University of Cape Town

Optimisation

Garden by Numbers
Designing the Optimal Garden

Author:
Andomei Smit

Student Number:
SMTAND051

April 20, 2025

Contents

1 Introduction

Designing the perfect garden is usually a deeply personal endeavour. The average gardener or landscape architect would much rather consult an encyclopedia of local fauna and flora to determine which plants should feature in their design than rely on optimisation models to make these choices for them. However, there is much to be gained from using these models, making them worth exploring.

Humans are limited in their ability to solve complex problems that involve many unknowns. In a garden, the placement of plants to design an optimal garden is by no means independent of the plants in the rest of the garden. In fact, there are clear design principles that will determine if a plant will contribute or subtract from the overall aesthetic of the garden. When considering the commercialisation of garden design (for example designing public parks or large green spaces for private clients), the optimal garden could mean saving the client a substantial amount of money.

Many more aspects of garden design could benefit from optimisation techniques. Think of a landscaper who has various projects running at the same time and a limited number of workers with varying skill sets to allocate to these projects. Or even simply optimising the selection of plants to have the biggest visual impact, while using plants that are low maintenance and water-wise. The possibilities are endless!

This project will employ various optimisation models to determine what the optimal garden design is in terms of aesthetic appeal. It will use Simulated Annealing and Genetic Algorithms to determine which plants should be planted in different positions in a garden. Thereafter, it will use Mixed Integer Linear Programming to determine what the maximal visual impact is that can be achieved with a set budget. Finally, Multi-objective Linear Programming will be used to simultaneously maximise immediate visual interest, minimise the cost and minimise the plant care effort needed to maintain the plants in the garden.

1.1 Defining the "optimal" garden

The idea of defining "optimal" when it comes to garden design can be very subjective. Thus for the purpose of this project, the basic design principles of

1. Colour
2. Balance and Symmetry
3. Seasonal Variation

Optimality in terms of **Colour** is defined here as having mostly analogous colours

(those close to one another on the colour wheel) with a few complementary colours (those opposite one another on the colour wheel) for some contrast. Figure 1a is an excellent example of this.

(a) An example of a garden with mostly analogous colours (greens and purples) with a few flowers in red to add contrast.

(b) A beautiful example of balance and symmetry in a garden where the plant colours, heights and categories are mirrored around a central vertical line.

Figure 1: Two images showcasing the garden design principles of **Colour** and **Balance and Symmetry**.

Optimality in terms of **Balance and Symmetry** means trying to find a solution for a garden layout that is mirrored to some degree around central lines. Elements that can be mirrored are height of plants, colours and the type of plant or plant category (for example a ground cover, rose or non-flowering plant like an evergreen). Finally, the optimal garden in terms of **Seasonal Variation** simply means that there is visual interest throughout the year in terms of flowering plants and evergreens (plants that do not lose their leaves in winter).

These three elements will be combined into a single **Aesthetic Score** that will be used as the cost function to be maximized on some of the optimisation models. Section 1.5 below will explain in great detail how these three elements will be quantified in this project.

1.2 Defining a basic Garden Grid

To simplify the project, it will use a set garden grid as shown in Figure 2. The garden is a rectangle of 8 meters by 4 meters. The garden is divided into 128 blocks, each of 0.5 by 0.5 meters. All the non-white blocks represent unavailable blocks that already have something placed on it. For example, the blocks in yellow represent a

seating area. The remaining open blocks are where plants will be placed to try to find the optimal plant placement to maximize the garden aesthetic.

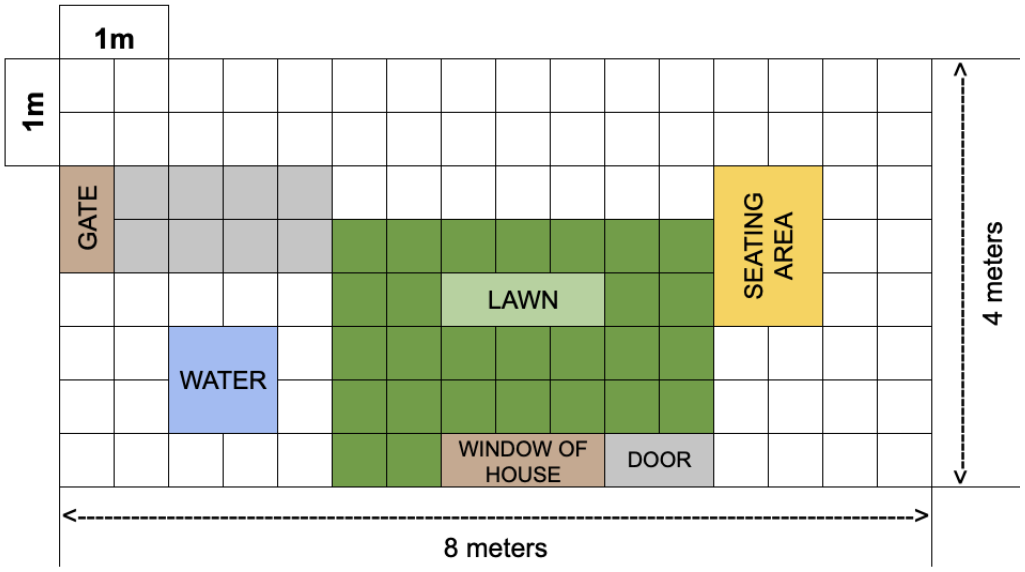


Figure 2: The basic garden design grid. The area in grey is a garden path. All the white blocks are available positions where plants can be placed. Each block is a square of 50 × 50 centimeters. In total there are 73 blocks where plants can be placed.

1.3 Defining some constraints

In this garden design there are three constraints. These are

1. The water feature must be visible. The water feature will be assumed to be 1 meter high, thus any plant around it must be 90 centimeters or less high at maturity.
2. The number of unique plants used in the garden, k must be $15 \leq k \leq 25$.
3. All plants must be visible from specific vantage points as defined in Figure 3 by the red arrows. In other words, taller plants cannot be placed in front of shorter plants, hiding them completely.

