_{i=1}^{SubC} SubC_i$: Satisfy as many of the subjective constraints

3.6.7. Hard constraints

1. $\forall dl \in Dl\ 0 \leq A(dl) \leq 80$: Angles of the doglegs should be between 0 and 80 degrees

[1] https://stackoverflow.com/questions/849211/shortest-distance-between-a-point-and-a-line-segment
[2] https://stackoverflow.com/questions/3838319/how-can-i-check-if-a-point-is-below-a-line-or-not

2. $D(g_i, t_{i+1}) \geq 30$: Distance between the green and the next tee should be at least 30
3. $D(g_i, w) \geq 15$: Distance between the green and a body of water should be at least 15
4. $D(pl_i, b) \geq 60$: Distance between a playing line and outer boundary should be at least 60
5. $D(pl_i, n) \geq 30$: Distance between a playing line and woodlands should be at least 30
6. $D(pl_i, pl_j) \geq 60$: Distance between two playing lines should at least be 60
7. $100 \leq D(dl_{1,i}, dl_{2,i}) \leq 200$: Distance between the first and the second dog leg should be in this range
8. $180 \leq D(t_i, dl_{1,i}) \leq 250$: Distance between the tee and the first dog leg of a hole should be within this distance

3.6.8. Soft constraints

1. $D(t_1, C)$: Distance between the first tee and the club house
2. $Dir(n_1)$: Direction of the first hole
3. $Dir(n_N)$: Direction of the last hole
4. $Sv(Pl, 500)$: Variation in playing length
5. $Sv(D, 360)$: Variation in playing direction from the tee to the green site.
6. $\min \sum_{i=1}^{|Dl|} |LorR(i)$: Minimize the difference between left and right dog legs
7. $\min \sum_{i=1}^{N-1} D(g_i, t_{i+1})$: Minimize the walking distance between holes
8. $\sum_{i=1}^{|Dl|} A(i)$: Sum of all dogleg angles
9. $\sum_{i=1}^{N} p_i$: The sum of all pars should be in the indicate range
10. if $N \geq 9 \rightarrow D(g_9, C) < 150$: Distance between the ninth green and the club house
11. if $N == 18 \rightarrow D(t_{10}, C)$: Distance between the tenth tee and the club house
12. if $N == 18 \rightarrow D(g_{18}, C)$: Distance between the last green and the club house
13. if $N == 18 \rightarrow |\sum_{i=1}^{9} p_i - \sum_{i=10}^{18} p_i|$: The difference in par between the first and second 9 holes

3.6.9. Ranges soft constraints

Soft constraint	Great	Good	Pass	Below pass	Poor	Fail
(1)	< 30	30-40	40-50	50-70	70-100	> 100
(2)	225-315	0-45, 135-225,315-360		45-135		
(3)	45-135	0-45, 135-225,315-360		225-315		
(4)	1.0-0.8	0.8-0.6	.06-0.45	.45-0.2	0.2-0.1	0.1-0.0
(5)	1.0-0.8	0.8-0.6	.06-0.45	.45-0.2	0.2-0.1	0.1-0.0
(6)	0	1-2	3-4	4-5	6-7	>8
(7)	540-700	700-900	900-1200	1200-1500	1500-2000	>2000m
(8)	0-200	200-400	400-600	600-800	800-1100	>1100
(9)	71-72	70	69 or 73	68 or 64	67	< 67 or > 74
(10)	< 30	30-60	60-90	90-120	120-150	> 150
(11)	< 30	30-60	60-90	90-120	120-150	> 150
(12)	30	30-50	50-70	70-90	90-120	> 120
(13)	0-1	1-2	2-3	4	5	>= 6

Table 3.1: Ranges for the soft constraints for 18 holes

3.6.10. Subjective constraints

List similar to the soft constraints based on the preferences for each architect. This lists varies based on personal preferences, so it only contains a few examples:

1. $p_1 = 5$: The first hole should be par 5
2. $Dir(h_5)$ between 45-135: The direction of the fifth hole should be east.
3. $\max g_i \in PG \,\&\, g_i \in G$: Maximize the preferred green sites used

4

Constraint Programming

Constraint programming seems to be a viable option for the golf course routing problem, as discussed in the related work in chapter 2. This chapter will look at how the mathematical model from chapter 3.6 can be transformed into a constraint programming model and the potential challenges that come with constraint programming. Then some of the results and lessons learned will be discussed.

4.1. Transforming the model

For the implementation, MiniZinc [11] was chosen as constraint modelling language as it can be used with a wide variety of solvers. In MiniZinc, one can program a model in a language that is similar to mathematical notation [9]. It is essential to consider that many provided solvers of the MiniZinc IDE do not work well with floating point numbers. Therefore all components of the model should work with integers.

Before transforming the model, a way to represent the holes must be chosen. The holes need to be described as decision variables. A hole can be represented by its playing line, which consists of the line segment from the tee to the green and could contain bends in the form of one or two doglegs. By representing a hole using the two-dimensional Cartesian coordinate system, the location of the tee, the possible doglegs and the green can be represented as a pair of x and y coordinates. The rest of the decision variables consist of the walking distance between the holes, the length, direction and par of each individual hole.

Next, there are some parameters which are fixed for specific scenarios. The location of the clubhouse, the number of holes and the size of the area in which the golf course routing should fit. These variables can be put into a data file and loaded into the MiniZinc model. In the appendix B.1, which contains the first version of the transformed model, these parameters are put at the top of the file.

4.1.1. Constraints

Constraints are what restrict a large number of possible values for each decision variable and ensure that the desired criteria are met. Constraints come in different forms, and each has its uses to translate the hard and soft constraints from the mathematical model from chapter 3.6. A simple constraint is used to ensure the minimum walking distance between the first tee and the clubhouse by stating that the minimum distance should between these two should be at least 30 meter. While an example of more complicated constraints is ensuring that longer holes have their doglegs at the appropriate distance from the green or tee, which should be upheld for all holes.

4.1.2. Helper functions and predicates

In MiniZinc, predicates and functions can be used to formulate more complex constraints. While MiniZinc has some global constraints, which provide a wide variety of implemented predicates [9], these do not cover all the needs for the golf course routing problems. Trigonometric functions are required to calculate the directions of the hole and doglegs. Geometric functions are needed to ensure the minimum safety margins and the custom functions to represent all the objective functions and satisfaction of the soft and subjective constraints. MiniZinc itself supports the sine and cosines functions [9], but many of the available solvers do not. Therefore the trigonometric functions needed to be implemented manually. The atan2 function can be used to calculate the degree of a point in a 360-degree range. The implementation of the atan2 function requires

the inverse of the tangent function, which is not supported by most of the solvers and works with decimal numbers. A feasible solution is using a lookup table with the stored ranges for each value as a substitute for the atan2 function. This does not require any trigonometric functions and works with only integers.

For the geometric functions, functions for calculating the distance between a point and a line and the distance between two lines are required to check if the safety requirements are met. The distance function 2.1 requires both the square and square root. These functions are contained in the MiniZinc built-ins [9], but are also not supported by many of the available solvers, so they require a manual implementation. Calculating the distance between two lines, as described in function 2.2, turned out to be more complicated as it needed the dot product and conditional statements. By only being able to use integers, the function becomes more complex and contains several if-statements. The objective function partially consists of the level of satisfaction of the soft constraints, which all can be transformed into separate custom functions. All of these functions take a decision variable as input and return the corresponding level of satisfaction. One variable can then be used to express the total satisfaction of the objective function by adding the results of all the individual parts together. The implied variables of the model can be turned into MiniZinc predicates. These predicates make sure that the concept of par is used correctly in the model and that par corresponds to the right amount of doglegs for each hole.

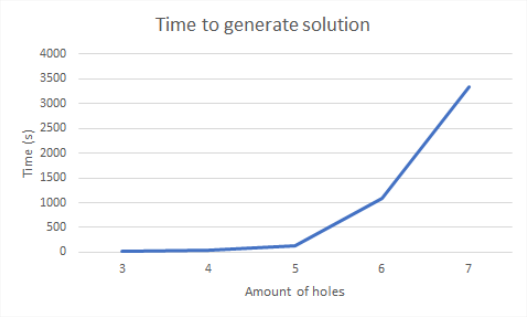


Figure 4.1: Graph showing the speed of the first model

With each part of the model adapted to the MiniZinc model, it could be run by maximizing the objective function. The output of the model can be visualized by drawing the playing lines. First, a small area and a small number of holes were used to test the model. Several different solvers were used to test the first model, but the solver Chuffed outperformed both OR-Tools and Gecode. Figure 4.1 shows the speed of the first constraint programming model using the solver Chuffed. As the graph indicates, generating golf course routings of 3 to 5 holes can be done in 2 minutes. However, from 6 holes on, the time it takes to create a solution increases substantially. The current model could not generate a routing for 8 holes after 4 hours. With the exponential time increase per adding an extra hole, the model needs some changes and optimization in order to be able to generate a golf course routing.

4.2. Challenges

The first MiniZinc model worked well for a small area with less than 6 holes, but it still took considerable time to generate solutions of larger size. Scaling up to an 18 hole golf course did not seem feasible as it took an hour before a solution for 7 holes was found, and the time to find a solution increased exponentially when increasing the number of holes. Multiple parts of the model could be improved to gain better results in a smaller time period.

4.2.1. Variables domain

The scale of the area could cause a challenge as the search space for an area of around 70 hectare is quite big for a scale in meters. This reflects on the decision variables as well as their range is also large as the model worked with the same scale. By reducing the scale of the decision variables from meter to decameter, the domain decreases by a factor 10 for each of the variables related to the playing line.

4.2.2. Functions

The distance functions are one of the slower parts of the model, as these functions require square root, square and conditional functions which are hard for solvers. The complexity of the function in combination in com-

bination with a large number of times the distance function is called in the model makes the distance between two lines slow. On a small scale with a few holes, these functions can be used, but when scaling up, the number of times each of the distance functions is called increases for each hole because each playing line needs to be checked against all the other holes. Par 4 and par 5 holes require more use as they contain extra line segments between the doglegs and green and tee. To solve this, an approximation for the safety margins between the playing lines could be used.

4.3. Split generation

Instead of only using one model, a two-model setup can be used to take away some of the challenges. By separating the model into two separate MiniZinc models, the first MiniZinc model can take care of generating the estimated playing line, while the second MiniZinc model adds the doglegs and makes minor changes to the holes while checking the safety margins. Figure 4.2 indicates this.

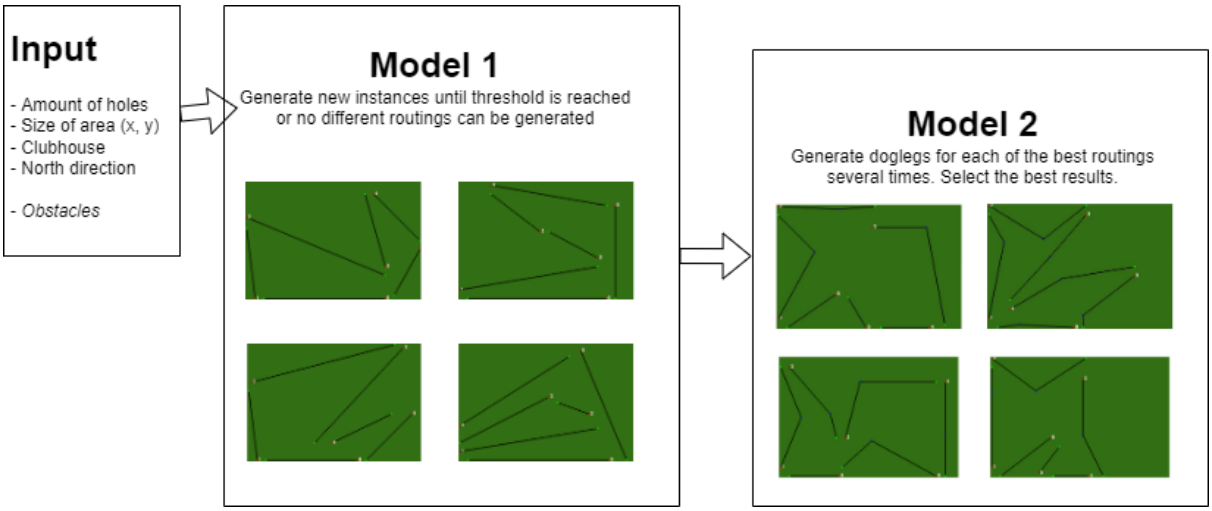


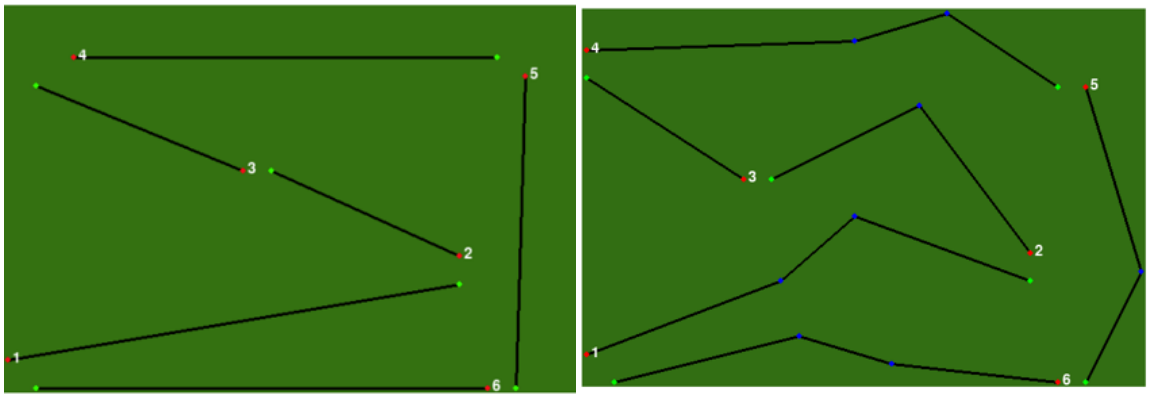
Figure 4.2: Model for two step generation

