Solving Systematic Conservation Planning using Integer Linear Programming

Simon Schreinermacher

A thesis presented for the degree of
Bachelor of Science

Algorithmic Bioinformatics
Heinrich Heine University Düsseldorf
Germany
6th September, 2022

Acknowledgments

I am very grateful to Prof. Dr. Gunnar Klau for proposing this interesting topic to me and for being the first assessor for this thesis. I would also like to thank Prof. Dr. Michael Leuschel as the second assessor for this thesis. Finally, I would especially like to express my gratitude to Eline van Mantgem for her feedback and advice during our weekly meetings.

Abstract

Systematic Conservation Planning is a process that aims to efficiently implement conservation actions in a nature reserve to conserve and protect the biodiversity in the area. One step in this process is to find a suitable selection of space in which to apply the conservation efforts. To do this, the area is divided into many different smaller tiles, called planning units. The goal is to conserve a selection of planning units so that the conservation process results in the biodiversity being preserved as intended while minimizing the conservation costs. In this thesis, we implement an algorithm to solve this problem by formulating it as an instance of Integer Linear Programming (ILP). We conduct several different experiments that solve the base problem and variants of it using mainly synthetic data and one real dataset. We compare our results to the results of Marxan, another application that solves this problem by using Simulated Annealing. Furthermore, we propose options to generate synthetic data instances ourselves and show how the realism of said synthetic data can be improved through the use of Perlin noise. Our results show that while the ILP cannot find the optimal solution in a reasonable time and while it displays a varying performance for different scenarios, it can reliably get much closer to the optimum than Marxan when both use the same timeframe. We conclude that the ILP implementation can be a useful alternative to Marxan for real conservation projects.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Scientific Background

Protecting valuable natural environments is an important task to conserve the biodiversity that exists on Earth. Doing so can prove to be difficult, as many factors contribute to this situation, and the required actions to achieve effective conserving can differ from area to area. Margules and Pressey [1] introduced Systematic Conservation Planning, a multi-step framework that aims to bring structure to this complex task. First, the given area is analyzed to clearly define its specific conservation needs, namely the biodiversity features that need to be protected. With this information, explicit actions that need to be implemented can be formulated. Existing conservation areas, if present, are evaluated on their contribution to the conservation needs, and the best new locations for effective conservation are identified. In these new areas, the determined actions are applied. Ultimately, mechanisms are adopted to maintain the conserving efforts over an extended period of time.

Applying this framework comes with several challenges during each phase of implementation as it is still a complex task, even with careful planning and assessment. One of these problems is the financial aspect. In most cases, it is not feasible to implement conservation efforts across the entire natural reserve since it can extend over several thousands of square kilometers. A crucial step in the planning process is to decide where to take actions to be as efficient and inexpensive as possible. An idea is to divide the entire nature reserve into many smaller sub-areas, i.e., planning units [2]. These can have arbitrary sizes and shapes, e.g., squares or hexagons, to cover the entire area easily. Alternatively, their shape could be defined by the landscape they contain, e.g. by mountains, forests, or man-made boundaries.

Figure 1 and Figure 2 show how a nature reserve could be divided into planning units and display the selection of planning units for conservation to complement existing conservation areas.

Planning units possess several attributes, such as the conservation cost for implementing conservation actions in the unit‘s area or the biodiversity, i.e., the conservation features contained in the unit. The goal is to strategically select units where conservation measures will be implemented to minimize the total conservation costs while satisfying target values regarding several conservation features. This is known as the Reserve Selection Problem [2].

These features could include a minimum number of occurrences of a certain species or a valuable natural resource across all selected planning units. A marine conservation project could for example demand that all selected units combined must contain at least 3000 individuals of an endangered fish species and 5000 hectares of coral reefs.

Besides the conditions for conservation features, there could be additional criteria the selected units need to satisfy. A few examples are mentioned in [2]. Due to the biological or geological processes in the area, there could be situations, where conserving a unit A is only possible or profitable, if another unit B is selected as well. In this case, we call the unit A to be dependent on the unit B. Satisfying these dependencies is one such additional criterion. Additionally, we

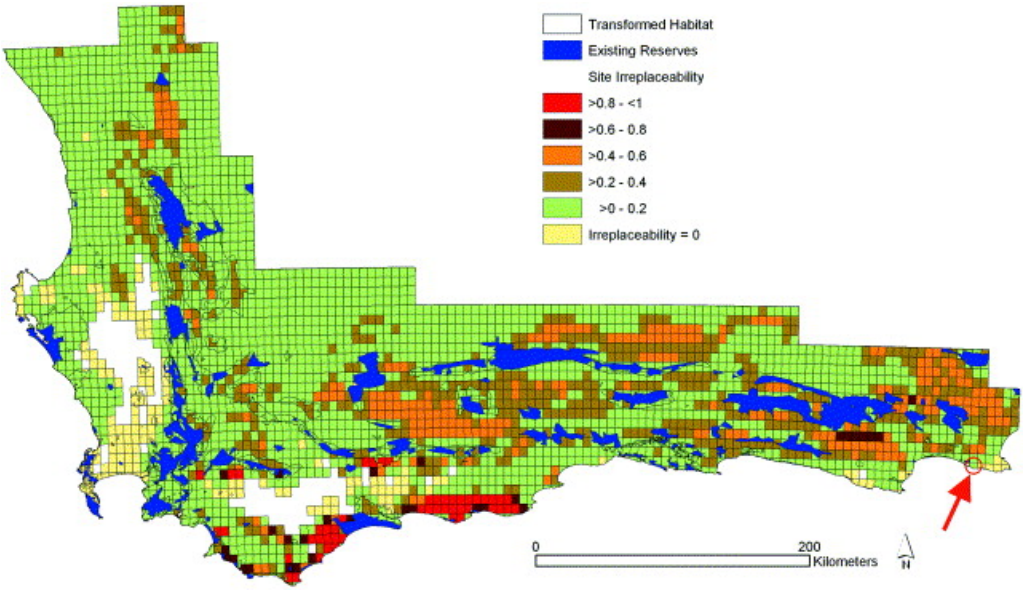


Figure 1: An example for a conservation planning project, located in the Cape Floristic Region in South Africa. The figure shows how the area is divided into several planning units in a square-shaped grid. The blue squares represent the already selected area, the other colors indicate the irreplaceability value for each non-selected unit (a calculated probability for this unit to be included in extended conservation efforts). Reused with permission from [3].

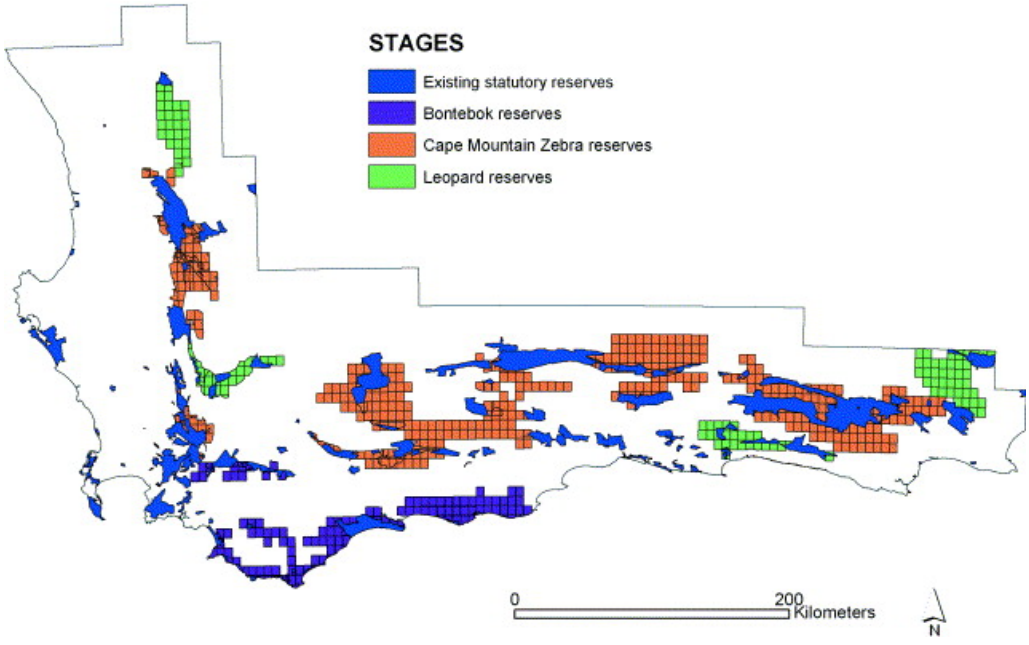


Figure 2: The reserve from Figure 1. Here, the results of a solution for extended conservation planning, found by an algorithm, are shown. The orange, green and purple units represent the area that was excluded from the initial conservation efforts but included in the extended planning to meet three additional conservation targets. It is visible, that these areas coincide with the areas of high irreplaceability value from Figure 1. Reused with permission from [3].

could demand the boundary length of all selected units be as minimal as possible, in which case clusters of selected units would be preferred over single selected units all over the area. Therefore, the right selection of planning units is key to ensuring the efficiency of conservation projects. However, depending on the demands on the planning units, this problem can get complex very quickly.
In this thesis, we implement an algorithm that solves the Reserve Selection Problem by deciding which planning units should be included in the conservation efforts to minimize the conservation costs while satisfying all demanded criteria. For this, we transform the problem into an instance of an Integer Linear Program (ILP) as described in [2], which is solved using the optimizing software Gurobi [4]. Afterward, we compare the results and performance of our implementation to the established software Marxan [5] which is also able to find solutions for the Reserve Selection Problem.

1.2 Related work and problems

There are many scientific papers regarding the biological aspect of systematic conservation planning ([1], [6]). Other publications focused on how to solve this problem algorithmically. One of the most common approaches is to formulate the problem as an ILP by minimizing the total expenses for all selected units while satisfying all conservation features through (in)equations as seen in the paper by Beyer et al. [2], which is the formulation this thesis builds upon as well. In their paper, they describe, how to formulate the basic problem of minimizing the cost for the selection of units that satisfy target values for biological features and how to include boundary penalties and dependencies. We reproduce their discoveries and extend their investigations. They did not publish their code, therefore we needed to implement our algorithm from scratch based on the mathematical formulation of the ILP they defined.
In addition to ILP, another approach is to use heuristic algorithms. A commonly used software is Marxan [5], which uses Simulated Annealing (Section 2.2) to decide which planning units should be selected. Marxan also incorporates the possibility to include potential boundary costs in their calculations. As with every heuristic algorithm, it finds a solution fast, whereas optimal algorithms would require much more time, but it can neither guarantee to find the optimal solution nor guarantee to satisfy all conservation targets.
There are a few variants of the Reserve Selection Problem, e.g., one reformulation that only searches for selections that cover each conservation feature at least once instead of demanding a certain required target value for each conservation feature to be satisfied. This is an applied instance of the Minimum Set Cover problem, whose related decision problem is known to be NP-complete [7]. Other papers also formulated an optimization problem where the amount of conservation features, that were covered at least once, is maximized while the total expenses must be kept below a certain threshold ([8] and [9]). These two variants deserve to be mentioned when introducing algorithmic approaches to systematic conservation planning, but they are not the subject of this thesis.

2 Preliminaries

2.1 Linear Programming

The following definition of linear programming is taken from the book "Understanding and Using Linear Programming", written by Jiří Matoušek and Bernd Gärtner [10]:

> A linear program is the problem of maximizing a given linear function over the set of all vectors that satisfy a given system of linear equations and inequalities. Each linear program can easily be transformed to the form:
>
> $$\text{maximize } c^T x \text{ subject to } Ax \leq b \tag{1}$$

Here, $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$ and $A \in \mathbb{R}^{m \times n}$ are known constants with $c^T x$ being the objective function that must be maximized and $Ax \leq b$ the linear constraints that need to be satisfied. $x \in \mathbb{R}^n$ represents the solution vector, which is called feasible if it satisfies all linear constraints and optimal if it simultaneously maximizes the objective function. Algorithms that can solve any instance of a linear program in a polynomial runtime exist, e.g., the Simplex method [10], which makes this optimization problem efficiently solvable.

If we restrict the solution vector to only allow integer values, i.e., if we define the solution vector through $x \in \mathbb{Z}^n$, we face a different optimization problem called Integer Linear Programming (short: ILP). Unlike regular linear programs, ILP is proven to be NP-hard [7].
Each maximization problem can also be formalized as a minimization problem by multiplying the objective function $c^T x$ with -1. This is more useful in our case, as we wish to minimize the total cost of the conservation efforts, which will be represented through the objective function of the ILP.