1.4 Description of the plant data

The data that will be used for this project is a list of 50 plants that are common to a typical cottage garden-style that can be found in South Africa. These plants are

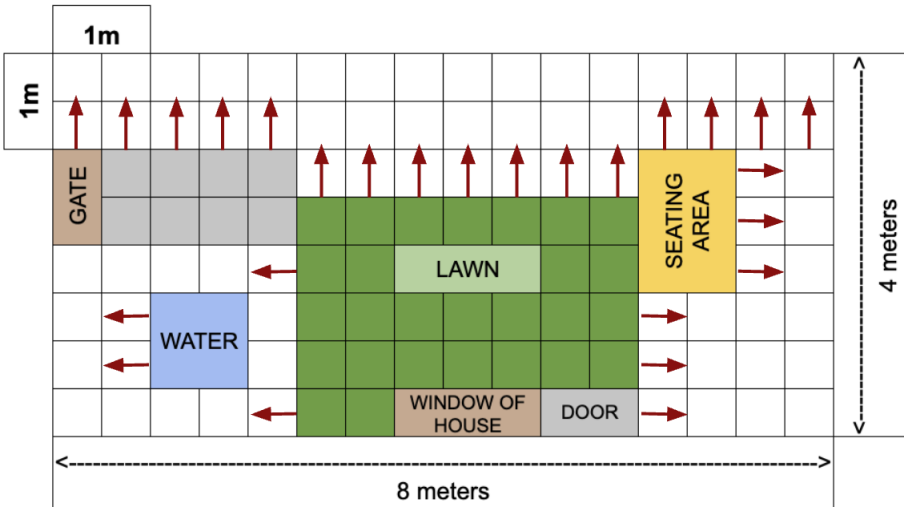


Figure 3: The basic garden grid outline showing in which direction the heights of the plants will be ordered to ensure the plants are visible from that particular vantage points. The two blocks to the right of the water feature are not of concern as another constraint will ensure that the water feature is visible.

listed in Section **??**. The following characteristics of each plant is captured in the data:

1. Scientific Name
2. Common Name
3. Description
4. Category (the 5 categories are: Perennials and Shrubs, Roses and Traditional Cottage Flowers, Bulbs and Seasonal Flowers, Ground Covers and Edging Plants, Foliage and Textural Interest)
5. Maturity Height
6. Cost of the plant when Small, Medium and Large
7. Flower Colour Hue (numerical representation of the flower colour)
8. The flowering time of the plant represented by 12 binary variables, one for each month of the year
9. The visual impact a plant has at three different sizes: Small, Medium and Large

This is the data that will be used to calculate the Aesthetic Score and do the

optimisation.

1.5 Key optimisation functions

Both the Simulated Annealing (SA) Algorithm and the Genetic Algorithm (GA) make use of some key functions. These will be explained in this section. The code for these functions can be found on GitHub. Each function is listed and briefly explained below.

generate_initial_sol: a function that will create an initial solution by sampling random plants from the plants data to fill an empty grid. The empty grid is made up of a 8 by 16 matrix representing the blocks in the garden grid from Figure 2. All the coloured blocks are marked as being unavailable and will not be filled with plants.

Since the initial solution may violate some of the constraints in Section 1.3, the solution first needs to be made viable. This is done with six different functions in total. These are:

1. **correct_amount_unique_plants**: ensures that the number of unique plant species, k, is such that $15 \leq k \leq 25$. If there are too few plants species, it will replace the most common plant species with new plant species not currently in the garden. If there are too many plant species, it will replace some of the species with other species already in the garden until the constraint is not violeted.
2. **water_feature_visible**: ensures all plants around the water feature are shorter than 90 cm. If they are not, it will replace those plants with a random sample from a list of plants whose heights are less than 90 cm.
3. **sort_plants**: a function that will be called in the next function that takes a vector of plants and orders them from shortest to tallest.
4. **all_plants_visible**: ensures all plants are visible according to the vantage points in Figure 3.
5. **make_viable_solution**: combines all the above functions into a single function that will take a candidate grid and return a viable solution, i.e. one that does not violate any of the constraints.

Finally, a number of functions are needed that will calculate the total **Aesthetic Score**. These are:

1. For **Colour Harmony**:

   (a) **colour_harmony_score**: calculates the colour harmony score as a weighted sum of the number of plants that are analogous, complementary of discordant (not analogous or complementary) in colour.

2. For **Balance and Symmetry**:

   (a) **colour_harmony_score**: calculates the circular distance between two colour hues (in order to determine if they are on opposite ends of the colour wheel or closer together). It returns a colour similarity score between 1 to 0

   (b) **height_similarity**: calculates a height score as 1− (height difference)/(max height of the two plants) between two plants. It returns this score as the height similarity score.

   (c) **category_similarity**: returns a 1 or 0 if the plants are in the same category or not.

   (d) **symmetry_score**: combines the above functions to calculate one overall **Balance and Symmetry** score as a weighted sum of the three individual scores.

3. For **seasonal_variation_score**: calculates the seasonal variation score as the inverse of the standard deviation of the column sums of the number of plants that flower each month. Thus, if the plants flower evenly throughout the year, the standard deviation would be close to 1 and this score also close to one (the maximum). Non-flowering plants are given a small score for each month of the year. The final score is the sum of these two scores.

4. The final **Aesthetic Score**: **total_aesthetic_score**: Combines all the above functions into one where the final aesthetic score is the sum of the three scores above.

These functions will be employed in Section 2 for Simulated Annealing and Section 3 for the Genetic Algorithm.

2 Simulated Annealing

2.1 Overview of the algorithm

Simulated Annealing is an optimization algorithm that is able to find optimal solutions to complex problems even when local optima exist. It does so by simulating the process of annealing in metallurgy, where defects are reduced by heating and gradual cooling, allowing the material to settle into a lower-energy state.

Formally, the algorithm can be broken into these three broad steps:

1. Set initial solution s, temperature T_0, and iteration counter $j = 0$.
2. Repeat the following until the stopping criteria are met (in the case of this project until the maximum number of iterations is met):
   (a) Generate a neighbouring solution of s, s' by applying some perturbing function to s.
   (b) Compute the change in the Aesthetic Score between these solutions, $\Delta = \text{Score}(s') - \text{Score}(s)$.
   (c) If the Aesthetic Score increased ($\Delta \geq 0$), accept s'. But, if the score decreased ($\Delta < 0$), accept the suboptimal solution with a probability:
   $$P(\text{accept}) = \exp\left(\frac{\Delta}{T_j}\right)$$
   This solution will be accepted if a random number drawn from the uniform distribution $U(0, 1)$ is less than this probability.
   (d) Update the temperature:
   $$T_{j+1} = \alpha^j \cdot T_0$$
   where α is the cooling rate.
   (e) Increment the iterations: $j \leftarrow j + 1$
3. Return the solution with the highest Aesthetic Score after the search is completed.