Model 1 takes care of generating the playing lines by increasing the safety margin so that there is space to add the doglegs. Instead of using all objective functions, only the walking distance and set par goal and directions were used. In order to create diversity, extra constraints were added to ensure a second solution was different from the first. The second model uses these playing lines and adds the doglegs, and makes sure that no safety constraints are broken.
Figure 4.3 shows a simple example of 6 holes. The location tee of hole 1, indicated by the red dot in the bottom left corner of image 4.3.a, and the green site, indicated by the green dot at the end of hole 6, were the only two parts that were set. The rest was generated by the constraint programming model. In image 4.3.b, the doglegs are added to the playing lines from the first model. An important observation is that the doglegs influence one another. The doglegs of the final hole are pointed upwards, reducing the space for the dogleg locations for the first hole, which therefore needs to move upwards as well. This again influences the space for the dogleg of hole 2, which influences the space of the doglegs of hole 4.

The split generation model works well on a small scale on this area and therefore, it was time to focus on the two challenges of diversity and scalability. The approach was to work on these two challenges separately and combine them afterwards. This was to see the effect of the solution to each of these challenges on the time it takes to generate a solution.

4.4. Diversity

Constraint programming models lead to the same solution when rerunning a model. This is because solvers approach the model in the same way each time. An option to tackle this problem is working with the solver configurations. MiniZinc allows the user to select several different solver options if available to that specific solver, such as random start and free search. Experimentation with both these options did not result in diverse results, so a way to ensure diversity had to be added to the model itself. A result is different if the tees and greens of the holes are not similar to a previous result. A simple measure of how similar two results are can be



(a) Results after first part

(b) Final result

Figure 4.3: Example of a 6 hole run from model 2

calculated by measuring the distance between the tees and greens from each result, so the distance between the tee of hole 1 of the first result and the second result. By adding this all together, the total distance between the two solutions is found. By adding an extra constraint after each run that the result of the next run has a minimum distance compared to the previous runs, diversity can be ensured. Adding the distance between results slows down the speed of the model, but only by a few seconds, which is acceptable.

So far, the diversity only makes sure that the results are different but does not do anything regarding the diversity in par for the generated routings. By setting a minimum or maximum for each of the possible par for the whole routing, diversity in par can be added. In the first part of appendix D, several results of setting the par are shown. Note that there is a big difference between setting only a minimum for each of the par while leaving some room for the remaining holes and setting the total of the requirements equal to the number of holes. As seen in the appendix, the par constraints help generate diversity.

Another way to add diversity is by looking at the hole-specific constraints in addition to the two previous diversity constraints. By requiring a specific hole to be a specific par or have at least a length of 300 meter, the results also become different from the current results. Next to constraints regarding the length of a hole, there can also be constraints regarding the location of a hole, for example by requiring the green site of hole 3 to be in the bottom left quarter of the available area. The problem with these types of constraints is that not all of these constraints lead to results with good routings, as indicated on the final page of appendix D. Out of the three different routings, only the third has a good routing, while for the first two the additional constraints seem to have been the cause for the bad routings as the first and second holes are a bit out of place.

Finally, changing the starting point and ending point around could also lead to more diverse results. By swapping the location of the tee of the first hole with the green of the final hole, the course goes in the opposite direction. This can create new results, but there is a danger that it creates similar holes as designed with the initial start and end of the course, as the whole can be the same only played backwards. Pages 2 and 3 of appendix D show this pitfalls.

4.5. Scalability

Besides diversity, the model also needed to be able to scale to 18 holes. Simply scaling up the number of holes from six to eighteen already showed some promise, but the quality of the generated routing leaves a lot of place for improvement. In image 4.4 the results of simply scaling up are shown. The result was generated in 5 minutes of running the improved model.

An important observation is that the holes are mostly placed at the side of the available area, and many of the holes are similar. Holes 3, 4, 5 and 6 are almost equivalent to each other. These holes are all par 3, so have a short length and all of these holes are played in the same direction. Another remark about image 4.4 is the variance in par. 9 out of the 18 holes have a par of 3, while the most common hole should be the par 4 hole.

The essential part of the scaling up is that an 18 hole golf course was generated in 5 minutes. As the diversity constraints make the whole model slower, it is critical that it still is able to create solutions. Below are three examples of an 18 hole golf course routing. In this scenario, the tees of holes 1 and 10 and the green sites of holes 9 and 18 are set. The first result in figure 4.5 has no diversity constraints and although the holes look compact, the routing of the course in figure 4.5.a is not a good one as it contains too many par 3 courses

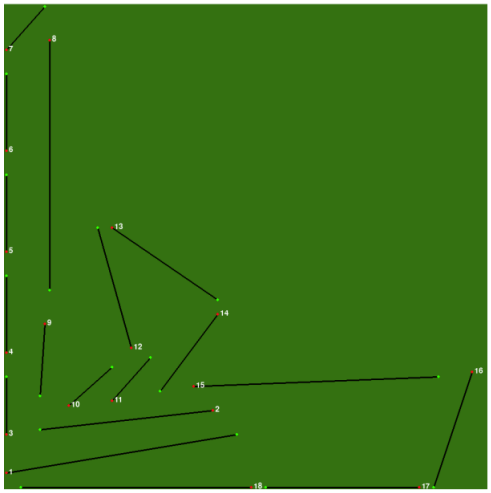
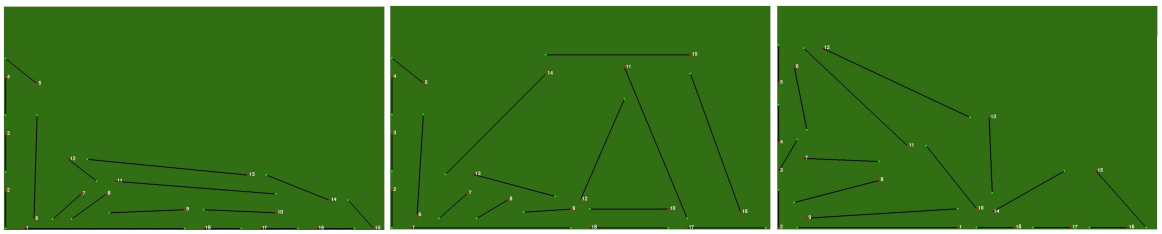


Figure 4.4: Image showing the first result of scaling up to 18 holes

and lacks diversity because many of the holes are played in the same direction and consist of similar sizes. The second generated course in figure 4.5 has constraints to the par. This results in a more balanced routing in terms of variety and usage of par. A third tactic to create diversity was used on the third image in figure 4.5. The constraint is to have the green of the third hole be located right of its tee. It does not seem to a big influence on the whole routing, but by moving the green more towards the center of the golf course, the other holes are also influenced by this and have to move around.



(a) No additional diversity constraints (b) Diversity in par (c) Diversity in par and one manual constraint

Figure 4.5: Example of a 18 hole golf course routings from model 2

While the diversity of par on a macro level is acceptable, as there is a good division between the amount of par 3, 4 and 5 holes, but this does not take into account the diversity of par on a micro level. Many of the holes following one another are in the same direction and of the same par. The desired outcome would be that the generated courses are also diverse on a micro level and have no repetition of a par 3 after a par 3 hole, and the same applies to par 5 holes. The problem is not with the difficulty of creating the constraints to prevent this but with the effect these constraints have on the runtime. Adding simple constraints that the par of the previous hole cannot be equal to the par of the next hole did not result in any generated golf courses. After 2 hours, no results were found.

Another issue with the diversity is the balance between the first nine holes and the second nine holes in par. The difference between the par of the first nine and the par of the second nine holes should be equal or differ only slightly. The generated results do not take this into consideration. Copying the same constraint for the overall par diversity and focusing it on only the first 9 holes also did not create any results, as after 2 hours, again, no results were found.

The diversity of the par is not the only problem with the current implementation, as the second part of the model creates problems as well. The issue is with the distance between the playing lines. Some holes need to have space for doglegs, but with the current set-up, there is not always enough room. The minimum distance of 60 meter between two playing lines is used for the first part of the model, and in all of the results shown in figure 4.5, the second part of the model could not find a solution due to the lack of space around the playing lines. However, by increasing the minimum distance between playing length, the total volume for the whole golf course increase a lot. The current area would need to be one-fifth larger in order to generate a solution

with enough space for the doglegs.

Finally, another issue would be the manoeuvring of holes around an obstacle or differently shaped areas. Currently, the area does not include any bends or obstacles. With the chosen two-step approach, a hole cannot move around a corner of an obstacle as it is only considered as a straight line. Moving around corners and obstacles would require manual constraints per area in order to deal with them, as the current strategy is not able to generate golf courses that can work with obstacles or with the bending of holes.

4.6. Conclusion of constraint programming

The split generation strategy worked significantly better than the first model. It could generate golf courses of 18 holes and could do so in a reasonable time. However, the scaling up brought additional challenges in the form of diversity. Some diversity could be enforced by adding additional constraints to create diversity in par and between generated results. However, there remained still some diversity issues on a micro level, which could not be solved by adding additional constraints as the model became too complex to generate solutions. Finally, the model itself contained a flaw. By only working with straight lines, different area shapes and additional obstacles cannot be used as straight lines cannot move around those effectively. This all makes constraint programming not a feasible approach to the golf course routing problem.

5

Heuristics approaches

In this chapter, two of the proposed solutions to the golf course routing problem will be discussed. First, a new way to formulate will be presented based on the lessons learned from the constraint programming solution, followed by the building blocks required to build the solutions. Finally, a model for the random search algorithm and the genetic algorithm will be discussed.

5.1. New approach

Looking back at the constraint programming solution from the previous chapter, a different approach was needed in order to generate golf course routings. While the constraint programming approach got many parts right about the generation process, some parts could use improvement. The approach of first creating the playing line and later adding the doglegs is incorrect to use, so a new approach needs to be created.

With the constraint programming approach, the distance between the playing lines was checked to be of at least the minimum safety margin. While this works, there is another way to ensure a minimum distance between the holes. By creating a polygon around the playing lines of the individual holes at a distance of 30 meter, the safety constraints can be satisfied by checking if any of the safety polygons intersect with one another. So instead of checking that the distance between the holes is at least 60 meter at each part of the hole, one can check if the bounding box of two holes intersects. The safety margin between two holes is at a minimum 60 meter if the bounding boxes do not intersect, as both holes have the bounding box at a distance of 30 meter. In figure 5.1 the bounding box is visualized. The playing line is surrounded by the safety margin indicated by the blue-colored space. This way of checking the safety margins is used by both of the created algorithms.

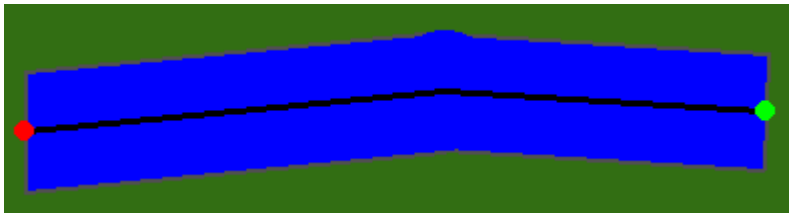


Figure 5.1: Example of a par 4 hole. Played from the tee (red dot) to the green site (green dot) with a dogleg with the safety margins in blue

5.2. Building blocks

After setting on a new approach, the components of the approach needed to be implemented. First, the objects and functions needed to be defined. On a high-level view, a golf course consists of four different components. The available area in which the golf course should fit, the individual holes, potential obstacles and the clubhouse.