2.2 Simulated Annealing

This entire explanation is loosely based on the definition by Kathryn A. Dowsland and Jonathan M. Thompson given in their book "Simulated Annealing" [11].

Simulated Annealing is a heuristic approach to minimize a given optimization function. It is based on the Local Search heuristic, which searches for better solutions in the neighborhood of the current solution. It expands Local Search by implementing a feature to escape local optima. The procedure is inspired by the cooling process of molten materials into their solid form, hence the name Simulated Annealing. Due to the laws of thermodynamics, a material, that has been first heated up to a certain temperature at which the material has turned into a liquid and afterward started to cool down has a certain probability of altering its internal

energy by a value of δE. This probability is described by :

$$P(\delta E) = exp(\frac{-\delta E}{k_B T}) \tag{2}$$

with temperature T and Boltzmann's constant k_B. It can be seen, that the probability $P(\delta E)$ decreases when the temperature decreases.

Simulated Annealing uses this physical analogy by representing the solution of a minimization problem, a cost function, through the energy of the material. In the beginning, a value T is set, which is analogous to the temperature of the physical system that keeps decreasing over time. In each iteration, with x being the currently selected solution, the algorithm inspects an element x' from the neighborhood of x, which is defined as all solutions that are only slightly different from x. This difference between the cost for x and x' is analogous to δE. $P(\delta E)$ is then calculated as above (although Boltzmann's constant is not required) and is used as the probability of selecting x' as the new current solution. If x' costs less than x, then $P(\delta E) > 1$ in which case x' is guaranteed to be selected. If x' is not selected, then x remains the selected solution, and another solution from the neighborhood of x is inspected.
As the algorithm progresses, the temperature T decreases, and so does $P(\delta E)$. While there is a good probability at the start to select a worse solution compared to the current solution, over time the algorithm becomes less likely to do so and will only accept better solutions. This means that during the first iterations, the algorithm tries to explore the different possible solutions and might even escape local minimums in hopes of finding better selections elsewhere. However, with the temperature slowly decreasing, the algorithm becomes more reluctant to accept worse solutions than the currently selected one.
Since Simulated Annealing is based on randomness, the algorithm cannot guarantee anything concerning the solution quality. The algorithm will continue to run until a stopping condition is reached, e.g., a verification if the current solution is within certain acceptance bounds. Alternatively, the algorithm could terminate after a set amount of iterations has passed. This means the runtime cannot be explicitly described since the solution quality depends on randomness. However, unlike ILP, it is not NP-hard, which makes Simulated Annealing especially useful when dealing with large instances.

According to the Marxan User Manual [12], Marxan uses Simulated Annealing to generate its solutions as results from a minimization function that calculates the total conservation cost based on the selected planning units. A new solution is found in each iteration by randomly flipping the state of one planning unit from selected to unselected or vice versa. The user can configure, how many Simulated Annealing runs should be processed and also how many iterations Marxan's Simulated Annealing should do in each run.

3 Formulating Conservation Planning as an ILP

In this section, we transform the conservation planning problem into a mathematical optimization problem in the form of an ILP. The ILP formulation is taken from the paper by Beyer et al. [2]. We define the sets and variables used slightly differently than them for clarity reasons.

3.1 The cost function

Our objective function minimizes the sum of the costs for all selected planning units. We define each planning unit i through a binary variable $x_i \in \{0, 1\}$ in which $x_i = 0$ means, that the planning unit i is excluded for conservation. With c_i, we define the cost associated with planning unit i. Due to the decision variable x_i, we can model the cost required for a single planning unit i as $c_i \cdot x_i$, regardless of whether i was selected. This brings us to the sum of all costs for the selected planning units, which can be expressed through:

$$min \sum_{i \in I} c_i x_i \tag{3}$$

Here, I is the set of all planning units involved in the nature reserve. Thus, this function minimizes the costs for all selected planning units.

3.2 Conservation features

The main focus of this biological problem is on the conservation features, which are the biodiversity goals we wish to accomplish with the conservation efforts. Let r_{ik} be the numerical contribution of the planning unit i to the conservation feature k. $r_{ik}x_i$ therefore represents, how much x_i adds to this feature, regardless if i was selected or not. We define the target value for conservation feature k as T_k. Summing up $r_{ik}x_i$ for each planning unit gets us the total amount of occurrences of feature k gained from the conservation actions. This value must be greater or equal to T_k. Therefore, we define the first inequation of the ILP as:

$$\sum_{i \in I} r_{ik} x_i \geq T_k, \forall k \in K \tag{4}$$

Note the $\forall k \in K$. Since we can have multiple conservation features, we require one inequation for each feature in the set of all conservation features K. Therefore, we add $\forall k \in K$ to the constraint.

3.3 Boundary costs

In some cases, we might wish to apply a penalty for selecting non-adjacent units to encourage finding solutions that select clusters of units, e.g., to reduce transportation costs. This penalty is comparable to a cost for the total border length of our selected units. For two adjacent units i and j, the border between i and j must count towards the total border length if exactly one of i and j is selected for conservation. Additionally, for all selected planning units at the edge

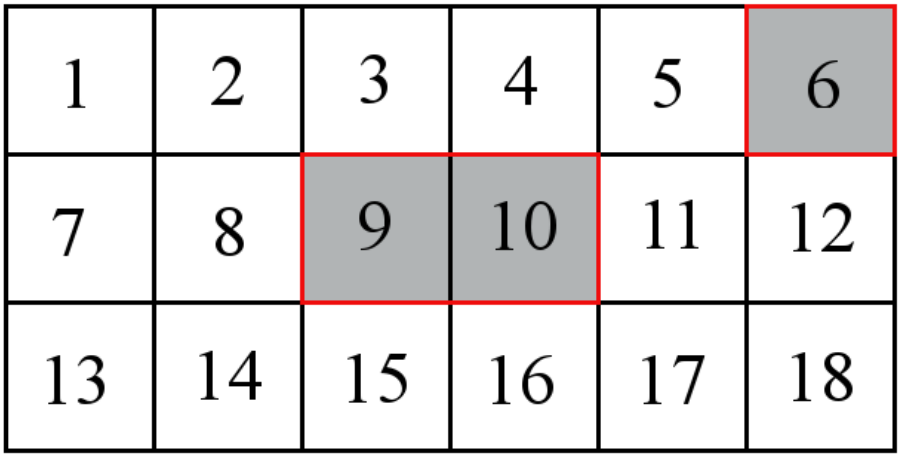


Figure 3: A sample instance with 18 planning units. The grey units were selected for conservation. The red boundaries are the ones that count towards the boundary penalty, these are all the boundaries between two adjacent units from which one unit is selected and one excluded. For example, the border between 8 and 9 counts towards the boundaries since 9 is selected and 8 is not. On the contrary, the border between 9 and 10 does not count because both 9 and 10 are selected. Additionally, for unit 6, all edges are included, even if the top and right edges are not connected to another planning unit.

of the reserve, the boundaries these units share with the area outside of the nature reserve must also be included in this penalty. Figure 3 shows a practical example of which boundaries must be included. Hereinafter, we refer to the penalties as boundary costs, since this is a more graspable concept.

If we define v_{ij} as the border length between planning unit i and j, then $x_i \cdot (1 - x_j) \cdot v_{ij}$ describes a formula, which results in v_{ij} if i is included and j excluded. Otherwise, it equals 0. We can find the border length for one unit i by summing the border length between all its neighbors j. Let $N(i)$ be the set of all planning units adjacent to i. Therefore,

$$\sum_{j \in N(i)} x_i(1 - x_j)v_{ij} \tag{5}$$

calculates the border length between unit i and all its neighbors j. For the total border between any two planning units, we sum equation (5) for each planning unit:

$$\sum_{i \in I} \sum_{j \in N(i)} x_i(1 - x_j)v_{ij} \tag{6}$$

With this, we covered all borders between two units. However, boundaries at the edge of the reserve have not been included yet as they are not borders between two units but rather borders between a unit and some area outside of the reserve. If we do not account for these boundaries, the algorithm would be biased to select units at the edge of the reserve. We

can account for these edge boundaries by adding their border length to the equation if the belonging planning unit is selected. Let E be the set of all planning units that are adjacent to the area outside of the reserve and $v_{i,outside}$ the length of the edge boundary a planning unit i shares with the area outside of the nature reserve. With

$$\sum_{i \in E} x_i \cdot v_{i,outside} \tag{7}$$

we get the sum of all edge boundaries for all selected units. Therefore, adding both equations gives us the total border length for the entire selected area as

$$\sum_{i \in I} \sum_{j \in N(i)} x_i(1-x_j)v_{ij} + \sum_{i \in E} x_i \cdot v_{i,outside} \tag{8}$$

Multiplying this equation with a parameter b gives us the possibility to control the importance of the boundary cost. If we do not want to account for boundary cost at all, we set $b = 0$. If boundary cost is very important, we set b to a high value. Unlike the conservation features, we do not add these terms as constraints but rather append them to the cost function since we wish to minimize the penalty of selecting non-adjacent units. The improved objective function is now defined as:

$$min \sum_{i \in I} c_i x_i + b \sum_{i \in I} \sum_{j \in N(i)} x_i(1-x_j)v_{ij} + b \sum_{i \in E} x_i \cdot v_{i,outside} \tag{9}$$

However, we are now faced with an issue. The term $x_i(1-x_j)v_{ij}$ can be rewritten as $x_i v_{ij} - x_i x_j v_{ij}$ and a multiplication between two binary variables is not allowed in linear programs. We need to add a workaround to avoid the term $x_i x_j$.

$x_i x_j$ will only be 1 if both x_i and x_j are 1. We can therefore substitute $x_i x_j$ through a new input variable z_{ij}, which must be 1 if i and j are both selected and 0 otherwise. We cannot simply define z_{ij} like this since all our input variables are independent of each other. Instead we need to add additional constraints that enforce $z_{ij} = 1$ if and only if $x_i = 1$ and $x_j = 1$. To ensure that z_{ij} cannot be 1 if either x_i or x_j are not 1, we add the constraints

$$z_{ij} - x_i \leq 0 \tag{10}$$

$$z_{ij} - x_j \leq 0 \tag{11}$$

To enforce z_{ij} to be 1 if both x_i and x_j are 1, we add

$$z_{ij} - x_i - x_j \geq -1 \tag{12}$$

Substituting $x_i x_j v_{ij}$ in the equation above through $z_{ij} v_{ij}$ and adding the three constraints to enforce the properties of z_{ij} will finally get us the linear constraints for the ILP that satisfy any demands concerning the boundary cost of our solution. Note that we do not have just a single additional input variable z_{ij} but rather a z_{ij} for each combination of two adjacent units

i and j. Looking at the constraints added for z_{ij}, it is apparent, that $z_{ij} = z_{ji}$. To shorten the mathematical requirements on the constraints, we introduce a set of neighboring units. Let $B \subset I^2$. A set $\{i, j\} \in I^2$ is part of B, if i and j are adjacent to each other. This way, our constraints regarding z_{ij} need to be added for each $\{i, j\} \in B$. Since we use $\{i, j\}$ instead of (i, j), the order of i and j does not matter, therefore $z_{ij} = z_{ji}$. In total, we get our new objective function as

$$min \sum_{i \in I} c_i x_i + b \sum_{i \in I} \sum_{j \in N(i)} x_i v_{ij} - z_{ij} v_{ij} + b \sum_{i \in E} x_i \cdot v_{i,outside} \tag{13}$$

and our added constraints as

$$z_{ij} - x_i \leq 0, \forall \{i, j\} \in B \tag{14}$$

$$z_{ij} - x_j \leq 0, \forall \{i, j\} \in B \tag{15}$$

$$z_{ij} - x_i - x_j \geq -1, \forall \{i, j\} \in B \tag{16}$$

3.4 Dependent units

There is a multitude of reasons why units would have a dependence on other units. A unit i is dependent on unit j if unit j must be selected in the case unit i was selected for conservation. Dependence is directional, i.e., j does not have to be dependent on unit i if i is dependent on j. We can define $D \subset I^2$ as the set of all dependencies, thus $(i, j) \in D \Leftrightarrow i$ is dependent on j. Mathematically, we can express dependence through

$$x_i - x_j \leq 0, \forall (i, j) \in D \tag{17}$$

This inequation therefore forces $x_j = 1$ if $x_i = 1$ but neither does it prevent j from being excluded if i is not part of the conservation units, nor does it enforce i to be selected, if j was selected.