Since most of the basic functions to generate an initial solution, make it viable in terms of meeting all constraints and to calculate the Aesthetic Score of a solution have been defined above, all that is left to do is to define a perturbing function and to iterate through this process. These are defined in the functions **perturbing** and

simulated_annealing, which can be found in the Simulated Annealing.Rmd file on GitHub.

In brief, the **perturbing** function will:

1. Randomly choose the amount of plants to change in the current grid design. Call this l, where $5 \leq l \leq 10$.
2. Randomly sample l new plants (with replacement) to replace these plants.
3. Ensure the solution is viable using the **make_viable** function above.

The **simulated_annealing** function will set the number of iterations, a starting temperature and a cooling rate, apply step 1 in the algorithm above and repeat the sub-steps in step 2 until the number of iterations is complete.

2.2 Results and Discussion

Since the global optimum is not known or easily computable, the algorithm was applied for a number of different iterations, cooling rates and starting temperatures to see what the most optimal solution is that the algorithm can produce. These were used to define a parameter grid to apply the algorithm to each combination of parameters. The different values for each parameter that were tested are:

1. Number of iterations: 5000, 10 000 and 15 000
2. Starting Temperature: 1, 5 and 10
3. Cooling Rate: 0.85, 0.9 and 0.99

This resulted in 36 unique combinations. The best aesthetic score, final temperature, final score, mean score and standard deviation of all scores was calculated for each. Due to the size of this table, it can be found in the Appendix, in Section 6.2. All important results will be included in the main body of the report.

Table 6 shows all of these results. It is clear that the most important parameter is the number of iterations. For all but one combination (where T_0 is 5 and α is 0.85), the best score is at least 2.94 when the number of iterations is set to 15 000. Thus T_0 and α matter much less than the number of iterations. Overall the best score achieved was 2.97 when $T_0 = 10$, $\alpha = 0.99$ and the number of iterations is 15 000, although this is a very small (almost insignificant) improvement from the other configurations with 15 000 iterations.

When setting the three parameters to this value, the Aesthetic Score over the iterations is plotted in Figure 4. The highest Aesthetic Score achieved was 2.92. The Garden Grid can be seen in Figure 5. The numbers in each block represent the

row number of the plants in the Plant List in the data file on GitHub, Plant Data for Optimisation.xlsx. The number in brackets indicates the height at maturity of each plant and the colour of the block is the colour of its flowers. From this figure it is clear that the plant colours are mostly analogous in shades of pink with some complementary colours in yellow. All plants and features are visible and none of the constraints are violated. The layout also exhibits some degree of mirroring around the vertical line down the middle of the garden. In total there were 25 unique plant species in this garden design.

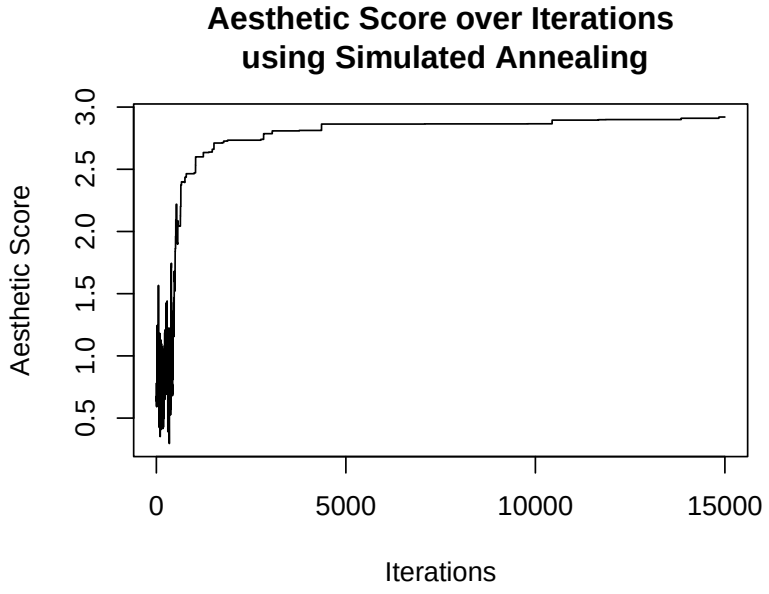


Figure 4: The Aesthetic Score for each iteration of the Simulated Annealing algorithm with $T_0 = 10$, $\alpha = 0.99$ and the maximum number of iterations at 15 000.

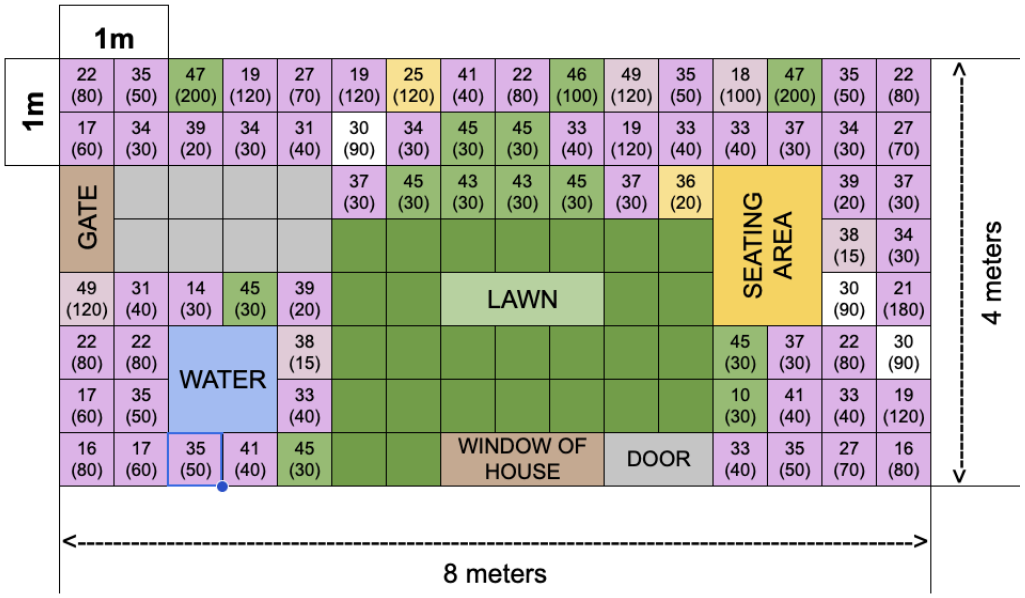


Figure 5: The Aesthetic Score for each iteration of the Simulated Annealing algorithm with $T_0 = 10$, $\alpha = 0.99$ and the maximum number of iterations at 15 000.