The area object contains a polygon to indicate the outer edges of the available space. The area contains all the other components of a golf course. The parts that are not available inside the area are defined as

obstacles. Similar to the area object, the obstacles are also defined by a polygon defining the outer edges of the space that cannot be occupied by holes. The holes are more interesting to define as using one polygon is not possible for all hole types. The simplest hole, a par 3, can still be defined by only using one polygon. In order to create this polygon, two perpendicular lines can be drawn from both the green and the tee of the hole 30 meters in both directions. The endpoints of these lines can be connected to the other side with a line parallel to the playing line. However, holes with a dogleg require additional steps to satisfy the safety constraints. By simply creating an additional polygon for each additional part of the hole, not all the safety margins can be guaranteed. In figure 5.2 a par 5 hole is shown with some gaps in the safety margin at the location of the start of the doglegs. By adding a circle with a radius of 30 meters, these gaps can be properly filled. Finally the clubhouse works the same as an obstacle and is also defined by a polygon of the outer edges of the clubhouse.

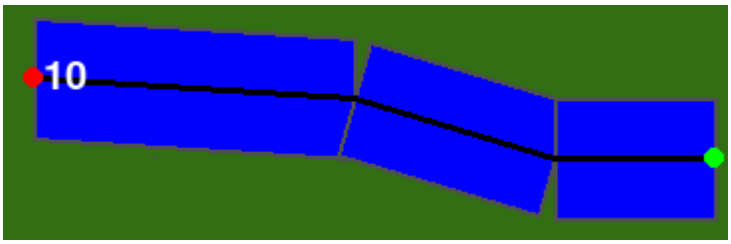


Figure 5.2: Par 5 hole with just polygons

Function 2.3 from the related work chapter can again be used to check if the bounding boxes of two holes intersect. However, only checking if any of the lines of the bounding boxes intersect is not enough. A hole can be parallel to another hole and have a side of its bounding box overlap with one of the other holes. By keeping track of how often a side touches but does not crosses a side of the safety box of another hole, these situations can be allowed. In figure 5.3, two par 3 holes are shown, and the safety margins of both holes overlap with each other. As long as they do not overlap, this is allowed.

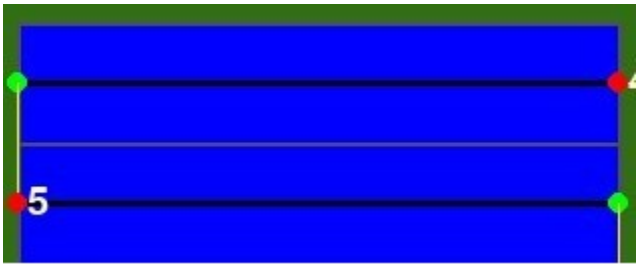


Figure 5.3: Example of 2 par 3 holes being side by side with the overlapping edge of the safety margin

5.3. Random search

In many of the related work from chapter 2, a generative model was used to generate candidate solutions to their respective problems. These candidate solutions could either serve as real solutions or be improved by another algorithm. Before creating an algorithm that can generate golf course routings, a model is needed to formulate how the generation process works.

In figure 5.4 the chosen model for the generation process is shown. Similarly to the constraint programming model, the model for the area approach is again split into two parts.

The decision to divide the generation process into designing the first 9 holes in the first part and after creating the second 9 holes of an 18 hole golf course was made with the reasoning the routing for the first nine holes has an influence on the generation process of the second 9 holes. The most important effect of the first nine holes comes from the space taken up by them, which reduces the space for the second nine holes. Furthermore, the filtering of the best results after generating the first nine holes ensures that no lousy candidate solutions are used in the second part of the model. Similarly to the constraint programming model, there are again several inputs for the model the same: the area in which the golf course should fit, obstacles which should be avoided and the clubhouse. Additional inputs come in the form of the design of the first, ninth,

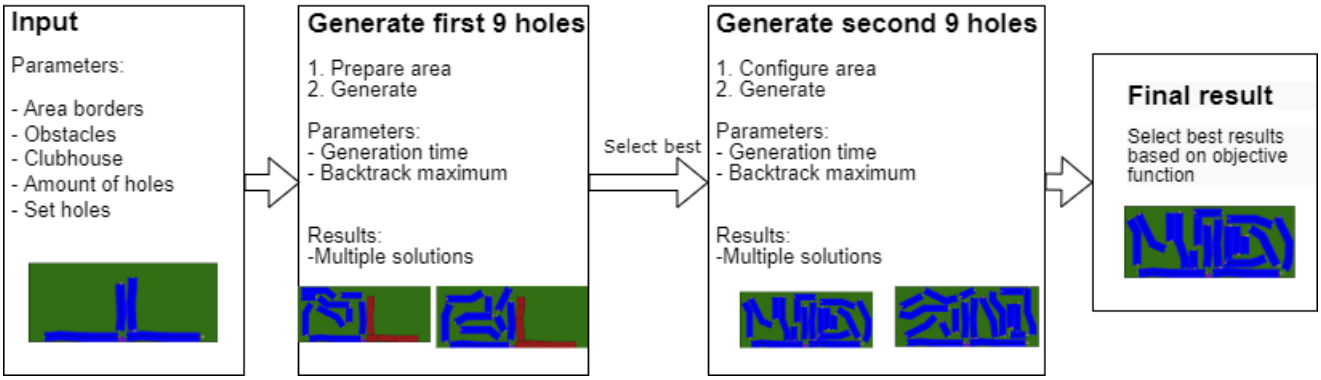


Figure 5.4: Model for two-step generation

tenth and eighteenth holes. While this takes away flexibility from the algorithm, it handles the constraint and preference regarding the direction of these holes, the walking distance to the clubhouse, as well as always making sure that enough space is left for these holes.

With all the components complete and a model in place, the next step is to look at the generation algorithm. In the pseudocode from algorithm 1, a general outline of the generation process is shown. With this approach, the golf course routings are generated in chronological order, starting with the green of the first hole, which is set and working towards the tee of the ninth hole. This strategy is then repeated again for the second half of the golf course.

Algorithm 1 Algorithm for generating golf course routings

```
input area object A
while A not having all the holes do
    p ← possible holes to chose
    if p then                                   ▷ If there are any holes found
        select the best hole from p and add to A
    else
        backtrack
    end if
end while
```

After having the pseudocode as a plan to create the generation algorithm, it is time to look deeper into each of the components: the designing of the holes, the backtracking and selecting the best solutions.

5.3.1. Designing holes

The process of generating the first nine holes and the process of generating the second nine holes work similarly. They both start with a set hole in the form of the first and tenth hole and a target hole in the form of the ninth and eighteenth holes. Between these holes, the generation algorithm should design the remaining holes.

The first step in the process of generating possible holes consists of finding a suitable starting location for the next hole. The walking distance between the green of the previous hole and the start of the next hole should be low, and therefore only starting locations within 60 meter are considered as candidate starting spots. 60 meter is chosen because then a hole can be played in the opposite direction compared to the previous hole, as shown in figure 5.3.

In the initial version of the generation, algorithm doglegs were not considered yet and each hole consisted only of a straight playing line. Although it was concluded in the constraint programming chapter that this is not the desired approach, it was used in order to test for speed of the generation process and have indication of the diversity. When the initial version proves its value, the doglegs will be added in this step. Not all the ranges for the holes were considered. As there is little difference in a par 3 hole of 130 or 140 meters, only a selection of the range of a par 3 hole is used. The same principle applies to par 4 and 5 holes. Similarly to the length of the holes, the direction of the hole also makes use of a predefined range of possible angles the hole

can have.

After designing all possible holes, the holes are rated based on parts of the objective function: the direction, par and when nearing the next set hole, the distance to that set hole. For the latter one, the last 3 holes before a targeted hole should go in the general direction of the tee of the targeted hole otherwise, the walking distance to the tee of the targeted hole from the green of the previous hole will be too big. The hole with the highest score is attempted and added to current holes and the design process starts again from the green of the last added hole.

5.3.2. Backtracking

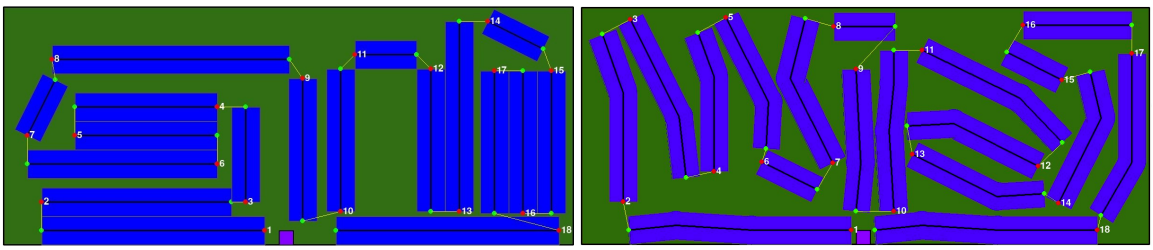
While the generation works, selecting the best hole on an individual basis does not guarantee a good solution. One can visualize the possibilities of golf course routing as a tree in which the nodes represent the possible holes in the routing. The root node consists of the first hole. The children of each node consist of all the possible holes that can be chosen in each step. As one goes down the tree, there can come nodes which do not have any children, which means that there are no more candidate holes from that hole forward. There could be no more space for the next hole, or the target hole cannot be reached anymore. In that case, backtracking can be used to traverse the tree back up to the parent and try a different hole.

The way to implement backtracking is to deal with the two situations that it can get stuck. In the first case, when there is not enough space for the next hole, the program backtracks to the hole before the last one. Similar candidate holes with the same direction and similar end location are pruned from the possibilities before a new hole from the remaining candidate holes are chosen. The generation process also keeps track of the number of times it backtracks to a specific hole. If this happens too often, then that hole is also removed, and the program resumes from the hole before.

In the second case, when the targeted hole is not within range, the program makes use of back jumping[2] and backtracks several holes to try a different configuration to get close enough to the targeted hole. Due to the green of the last hole being too far from the tee of the targeted hole, the program needs to back jump. There is little use only backtracking one hole, as the best candidate hole was chosen and could not get close enough to the targeted hole. By jumping back several holes, a new branch of the tree can be tried. Similarly to backtracking, comparable holes are pruned from the candidate solutions again.

5.3.3. Selecting the best solutions

When the generation algorithm creates a good solution to the first or second nine holes, the solutions are stored with their score based on the objective function. After the first part of the model, not all solutions are used for the second part, as that would take up a lot of time. Instead, the solutions with the highest scores are selected to be used for generating the second nine holes. These solutions need to be different from one another, and therefore the best solutions are chosen for the second part, and similar solutions are then pruned from the remaining solutions in order to have diversity between the first nine holes. This process is repeated up to the desired amount of solution for running the second part. When also the second part of the model is done, the final solution is scored by the objective function again, and the results with the highest score are selected similarly to the selection after nine holes to create a set of diverse final solutions.



(a) Generative algorithm without doglegs

(b) Generative algorithm with doglegs

Figure 5.5: Example of the two versions of the generative algorithm

In figure 5.5 the result of the generative algorithm without doglegs can be found. After verifying that it worked, it was time to add the doglegs. The difference between par 3 and par 4 or 5 holes is the addition of doglegs. This means that next to having different lengths for each segment, there are also different possible directions for each dogleg. Although the maximum degree a dogleg can be is 80 degrees, this is uncommon and unrealistic. Therefore a more realistic range between 10 and 40 degrees was chosen for a dogleg. This

dogleg can be both to the left and to the right of the playing lineup to the dogleg. This increases the number of possibilities drastically. In the case that there are 2 ranges for the distance between the tee and the dogleg and 2 possible values for the remaining part between the dogleg and the green. Without using the angles for the dogleg, there would be 4 different possibilities total. Adding 2 values for the doglegs to the left, which also means to the right, creates a total of 16 options. This again increases for each direction the initial hole can be played in and by adding a second dogleg. This results in this type of generation being a lot slower than the first version, but the results look a lot better, as can be seen in image 5.5. The difference between the results will be analyzed in the next chapter.

5.4. Genetic programming

Another viable heuristic technique which was discussed in the related work from chapter 2 is genetic programming. Similarly to the random search algorithm, a genetic algorithm does not guarantee an optimal solution. Instead, the genetic algorithm will try to find better solutions from a set of starting solutions. This initial population will be improved with regard to the objective function [10]. The objective function is the same one used for the random search algorithm. A genetic program uses 2 different actions to generate new candidate solutions, mutation and crossover. In model 5.6 an overview is given of the workings of a generative algorithm for the golf course routing problem.