3.5 The complete formulation

With this, we can present the entire ILP formulation through the terms and functions we defined in the previous subsections:

$$min \sum_{i \in I} c_i x_i + b \sum_{i \in I} \sum_{j \in N(i)} x_i v_{ij} - z_{ij} v_{ij} + b \sum_{i \in E} x_i \cdot v_{i,outside} \tag{18}$$

subject to

$$\sum_{i \in I} r_{ik} x_i \geq T_k, \forall k \in K \tag{19}$$

$$z_{ij} - x_i \leq 0, \forall \{i,j\} \in B \tag{20}$$

$$z_{ij} - x_j \leq 0, \forall \{i,j\} \in B \tag{21}$$

$$z_{ij} - x_i - x_j \geq -1, \forall \{i,j\} \in B \tag{22}$$

$$x_i - x_j \leq 0, \forall (i,j) \in D \tag{23}$$

with the variables x_i, c_i, r_{ik}, T_k, b, v_{ij}, $v_{i,outside}$, z_{ij} and the sets I, $N(i)$, K, E, B, D as defined before.

4 Methods

4.1 Implementation

This project is implemented in Python 3.10.5 using the optimization library Gurobi 3.9.5 [4] as an ILP solver. The implementation can be found on GitLab [1].

Data can be input into the algorithm either through a single input file called master.dat or multiple input files. All files must contain one entry per line, with commas separating the values in each line. If multiple input files are used, the following files are mandatory; pu.dat, which contains a list of the planning units, spec.dat, which contains all conservation features and their target values that need to be satisfied, and puvspr.dat, which lists information about which planning units contain which features. Additionally to these files, an optional file bound.dat may define the border length for each border between two adjacent units, and ultimately, an optional file dependencies.dat contains the dependencies between the units.
If a single input file is used, the file must be called master.dat and list all the necessary information from the multiple input files in sections. Each section starts with "=SECTIONNAME" with SECTIONNAME being "pu", "spec", "puvspr", "bound" or "dependencies". Similar to the multiple input files, the sections "pu", "spec" and "puvspr" are mandatory, while "bound" and "dependencies" are optional. The order of the sections does not matter. To further illustrate this format, the code folder contains some sample inputs and more detailed explanations.

When all necessary input files to define the instance are present, the program can be executed using

```
python main.py [OPTIONAL PARAMETERS]
```

The optional parameters may be added to configure the program to the user's needs:
-d: Include dependencies.
-b [INTEGER]: Sets the boundary multiplier (b in the ILP definition of Section 3.5). Setting this to 0 will remove the boundaries entirely. (default: 0)
-g [FLOAT]: Sets an acceptance threshold, which causes Gurobi to prematurely terminate the solving as soon as the percentage difference between the current best solution and the current theoretical lower bound for the optimal solution is less than the value given. -g 5 represents an acceptance threshold of 5 percent.
-t [FLOAT]: Sets a time limit in seconds that defines how long Gurobi may try to solve for the optimal solution.
-w [DIRECTORY]: Working directory for input and output files. (default: empty = same directory as the source code files)

[1] *https://gitlab.cs.uni-duesseldorf.de/albi/albi-students/ba-simon-schreinermacher*

-lp: Print the ILP instance into a .lp file. Will be inserted into the output folder.
-fmult: Force to use multiple input files (by default, the algorithm will prefer a master.dat if one is present, and only if none is available, it uses multiple input files. Using this flag forces the algorithm to use multiple input files, even when a master.dat is present).
-heatmap: Generate a heatmap that shows which units got selected in the solution. Will be inserted into the output folder.
-multobj: Splits the objective function into two parts and solves this multi-objective function hierarchically (see Section 5.4)

The algorithm will then process the input, create and solve the ILP instance and create an output directory inside the input folder. This output will contain a file that lists for each planning unit, whether it was selected (1) or excluded (0) from conservation planning, a file that lists statistics about that run, and the .lp file or the heatmap if configured.

4.2 Data creation

There is only a very limited amount of suitable data publicly available. To be able to measure different aspects of the algorithm and efficiently compare these to their Marxan equivalents, input instances of several sizes and forms are required. To surpass this bottleneck, we need to create data ourselves. Since the output consists solely of numbers, we can create synthetic data using random number generation. For this, we have adopted the method presented in the paper by Beyer et al. [2] and made slight adjustments. Here, we define how our data is created:

The ids of planning units and conservation features are modeled through ascending numbers, starting with 1. For the cost of each planning unit, we use a uniform distribution with lower bound 100 and upper bound 10000.

The amount of how much a planning unit contributes to a conservation feature is listed in puvspr.dat. This amount is generated by a normal distribution with expectation value $\mu = 0$ and standard deviation $\sigma = 5$. In case of a negative value for a given conservation feature in a given planning unit, this value is instead set to 0, i.e., this planning unit does not contribute to this conservation feature at all. Due to the selection of μ and σ, roughly half of all units are expected to contain a non-zero amount for a conservation feature.

The target value for each conservation feature is set to be 30 percent of the total contributions of all planning units to the respective conservation feature.

To simplify the boundaries, the planning units are rectangular and the area itself is a rectangular grid with a configurable amount of rows and columns, as well as width and height for

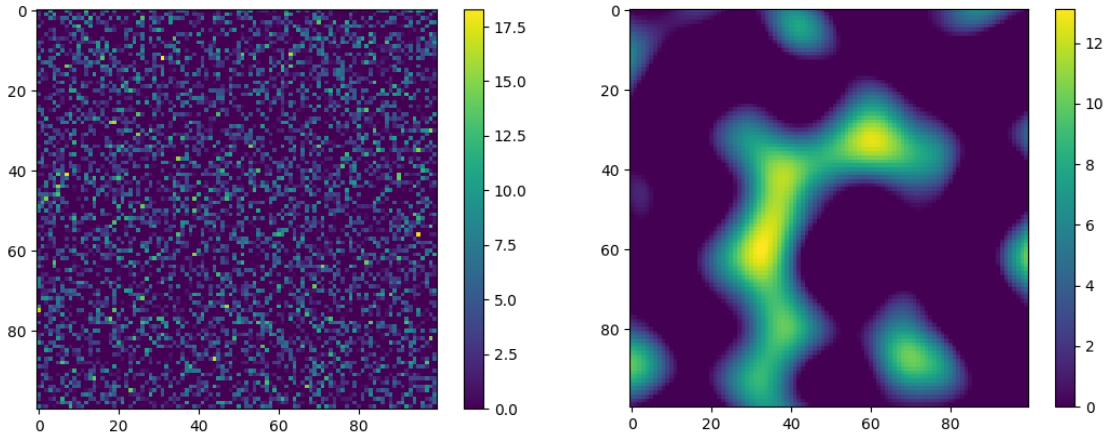


Figure 4: Comparison between both methods to create the feature distribution. Both instances consist of 10,000 planning units, arranged in a 100 × 100 rectangular grid. The left figure shows a feature distribution through random number generation, the right figure shows a feature distribution through Perlin noise.

the units. Each planning unit is therefore adjacent to at most one planning unit each to the left, right, top and bottom. Each unit‘s boundaries with its neighboring units are dependent on the chosen unit width and unit height. In our case, we used a height of 50 and a width of 30.

Finally, the dependencies were also created at random. Each unit has a 25 % probability for each neighbor to be dependent on them. In a real case scenario, a unit might be dependent on non-adjacent planning units as well, but we wish to keep our data generation simple and effective. Therefore, our synthetic data only considers adjacent units for possible dependencies.

Alternatively, the data creation algorithm can use Perlin noise to create both the planning unit costs and the conservation feature distribution, i.e., how much each unit contributes to each conservation feature. Perlin noise is a random number generator commonly used for synthetic terrain generation, e.g., mountains and oceans by creating different elevation levels for each coordinate on a two-dimensional plane. Unlike simple random number generation, two adjacent coordinates will have similar elevation levels to create realistic-looking terrain. Here we use these elevation levels as numbers for conservation features and costs for our planning units. The motivation behind this alternative approach is to generate more realistic datasets, as we would expect real data to have similar values for adjacent units. In Figure 4, we display an exemplary feature distribution for both approaches. Hereinafter, we will refer to the instances as random number instances or Perlin instances, depending on the creation method used to generate these instances.

4.3 Configuring instances and tests

As mentioned in Section 1.2, Marxan features the possibility to find solutions that may not fully satisfy all conservation features in exchange for an added penalty to the solution value. There is a configuration option that controls, how much the penalty affects the total solution cost. We have the option to raise this variable to a value at which point it would be highly unlikely that the solution does not meet all target values for the conservation feature. However, this considerably increases the result value, even for solutions that satisfy all targets, which is most likely due to the way the Simulated Annealing iterates between neighboring solutions. This is why we set this value to 1, which means that most of Marxan‘s solutions will not meet all targets, but in exchange, we get better solution values that are more comparable to the solutions of our algorithm.

Furthermore, multiple parameters control the calculation effort of Marxan while searching for solutions, namely the number of iterations the Simulated Annealing algorithm performed in each run and the number of runs. Marxan lists the final result for each run and highlights the best results across all runs. We ran all instances first with 1 million iterations per run and ten runs total, then with 10 million iterations per run and 100 runs total.

When creating instances for testing, it was important to choose instance sizes comparable to real-world applications. Unfortunately, early tests proved that the algorithm takes too long to search for the optimal solution, even when it finds an almost perfect solution early on. An instance with 10,000 planning units and ten conservation features ran over 11 hours before being manually interrupted after it had not made any progress towards better solutions or lower bounds for over two hours. This resulted in a solution that was 0.0172% off the possible lower bound for an optimal solution after it had reached 0.02% difference already after 360 seconds of running, which means that the remaining 11 hours of runtime brought a negligible improvement to the solution found. Since the instance with 10,000 planning units and ten conservation features was supposed to be one of the smaller test instances, as most real-life applications use larger instances, it became clear that searching for the optimal solution for each test was impossible. Instead, we reconfigured Gurobi to use two alternatives. First, we conducted experiments in which Gurobi terminated solving after finding a solution less than 0.05 percent away from the lower bound for the optimal solution. Furthermore, we repeated these tests with a time limit instead of a gap percentage. We chose 40 seconds for our time limit as this is the time Marxan requires for its Simulated Annealing for 100 runs with 10 million iterations each.

5 Results

After introducing the theoretical background and the mathematical interpretation, we now evaluate the runtime and solution quality of our algorithm and compare our results to Marxan's solutions.
Our goal was to identify the influence of a single instance size with all other factors isolated. We created instances that only change single attributes compared to the other instances, each instance once using random number generation and once using Perlin noise. For our planning units, we used numbers between 10,000 and 1,000,000. We limited our boundary multiplier to values between $b = 0$ and $b = 100$. Since the boundary multiplier is a configuration of our algorithm and Marxan instead of a fixed attribute of our instance, we can use different boundary multipliers with the same instance. We used the same instance for all boundary tests to see how increased boundaries affect the solution for the same instance. We did not know a good upper limit for our conservation features in advance. Therefore, we started with ten conservation features and steadily increased the number per instance by 1. The goal was to continue until we reached an instance that would take over eight hours to finish solving. The first instance that surpassed this runtime had 18 conservation features. For all our different numbers of planning units, we measured the instance once without dependencies and once with dependencies included. We did not conduct any tests with Marxan regarding dependencies as Marxan does not support those. Each of these tests was conducted once with a gap percentage and once with a time limit (see Section 4.3).
All tests were run on a desktop computer with an Intel (R) Core(TM) i7-4790K CPU @ 4.00 GHz x 8 and 16 GB DDR3 RAM on Windows 10 Pro.

5.1 Solution analysis

First, we evaluate the solution quality for different instances and compare the results of our implementation with different configurations for Marxan. Some selected results from our tests are represented in Table 1 and Table 2. Tables with all solutions for all tests can be found in the Appendix.