3 Genetic Algorithm

3.1 Overview of the algorithm

A genetic algorithm is also an optimisation algorithm that seeks to evolve a population of solutions to be optimal. It relies on theories of evolution that involve natural selection where the fittest individuals tend to survive. Broadly, it does the following:

1. **Initialise a population**: Begin by initialising a population of candidate solutions, known as a generation.
2. **Selection**: From these solutions, select the fittest parents to create the next generation. In the case of this project, the fitness of an individual garden grid is determined by its Aesthetic Score.
3. **Crossover**: Create offspring as combinations of the parents, mimicking biological reproduction.
4. **Mutation**: Mutate these offspring by making small changes to ensure there are some new traits in the population.
5. **Replacement**: Replace some of the individuals in the population with the new offspring.

This process is repeated until some stopping criteria are met. In this project, the steps above are implemented as follows:

1. **Initialise a population**: A new function was created, **generate_population**, that generates n initial solutions.
2. **Calculate its fitness**: this is simply done by applying the **total_aesthetic_score** function over the population of solutions.
3. **Selection**: parents are randomly selected based on a probability proportional to their fitness ranks[1]. This function also allows the user to specify what proportion of the generation to select as parents for the next generation.
4. **Crossover**: A probability based, patch crossover is applied[2]. It works by:

[1]Ranks are used since the fitness scores tend to be very close in absolute terms as the algorithm progresses. Thus to be able to distinguish between different fitness levels, the ranks are used to calculate a probability of selection.

[2]Note that due to the nature of the garden grid that is made up of plant species that can repeat (and are thus not unique as in the case of the traveling salesman problem) and some species are allowed to not appear in a garden grid at all, partially mapped crossover and cycle crossover cannot be applied. Instead, this unique patch crossover technique is applied.

   (a) Divide the offspring garden grid into 2×2 patches where the patches do not overlap (See Figure 6a).

   (b) Define parent pairs to create two offspring from[3].

   (c) For each parent pair and each patch, determine which parent's patch should make up the offspring's patch with a probability proportional to the parent's fitness, i.e. each 2×2 patch will be filled with the corresponding patch of either parent 1 or parent 2. This is done in the **patch_crossover** function.

   (d) Apply this crossover to all parent pairs to generate the full set of offspring. This is done in the **generate_all_offspring** function.

5. **Mutation**: The mutation is done in three steps that are all in the **mutate_grid** function:

   (a) Randomly choose how many 2×2 patches to mutate, call this k, (where $1 \leq k \leq 3$).

   (b) Randomly choose which k 2×2 patches to mutate (these can now be any 2×2 patches over the whole grid, i.e. the patches may overlap by one row or column, for example in matrix notation, the first patch will be $[1:2, 1:2]$ and the second $[2:3, 1:2]$, and so forth.) (See Figure 6b).

   (c) Randomly replace the plants in these patches with new plants (if the block is not marked with "U"- i.e. is a garden path or lawn or other feature).

6. **Ensure the solution is viable**: Apply the **make_viable** function over the population.

7. **Replacement**: The new offspring will replace the worst individuals (based on their fitness) from the previous generation to form the new generation.

8. **Repeat until stopping criteria is met**: The Genetic Algorithm will keep generating new offspring and generations until one of the following criteria is met:

   (a) Reaching a maximum number of generations (i.e. iterations)

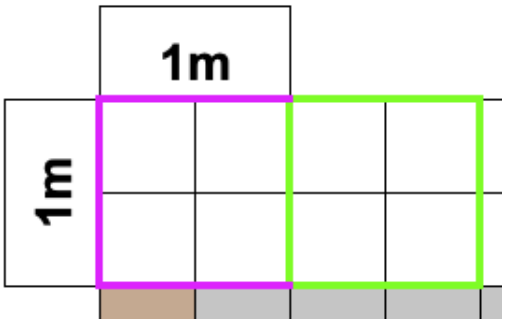
   (b) No improvement in fitness greater than some small ϵ for at least n generations.

   (c) More than $p\%$ duplicate individuals in the population.

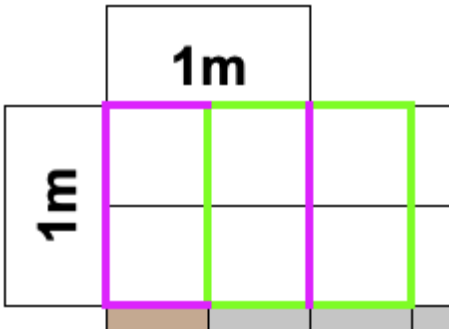
[3]By specifying that each parent pair must create two offspring the size of the population is maintained throughout all generations.

The stopping criteria are essential to ensure the algorithm does not run indefinitely, but knows when to stop. The first criteria ensures that if none of the other criteria are met, the algorithm will stop eventually. The second and third go hand-in-hand to ensure that the algorithm will stop once we do not see significant improvement in fitness over a number of generations. Finally the last criteria ensures that the final generation is not entirely made up of duplicates of the same garden grid.

Each one of the stopping criteria parameters was used to create a grid of parameters to apply the genetic algorithm to and return some key statistics. This will be discussed on the next section.



(a) Two patches, one in pink and one in green as an example of patches used in the crossover step. Here the patches do not overlap. These are used in the Crossover step of the Genetic Algorithm.



(b) Two patches, one in pink and one in green as an example of patches used in the crossover step. Here the patches are allowed to overlap. These are used in the Mutation step of the Genetic Algorithm.