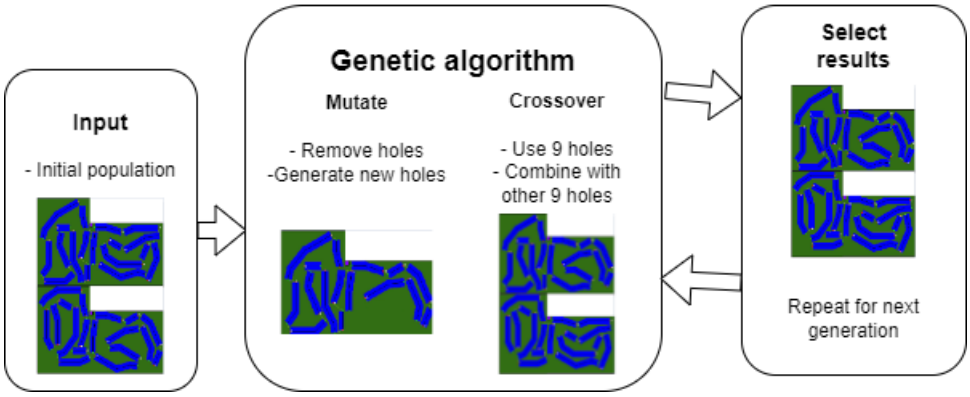


Figure 5.6: Model for a genetic algorithm

As one can see from the model in figure 5.6, the genetic algorithm will start with an initial population. This initial population can be provided by the random search algorithm from the previous section. Both versions, with and without doglegs can be used in order to create better solutions. The genetic algorithm will use two actions in order to create a better generation. These actions are the mutation action and the crossover action. The two actions from the literature do not directly translate to the golf course routing problem and, therefor require some modifications. With the mutation action, one cannot simply mutate a selection of holes with random other ones as these random holes do not necessarily respect the safety margins of the other holes. The same problem occurs when using the crossover action to combine two generated golf course routings. By modifying these actions to fit the golf course routing problem, a genetic program can be created to create new golf course routings. The only part of the golf course that often is exchanged is the first nine or second nine holes, therefor the crossover action uses the first nine holes from one of the candidate solutions and combines this with the second nine from another solution. The mutation can be done by removing a part of the golf course routing and using the generative algorithm again to find a solution to removed holes with the first hole after serving as a target hole. After creating several new candidate solutions, the best solutions are selected to serve as the population for the next generation of the genetic algorithm. After a predetermined amount of generations, the genetic algorithm will stop and put out the best solutions found after the final generation.

6

Results

In this chapter, the designed algorithm explained in the previous chapter will be tested. First, the test setup will be explained, followed by the analysis of individual approaches, followed by a comparison between the different proposed solutions.

6.1. Test setup

Before each of the algorithms could be tested, a diverse test setup was needed. Frank Pont from Infinite Variety Golf Design suggested four different types of areas with each having a different shape and different total size. Since most golf courses are played on sites between 60 and 70 hectare, the four different test areas should also be within this range or close to this range.

(a) Rectangular area (b) Area in shape of the letter L (c) Area in shape of the letter T (d) Area in shape of the letter O

Figure 6.1: The different areas

Figure 6.1 contains an overview of all of the four different shapes used for the experiments. The rectangle has a total area of 60 hectare as it is the simplest and does not require any turns. Both the L-shaped and T-shaped areas are slightly more challenging and therefore have a higher available space, 70 and 68,5 hectare, respectively. The O-shape is the most difficult shape and therefore the largest available space, namely 77 hectare. Since the O-shaped area is the most difficult, the amount of available land was increased a little in order to be able to generate solutions.

Each of the algorithms was run on an Intel Core i7-7700HQ CPU with the constraint programming solution and the random search algorithm running on five different cores in parallel.

6.2. Constraint programming model

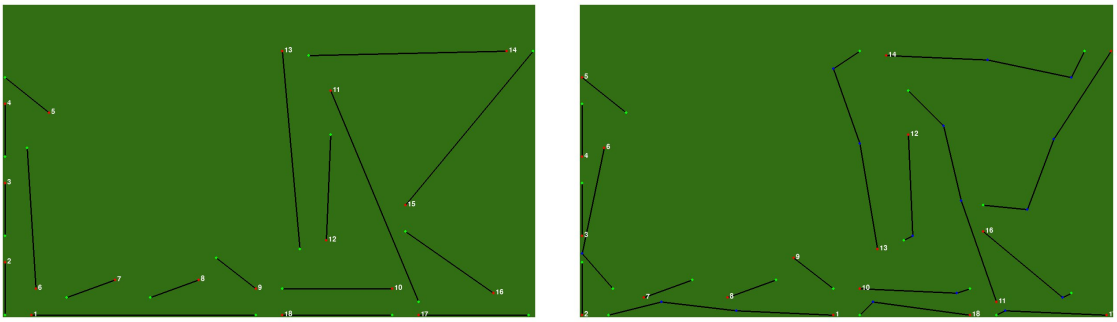
The constraint programming solution from chapter 4 was only tested on the rectangular-shaped area. There are some limitations with this implementation, such as not being able to deal with obstacles. Formulating the other 3 areas by adding additional constraints to restrict the parts where no hole could go did not yield any solutions from the solvers.

For the rectangular-shaped area, two different experiments were run to show the effectiveness and flaws of the constraint programming approach for the golf course routing problem. The constraint programming

approach is a 2-staged approach in which the first part of the model generates for 18 holes the playing lines without doglegs, and in the second part, the doglegs are added. The first part of the model is solved using the solver Gecode and uses five cores to find solutions. The model tries to optimize the solution for 30 minutes, and after 30 minutes, the best solution is stored to be later used by the second model. The first part runs again with the additional constraint that the next solution should be different from the first. This process repeats until either 15 solutions are found or no new solution cannot be found.

The second part of the constraint programming approach uses the solver Chuffed to add the doglegs. This is not always possible, but each input is tested. A maximum of 5 different solutions can be created this way with a time limit of 10 minutes per solution. A maximum number for the solutions is set in order to be efficient. The general thought is that the first solution will be the best according to the objective function, and when the model runs again, the solutions need to differ from the previous solution, which often results in a worse score based on the objective function.

In the first series of tests, the distance for the distance approximation was set at 6. This means that there is limited space between the holes, but there is some difficulty with placing the doglegs. There is not always enough space left to find a solution with doglegs for every solution from the first part. It is not a matter of runtime, as after a short time, the solver indicates that no solution can be found with the given input. This means that from the 15 different results found, and only 5 had enough room for doglegs. Below in figure 6.2 are two examples of the results found after the first part.



(a) Constraint programming solution without doglegs

(b) Constraint programming solution with doglegs

Figure 6.2: Constraint programming solution with a minimum distance of 60 meter

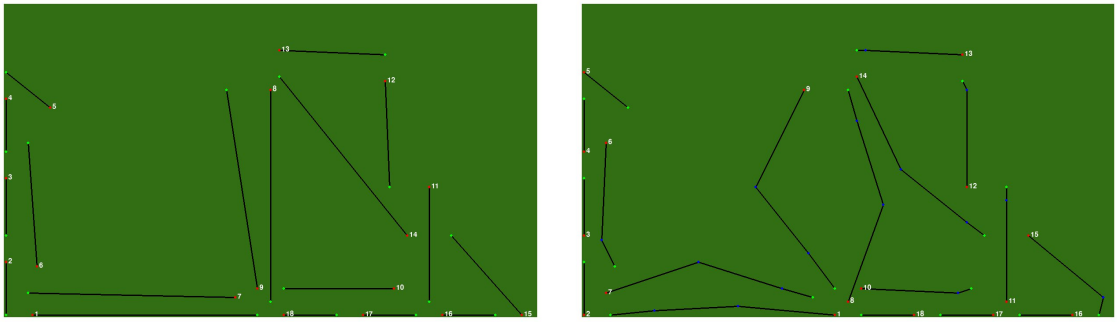
As you can see, although the safety margins are mostly satisfied in both of the results in figure 6.2, there are some violations as the approximation for the distance between playing lines is not capable of checking each point on a line. Furthermore, there are some issues with the doglegs, as can be seen in figure 6.2b. Some dogleg angles are alright, like the doglegs of hole 1 and hole 11, but many contain a sharp angle of more than 80 degrees, which is not allowed according to the mathematical model from chapter 3.

Working with angles these angles in a constraint programming model is difficult as the default sin, and cosine functions are not supported. The safety violations could potentially be solved by enlarging the minimum distance in the distance approximation function. By increasing this to 90 meter, there should be more room for doglegs and fewer or no safety violations. In figure 6.3 below the best result is shown. The image on the left contains the solution generated by the first part with the solver Gecode, while the final result used the input from the first solution to add the doglegs using the solver Chuffed.

The results from figure 6.3 are much better than the results from figure 6.2. The extra space between the holes allows for better placement of the doglegs and the holes being much less compressed together in the same area. Although parts are very similar between the two figures, such as the first three holes, the big differences are in the addition of the doglegs, whereas the version with a little more space has nicer looking and less extreme angled doglegs.

	Average	Highest	Amount without DL	Amount with DL
Margin 6	321.98	332.94	15 (5)	17
Margin 9	321.91	393.72	4 (3)	10

Table 6.1: Results from the constraint programming solutions



(a) Constraint programming solution without doglegs

(b) Constraint programming solution with doglegs

Figure 6.3: Constraint programming solution with a minimum distance of 90 meter

In table 6.1 the results of the two experiments for the constraint programming approach are shown. Although the average for both versions is almost identical, the best result for the run with a minimum distance of 90 meter is significantly larger than the one from the 60 meter version. The similarity in the average score can be explained when looking at the setup. The 60 meter version also contained constraints regarding the par of the holes, having at least 5 of each. However, this was not possible for the 60 meter version as it yielded no results, also not when increasing the run time by multiple hours. The margin 6 scores on average higher on all the par-related constraints due to the additional constraint, but when the margin 9 version scores decent on par, it outperforms the other version because, on average, it scores higher on the other parts of the objective function.

6.3. Random search algorithm

6.3.1. Problems halfway results

When running the first part of the random search algorithm, there was an issue with the results after the first nine holes. A significant portion of the results was not usable as there were several problems with the hallway results. These problems were present in roughly 40% of the results over running the algorithm 10 times. Two of the problems with these results are shown in image 6.4

The first problem regards the start of the second part of the algorithm, as there is a problem with the possible starting locations for the eleventh hole. The tenth hole is visualized as the red hole in the middle of figure 6.4, with the green being on the right. The only possible start location for the tee of the eleventh hole is just above the tenth hole. Furthermore, the eleventh hole would have many options as many of the directions are blocked by the routing of the first nine holes.

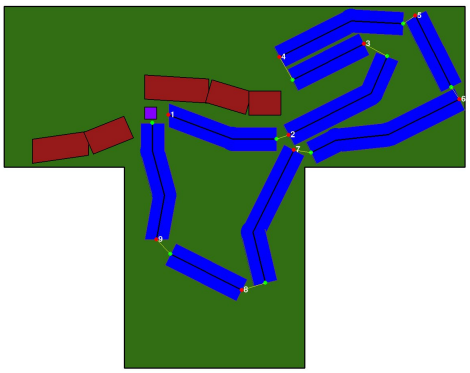


Figure 6.4: Example of an unusable routing for the first 9 holes

The second problem with the routing from figure 6.4 is the use of space. The first nine holes use a lot of space which should be used by the second nine holes. There is still some space between holes 10 and 18 in

the top left. However, this is not enough to place 9 holes. The lower part of the T-shaped area also has a room available, but this bottom part cannot be properly reached. This means that there will not be any complete solutions when using these types of routings for the first nine holes.

In order to prevent these two problems from occurring after generating the first nine holes, an additional obstacle is used to prevent this. A temporary obstacle is added around the green of the tenth hole when generating the first nine holes. This prevents taking up many of the possible tee locations while also allowing for more space around the tenth hole for more flexibility for the second nine holes. Figure 6.5 depicts the additional obstacle after the tenth hole. The obstacle is also added at the tee of the eighteenth hole in order to have enough space to reach it.

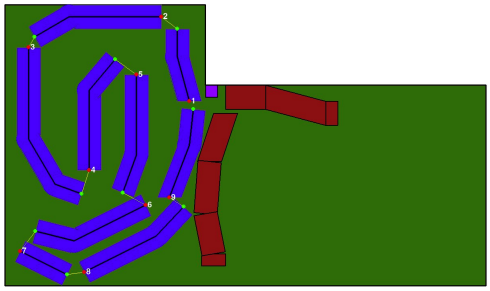


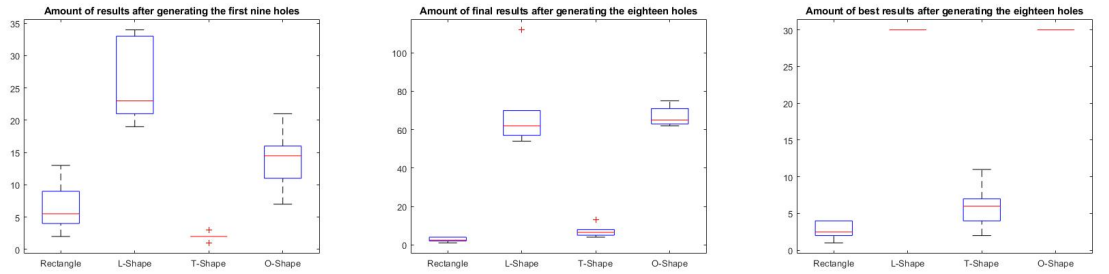
Figure 6.5: Example of using an extra obstacle after the tenth hole

6.3.2. Results random search algorithm

With the addition of the temporary obstacles, each of the areas was run with the random search algorithm from chapter 5. The time limit for both the first and second parts of the model consisted of one hour. After one hour, the best five halfway results were selected and used for the second part of the model, which each also ran for one hour. In appendix C one example run is shown of the L-shaped area.