As mentioned in Section 4.3, we tested all instances once with an acceptance gap threshold for our ILP algorithm of 0.05 percent and once with a time limit of 40 seconds to allow suboptimal solutions in exchange for a reasonable runtime. We can see that the time limit results in better solutions on most occasions, albeit with only minor differences. There are instances where 40 seconds were not enough to finish preprocessing. Those instances would have resulted in very large solution values. We adjusted the time limit for these instances to be just enough to finish preprocessing. These instances are listed in Table 1 in bold font and the number in brackets signifies the necessary amount of seconds to finish preprocessing.
We can observe multiple properties when we compare the results of our implementation and Marxan. Solutions found by Marxan had higher costs of up to 30 percent compared to the ILP.

Instance	Gap: Solution Value	≤ 0.05 % Gap to lower bound	Time limit: Solution Value	40 seconds Gap to lower bound
PU10000_F10_B0	3.726e06	0.0408%	3.725e06	0.0234%
PU100000_F10_B0	3.934e07	0.0032%	3.934e07	0.0016%
PU500000_F10_B0	1.959e08	0.0174%	1.959e08	0.0174%
PU1000000_F10_B0	3.914e08	0.0073%	**3.914e08** (67)	**0.0073**%
PU10000_F10_B0_Perlin	6.906e06	0.0181%	6.905e06	0.0128%
PU100000_F10_B0_Perlin	8.658e07	0.0198%	8.656e07	0.0021%
PU500000_F10_B0_Perlin	3.791e08	0.0029%	3.791e08	0.0005%
PU1000000_F10_B0_Perlin	7.574e08	0.0014%	7.574e08	0.0014%
PU10000_F12_B0	3.96e06	0.0478%	3.96e06	0.0337%
PU10000_F14_B0	4.234e06	0.0467%	4.233e06	0.0349%
PU10000_F16_B0	4.32e06	0.0474%	4.32e06	0.0608%
PU10000_F18_B0	4.258e06	0.0509%	4.259e06	0.0754%
PU10000_F12_B0_Perlin	8.938e06	0.0197%	8.938e06	0.0108%
PU10000_F14_B0_Perlin	7.426e06	0.0227%	7.425e06	0.0118%
PU10000_F16_B0_Perlin	6.831e06	0.0234%	6.83e06	0.0110%
PU10000_F18_B0_Perlin	6.744e06	0.0353%	6.743e06	0.0259%
PU10000_F10_B10	6.427e06	0.0386%	6.426e06	0.0202%
PU10000_F10_B20	8.611e06	0.04%	8.611e06	0.0327%
PU10000_F10_B50	1.29e07	0.0486%	1.29e07	0.0475%
PU10000_F10_B100	1.502e07	0.0499%	1.587e07	5.5063%
PU10000_F10_B10_Perlin	7.156e06	0.0403%	7.156e06	0.0352%
PU10000_F10_B20_Perlin	7.381e06	0.0420%	7.381e06	0.0365%
PU10000_F10_B50_Perlin	7.979e06	0.0388%	7.978e06	0.0330%
PU10000_F10_B100_Perlin	8.927e06	0.0405%	8.936e06	0.2866%
PU10000_F10_B0_D	6.827e06	0.0401%	6.827e06	0.0287%
PU100000_F10_B0_D	7.048e07	0.0284%	7.046e07	0.0033%
PU500000_F10_B0_D	3.519e08	0.0110%	**3.519e08** (155)	**0.0110** %
PU1000000_F10_B0_D	7.035e08	0.0127%	**7.035e08** (420)	**0.0127** %
PU10000_F10_B0_D_Perlin	6.959e06	0.0379%	6.958e06	0.0185%
PU100000_F10_B0_D_Perlin	8.686e07	0.0067%	8.686e07	0.0028%
PU500000_F10_B0_D_Perlin	3.792e08	0.0304%	**3.792e08** (112)	**0.0304** %
PU1000000_F10_B0_D_Perlin	7.576e08	0.0178%	**7.576e08** (330)	**0.0178** %

Table 1: Solutions for different instances for the ILP implementation, each once for an acceptance gap of 0.05 percent and once for a time limit of 40 seconds. The instance names are abbreviated through a unique string defining the instance sizes. The number after PU represents the number of planning units, the number after F represents the number of conservation features and the number after B represents the boundary multiplier. If "D" or "Perlin" is appended to the instance name then the instance includes dependencies or Perlin noise generation respectively. Values in bold font required more than 40 seconds to finish preprocessing, the number in the brackets signifies the required amount of seconds for those cases.

Instance	10 runs Solution Value	1M iterations Gap to ILP	100 runs Solution Value	10M iterations Gap to ILP
PU10000_F10_B0	3.831e06	2.846%	3.769e06	1.181%
PU100000_F10_B0	4.205e07	6.889%	4.074e07	3.559%
PU500000_F10_B0	2.278e08	16.284%	2.075e08	5.921%
PU1000000_F10_B0	4.746e08	21.257%	4.185e08	6.924%
PU10000_F10_B0_Perlin	7.207e06	4.374%	7.032e06	1.839%
PU100000_F10_B0_Perlin	9.604e07	10.952%	9.211e07	6.412%
PU500000_F10_B0_Perlin	4.384e08	15.642%	4.064e08	7.201%
PU1000000_F10_B0_Perlin	9.438e08	24.611%	8.324e08	9.902%
PU10000_F12_B0	4.097e06	3.460%	4.017e06	1.439%
PU10000_F14_B0	4.41e06	4.181%	4.307e06	1.748%
PU10000_F16_B0	4.509e06	4.375%	4.393e06	1.690%
PU10000_F18_B0	4.429e06	3.992%	4.340e06	1.902%
PU10000_F12_B0_Perlin	9.405e06	5.225%	9.144e06	2.305%
PU10000_F14_B0_Perlin	7.75e06	4.377%	7.564e08	1.872%
PU10000_F16_B0_Perlin	7.171e06	4.993%	6.986e06	2.284%
PU10000_F18_B0_Perlin	8.809e06	30.639%	8.505e06	26.131%
PU10000_F10_B10	6.788e06	5.633%	6.571e06	2.256%
PU10000_F10_B20	9.25e06	7.421%	8.855e06	2.834%
PU10000_F10_B50	1.445e07	12.016%	1.346e07	4.341%
PU10000_F10_B100	1.933e07	21.802%	1.615e07	1.764%
PU10000_F10_B10_Perlin	7.554e06	5.562%	7.269e06	1.579%
PU10000_F10_B20_Perlin	7.778e06	5.379%	7.493e066	1.517%
PU10000_F10_B50_Perlin	8.601e06	7.809%	8.081e06	1.291%
PU10000_F10_B100_Perlin	1.014e07	13.474%	9.038e06	1.141%

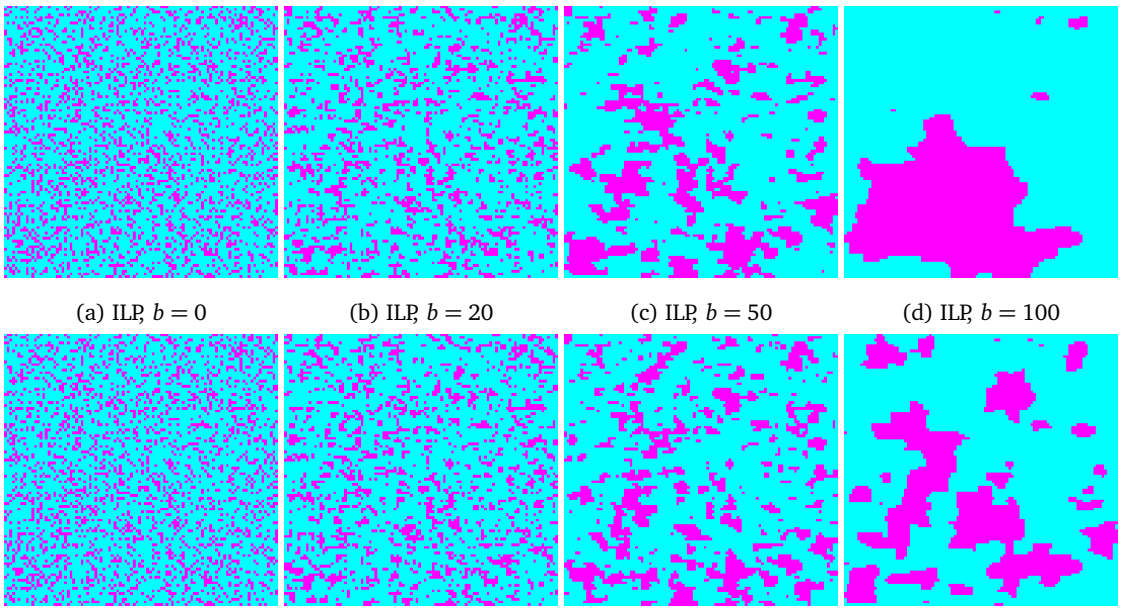
Table 2: Solutions for different instances for Marxan using ten runs and 1 million iterations per run as well as 100 runs and 10 million iterations per run. The column “Gap to ILP” shows the relative difference between the Marxan solution value and the ILP time limit solution value for the respective instance. For each instance, dependencies are excluded and the amount of features is set to ten. The abbreviations used for the instance names are the same as in Table 1.

This suggests that the ILP might be more suited to find solutions closer to the optimum. We also see that if we increase the number of runs and iterations per run for Marxan's Simulated Annealing, we find solutions closer to the optimal value at the cost of an increased runtime.

Instances with more planning units result in higher solution values. Doubling the planning units also roughly doubles the cost of the solutions, both for the random number instances and the Perlin instances. Judging by the solution value of our different test sets, the number of planning units seems to be the most important factor for increases in the solution values. If we also include dependencies while increasing the number of planning units, the solution value is increased even further, although for the Perlin instances much less than for the random number instances. It must also be mentioned that with different numbers of dependencies, we would see a different relative increase in runtime. In our synthetic data, each planning unit has a 25 percent chance for each adjacent unit to be dependent on them.

We only see a slight increase in the solution value if we keep the number of planning units constant and slowly raise the number of conservation features that must be satisfied, both for the random number instances and the Perlin instances, with some exceptions. This implies that with only minor increases in the expenses, a conservation project could achieve to protect additional conservation features.

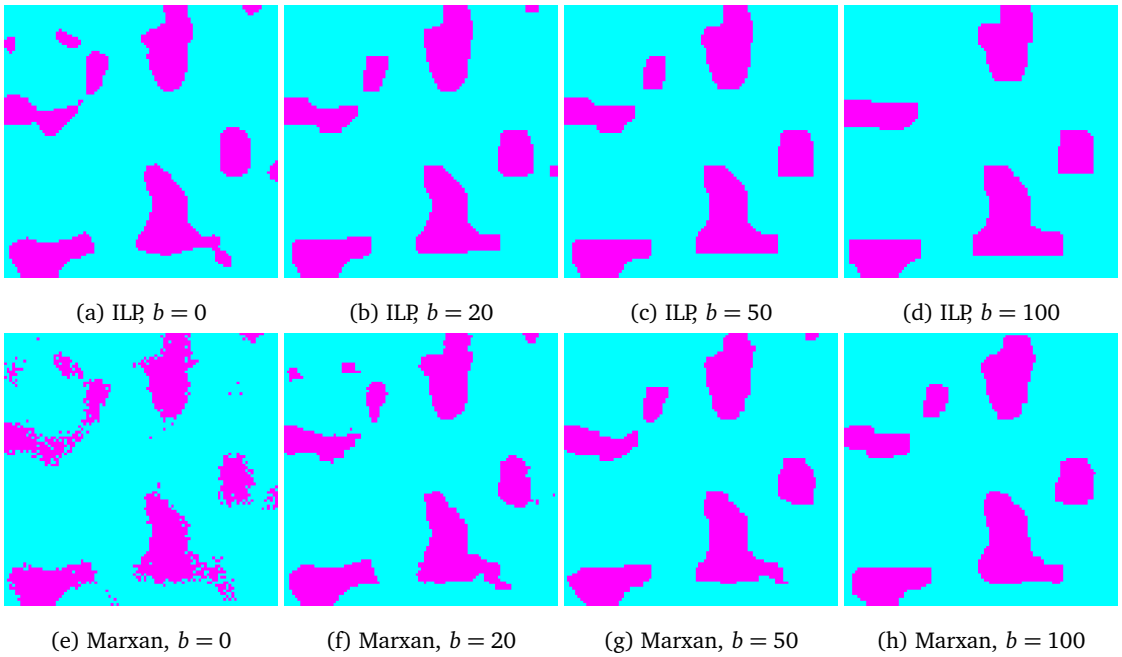
Increasing the boundary multiplier also raises the costs. However, as we increase the boundary multiplier, the increase in solution value becomes weaker. This is both visible for the random number instance and the Perlin instance. While the initial expenses for including boundaries are too high to be effective, increasing the boundary penalty even further when it is already a high value becomes much cheaper. We deduce that it is best to either exclude boundaries completely or to use high boundary values. This enables us to make efficient use of border-related cost increases.
Altering the boundary multiplier can drastically change which planning units are selected. With a higher value, the algorithm tends to find solutions where the selected planning units are more clustered, as adjacent selected units reduce the boundary length. Figure 5 shows solutions for the same instance only with different boundary penalties in which the progressive clustering is visible. Without any boundary penalty, the algorithm cherry-picks the best single planning units as it does not need to pay attention to boundaries. With rising boundary penalty, finding larger clusters becomes more important and selecting single planning units becomes less frequent. Figure 6 shows the same experiment with a Perlin instance. Here we see clusters of selected units even for lower boundary values.