Figure 6: An illustration of the patches defined in the **Crossover** (left) and **Mutation** (right) steps of the Genetic Algorithm.

All of these functions can be found in the Genetic Algorithm.Rmd file on Github.

3.2 Results and Discussion

Table 1 below shows each combination of:

1. Maximum number of iterations, i
2. Threshold for improvement, ϵ
3. Number of iterations allowed with improvement less than ϵ, n
4. Maximum proportion of allowable duplicate individuals in the population, p

that the algorithm was run for, as well as some key results. In all of these iterations, n, the population size, was set to 100. Mean Max Score is the mean score of the garden grids with the maximum fitness over the 10 runs for each parameter combination. Mean Overall Score shows the mean fitness score over all individuals in the final generation of all 10 runs for each parameter combination. Total iterations shows on average, how many iterations each run ran for until one of the stopping criteria was met. Prop. Duplicates shows the average proportion in the population that are duplicate garden grids in the final generation across the 10 runs. Finally, Stopping Criteria Met specifies which criteria most commonly resulted in the algorithm stopping and returning the final results.

Table 1: Summary of Genetic Algorithm Results for different combination of stopping criteria parameters. These are the maximum number of iterations (or generations), i, the threshold for improvement ϵ, number of iterations allowed with improvement less than ϵ, n, and finally, the maximum proportion of allowable duplicate individuals in the population, p.

i	n	ϵ	p	**Mean Max Score**	**Mean Overall Score**	**Mean Iterations**	**Prop. Duplicates**	**Stopping Criteria Met**
50	5	0.05	0.85	2.73	2.61	28	0.03	n
50	5	0.05	0.90	2.74	2.62	28	0.02	n
50	5	0.10	0.85	2.21	1.77	12	0.02	n
50	5	0.10	0.90	1.98	1.47	10	0.02	n
50	10	0.05	0.85	2.78	2.70	33	0.02	n
50	10	0.05	0.90	2.80	2.72	33	0.02	n
50	10	0.10	0.85	2.39	2.16	19	0.02	n
50	10	0.10	0.90	2.21	1.92	15	0.02	n
50	15	0.05	0.85	2.79	2.72	36	0.03	n
50	15	0.05	0.90	2.81	2.73	36	0.02	n
50	15	0.10	0.85	2.50	2.30	21	0.02	n
50	15	0.10	0.90	2.37	2.15	19	0.02	n

From Table 1 it can be seen that the only stopping criteria being met is the number

of iterations for which there is not sufficient improvement in fitness (i.e. an improvement greater than ϵ). This is true for all n. However, it is also noted that for all combinations, those with a smaller ϵ always produce better fitness scores than those with larger ϵ, holding all else constant. It is also noted that for $\epsilon = 0.05$, the fitness marginally increased for larger n, starting with a Mean Max Score of 2.73 and 2.74 for $n = 5$, to 2.78 and 2.8 for $n = 10$ and finally to 2.79 and 2.81 for $n = 15$. As a last observation, it does not seem that value of p bears any significance since that stopping criteria is never met and the proportion of duplicates is at most only 3% with the threshold at 85% or 90%.

All of the above seems to suggest that if the algorithm is allowed to make tiny improvements, that over time the fitness would increase. To test this hypothesis, the algorithm was again applied for 10 runs each over a new set of parameter combinations. This time, p is dropped and ϵ is set to be even smaller to allow smaller changes of fitness over time to be acceptable.

Table 2: Summary of Genetic Algorithm Results for a new set of combinations of stopping criteria parameters. These are the same as in Table 1, but dropping p.

i	n	ϵ	**Mean Max Score**	**Mean Overall Score**	**Mean Iterations**	**Prop. Duplicates**	**Stopping Criteria Met**
100	10	0.01	2.90	2.89	67.60	0.57	n
100	10	0.02	2.87	2.82	45.40	0.05	n
100	15	0.01	2.91	2.91	72.00	0.69	p
100	15	0.02	2.89	2.87	57.30	0.12	n
100	20	0.01	2.90	2.89	71.40	0.86	p
100	20	0.02	2.88	2.86	58.70	0.19	n

Table 2 shows that the algorithm could find a more optimal solution when allowed to make very small incremental improvements over time. Also of note is that the most common stopping criteria in some cases was p, the number of duplicates in the algorithm. Even though in this iteration it was removed from the parameter grid, it still has a default in the main algorithm set at 90%. There are very small marginal gains for different values of n and holding all else constant. It seems that the most sensitive parameter is, in fact, ϵ. Overall, there seems to be a gain in mean score of between 0.02 and 0.07 when moving from $\epsilon = 0.02$ to $\epsilon = 0.01$. The algorithm does on average run for more iterations for lower values of ϵ, which makes sense.

It is noted, however, that each of these parameter combinations was only run for 10 rounds each. The computation time would have been increased substantially should more runs have been tested. This does mean though that there is still variability in the mean scores and improvements as small as 0.02 should be taken lightly and could be due to randomness. Nevertheless, it seems that the optimal garden has a fitness or aesthetic score of 2.9. Since all of the configurations in Table 2 give similar results, for the sake of parsimony, a simpler model that converges more quickly (i.e. over fewer iterations) and have fewer duplicates on average may be preferred. Thus something like a model with $n = 10$ and $\epsilon = 0.02$ resulting here in a mean overall score of 2.82 and maximum mean score of 2.87. Considering that reducing ϵ to 0.01 only results in an increase of maximum score of 0.03 on average (roughly a 1% increase), but runs for more than 20 iterations longer, the increase seems negligible and not worth the increase in computation time.

Thus, solving for the optimal population of gardens again with $n = 10$, $\epsilon = 0.02$ and p left to the default of 90%, the algorithm met the stopping criteria for n after 48 iterations. The resulting mean and max fitness (or aesthetic) scores across the generations can be seen in Figure 7. The maximal fitness score was 2.84. It is clear that the mean and max for each generation converge, confirming that overall, the maximum solution has been found.

Finally, the resulting Garden Grid is shown in Figure 8, with the same interpretation from the Simulated Annealing final grid. Here it can be seen that the constraints of all plants and the water feature being visible are being met. It is also noted that the principle of Symmetry and Balance is being applied, especially at the top section where the yellow coloured plants are being mirrored across the central vertical line. There are clearly also mostly analogous flower colours in pink with some contrasting colours in yellow. In total there are 23 unique plant species.