	Halfway average	Halfway SD	Final average	Final SD	Best	Best SD
Rectangle	6,5	3,17	2,67	1	2,67	1
L-shape	26,4	5,68	69,4	17,04	30	0
T-shape	2	0,5	7,5	3	6	3
O-shape	14	3,67	66,83	4,11	30	0

Table 6.2: Table with the results of 10 runs for each shape



(a) Box plot of the number of diverse results (b) Box plot of the number of final results (c) Box plot of the number of diverse results

Figure 6.6: Box plots visualizing the number of results found

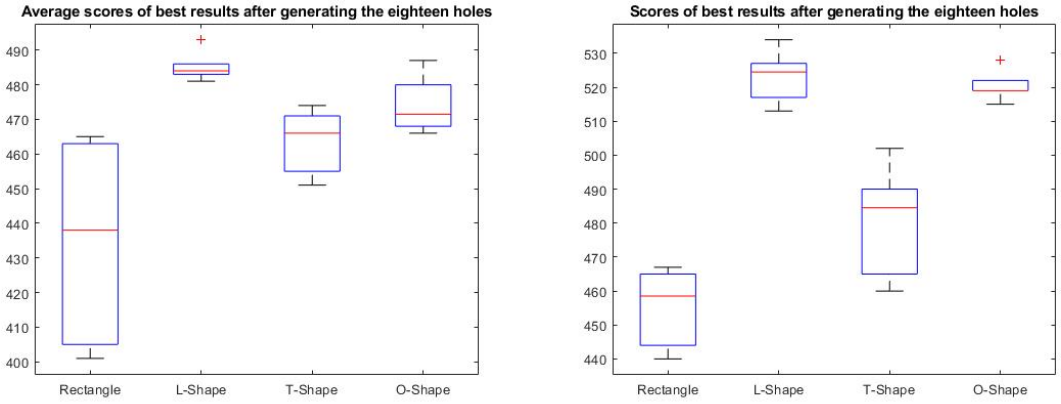
Amount of results found In table 6.2 the amount of results found by the algorithm is shown. The first column shows the number of results found for the first 9 holes and the second column indicates the standard deviation of the number of results found. The next two columns show this for the final results, which are all the generated solutions consisting of 18 holes. The final two columns show this for the best results. The best results are the 30 best and most diverse results of the final routings. When there are more results found, only

the 30 results with the highest scores are part of the best. Furthermore, these results need to vary in layout, so when multiple results have a similar routing, only the one with the highest score can be selected for the best results, while the others are not considered. For the T-shaped area, this occurred as there were on average more final routings were found than were selected for the best routings.

Looking at the results in table 6.2, there are big differences between different areas. The rectangular area has on average more results for the first 9 holes compared to the average amount of final results. This is the opposite for the T-shaped area, where there are on average only 2 routings found for the first 9 holes, while there are more final routings. This indicates which parts of an area are difficult to solve. Comparing both the T-shaped area and the rectangular area to the L-shaped area, the results show that the first two areas are more complex to solve than the L-shaped area. Reasons for this are the total amount of available space and the figure of the free space. The rectangular area is significantly smaller than the L-shaped one, while the T-shaped area has a more complex area for both the first and second nine holes compared to the L-shaped area.

	Average score	SD	Highest score average	Highest score SD	Highest
Rectangle	435,43	21,28	455,75	9,25	465,04
L	485,66	3,28	523,70	6,75	527,91
T	464,93	8,95	484,43	15,96	502,23
O	474,56	6,22	420,94	3,02	528,59

Table 6.3: Table with the scores of 10 runs for each shape



(a) Box plot visualising the average score of the final results

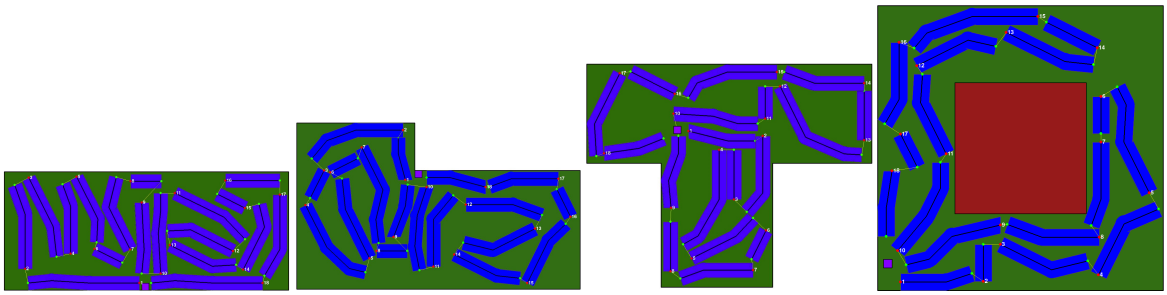
(b) Box plot visualising the highest score of the final results

Figure 6.7: Box plots visualizing the score of the results found

Score of results In table 6.3 the average scores and high scores for each of the different areas are presented. On average the scores of the results from the L-shaped are the highest, followed by the T-shaped area, while the results from the rectangular area have the lowest score on average. This is also the case for the highest scoring routing in each run, as the average high score from the L-shaped area is significantly higher compared to the other shapes. The standard deviation is, of course, higher for the rectangular and T-shaped areas as there are on average notably fewer results were found compared to the L-shaped area, as shown in table 6.2. Figure 6.8 shows the best results for each of the areas.

Impact of pre-set holes As both of the previous subsections have shown, the number of results and the quality vary a lot between the different shaped areas. Two of the discussed issues are part of this large deviation. The size of the available area and the complexity of the shape is not the only reason for this difference. The pre-set components, namely the first, ninth, tenth and eighteen hole as well as the clubhouse also have an impact on the number of solutions found. In both the rectangular area and the L-shaped area, a clear division can be used to separate the first and second part of the routing, while in the T-shaped area this is

much harder. Furthermore changing one of the set holes to a different par can make the difference between finding a lot of results or non at all.



(a) Best rectangular area (b) Best L-shaped area (c) Best T-shaped area (d) Best O-shaped area

Figure 6.8: The best result for each of the areas

More time to generate While the results so far are good, a potential improvement could be found in adding additional time for two parts of the model, so first, a higher time limit for generating the first nine holes and secondly, a higher time limit for generating the final solutions from each of the candidate solutions provided by the first part of the random search algorithm. In order to see if the increased time had a positive effect on the quality of the generated solutions, the extended version was used on two of the area: the L-shaped area and the T-shaped area. The L-shaped area already had lots of solutions as seen in table 6.2, so maybe the additional time would create even better results. The T-shaped area lacked solutions to the first nine holes, so additional time could lead to more results. In table 6.4 the results of this version are shown.

	Halfway	Final	Best	Score	Highest
L 1 hour	26,4	70	30	485,66	527,91
L 2 hour	34	125	30	486,31	515,19
T 1 hour	2	7,5	6	464,93	502,23
T 2 hour	3.5	15	13	466,92	501,78

Table 6.4: Results from the random search with different times

Looking at the results for the L-shaped area, the number of results found increased by a lot compared to the 1 hour runs, but the quality of the results is similar. The highest recorded score of a solution differs a little in favour of the 1 hour run, but the average is just higher for the long run. Overall the difference between running 1 hour or 2 hours does not seem significant for the L-shaped area. For the T-shaped area, similar observations can be made. Both the average and highest recorded scores are similar between the two different run times. There are slightly more results, but not significantly. Therefore a time of 1 hour per part seems better as it results in a better score to time ratio.

6.3.3. Genetic algorithm

Starting population with no doglegs The idea to use the version The genetic algorithm was first tested with the initial version of the random search algorithm. This was the version without any doglegs. The concept to use the version came from the fact that the initial population could be generated a lot faster compared to the final version of the random search algorithm and that the lack of doglegs could be solved by the genetic approach. The idea was the genetic algorithm would improve the initial population and, with each generation, add some doglegs to the par 4 and 5 holes. Below in image 6.9 one of the inputs without doglegs for the genetic algorithm is shown.

As is clear in the picture, there is little room to add doglegs. Even when removing part of the holes, there is still not enough room to generate new holes with doglegs. After 15 generations, which took over 7 hours, this kind of input for the genetic algorithm was discarded and only a starting population consisting of the results from the random search algorithm were considered.

Final version genetic algorithm The genetic algorithm was tested with inputs from some of the best results for both the L-shaped area and the rectangular area. Each run started with 10 good results as the initial

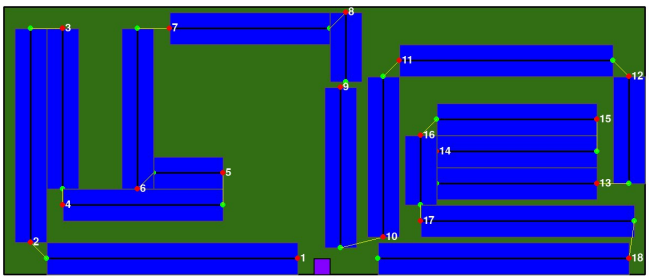


Figure 6.9: Example of one routing for the starting population of the genetic algorithm

population. There are several parameters to the implemented version of the genetic algorithm, such as the ratio between the mutation action and the crossover action and the number of generations. The size for each generation was kept equal to the input at 10. The ratio between the two actions was set 3 to 2 in favour of the mutation action, but over the course of the generations, this ratio becomes equal. Over the course of 30 generations, each solution was modified 10 times each and after all the actions, the best diverse results were selected for the next generation. The runs took 15 hours each to run. The main reason for this is the mutation action as it has to find a new solution with regard to the removed holes and this does not always happen within a reasonable time. Below in table 6.5 the results from the rectangular and L-shaped areas are shown.

	Average	Highest
Initial Rectangular	425,26	462,51
GA Rectangular	459,48	471,57
Initial L-shaped	476,93	510,02
GA L-shaped	484,56	512,23

Table 6.5: Table showing the results for the genetic algorithm

The results from the genetic algorithm are better than the initial population. The average improves quite a lot for both the rectangular-shaped area as for the L-shaped area. While the scores for the L-shaped area were already quite good to start with, the scores of the rectangular area were a lot worse. This means that by using the genetic algorithm after the random search algorithm, the results get better. However, this comes at the cost of an additional 15 hours. Looking at some of the final solutions provided by the genetic algorithm, a problem can be observed. The mutation action has trouble creating new holes that connect well to the existing holes. In image 6.10 an example of this is shown.

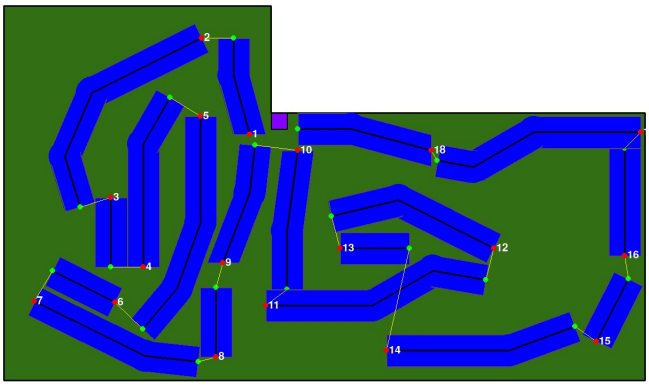


Figure 6.10: Example of the problem with the mutation action

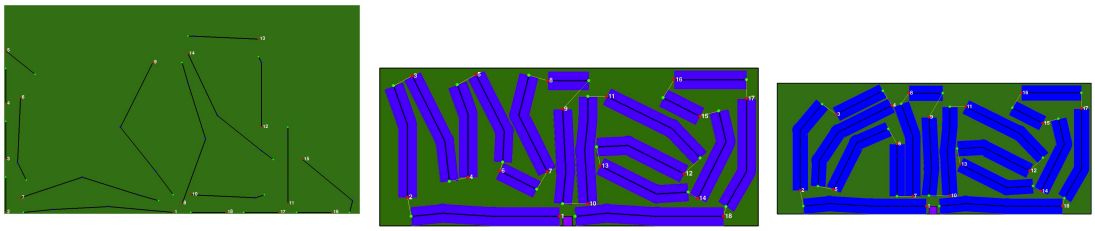
The two holes that were removed were hole 12 and hole 13, but when trying to generate a new solution, it is hard to get close to hole 14. This solution ends as close as it can get to hole 14 without making use of

the space between hole 11 and 14, but this creates two problems: 1. the walking route crosses a hole, which should not be allowed and 2. the walking distance increases a lot making this solution worse than the solution it started from.

The mutation action takes the longest time as the generation process is not as fast as simply swapping the holes in the crossover action. Furthermore, the mutation could also worsen the solution a lot, which is not desirable. Therefore most solutions found after the generation did not include mutations but were created using mostly the crossover action.