(a) ILP, $b = 0$ (b) ILP, $b = 20$ (c) ILP, $b = 50$ (d) ILP, $b = 100$

(e) Marxan, $b = 0$ (f) Marxan, $b = 20$ (g) Marxan, $b = 50$ (h) Marxan, $b = 100$

Figure 5: Map of selected units for boundary multipliers b on a quadratic planning unit grid consisting of 100×100 planning units. The top row shows the results of the ILP, and the bottom row shows the results of Marxan with 100 runs and 10 million iterations per run. The purple units are selected, and the blue units are excluded from conservation. The instance was generated using simple random numbers for the conservation feature distribution.



(a) ILP, $b = 0$ (b) ILP, $b = 20$ (c) ILP, $b = 50$ (d) ILP, $b = 100$

(e) Marxan, $b = 0$ (f) Marxan, $b = 20$ (g) Marxan, $b = 50$ (h) Marxan, $b = 100$

Figure 6: Map of selected units for boundary multipliers b on a quadratic planning unit grid consisting of 100×100 planning units. The top row shows the results of the ILP, and the bottom row shows the results of Marxan with 100 runs and 10 million iterations per run. Here we used an instance generated with Perlin noise for the conservation feature distribution and the cost.

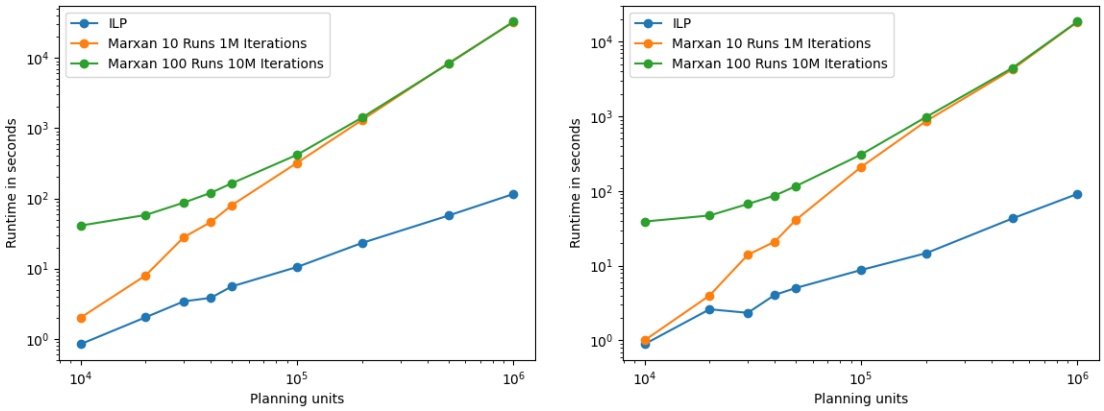


Figure 7: Comparison of runtime for different numbers of planning units. The left graph shows the random number instances, the right graph shows the Perlin instances.

5.2 Runtime analysis

Apart from solution quality, it is equally important to consider the runtime of each algorithm. For this, we used all the instances that we have also used in Table 1 and Table 2. We only used the acceptance gap percentage for runtime tests, as the time limit tests all required the same time, i.e., the time limit itself. We split these tests into four categories, each with random number instances and Perlin instances.

First, we investigate how the runtime changes with increasing numbers of planning units. This is shown in Figure 7. We see that the ILP approach is faster than Marxan in both cases for all numbers of planning units. We expected our algorithm to take longer for instances with more planning units but did not expect Marxan to increase its runtime as well since Simulated Annealing is configured to run for a fixed number of iterations. The reason for this increase in runtime is a preprocessing step Marxan does before even starting the annealing. While the annealing takes about the same time for each instance, the preprocessing time seems to be scaling with the number of planning units. Figure 7 shows that on a logarithmic scale, the curve for the runtime of Marxan with ten runs and 1 million iterations per run seems to be a straight line which highly suggests that Marxan‘s total runtime is polynomially correlated to the number of planning units. The results for 100 runs and 10 million iterations per run are very similar to the other Marxan tests, only with an offset of roughly 40 seconds, which is the increased time for the annealing due to the use of more iterations and more runs.
We identify a straight line for both graphs of the ILP as well, which suggests that the ILP runtime is also polynomially correlated to the number of planning units. The slope of this curve is in both cases less steep, i.e., the degree of the polynomial function is smaller for the ILP than for the Marxan curves, e.g., linear compared to quadratic. It must be mentioned that the runtime for our ILP is also affected by an increased time of creating the ILP, which is noticeable for instances with large numbers of planning units. We can also see that the Perlin instances behave similarly but are slightly faster than the random number instances.

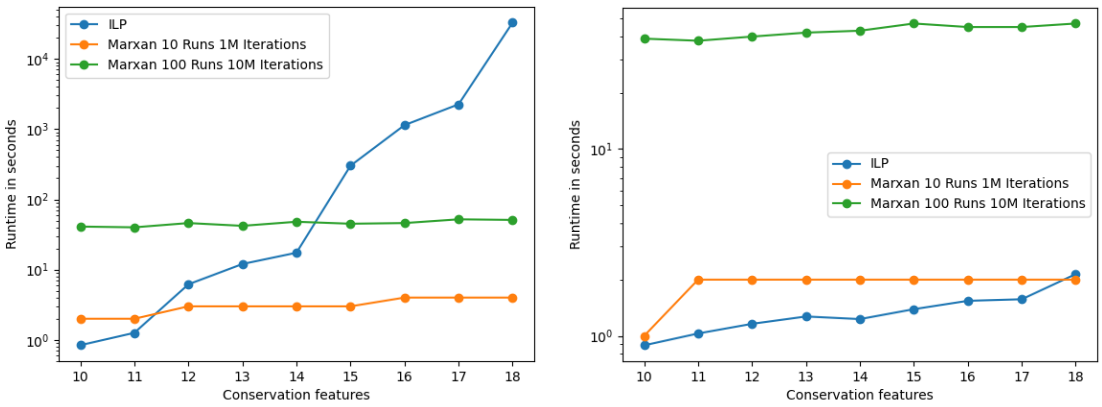


Figure 8: Comparison of runtime for different numbers of conservation features. The left graph shows the random number instances, the right graph shows the Perlin instances.

Figure 8 shows the results of our tests with different numbers of conservation features. During these tests, we noticed that different instances of the same size, i.e., the same amount of planning units, conservation features, and boundary multiplier could require drastically different solving times. For a random number instance with 18 conservation features, we conducted three tests which resulted in 32,800, 5,200, and 1,500 seconds of runtime. We found that Gurobi can solve some instances "easier" than others based on additional factors, i.e., the random distribution of the conservation features to the planning units. In some instances, Gurobi may find good solutions quickly and ignore a lot of obvious worse solutions while other very similar instances might even require several times that amount of runtime.
Due to the fluctuations, the results for this experiment, depicted in Figure 8, should not be relied on without using additional sources. Repeating this experiment might lead to different results. Still, we can notice a steep increase in runtime for the ILP measures concerning the random number instances. We can also see the significant difference in runtime for the Perlin instances. These instances are even faster than Marxan. Therefore, this experiment underlines how much the cost and feature distribution can influence how easily Gurobi can find the desired solutions. Marxan shows little to no increase in runtime for these instances at all. Here, Marxan's preprocessing time does not increase with different numbers of conservation features, and the annealing always requires the same amount of time for each instance.

Figure 9 shows a constant runtime for the Marxan results again and slow growth in runtime for an increasing boundary multiplier with a jump between $b = 50$ and $b = 60$ for our implementation for the random number instance. We assume that a higher boundary penalty makes it more complicated for Gurobi to find the best solutions and might therefore be correlated to an increase in runtime. For this specific instance, boundary multiplier values of $b = 60$ and upwards seem to be much more complicated for Gurobi to solve.
We see that, again, the Perlin instance tests terminated much earlier compared to the boundary tests of the random number instance. This is further proof of our suggestion that increases

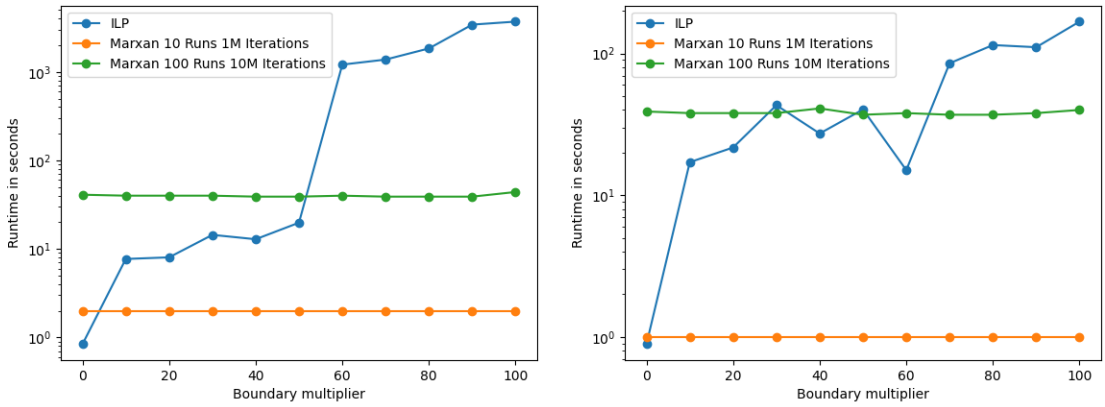


Figure 9: Comparison of runtime for different boundary multiplier values. The left graph shows the random number instance, the right graph shows the Perlin instance.

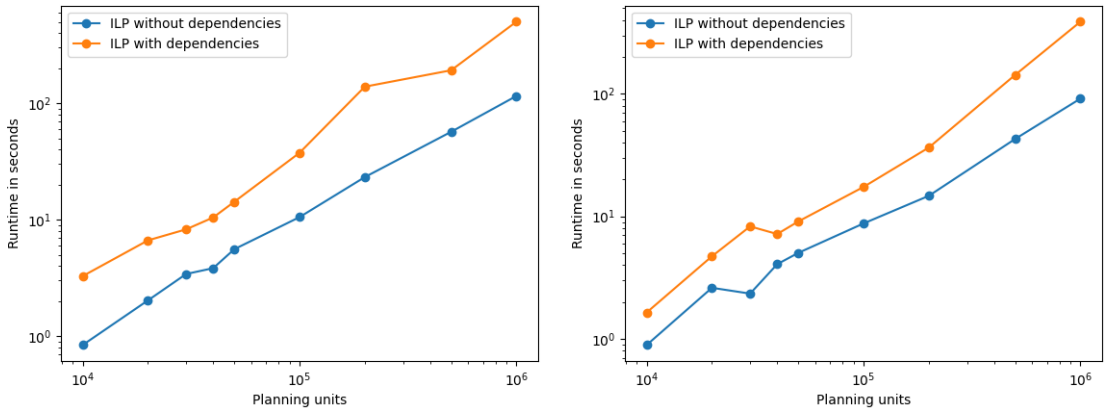


Figure 10: Comparison of runtime for instances with and without dependencies. The left graph shows the random number instances, the right graph shows the Perlin instances.

in runtime for larger boundary multipliers may be due to Gurobi being able to solve some instances easier than other similar ones. The case for $b = 0$ was already clustered, as seen in Figure 6. Therefore, here it is easy for Gurobi to find suitable solutions for larger boundary penalties, which results in lower runtimes compared to the boundary tests for random number instances.