3.3 Comparing the results with Simulated Annealing

The maximum Aesthetic Score for both algorithms were around 2.9, serving as some validation that both algorithms were able to find a global maximum and were not getting stuck in local maxima. Since the scores are very similar, the simplest solution will be taken to proceed with in this project, namely, the Garden Grid from the Genetic Algorithm that had a total of 16 unique plant species in the final design.

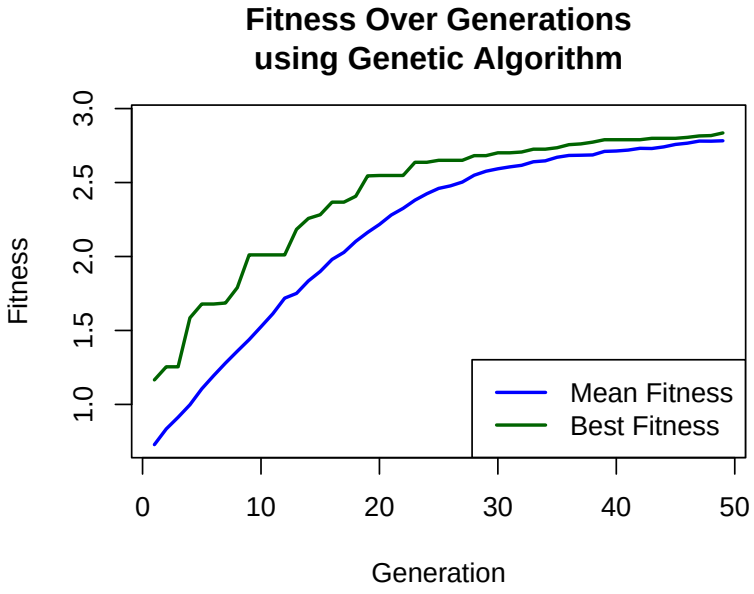


Figure 7: The mean (in blue) and max (in green) fitness scores for each generation after applying the Genetic Algorithm with $n = 10$, $\epsilon = 0.02$ and p left to the default of 90%.

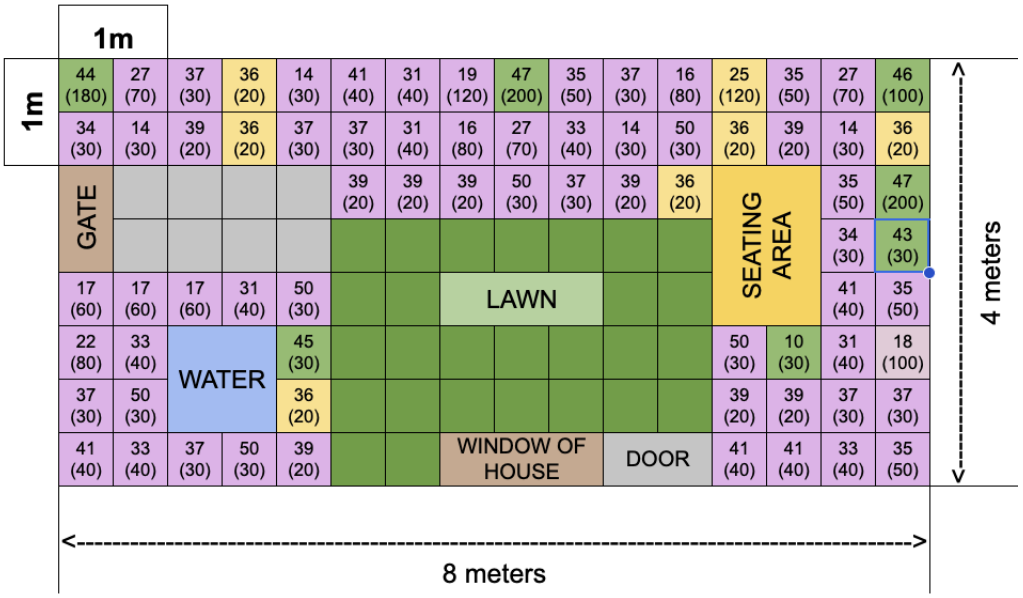


Figure 8: The mean (in blue) and max (in green) fitness scores for each generation after applying the Genetic Algorithm with $n = 10$, $\epsilon = 0.02$ and p left to the default of 90%.

4 Mixed Integer Linear Programming

4.1 Overview of the algorithm

Mixed Integer Linear Programming (MILP) is an optimisation algorithm that aims to find the best solution to a problem by minimizing or maximising some objective function subject to a list of linear constraints.

The aim of the MILP algorithm here is to find the maximum immediate visual impact achievable by buying the most mature plants, given a set budget. The older a plant is, the larger its size and immediate visual impact in a garden. Ideally, one would want to buy only the most mature plants to avoid having to wait potentially several years for them to have the desired visual effect.

The MILP algorithm can be formulated as follows. Let

1. x_{is} be the number of units of plant i at size s purchased
2. c_{is} be the cost of plant i of size s
3. I_{is} be the impact that plant i has when it is of size s, $0 < I_{is} < 100$
4. Q_i be the quantity of plant i required for the optimal garden layout found by the SA and GA algorithms
5. B be the total available budget

We then define our objective as:

$$\max \sum_i \sum_s I_{is} x_{is}$$

Subject to the following constraints:

$$\sum_s x_{is} = Q_i \quad \forall\ i$$

$$\sum_i \sum_s c_{is} x_{is} \leq B \quad \forall\ i, s$$

$$x_{is} \geq 0 \quad \forall\ i, s$$

where each x_{is} is an integer.

In this model, the plant variables x_{is} correspond to the plants selected by the Genetic Algorithm above. The visual impact of a plant is a theoretical number between 20 and 100. It is randomly generated, but is designed to increase with plant size, thus simulating the phenomena that an older plant would have a larger visual impact.