6.3.4. Comparison between the three approaches

The scores from the previous results section do not provide much context without comparing the results of the three approaches together. Since the constraint programming approach only can work with the rectangular area, the three approaches will be compared to each other on that area. In figure 6.11 the best result are shown and the average scores and high scores are presented in table 6.6.



(a) Best constraint programming solution (b) Best random search solution (c) Best genetic solution

Figure 6.11: Best solutions from the three approaches

	Average	Highest
Constraint programming	321	393
Random search	435	465
Genetic algorithm	459	471

Table 6.6: Table showing the results for the three approaches

It is clear that the results from the genetic algorithm and random search are a lot better than the solutions found by the constraint programming approach. The constraint programming lacks the ability to adapt to additional constraints regarding doglegs and with the imperfect distance approximations, the safety margins are not always ensured. This makes it the worst of the three approaches. The other two are a lot closer together, which can also be seen in figure 6.11, where the two solutions share the same second half. The genetic algorithm outperforms the random search on both the average and best solutions, but the best solutions are very comparable. The genetic algorithm just produces overall a better set of solutions. It is important to note that the genetic algorithm cannot run on its own without first running the random search algorithm to generate the initial population. When looking at the run time of both the algorithms, the random search seems preferable as the final results are produced after six hours compared to the additional 15 hours used by the genetic algorithm. Instead of running the genetic algorithm after the random search, the random search could also be run twice again to generate new possibilities in less time than the genetic algorithm would take. Considering the results and runtime, the random search algorithm is the best solution to the golf course routing problem.

7

Conclusion & Recommendations

The goal of this thesis was to find a way to automatically generate a diverse set of golf course routings. In order to find a solution to this problem, the problem was divided into smaller parts using one research question and three sub-questions in the introduction. In this conclusion, those questions will be answered. First, the sub-questions will be answered and finally, the main research question will be answered.

Which techniques can be used to effectively design golf course routings?
The first step in finding good approaches for the golf course routing problem was to look at related work, which was presented in chapter 2. While there were many problems which showed some similarities to the golf course routing problem, like city and building design, none of the approaches used in the related work translates to the golf course routing problem. However, some of the techniques found, like a random search algorithm, a genetic algorithm and a constraint programming solution can be used to generate golf course routings.

What are the constraints for a golf course routing?
Before any of the techniques can be used to solve the golf course routing problem, all the requirements and rules for a golf course need to be defined. Infinite Variety Golf Design provided a set of guidelines which were turned into a mathematical model formulating all the constraints in chapter 3. Hard constraints were used to describe the safety margins and requirements of each hole as these constraints cannot be violated. Soft constraints were used to formulate the satisfactory level of the golf course routing by using physical programming.

How to provide the architect with a diverse set of routings?
Each of the different approaches has its own way of creating diversity. For the constraint programming solution from chapter 4 diversity can be added in different ways: adding specific constraints to the par of the whole routing, using additional hole-specific constraints and using constraints to make it differ from previously found solutions.

The random search algorithm from chapter 5 provides diversity by selecting results which are different from the others. By selecting the best results which differ by measuring the distance between the previous solutions found, the results after generating the first nine holes as well as the second nine holes are diverse.

The genetic algorithm from 5 creates solutions based on the initial population. By having a diverse initial population, the output will also contain some of this diversity. The other way to create diversity comes by means of the two actions, mutation and crossover.

How can one automatically design a diverse set of golf course routings considering the architect's preferences?
With the answers to the sub-questions, the main research question can be answered. Three different techniques can be used to automatically design a diverse set of golf course routings while considering the architect's preferences. The constraint programming solution is the least effective of the three techniques. It can generate golf courses but is limited by the quadratic complexities in the constraint programming model. Some parts of the mathematical model do not translate well to the constraint programming model, making

its options limited. The genetic approach and random search approach both perform very similar, as shown in chapter 6, with the genetic algorithm creating slightly better golf course routings according to the objective function. The runtime of the random search algorithm is much better and, therefore the preferred solution to the golf course routing problem.

7.1. Recommendations

There are several points on which each of the discussed techniques can improve upon and some limitations that were not considered in time. The constraint programming approach struggles with all the quadratic functions that come with designing golf course routings. When a solver comes available, that can effectively deal with these quadratic constraints on a large scale, this option can be worked out further. As it is an exact approach, the results could be better than the current solutions found by both the genetic algorithm and random search algorithm.

For the approach of both heuristic techniques, the chosen tactic to split the golf course into two loops of nine holes limits its possibilities, as the holes can also cross over into each other's available space, for example having holes 12 and 5 being close together. However, this is not possible for these two approaches. By designing a way to allow for one loop to cross over between two holes to the other side while making sure there is enough space and there is room to potentially cross back. The crossing of routings could result in more diverse solutions and more flexibility for the pre-set holes.

Another improvement for the random search algorithm concerns the number of options for each hole. By enhancing the feedback from getting stuck to order to upgrade the pruning process and to improve selecting the next hole, more solutions in the form of a larger variety of doglegs, starting locations and lengths can be considered. This could lead to even better results.

The genetic algorithm could be improved by changing the mutation action to be faster and better, as now the results after the mutation are not always heading in the right direction. The creation of the newly mutated holes takes too long for the mutation action to be effective in the genetic algorithm. Maybe a different approach to the mutation could improve the genetic algorithm and result in even better golf course routings.

A

Appendix A

A.1. Problem desciption

Problem description “Optimization of Golf Course Routings”

Context

When designing a golf course, there are two discrete skills that a golf architect must possess. First, he or she must determine the layout of the 18 golf holes on the site, which is called a “routing” (see picture below: green circles are the greens, red lines are the lines of play, the blue line is the outer boundary, brown lines are 5 m contour lines, the black rectangle is the club house location).

Secondly, she or he must then fill in each golf hole in detail for the chosen routing.

In practice we see that many golf architects are fairly good at designing individual holes, but are not very good at creating optimal routings. The question is whether an artificial intelligence / optimization program can be made that can help these architects.

Routing Problem

The routing problem consists of two elements.

1. First, all possible ways must be found by which an 18-hole golf course can be routed within certain given boundary conditions (eg the size of the available terrain, the existing topography of the landscape, external security (houses, roads around terrain), internal security (between holes), club house location, preferred green and tee locations etc.).

2. These routings then can be ranked according to a number of general (objective/subjective) criteria to be defined in by the architect (eg good distribution of direction and length of the holes, short walking distances between holes, etc.).

The output consists of the best 20-30 routings found according to this method.

Boundary conditions (examples)

- Golf course is 9 or 18 holes
- Holes can be either par 3, 4 or 5
- Par 3 hole is 100 to 220 m long
- Par 4 hole is 220 to 425 m long
- Par 5 hole is 425 to 620 m long
- The par sum of the holes preferably should be between 69 and 73
- The par sum of holes 1-9 and holes 10-18 must not differ by more than 2 strokes
- Tees of hole 1 and green of hole 18 must be within 100 m of the club house
- Tee of hole 10 and green of hole 9 must be within 150m of the clubhouse
- A par 3 hole has no dog leg (kink in the line of play)
- A par 4 hole has a maximum of one dog leg
- A par 5 hole has a maximum of two doglegs
- Length from the tee' to the first dog leg point is 180 - 250 m
- Length from the first dogleg point to the second dogleg point is 100 - 200 m
- Dogleg angle is maximum 40 degrees (0 degrees is no dogleg)
- Diameter of the green circle is 20 m
- Distance between center green and next tee must be at least 30 m
- Distance between center green and water must be at least 15m
- Distance between the playing lines of two holes must be at least 60 m
- Distance between the playing line of a hole and outer boundary must be at least 60 m
- Distance between the playing line and a eco zone/woodland must be at least 30 m
- Maximum vertical slope of playing line up is 4%
- Maximum vertical slope of playing line down is 25%

- Where possible, use the preferred green positions as indicated by the architect

- Topographic map on which is indicated

    - Boundaries of the site
    - Contour lines
    - Woodland/eco areas
    - Water areas
    - Clubhouse position
    - Roads on site
    - Areas with potential security issues (roads, houses)

Assessment criteria (examples)

- How many of the preferred green sites are used?
- How much variation is there in the length and direction of the golf holes?
- What is the total par and length of the golf course?
- The balance in the number of left and right doglegs
- Length of walking distances between holes
- Avoid a starting hole to the east (morning sun)
- Avoid a finishing hole to the west (evening sun)
- Etc. etc.

Update

I wrote the above description in 2006 when artificial intelligence, and in particular deep learning, was not as well known yet in the general public. My idea then was to create a dumb crawler routing creation program that would create millions of routings, which would then be filtered on quality afterwards so that the best 20-30 routings could be selected.

The question is whether a AI / deep learning process using the approximately 40,000 existing golf courses in the world would be a better approach.

Acquiring the data could be an issue, although good aerial photos of almost all golf courses can be collected via Google Earth or Bing Maps.

Another problem is of course that many of these 40,000 golf courses have poor routings. The question is should you include that as input per golf course, or can the system learn and / or determine this itself.

I would like to work on this problem in the coming years with people who can supplement my knowledge golf course design with deep learning knowledge.