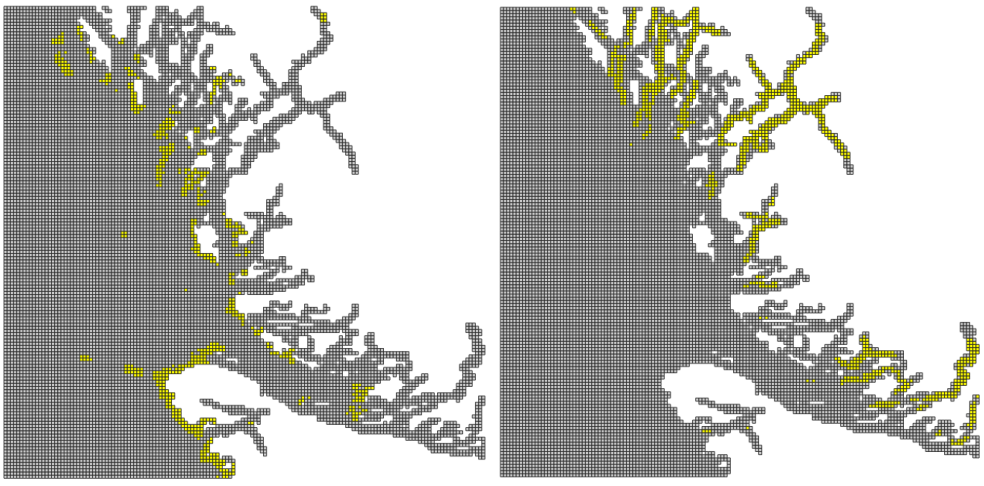
Another interesting observation is the jump in runtime when comparing the time required for $b = 0$ and $b = 10$, both for the random number instance and the Perlin instance. Our algorithm is designed to append the boundary penalty sum to the objective function and to include the z_{ij} constraints, defined in Section 3.3, only if the boundary penalty multiplier is above 0. This is done for efficiency reasons since all boundary terms would disappear regardless when multiplied by 0.

Ultimately, we see in Figure 10 that for each instance, the run with dependencies takes longer than the run without dependencies, both for the random number instances and the Perlin instances. This is expected, as dependencies add further constraints to the ILP model. Just like for the planning unit tests without dependencies, the Perlin instances terminate faster for

the case with dependencies as well.

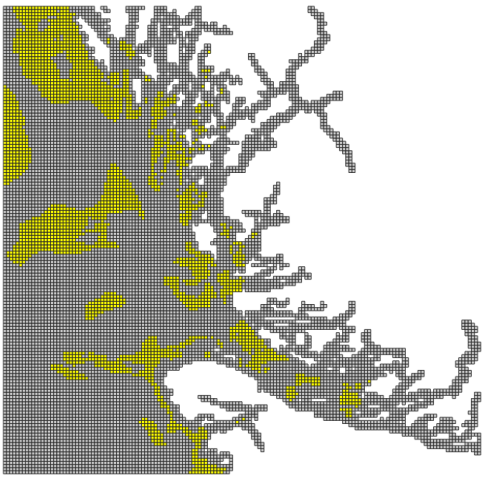
5.3 Real instances compared to synthetic instances

There are only a few instances of real conservation projects publicly available online, but we found one real dataset consisting of 12,179 planning units and 59 conservation features. This dataset is part of a project in British Columbia in Canada, which was carried out by the BCMCA organization [13]. We can use this instance to determine how realistic the synthetic Perlin instances are by comparing them to a real instance. The nature reserve in question is not rectangular. Therefore, we can not use our simple heatmap algorithm we used to create the heatmaps from the past chapters. Instead, we used the GIS software QGIS [14] to create a graphic representation of the planning units. Figure 11 and Figure 12 display the cost and some exemplary conservation features on these representations.

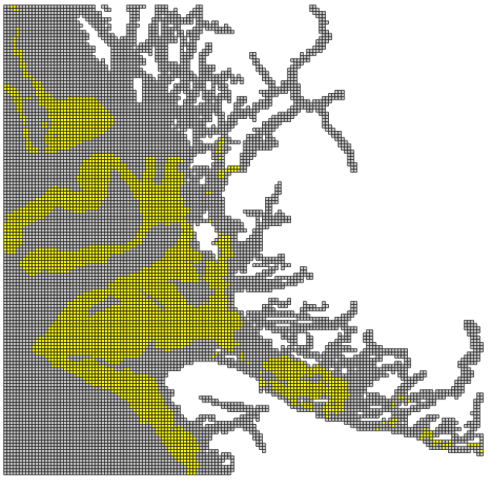


(a) Feature 2

(b) Feature 29



(c) Feature 42



(d) Feature 50

Figure 11: Feature distribution of the real instances for four different conservation features. The grey planning units do not contain the respective feature at all, the yellow planning units contain a value of above 0 for the respective feature.

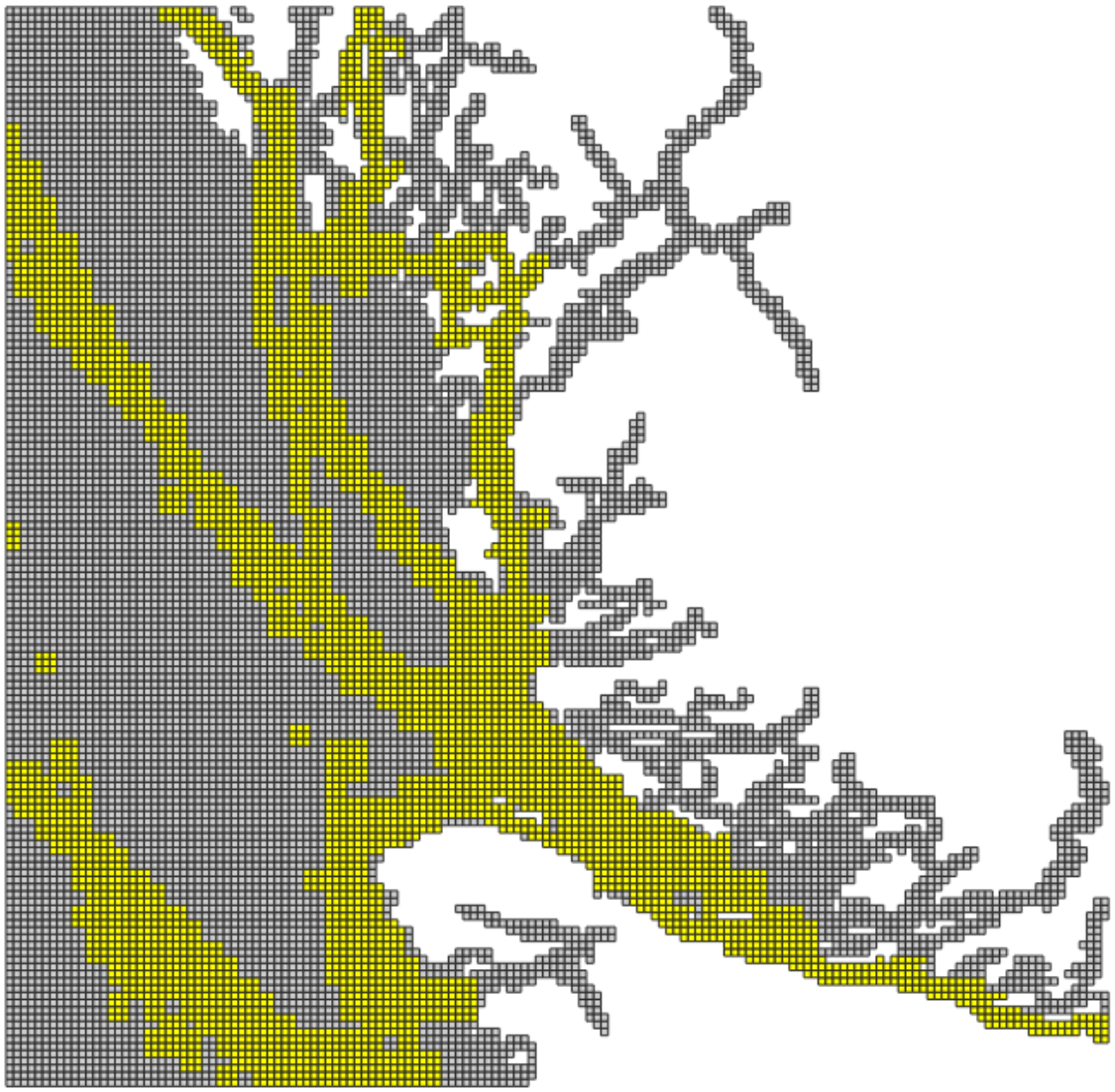


Figure 12: Cost distribution of the real instance. The grey planning units have a cost of 400, which was the lowest cost, the yellow planning units have a cost of above 400.

We can see that adjacent units indeed seem to have similar cost and feature values in real instances. This supports our theory that Perlin instances might be an improvement in realism compared to the random number instances.

We can also use this real instance to evaluate the usefulness of our synthetic data creation. For this, we created synthetic datasets of the same size, twenty random number instances, and twenty Perlin instances. We adjusted the multiplier for the cost and feature distribution so that the ILP coefficients of our synthetic datasets would be close to the ILP coefficients for the real instance since these values may influence the runtime as well. We set a time limit of 40 seconds for each of these instances to find their best solution possible. Table 3 shows the results for the real instance as well as the mean and variance for the twenty random number instances and the twenty Perlin instances.

We can see that the Perlin instances were closer to their optimum, and the random number instances were farther away from their optimum compared to the relative gap of the real

Instance	Gap to lower bound
Real instance	0.1058%
Random number generation	Mean: 0.1244%, Variance: 3.323e-05
Perlin noise generation	Mean: 0.0497%, Variance: 4.562e-05

Table 3: Comparison of a real dataset to synthetic datasets. For both the random number generation and the Perlin generation, twenty instances were run to calculate mean and variance. Each of these instances consisted of 12,179 planning units and 59 conservation features and used a time limit of 40 seconds for solving. Boundaries and dependencies were disabled.

instance to its optimum. However, these relative gaps are all fairly similar, although the data creation is kept simple. Additionally, we have a low variance for both the random number instances and the Perlin instances, which implies that different synthetic instances of the same size tend to get similarly close to their optimum. Therefore, synthetic data seems to be a meaningful alternative to real data. We can assume that the Perlin instances might be too simplified and the random number instances might be too abstract, but both form bounds for how the algorithm performs on a real instance of the same size. However, since we only have one real instance to test this, we cannot conclude this as a clear statement.

5.4 A failed concept: Multi-objective boundary penalty

One of the main issues with the boundary multiplier is how aggressively it influences the solution quality. Selecting clusters of units keeps boundary penalty low but simultaneously comes with a great increase in unit selection cost. We have to keep in mind that the unit selection cost is a real expense the conservation project has to pay if they implement the selection of the algorithm, while the boundary penalty is an imaginary cost we use to force more clustered solutions. Selecting clusters of units certainly has advantages and most likely saves money on important processes like transportation, but it is financially not as important as the planning unit selection cost. In our current model, the boundary penalty influences the unit selection cost too much. This is why we examined an alternative ILP formulation. Gurobi offers the possibility of a hierarchical multi-objective function. Our objective function, which consists of the selection cost and the boundary penalty, is split into two parts, then the ILP is solved twice. First, only the selection cost is used as our objective function and the best solution is searched that minimizes the selection cost. Next, Gurobi uses the boundary penalty as our objective function but adds a linear constraint to ensure that the selection cost may only be a certain percentage higher than the value it found earlier. We set this percentage to 10 percent. This means that our algorithm searches for a selection that minimizes the boundary penalty under the condition that the selection cost differs from the optimal selection cost of a boundary-less scenario by less than 10 percent. The algorithm is only allowed to raise the selection cost by less than 10 percent to find more clustered solutions.

Unfortunately, this promising concept yielded results that were below our expectations. Due to the added constraint, the complexity increased and we were unable to get a result within eight hours, even for low boundary penalty values. We had to manually terminate all tests after each used an entire night of solving and ended up with solutions that all had roughly 0.07 percent difference from the optimal solution. Additionally, the selections we got after manually terminating the algorithm showed little to no clustering. Allowing to deviate from the optimal selection cost by only a small percentage is not enough for the algorithm to search for larger clusters of selections, but allowing it to deviate further will make it too expensive again. Therefore, we concluded this idea to have failed. With more enhancement, it might be useful in real application projects, but in the scope of this thesis, it proved to be nothing more than an interesting concept.