4.2 Results and Discussion

The quantities of the optimal solution of each plant and each size can be seen in Table 7. For some plants, larger sizes are selected at higher costs, but for others, the higher costs do not justify the small increase in impact. This is even clearer in Figure 9, where some plants clearly contribute greatly to impact from the larger sized plants. The distribution of plant sizes is shown in Figure 10. The results clearly indicate a preference for larger plants. This suggests that the budget was healthy and larger plants could be afforded while still meeting the exact quantity needed for each plant.

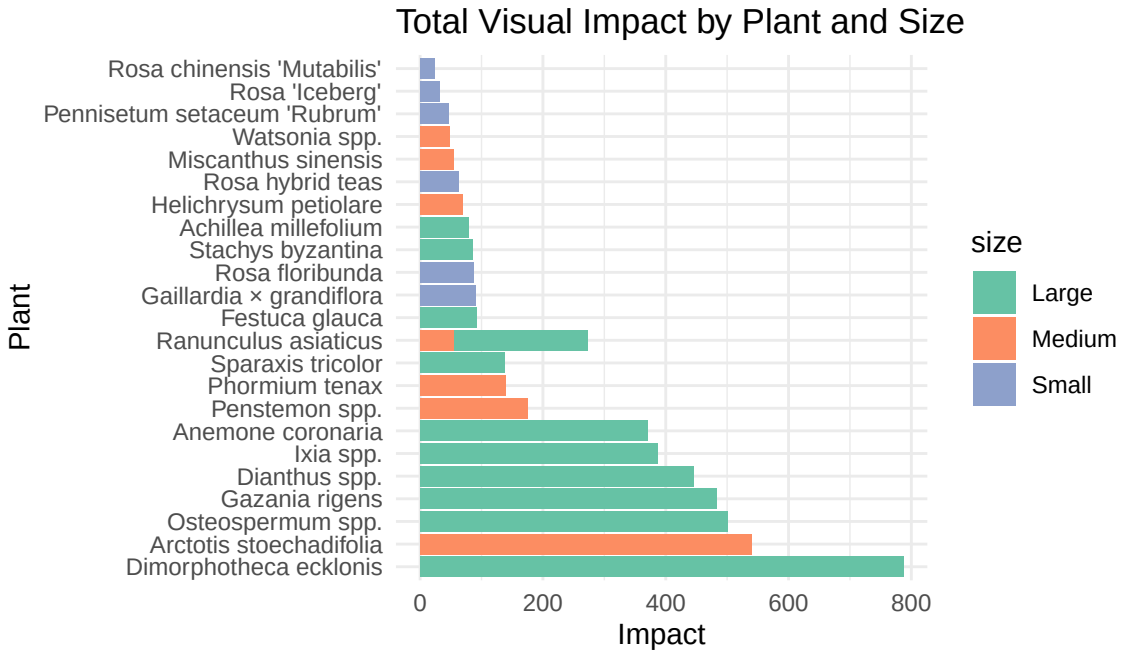


Figure 9: A bar chart showing the increase in impact for different sizes of each plant.

All of the code for this section can be found in the Mixed Integer Linear Programming.Rmd file on GitHub.

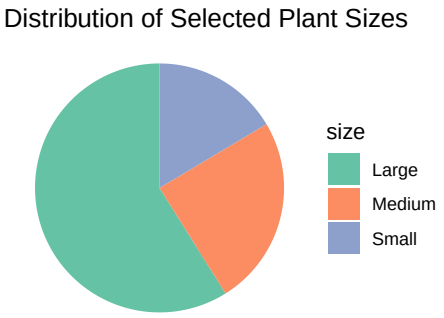


Figure 10: A pie chart showing the distribution of plant sizes.

5 Multi-objective Linear Programming

5.1 Overview of the algorithm

The Multi-objective Linear Programming Optimisation algorithm extends the MILP into multiple objectives. In this case, the objectives were:

1. Minimise Cost, $\sum_i \sum_s c_{is} x_{is}$
2. Maximise Impact, $\sum_i \sum_s I_{is} x_{is}$
3. Minimise Maintenance Effort, $\sum_i \sum_s M_{is} x_{is}$

where i and s are defined the same as in the MILP algorithm.

Maintenance effort refers to the amount of work that is needed to maintain a plant. Older plants, here assumed to be the larger plants, need less care than younger plants that would need pruning, feeding and regular watering to get established and grow into a healthy mature plant. This variable is simulated as follows:

1. For small plants, $M \sim N(3, 1)$
2. For medium plants, $M \sim N(2, 1)$
3. For large plants, $M \sim N(1, 1)$

The multi-objective problem is solved by both the Archimedean and Chebychev Goal Programming Algorithms. However, both of these algorithms make use of deviations for each goal that are optimised. In order to calculate these deviations, a payoff table is first constructed by solving a MILP algorithm for each objective individually. This procedure is described in the next section. All of the code for this section can be found in the Multi-objective Linear Programming.Rmd file on GitHub.

5.2 Constructing the pay-off table

For each MILP run that optimises one objective at a time, the cost, impact and maintenance effort is noted for the optimal solution. Table 3 shows these results. These results were used to scale each objective to be between 0 and 1.

The normalized deviations were calculated as follows:

1. $d_{\text{cost}} = d_1 = \frac{\text{cost-min}}{\text{max-min}}$
2. $d_{\text{impact}} = d_2 = 1 - \frac{\text{impact-min}}{\text{max-min}}$
3. $d_{\text{effort}} = d_3 = \frac{\text{effort-min}}{\text{max-min}}$

Table 3: A pay-off table constructed by solving the three objectives with MILP, one at a time.

	Cost	Impact	Effort
Impact Optimized	6520	5621.11	56.83
Cost Optimized	2595	2442.44	228.70
Effort Optimized	6130	5207.61	48.16

Each deviation is therefore expressed as a function of the decision variables, x_{is}.

5.3 Archimedean Goal Programming

Archimedean Goal Programming combines multiple goals into a single objective by minimizing a weighted sum of the deviations from ideal (target) values for each objective.