Frank Pont

B

Appendix B

B.1. First Model

```
include "FunctionsInt.mzn";
include "ObjectiveFunctions.mzn";
include "TrigonometryInt.mzn";

% Parameters
int: ClubhouseX = 1;
int: ClubhouseY = 1;
int: H = 3; % Number of holes
int: North = 0; % Offset of north
int: xMax = 3000; % Size of area x
int: yMax = 3000; % Size of area y

% Decision variables
array[1..H] of var 0..360: direction;
array[1..H] of var int: pars;
array[1..H] of var int: length;
array[1..H+1] of var int: walk;
array[1..H] of var 0..xMax: greenSitesX;
array[1..H] of var 0..yMax: greenSitesY;
array[1..H] of var 0..xMax: teesX;
array[1..H] of var 0..yMax: teesY;
array[1..H] of var -1..xMax: dogleg1X;
array[1..H] of var -1..yMax: dogleg1Y;
array[1..H] of var -1..xMax: dogleg2X;
array[1..H] of var -1..yMax: dogleg2Y;

predicate correctPar3(array[1..H] of var int: pars,
array[1..H] of var int: length) =
  forall(i in 1..H) (if length[i] < 200 then pars[i] == 3
endif);

predicate correctPar4(array[1..H] of var int: pars,
array[1..H] of var int: length) =
  forall(i in 1..H) (if length[i] < 600 /\ length[i] >=
200 then pars[i] == 4 endif);

predicate correctPar5(array[1..H] of var int: pars,
array[1..H] of var int: length) =
  forall(i in 1..H) (if length[i] >= 600 then pars[i] == 5
endif);

constraint correctPar3(pars, length);
constraint correctPar4(pars, length);
```

```
constraint correctPar5(pars, length);

constraint walk[1] = distanceInt(ClubhouseX, ClubhouseY,
teesX[1], teesY[1]);
constraint walk[1] > 50;
constraint walk[H+1] = distanceInt(greenSitesX[H],
greenSitesY[H], ClubhouseX, ClubhouseY);
constraint walk[H+1] > 50;
constraint forall(i, j in 1..H where j == i + 1) (walk[j]
= distanceInt(greenSitesX[i], greenSitesY[i], teesX[j],
teesY[j]));
constraint forall(i, j in 1..H where j == i + 1)
(distanceInt(greenSitesX[i], greenSitesY[i], teesX[j],
teesY[j]) > 30);

constraint forall(i in 1..H)(distanceInt(greenSitesX[i],
greenSitesY[i], teesX[i], teesY[i]) > 100);

% Length
constraint forall(i in 1..H where dogleg1X[i] == -
1)(length[i] = distanceInt(greenSitesX[i], greenSitesY[i],
teesX[i], teesY[i]));
constraint forall(i in 1..H where dogleg1X[i] != -1 /\
dogleg2X[i] == -1)(length[i] = distanceInt(greenSitesX[i],
greenSitesY[i], dogleg1X[i], dogleg1Y[i]) +
distanceInt(greenSitesX[i], greenSitesY[i], dogleg1X[i],
dogleg1Y[i]));
constraint forall(i in 1..H where dogleg2X[i] != -
1)(length[i] = distanceInt(greenSitesX[i], greenSitesY[i],
dogleg1X[i], dogleg1Y[i]) + distanceInt(greenSitesX[i],
greenSitesY[i], dogleg2X[i], dogleg2Y[i]) +
distanceInt(dogleg1X[i], dogleg1Y[i], dogleg2X[i],
dogleg2Y[i]));


constraint forall(i, j in 1..H where i < j)(greenSitesX[i]
!= greenSitesX[j] \/ greenSitesY[i] != greenSitesY[j]);
constraint forall(i, j in 1..H where i < j)(teesX[i] !=
teesX[j] \/ teesY[i] != teesY[j]);
```

```
constraint forall(i, j in 1..H where i !=
j)(distancePointLine(teesX[i], teesY[i], greenSitesX[j],
greenSitesY[j], teesX[j], teesY[j]) > 3600);

% Doglegs
predicate correctNoDL(array[1..H] of var int: pars,
array[1..H] of var int: dlX, array[1..H] of var int: dlY)
=
  forall(i in 1..H) (if pars[i] == 3 then dlX[i] == -1 /\
dlY[i] == -1 endif);

predicate correct1DL(array[1..H] of var int: pars,
array[1..H] of var int: dlX, array[1..H] of var int: dlY)
=
  forall(i in 1..H) (if pars[i] != 3 then dlX[i] != -1 /\
dlY[i] != -1 endif);

predicate correct2DL(array[1..H] of var int: pars,
array[1..H] of var int: dlX, array[1..H] of var int: dlY)
=
  forall(i in 1..H) (if pars[i] == 5 then dlX[i] != -1 /\
dlY[i] != -1 endif);

  predicate correctDL(array[1..H] of var int: pars,
array[1..H] of var int: dlX, array[1..H] of var int: dlY)
=
  forall(i in 1..H) (if pars[i] != 5 then dlX[i] == -1 /\
dlY[i] == -1 endif);

constraint correctNoDL(pars, dogleg1X, dogleg1Y);
constraint correctNoDL(pars, dogleg2X, dogleg2Y);
constraint correct1DL(pars, dogleg1X, dogleg1Y);
constraint correct2DL(pars, dogleg2X, dogleg2Y);
constraint correctDL(pars, dogleg2X, dogleg2Y);

% Check correct values
constraint forall(i in 1..H where dogleg1X[i] != -
1)(distanceInt(teesX[i], teesY[i], dogleg1X[i],
dogleg1Y[i]) > 100);
constraint forall(i in 1..H where dogleg1X[i] != -
1)(distanceInt(greenSitesX[i], greenSitesY[i],
dogleg1X[i], dogleg1Y[i]) > 100);
```

```
constraint forall(i, j in 1..H where dogleg1X[i] != -1 /\
dogleg1X[j] != -1 /\ j != i)(distanceInt(dogleg1X[i],
dogleg1Y[i], dogleg1X[j], dogleg1Y[j]) > 100);

constraint forall(i in 1..H where dogleg2X[i] != -
1)(distanceInt(teesX[i], teesY[i], dogleg2X[i],
dogleg2Y[i]) > 100);
constraint forall(i in 1..H where dogleg2X[i] != -
1)(distanceInt(greenSitesX[i], greenSitesY[i],
dogleg2X[i], dogleg2Y[i]) > 100);
constraint forall(i in 1..H where dogleg2X[i] != -
1)(distanceInt(dogleg1X[i], dogleg1Y[i], dogleg2X[i],
dogleg2Y[i]) > 100);
constraint forall(i, j in 1..H where dogleg2X[i] != -1 /\
dogleg1X[j] != -1 /\ j != i)(distanceInt(dogleg2X[i],
dogleg2Y[i], dogleg1X[j], dogleg1Y[j]) > 100);
constraint forall(i, j in 1..H where dogleg2X[i] != -1 /\
dogleg2X[j] != -1 /\ j != i)(distanceInt(dogleg2X[i],
dogleg2Y[i], dogleg2X[j], dogleg2Y[j]) > 100);

constraint forall(i in 1..H)(direction[i] =
trueDegree(teesX[i], teesY[i], greenSitesX[i],
greenSitesY[i], North));


var int: score = valueFirstTeeClubHouse(walk[1]) +
directionFirstHole(direction[1]) + sumPars2(sum(pars));

solve maximize score;

output ["Pars: \(pars)\n",
          "Length: \(length)\n",
          "Walk: \(walk)\n",
          "Green X: \(greenSitesX)\n",
          "Green Y: \(greenSitesY)\n",
          "Tee X: \(teesX)\n",
          "Tee Y: \(teesY)\n",
          "DL1X: \(dogleg1X)\n",
          "DL1Y: \(dogleg1Y)\n",
          "DL2X: \(dogleg2X)\n",
          "DL2Y: \(dogleg2Y)\n",
          "Direction: \(direction)\n",
```

```
        "Res: \(score) ",
    ];
```

B.2. Second Model generator 1

```
include "Parameters.mzn";
include "TestFunctions.mzn";

% Decision parameters
array[1..H] of var 0..360: direction;
array[1..H] of var 0..xMax: greenSitesX;
array[1..H] of var 0..yMax: greenSitesY;
array[1..H] of var 0..xMax: halfX;
array[1..H] of var 0..yMax: halfY;
array[1..H] of var 0..xMax: teesX;
array[1..H] of var 0..yMax: teesY;
array[1..H] of var 0..3000: length;
array[1..H+1] of var 0..100: walk;

constraint abs(teesX[1] - ClubhouseX) + abs(teesY[1] -
ClubhouseY) <= 15;
constraint abs(greenSitesX[H] - ClubhouseX) +
abs(greenSitesY[H] - ClubhouseY) <= 15;
constraint forall(i in 1..H) (length[i] ==
distanceInt(greenSitesX[i], greenSitesY[i], teesX[i],
teesY[i]));

constraint walk[1] = distanceInt(ClubhouseX, ClubhouseY,
teesX[1], teesY[1]);
constraint forall(i, j in 1..H where j = i + 1) (walk[j] =
distanceInt(greenSitesX[i], greenSitesY[i], teesX[j],
teesY[j]));
constraint walk[H+1] = distanceInt(greenSitesX[H],
greenSitesY[H], ClubhouseX, ClubhouseY);
constraint forall(i in 1..H+1)(walk[i] <= 150);
constraint forall(i in 2..H)(walk[i] >= 36);
constraint forall(i in 1..H) (distanceInt(greenSitesX[i],
greenSitesY[i], teesX[i], teesY[i]) >= 130);
constraint forall(i in 1..H) (distanceInt(greenSitesX[i],
greenSitesY[i], teesX[i], teesY[i]) <= 2704);

constraint forall(i in 1..H) (halfX[i] = (teesX[i] +
greenSitesX[i]) div 2);
constraint forall(i in 1..H) (halfY[i] = (teesY[i] +
greenSitesY[i]) div 2);

% Approximated distance
```

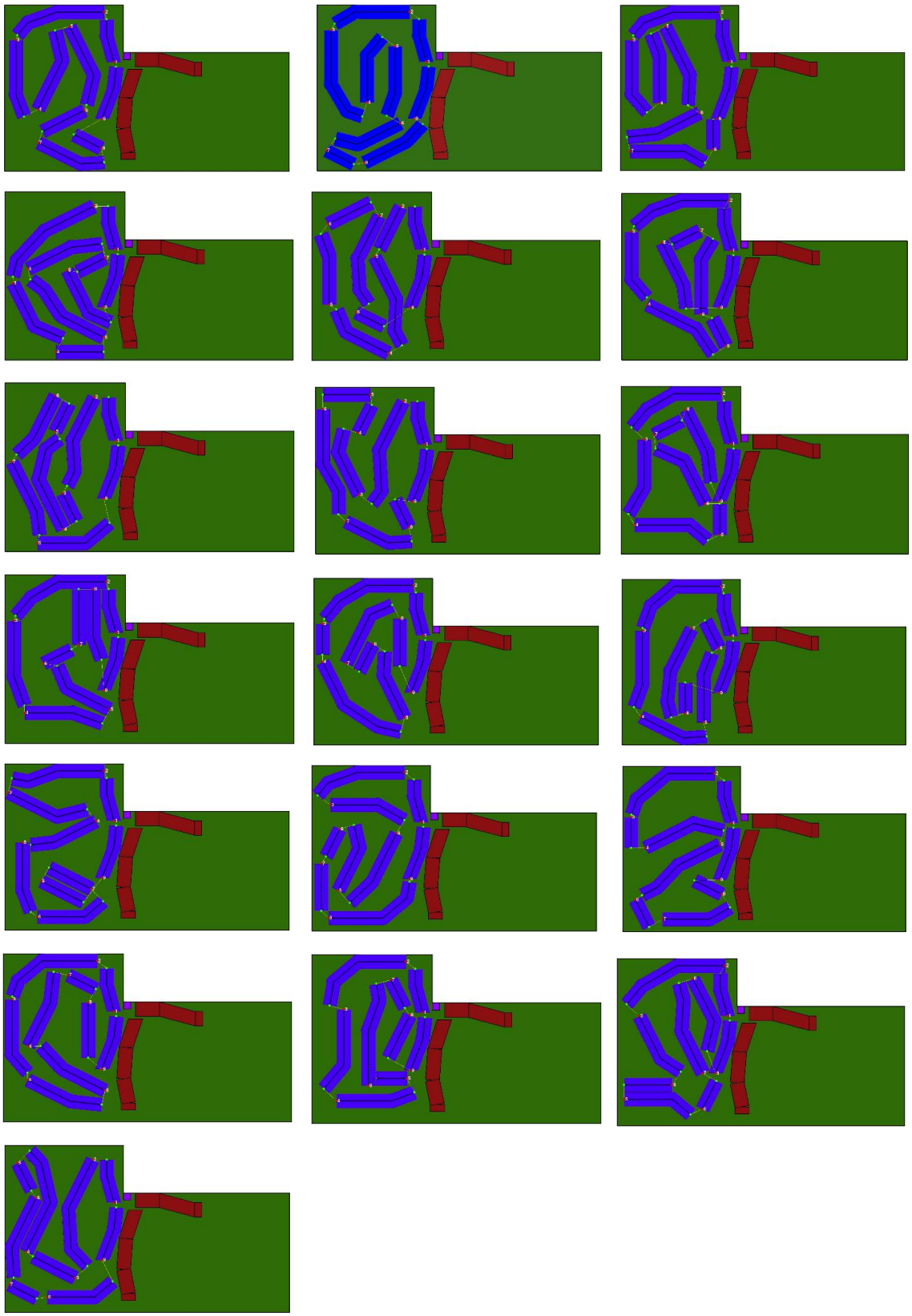
```
constraint forall(i, j in 1..H where i < j)
  (distanceLineLineApprox(teesX[i], teesY[i], halfX[i],
halfY[i], teesX[j], teesY[j], halfX[j], halfY[j], 6));
constraint forall(i, j in 1..H where i < j)
  (distanceLineLineApproxTeeGreen(halfX[i], halfY[i],
teesX[i], teesY[i], greenSitesX[j], greenSitesY[j],
halfX[j], halfY[j], 6));
constraint forall(i, j in 1..H where i < j)
  (distanceLineLineApproxTeeGreen(halfX[i], halfY[i],
greenSitesX[i], greenSitesY[i], teesX[j], teesY[j],
halfX[j], halfY[j], 6));
constraint forall(i, j in 1..H where i < j)
  (distanceLineLineApprox(greenSitesX[i], greenSitesY[i],
halfX[i], halfY[i], halfX[j], halfY[j], greenSitesX[j],
greenSitesY[j], 6));

% Direction
constraint forall(i in 1..H)(direction[i] ==
trueDegree(teesX[i], teesY[i], greenSitesX[i],
greenSitesY[i], North));

% Distance between tee and green
constraint forall(i, j in 1..H where i < j)
  (distanceLL(teesX[i], teesY[i], greenSitesX[i],
greenSitesY[i], teesX[j], teesY[j], greenSitesX[j],
greenSitesY[j]));

% Par
constraint greenSitesX[H] == 3;
constraint greenSitesY[H] == 0;


constraint teesX[1] == 0;
constraint teesY[1] == 3;


var 0..150*(H+1): res = sum(walk);
constraint countPar(length)[1] >= 2;
constraint countPar(length)[2] >= 2;
constraint countPar(length)[3] >= 1;

constraint countDir(direction)[1] >= 2;
constraint countDir(direction)[2] >= 2;
```

```
constraint countDir(direction)[3] >= 2;
constraint countDir(direction)[4] >= 2;


solve minimize res;

output [
          "teesX = \(teesX)\n",
          "teesY = \(teesY)\n",
          "greenSitesX = \(greenSitesX)\n",
          "greenSitesY = \(greenSitesY)\n",
          "approxLength = \(length)\n",
          "walk = \(walk)\n",
          "direction = \(direction)\n",
          "Res = \(res)\n",
          ];
```