6 Discussion

6.1 Evaluation of results

The results of our tests give us great insight into the performance of both our ILP and Marxan and enable us to compare the two different approaches based on the several different scenarios we investigated. Using the ILP can get us results that have a lower cost for all our tests, because, through the acceptance gap percentage, we can configure how close our results have to be to the optimal solution. Our ILP would even be able to find an optimal solution. However, this would take too much time for the instances we investigated. Marxan sometimes finds solutions close to the optimum, but especially for larger solution values, the relative difference becomes greater. With more iterations for the Simulated Annealing, Marxan finds better solutions, but even then, these solutions are inferior to the solutions found by the ILP. If we only care about solution quality, the ILP approach would almost always be the best choice.

However, in reality, we would most certainly also care about finding the solution fast. We first compare Marxan to the ILP formulation with the acceptance gap percentage. For our random number instances, Marxan was faster than the ILP when we increased the number of conservation features or the boundary multiplier. Our ILP performed better for the Perlin instances, sometimes even faster than Marxan. This proves that Gurobi can solve some instances easier than others based on the cost and feature distribution. Through the comparison to a real dataset, we evaluated the performance for a real instance to be better than the performance of random number instances but worse than the performance of Perlin instances that both use the same amount of planning units and conservation features. Therefore, a limitation of our ILP is the potential for increases in runtime for real instances with many conservation features or large boundary penalties, i.e., the ILP with an acceptance gap is only of limited suitability for some instances.

Comparing our results with the conclusion of Beyer et al. in their paper [2], we see similarities and differences. We agree with them about their assessment of the ILP‘s superior solution quality and the advantage of being able to estimate how close the found solution is to the optimum. However, our results include several investigations they did not conduct. These additional results demonstrate that the runtime of the ILP for the gap percentage runs is not always better than Marxan, for example for increasing numbers of conservation features.

We also investigated what happens to the solution if we remove the acceptance gap, replace it with a time limit, and let each ILP instance run for 40 seconds, which is the average time Marxan needs for its solving. Even then, our ILP still returned better results, with the few exceptions in which additional time was required to even finish preprocessing. Therefore, the ILP can be considered to be more time-effective than Marxan. For most tests, we also need to consider a few seconds of setup time for our ILP but this setup time is acceptable for in-

stances with realistic numbers of planning units and conservation features. Marxan, on the other hand, requires hours of preprocessing time for large numbers of planning units, rendering it unsuitable for such instances. Meanwhile, the ILP with a time limit is suitable for all evaluated scenarios, especially for the Perlin instances.

Comparing the acceptance gap approach and the time limit approach, we find that for the time limit approach, we get better results for most of our instances. For those instances in which the gap termination did result in better solutions, the ILP spent several hours to only reach a slightly better solution compared to the time limit solutions, which each took 40 seconds. We also have the advantage to know the calculation time in advance. If a conservation project requires the algorithm to find solutions within a certain percentage to the optimum, then the acceptance gap approach should be used. In all other cases, the time limit approach is more suitable.

Both the ILP and Marxan also have their limitations. One such problem is the complexity of the input data, a lot of different information is required for a complete instance, and realistically sized instances consist of tens of thousands or even millions of variables. This is noticeable in the ILP setup time and Marxan's preprocessing time. Additionally, due to this complexity, measuring real geological data is only possible as a member of an official conservation project, and there is only very limited real data available online. This increases the demand for synthetic data creation. Due to the complexity, it is not trivial for synthetic data to approximate realistic data.

Throughout the thesis, we considered the "best" solution to be the solution that minimizes the cost while satisfying all demands to the conservation features and dependencies. In reality, projects might prefer to move away from the cheapest solution to reach additional milestones such as conserving another conservation feature that was not listed in the instance or selecting units that are more expensive but still preferred over the cheapest solution due to their location, e.g., because they are better accessible. This can also be seen as a further limitation to our implementation. The ILP can only work with what is given through the instance, but ultimately, it is humans who decide where conservation actions should be implemented. The algorithm should always only serve as a decision-supporting tool and should never be the only deciding factor.

After comparing Marxan with the ILP as well as both ILP-solving approaches, we can now also take a look at some additional observations. We saw that the Perlin instances resulted in considerably higher solution values. This can be explained by overlaps between patches of high feature concentration and patches of high cost. Since we need to satisfy 30 percent of each conservation feature, we would be forced to select some rather expensive planning units.

For higher boundary penalties, we get solutions in which the selected units form clusters. Improving our data creation through Perlin noise gave us clusters in the solution for much smaller boundary multipliers compared to random number instances. The reason for this remarkably early clustering is the pattern of the Perlin noise distribution. There are larger areas that have both similar conservation feature values and costs. To find efficient solutions, we focus on finding patches in which areas with high conservation feature density and areas with low cost density overlap. In these larger patches, several planning units get selected. Therefore, our selection looks clustered, even for lower boundary penalties. Higher boundary multipliers do not change much on the selection of Perlin instances, therefore, the solution value does not increase much for higher boundary penalties.

6.2 Future work

With our examinations, we only investigated a few possible scenarios. Several other aspects may be changed in our implementation or added from scratch, which would give us additional insight into the topic and might even enable completely new future theses.

Since real, publicly available instances are very limited, synthetic data creation is a crucial step in the development of such an algorithm. The data should test the algorithm for its correctness and give us an idea of what to expect from solution value and runtime for real data instances. We made the first step towards more realistic data with the Perlin noise. However, there is much more that might be done to generate better data, e.g., even more realistic cost and conservation feature creation. In reality, the different conservation features could also influence each other, e.g., in areas with large numbers of coral reefs, there would be few trees. We could have conservation features that go well together and some that are not compatible with each other. Such feature dependencies are not supported by our current data creation. We also strictly limited ourselves to synthetic data in which the planning units are rectangles in a rectangular nature reserve. Different unit shapes and nature reserve shapes might influence the boundary calculations.

In our thesis, we implemented boundaries as a way to find solutions in which the selected planning units form clusters. This is done by introducing a penalty for adjacent units that are not both selected or excluded, resulting in the algorithm minimizing the frequency of such occurrences. Alternatively, we could try to minimize the total area that contains all selected planning units, hence increasing the compactness of our selection. Such solutions would not necessarily be clustered, but all selected planning units would be close to each other instead of spreading out across the entire nature reserve. In their paper, Beyer et al. mention several different mathematical ways to implement compactness, e.g., by expanding the objective function through the sum of all euclidean distances between any two planning units [2].

We also did not investigate how different the runtime of our ILP might be when we make changes to our acceptance threshold. The choice of the acceptance gap defines the runtime for our ILP. Varying this gap may help to find the right balance between good results and a fast runtime.

We had to abandon our experiments regarding the hierarchical multi-objective function due to the high runtime and the disappointing results. However, we still believe that this implementation could be made useful with the right configurations or computing power.

7 Conclusion

In this thesis, we implemented an algorithm based on ILP to solve the optimization problem of finding the best selection of planning units that minimizes cost while satisfying other criteria. Our goal was to investigate whether our implementation can compete with the established software Marxan.
We judge our algorithm to be an adequate alternative to Marxan based on our results, with the time limit being even more efficient than the acceptance gap. Our implementation consistently returned better solutions. For the gap termination tests, some instances with more conservation features or higher boundary penalties significantly increased the runtime of the algorithm. However, for the Perlin noise instances, the runtime of those test cases got much shorter. Based on one exemplary real instance, we assume that the performance of real projects would be better than the performance of random number instances but worse than the performance of Perlin noise instances. Therefore, both the random number instances and the Perlin instances are useful to form lower and upper bounds for the performance of a real instance of the same size.
If the ILP uses the same runtime as Marxan, it still consistently finds better results than Marxan. Therefore, we consider the ILP to be a more efficient algorithm. Through additional criteria, we observed additional insights, such as more clustered selections for an increase in the boundary penalty or altered solution values when dependencies were included. Both these additional criteria come at the cost of an increase in runtime.

References

[1] Christopher R. Margules and Robert L. Pressey. “Systematic conservating planning”. eng. In: *Nature* 405.6783 (May 2000), pp. 243–253. ISSN: 1476-4687. DOI: 10.1038/35012251. URL: https://doi.org/10.1038/35012251.

[2] Hawthorne L. Beyer et al. “Solving conservation planning problems with integer linear programming”. In: *Ecological Modelling* 328 (2016), pp. 14–22. ISSN: 0304-3800. DOI: https://doi.org/10.1016/j.ecolmodel.2016.02.005. URL: https://www.sciencedirect.com/science/article/pii/S0304380016300217.

[3] Graham I.H. Kerley et al. “Options for the conservation of large and medium-sized mammals in the Cape Floristic Region hotspot, South Africa”. In: *Biological Conservation* 112.1 (2003). Conservation Planning in the Cape Floristic Region, pp. 169–190. ISSN: 0006-3207. DOI: https://doi.org/10.1016/S0006-3207(02)00426-3. URL: https://www.sciencedirect.com/science/article/pii/S0006320702004263.

[4] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual*. 2022. URL: https://www.gurobi.com (visited on 09/06/2022).

[5] Ian R. Ball, Hugh P. Possingham, and Matthew E. Watts. “Marxan and relatives: Software for spatial conservation prioritisation.” In: *Spatial conservation prioritisation: Quantitative methods and computational tools.* Ed. by Atte Moilanen, Kerrie A. Wilson, and Hugh P. Possingham. Oxford University Press, Oxford, UK., 2009. Chap. 14, pp. 185–195.

[6] Robert L. Pressey et al. “Conservation planning in a changing world”. In: *Trends in Ecology & Evolution* 22.11 (2007), pp. 583–592. ISSN: 0169-5347. DOI: https://doi.org/10.1016/j.tree.2007.10.001. URL: https://www.sciencedirect.com/science/article/pii/S0169534707002807.

[7] Richard M. Karp. “Reducibility among Combinatorial Problems”. In: *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations, held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department*. Ed. by Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger. Boston, MA: Springer US, 1972, pp. 85–103. ISBN: 978-1-4684-2001-2. DOI: 10.1007/978-1-4684-2001-2_9. URL: https://doi.org/10.1007/978-1-4684-2001-2_9.

[8] Jeffrey D. Camm et al. “A note on optimal algorithms for reserve site selection”. In: *Biological Conservation* 78.3 (1996), pp. 353–355. ISSN: 0006-3207. DOI: https://doi.org/10.1016/0006-3207(95)00132-8. URL: https://www.sciencedirect.com/science/article/pii/0006320795001328.

[9] Richard L. Church, David M. Stoms, and Frank W. Davis. “Reserve selection as a maximal covering location problem”. In: *Biological Conservation* 76.2 (1996), pp. 105–112. ISSN: 0006-3207. DOI: https://doi.org/10.1016/0006-3207(95)00102-6. URL: https://www.sciencedirect.com/science/article/pii/0006320795001026.

[10] Jirí Matousek and Bernd Gärtner. *Understanding and Using Linear Programming*. Springer Berlin, Heidelberg, 2007. ISBN: 978-3-540-30717-4. DOI: 10.1007/978-3-540-30717-4. URL: https://doi.org/10.1007/978-3-540-30717-4.

[11] Kathryn A. Dowsland and Jonathan M. Thompson. “Simulated Annealing”. In: *Handbook of Natural Computing*. Ed. by Grzegorz Rozenberg, Thomas Bäck, and Joost N. Kok. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1623–1655. ISBN: 978-3-540-92910-9. DOI: 10.1007/978-3-540-92910-9_49. URL: https://doi.org/10.1007/978-3-540-92910-9_49.

[12] Norma Serra-Sogas et al. *Marxan User Manual: For Marxan version 2.43 and above*. The Nature Conservancy (TNC), Arlington, Virginia, United States, Pacific Marine Analysis, and Research Association (PacMARA), Victoria, British Columbia, Canada., 2020.

[13] British Columbia Marine Conservation Analysis. *Marine Atlas of Pacific Canada: a product of the British Columbia Marine Conservation Analysis (BCMCA)*. British Columbia Marine Conservation Analysis, 2022. URL: https://bcmca.ca/ (visited on 09/06/2022).

[14] QGIS.org. *QGIS Geographic Information System*. QGIS Association, 2022. URL: https://www.qgis.org (visited on 09/06/2022).