C

Appendix C

C.1. Halfway results L-shaped run

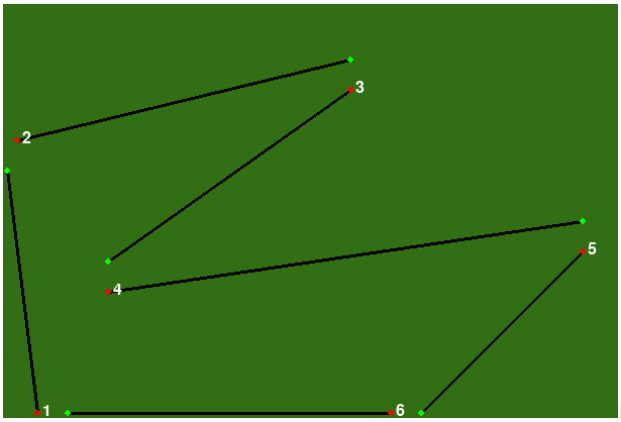


D

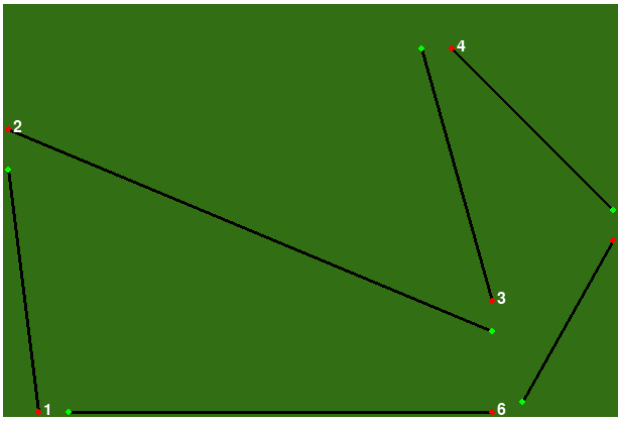
Appendix D

D.1. Diversity on 6 hole courses

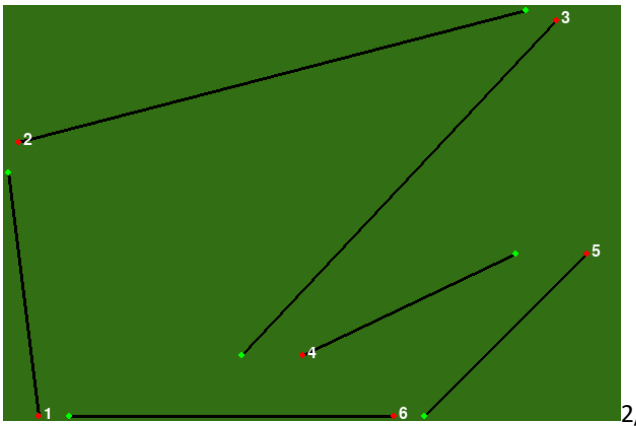
No variables set



Free

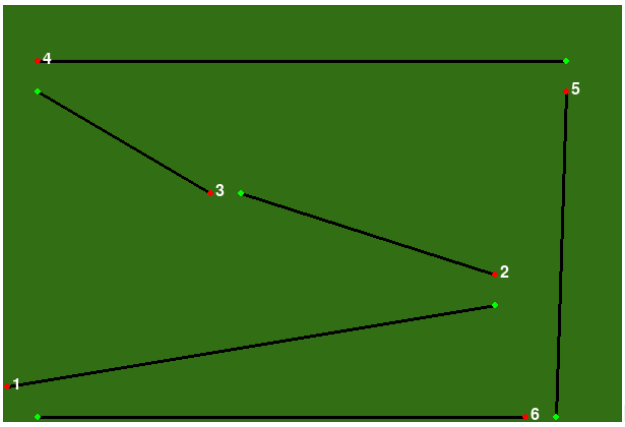


1/2/1

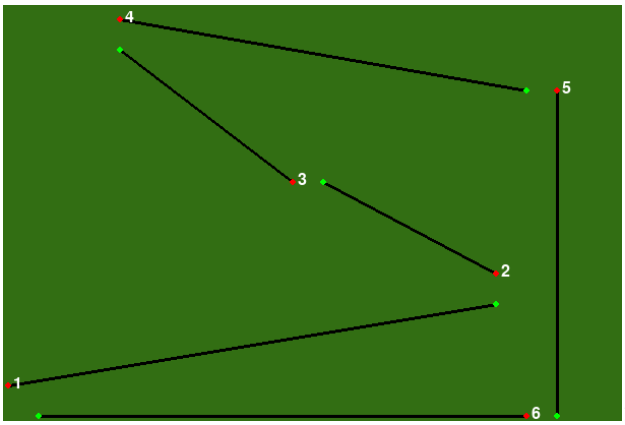


2/2/2

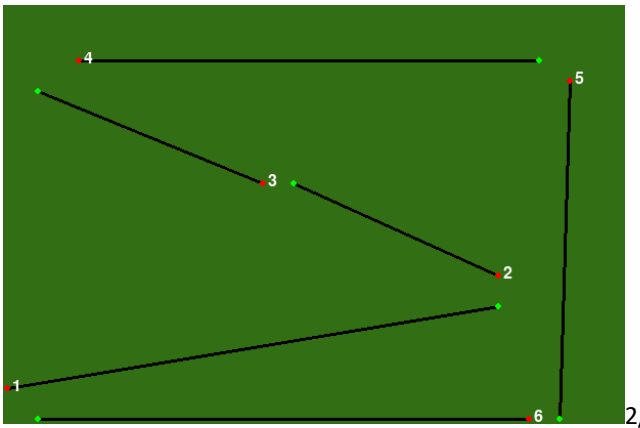
Start tee and final green set



Free

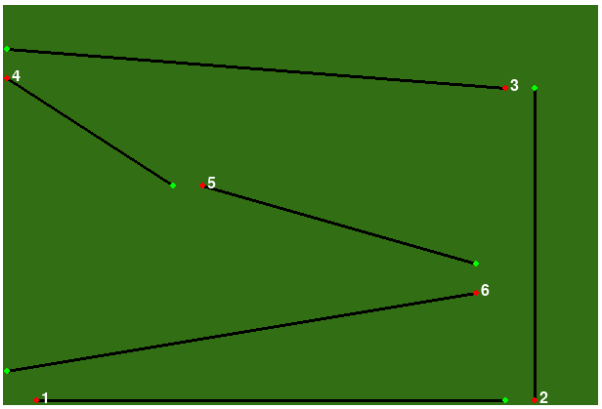


1/2/1

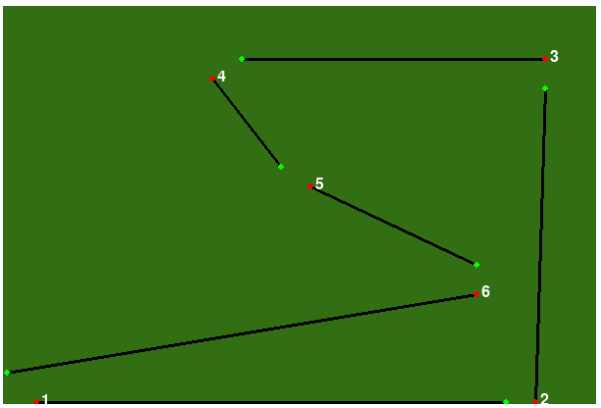


2/2/2

Start tee and final green flipped

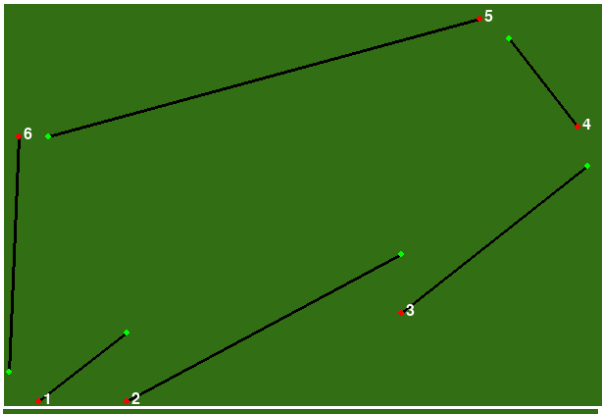


Free & 1/2/1

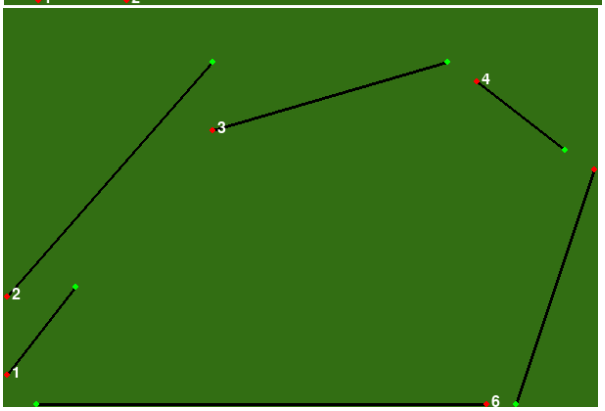


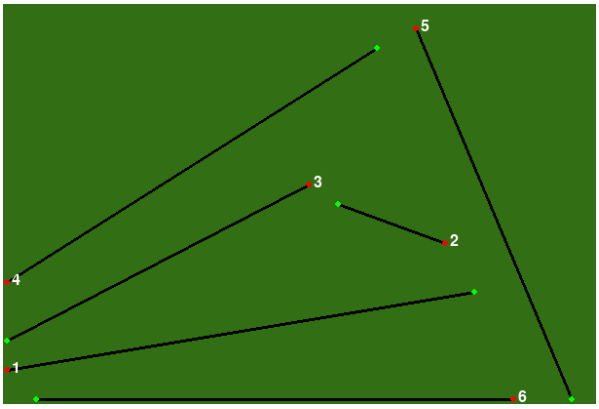
2/2/2

Experimenting with setting specifics



Restricting length of the first hole and second hole





Green site of the third hole must be in the first quadrant

Bibliography

[1] Franklin Antonio. Faster line segment intersection. In *Graphics Gems III (IBM Version)*, pages 199–202. Elsevier, 1992.

[2] Roman Bartak. Constraint propagation and backtracking-based search. *Charles Universität, Prag*, 2005.

[3] Elie Daher, Sylvain Kubicki, and Annie Guerriero. Data-driven development in the smart city: Generative design for refugee camps in luxembourg. *Entrepreneurship and Sustainability Issues*, 4(3):364, 2017.

[4] Robert Muir Graves and Geoffrey S Cornish. *Classic golf hole design: using the greatest holes as inspiration for modern courses*. John Wiley & Sons, 2002.

[5] Michael J Hurdzan. *Golf course architecture: evolutions in design, construction, and restoration technology*. John Wiley & Sons, 2005.

[6] Mehmet Ali Ilgin and Surendra M Gupta. Physical programming: A review of the state of the art. *Studies in Informatics and Control*, 21(4):349–366, 2012.

[7] Ryan Woodard Johnson. A comparison of perceptions among amateur and pga professional golfer to the five design principles of golf course architecture. 2010.

[8] Daniele Loiacono, Luigi Cardamone, and Pier Luca Lanzi. Automatic track generation for high-end racing games using evolutionary computation. *IEEE Transactions on computational intelligence and AI in games*, 3(3):245–259, 2011.

[9] Kim Marriott, Peter J Stuckey, LD Koninck, and Horst Samulowitz. A minizinc tutorial, 2014.

[10] Seyedali Mirjalili. Genetic algorithm. In *Evolutionary algorithms and neural networks*, pages 43–55. Springer, 2019.

[11] University of Melbourne Monach University, Data61 Decision Science. MiniZinc (2.5.5). URL `https://www.minizinc.org/`.

[12] Jaime Montemayor, Allison Druin, Allison Farber, Sante Simms, Wayne Churaman, and Allison D'Amour. Physical programming: designing tools for children to create physical interactive environments. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pages 299–306, 2002.

[13] Sangeun Oh, Yongsu Jung, Seongsin Kim, Ikjin Lee, and Namwoo Kang. Deep generative design: Integration of topology optimization and generative models. *Journal of Mechanical Design*, 141(11), 2019.

[14] Franco P Preparata and Michael I Shamos. *Computational geometry: an introduction*. Springer Science & Business Media, 2012.

[15] The R&A. *Golf around the world 2019*. 3rd edition, 2019. Research: National Golf Foundation, Jupiter, FL USA Narrative: Bradley S. Klein, Bloomfield, CT USA Report design: The PPL Group, Liverpool, UK.

[16] Forrest L Richardson. *Routing the golf course: the art & science that forms the golf journey*. John Wiley & Sons, 2002.

[17] Francesca Rossi, Peter Van Beek, and Toby Walsh. Constraint programming. *Foundations of Artificial Intelligence*, 3:181–211, 2008.

[18] Todor Stojanovski, Jenni Partanen, Ivor Samuels, Paul Sanders, and Christopher Peters. City information modelling (CIM) and digitizing urban design practices. *Built Environment*, 46(4):637–646, 2020.

[19] Quentin Vanhaelen, Yen-Chu Lin, and Alex Zhavoronkov. The advent of generative chemistry. *ACS Medicinal Chemistry Letters*, 11(8):1496–1505, 2020.

[20] John Waddington. How many golf players in the world?, Mar 2022. URL `https://golfeducate.com/how-many-golf-players-in-the-world/`.

[21] Mark Wallace. *Building decision support systems: using MiniZinc*. Springer, 2020.

[22] Zelda B Zabinsky et al. Random search algorithms. *Department of Industrial and Systems Engineering, University of Washington, USA*, 2009.

[23] Jingyu Zhang, Nianxiong Liu, and Shanshan Wang. Generative design and performance optimization of residential buildings based on parametric algorithm. *Energy and Buildings*, 244:111033, 2021.