A Full tables of solution values

ILP solutions

Instance	Gap: Solution Value	≤ 0.05 % Gap to lower bound	Time limit: Solution Value	40 seconds Gap to lower bound
PU10000_F10_B0	3.726e06	0.0408%	3.725e06	0.0234%
PU20000_F10_B0	7.575e06	0.0179%	7.574e06	0.0102%
PU30000_F10_B0	1.145e07	0.010%	1.145e07	0.0099%
PU40000_F10_B0	1.564e07	0.0065%	1.564e07	0.0065%
PU50000_F10_B0	1.956e07	0.0076%	1.956e07	0.0044%
PU100000_F10_B0	3.934e07	0.0032%	3.934e07	0.0016%
PU200000_F10_B0	7.777e07	0.0420%	7.774e07	0.0011%
PU500000_F10_B0	1.959e08	0.0174%	1.959e08	0.0174%
PU1000000_F10_B0	3.914e08	0.0073%	**3.914e08** (67)	**0.0073**%
PU10000_F10_B0_Perlin	6.906e06	0.0181%	6.905e06	0.0128%
PU20000_F10_B0_Perlin	1.472e07	0.0095%	1.472e07	0.0059%
PU30000_F10_B0_Perlin	2.010e07	0.0383%	2.010e07	0.0040%
PU40000_F10_B0_Perlin	2.261e07	0.0265%	2.260e07	0.0037%
PU50000_F10_B0_Perlin	2.578e07	0.0491%	2.577e07	0.0037%
PU100000_F10_B0_Perlin	8.658e07	0.0198%	8.656e07	0.0021%
PU200000_F10_B0_Perlin	1.453e08	0.0088%	1.453e08	0.0012%
PU500000_F10_B0_Perlin	3.791e08	0.0029%	3.791e08	0.0005%
PU1000000_F10_B0_Perlin	7.574e08	0.0014%	7.574e08	0.0014%
PU10000_F11_B0	3.913e06	0.0486%	3.913e06	0.0303%
PU10000_F12_B0	3.96e06	0.0478%	3.96e06	0.0337%
PU10000_F13_B0	4.100e06	0.0475%	4.100e06	0.0399%
PU10000_F14_B0	4.234e06	0.0467%	4.233e06	0.0349%
PU10000_F15_B0	4.137e06	0.0500%	4.137e06	0.0512%
PU10000_F16_B0	4.32e06	0.0474%	4.32e06	0.0608%
PU10000_F17_B0	4.331e06	0.0482%	4.332e06	0.0655%
PU10000_F18_B0	4.258e06	0.0509%	4.259e06	0.0754%
PU10000_F11_B0_Perlin	6.762e06	0.0182%	6.762e06	0.0107%
PU10000_F12_B0_Perlin	8.938e06	0.0197%	8.938e06	0.0108%
PU10000_F13_B0_Perlin	7.756e06	0.0327%	7.755e06	0.0190%
PU10000_F14_B0_Perlin	7.426e06	0.0227%	7.425e06	0.0118%
PU10000_F15_B0_Perlin	8.510e06	0.0390%	8.509e06	0.0195%
PU10000_F16_B0_Perlin	6.831e06	0.0234%	6.83e06	0.0110%
PU10000_F17_B0_Perlin	7.451e06	0.0377%	7.450e06	0.0199%
PU10000_F18_B0_Perlin	6.744e06	0.0353%	6.743e06	0.0259%

Table 4: ILP table containing all measurements for the planning units and conservation features

	Gap:	≤ 0.05 %	Time limit:	40 seconds
Instance	Solution Value	Gap to lower bound	Solution Value	Gap to lower bound
PU10000_F10_B10	6.427e06	0.0386%	6.426e06	0.0202%
PU10000_F10_B20	8.611e06	0.0400%	8.611e06	0.0327%
PU10000_F10_B30	1.038e07	0.0431%	1.038e07	0.0431%
PU10000_F10_B40	1.180e07	0.0401%	1.180e07	0.0401%
PU10000_F10_B50	1.290e07	0.0486%	1.290e07	0.0475%
PU10000_F10_B60	1.367e07	0.0456%	1.367e07	0.0628%
PU10000_F10_B70	1.422e07	0.0492%	1.423e07	0.1296%
PU10000_F10_B80	1.460e07	0.0500%	1.461e07	0.1298%
PU10000_F10_B90	1.485e07	0.0415%	1.577e07	5.9072%
PU10000_F10_B100	1.502e07	0.0499%	1.587e07	5.5063%
PU10000_F10_B10_Perlin	7.156e06	0.0403%	7.156e06	0.0352%
PU10000_F10_B20_Perlin	7.381e06	0.0420%	7.381e06	0.0365%
PU10000_F10_B30_Perlin	7.586e06	0.0421%	7.586e06	0.0421%
PU10000_F10_B40_Perlin	7.784e06	0.0476%	7.782e06	0.0209%
PU10000_F10_B50_Perlin	7.979e06	0.0388%	7.978e06	0.0330%
PU10000_F10_B60_Perlin	8.173e06	0.0434%	8.171e06	0.0121%
PU10000_F10_B70_Perlin	8.365e06	0.0498%	8.366e06	0.0845%
PU10000_F10_B80_Perlin	8.555e06	0.0476%	8.557e06	0.1343%
PU10000_F10_B90_Perlin	8.743e06	0.0481%	8.744e06	0.1535%
PU10000_F10_B100_Perlin	8.927e06	0.0405%	8.936e06	0.2866%
PU10000_F10_B0_D	6.827e06	0.0401%	6.827e06	0.0287%
PU20000_F10_B0_D	1.365e07	0.0137%	1.365e07	0.0122%
PU30000_F10_B0_D	2.083e07	0.0087%	2.083e07	0.0086%
PU40000_F10_B0_D	2.815e07	0.0087%	2.815e07	0.0075%
PU50000_F10_B0_D	3.521e07	0.0073%	3.521e07	0.0068%
PU100000_F10_B0_D	7.048e07	0.0284%	7.046e07	0.0033%
PU200000_F10_B0_D	1.408e08	0.0448%	**1.407e08** (127)	**0.0109**%
PU500000_F10_B0_D	3.519e08	0.0110%	**3.519e08** (155)	**0.0110** %
PU1000000_F10_B0_D	7.035e08	0.0127%	**7.035e08** (420)	**0.0127** %
PU10000_F10_B0_D_Perlin	6.959e06	0.0379%	6.958e06	0.0185%
PU20000_F10_B0_D_Perlin	1.478e07	0.0145%	1.478e07	0.0114%
PU30000_F10_B0_D_Perlin	2.016e07	0.0087%	2.016e07	0.0046%
PU40000_F10_B0_D_Perlin	2.274e07	0.0061%	2.274e07	0.0030%
PU50000_F10_B0_D_Perlin	2.593e07	0.0102%	2.593e07	0.0082%
PU100000_F10_B0_D_Perlin	8.686e07	0.0067%	8.686e07	0.0028%
PU200000_F10_B0_D_Perlin	1.455e08	0.0070%	1.455e08	0.0019%
PU500000_F10_B0_D_Perlin	3.792e08	0.0304%	**3.792e08** (112)	**0.0304** %
PU1000000_F10_B0_D_Perlin	7.576e08	0.0178%	**7.576e08** (330)	**0.0178** %

Table 5: ILP table containing all measurements for the boundary multiplier and dependencies

Marxan solutions

Instance	10 runs Solution Value	1M iterations Gap to ILP	100 runs Solution Value	10M iterations Gap to ILP
PU10000_F10_B0	3.831e06	2.846%	3.769e06	1.181%
PU20000_F10_B0	7.909e06	4.423%	7.693e06	1.571%
PU30000_F10_B0	1.197e07	4.541%	1.166e07	1.834%
PU40000_F10_B0	1.648e07	5.731%	1.600e07	2.302%
PU50000_F10_B0	2.063e07	5.470%	2.002e07	2.352%
PU100000_F10_B0	4.205e07	6.889%	4.074e07	3.559%
PU200000_F10_B0	8.500e07	9.339%	8.131e07	4.592%
PU500000_F10_B0	2.278e08	16.284%	2.075e08	5.921%
PU1000000_F10_B0	4.746e08	21.257%	4.185e08	6.924%
PU10000_F10_B0_Perlin	7.207e06	4.374%	7.032e06	1.839%
PU20000_F10_B0_Perlin	1.591e07	8.084%	1.521e07	3.329%
PU30000_F10_B0_Perlin	2.195e07	9.204%	2.094e07	4.179%
PU40000_F10_B0_Perlin	2.396e07	6.018%	2.328e07	3.009%
PU50000_F10_B0_Perlin	2.697e07	4.657%	2.632e07	2.134%
PU100000_F10_B0_Perlin	9.604e07	10.952%	9.211e07	6.412%
PU200000_F10_B0_Perlin	1.625e08	11.838%	1.537e08	5.781%
PU500000_F10_B0_Perlin	4.384e08	15.642%	4.064e08	7.201%
PU1000000_F10_B0_Perlin	9.438e08	24.611%	8.324e08	9.902%
PU10000_F11_B0	4.048e06	3.450%	3.964e06	1.303%
PU10000_F12_B0	4.097e06	3.460%	4.017e06	1.439%
PU10000_F13_B0	4.249e06	3.634%	4.164e06	1.561%
PU10000_F14_B0	4.410e06	4.181%	4.307e06	1.748%
PU10000_F15_B0	4.287e06	3.626%	4.206e06	1.668%
PU10000_F16_B0	4.509e06	4.375%	4.393e06	1.690%
PU10000_F17_B0	4.510e06	4.109%	4.408e06	1.754%
PU10000_F18_B0	4.429e06	3.992%	4.340e06	1.902%
PU10000_F11_B0_Perlin	7.030e06	3.963%	6.866e06	1.538%
PU10000_F12_B0_Perlin	9.405e06	5.225%	9.144e06	2.305%
PU10000_F13_B0_Perlin	8.219e06	5.983%	7.964e06	2.695%
PU10000_F14_B0_Perlin	7.750e06	4.377%	7.564e08	1.872%
PU10000_F15_B0_Perlin	9.155e06	7.592%	8.838e06	3.866%
PU10000_F16_B0_Perlin	7.171e06	4.993%	6.986e06	2.284%
PU10000_F17_B0_Perlin	7.889e06	5.893%	7.646e06	2.631%
PU10000_F18_B0_Perlin	8.809e06	30.639%	8.505e06	26.131%

Table 6: Marxan table containing all measurements for the planning units and conservation features

Instance	10 runs Solution Value	1M iterations Gap to ILP	100 runs Solution Value	10M iterations Gap to ILP
PU10000_F10_B10	6.788e06	5.633%	6.571e06	2.256%
PU10000_F10_B20	9.250e06	7.421%	8.855e06	2.834%
PU10000_F10_B30	1.127e07	8.574%	1.073e07	3.372%
PU10000_F10_B40	1.294e07	9.661%	1.221e07	3.475%
PU10000_F10_B50	1.445e07	12.016%	1.346e07	4.341%
PU10000_F10_B60	1.581e07	15.655%	1.432e07	4.755%
PU10000_F10_B70	1.687e07	18.552%	1.497e07	5.200%
PU10000_F10_B80	1.757e07	20.260%	1.548e07	5.955%
PU10000_F10_B90	1.846e07	17.058%	1.603e07	1.649%
PU10000_F10_B100	1.933e07	21.802%	1.615e07	1.764%
PU10000_F10_B10_Perlin	7.554e06	5.562%	7.269e06	1.579%
PU10000_F10_B20_Perlin	7.778e06	5.379%	7.493e066	1.517%
PU10000_F10_B30_Perlin	8.042e06	6.011%	7.673e06	1.147%
PU10000_F10_B40_Perlin	8.293e06	6.566%	7.893e06	1.426%
PU10000_F10_B50_Perlin	8.601e06	7.809%	8.081e06	1.291%
PU10000_F10_B60_Perlin	8.870e06	8.555%	8.283e06	1.371%
PU10000_F10_B70_Perlin	9.213e06	10.124%	8.467e06	1.207%
PU10000_F10_B80_Perlin	9.380e06	9.618%	8.661e06	1.215%
PU10000_F10_B90_Perlin	9.686e06	10.773%	8.846e06	1.167%
PU10000_F10_B100_Perlin	1.014e07	13.474%	9.038e06	1.141%

Table 7: Marxan table containing all measurements for the boundary multiplier