5.3.1 Problem Formulation

Formally, the problem can be formulated as below.

$$
\begin{aligned}
\min \quad & w_1 d_1 + w_2 d_2 + w_3 d_3 \\
\text{subject to:} \quad & \sum_s x_{is} = Q_i \quad \forall i \quad \text{(plant quantity constraint)} \\
& \sum_{i,s} c_{is} x_{is} \leq B \quad \text{(budget constraint)} \\
& \sum_{i,s} c_{is} x_{is} = \text{TotalCost}, \\
& \sum_{i,s} I_{is} x_{is} = \text{TotalImpact}, \\
& \sum_{i,s} M_{is} x_{is} = \text{TotalEffort}, \\
& d_1 = \frac{\text{TotalCost} - \text{BestCost}}{\text{WorstCost} - \text{BestCost}} \\
& d_2 = 1 - \frac{\text{TotalImpact} - \text{WorstImpact}}{\text{BestImpact} - \text{WorstImpact}} \\
& d_3 = \frac{\text{TotalEffort} - \text{BestEffort}}{\text{WorstEffort} - \text{BestEffort}} \\
& x_{is} \in \mathbb{Z}_{\geq 0} \quad \forall i, s
\end{aligned}
$$

Note here that for d_2 under-achieving this goal is undesirable, so the deviation is scaled accordingly to maintain consistency with the other objectives (where larger values imply worse performance). Thus all deviations are now in the same direction and it is not necessary to explicitly specify under- or over-achieving directions in the deviations. For all the goals, only overachievement needs to be tracked. For example, having a smaller overall cost will never be undesirable.

Since the deviations are not known, but rather functions of the decision variables, x_{is}, they are rewritten as linear combinations of the form

$$d_{\text{cost}} = \frac{\sum_{i,s} c_{is} x_{is} - \text{BestCost}}{\text{WorstCost} - \text{BestCost}} \tag{1}$$

Then setting

$$d = a \cdot \sum x + b \tag{2}$$

then it can be seen that

$$a = \frac{1}{\text{Worst} - \text{Best}} \tag{3}$$

$$b = -\frac{\text{Best}}{\text{Worst} - \text{Best}} \tag{4}$$

And similarly for the other two objectives.

Thus the final objective becomes

$$w_1 d_1 + w_2 d_2 + w_3 d_3 = \sum_k w_k (a_k \cdot \text{Total}_k + b_k) = \sum_k (w_k a_k) \cdot \text{Total}_k + \sum_k (w_k b_k) \tag{5}$$

where $\text{Total}_k = \sum_{i,s} f_{i,s} x_{is}$ for each objective k. And since the second part of the equation is constant for different values of x_{is}, the final objective becomes the weighted sum of these deviations:

$$w_1 d_1 + w_2 d_2 + w_3 d_3 = \sum_k (w_k a_k) \cdot \text{Total}_k \tag{6}$$

In order to test the results of this algorithm when different objectives are prioritised, four different weight vectors were used:

1. All equal weights
2. Cost prioritised above the other two
3. Impact prioritised above the other two
4. Effort prioritised above the other two

5.3.2 Results and discussion

After applying the algorithm to the four sets of weights mentioned above, the results are shown in Table 4. It can be seen that the formulation that prioritises all objectives equally actually derives a near-optimal result in terms of Impact and Maintenance Effort, only Cost is substantially higher than the Cost that is achieved when prioritising that objective above the others. The relationship between the size of plants and the objectives is also clear. When Impact and Effort are prioritised, the Cost is set to the maximum and as many large plants are selected as possible and only the minimum left to small plants. In contrast, prioritising cost results in a much lower cost is achieved, but mostly small plants are used.

5.4 Chebychev Goal Programming

Chebychev Goal Programming is slightly different to Archimedean Goal Programming in that it focused on fairness across objectives. Rather than minimising the weighted sum of deviations, it minimised the largest deviation from all the objectives.

Table 4: The results after applying the Archimedean Goal Programming algorithm with four different sets of weights. The first to prioritise the three objectives equally, then prioritising cost, impact and effort, respectively.

	Prioritisation	Cost	Impact	Effort	Large	Medium	Small
1	Equal	5000	4654.26	71.43	38	24	11
2	Cost	3410	3477.47	150.05	6	29	38
3	Impact	5000	4986.16	99.18	45	16	12
4	Effort	5000	4486.11	65.35	33	31	9

5.4.1 Problem Formulation

Formally, this can be written as

$$
\begin{aligned}
\min \quad & z = \max(w_1 d_1, w_2 d_2, w_3 d_3) \\
\text{subject to:} \quad & \sum_s x_{is} = Q_i \quad \forall i \quad \text{(plant quantity constraint)} \\
& \sum_{i,s} c_{is} x_{is} \leq B \quad \text{(budget constraint)} \\
& \sum_{i,s} c_{is} x_{is} = \text{TotalCost}, \\
& \sum_{i,s} I_{is} x_{is} = \text{TotalImpact}, \\
& \sum_{i,s} M_{is} x_{is} = \text{TotalEffort}, \\
& d_1 = \frac{\text{TotalCost} - \text{BestCost}}{\text{WorstCost} - \text{BestCost}} \\
& d_2 = 1 - \frac{\text{TotalImpact} - \text{WorstImpact}}{\text{BestImpact} - \text{WorstImpact}} \\
& d_3 = \frac{\text{TotalEffort} - \text{BestEffort}}{\text{WorstEffort} - \text{BestEffort}} \\
& w_1 d_1 \leq z \\
& w_2 d_2 \leq z \\
& w_3 d_3 \leq z \\
& x_{is} \in \mathbb{Z}_{\geq 0} \quad \forall i, s
\end{aligned}
$$

Thus the goal is to minimise the maximum deviation from the objective.

The deviations can be rewritten in the same way as above, i.e. as a linear combination of the decision variables x_{is}.

5.4.2 Results and Discussion

After applying the algorithm to the four sets of weights mentioned above, the results are shown in Table 5. It can be seen that this algorithm often assigns zero quantities to certain plant sizes to protect the "most critical" goal from having large deviations. It is able to achieve better solutions than the Archimedean formulation when Cost and Effort are prioritised, but does worse when Impact is prioritised. This means that the Archimedean Algorithm is able to give more balanced and stable results, whereas Chebychev results in more extreme solutions.

Table 5: The results after applying the Chebychev Goal Programming algorithm with four different sets of weights. The first to prioritise the three objectives equally, then prioritising cost, impact and effort, respectively.

	Prioritisation	Cost	Impact	Effort	Large	Medium	Small
1	Equal	3300	3208.57	151.96	0	33	40
2	Cost	2595	2442.44	228.70	0	0	73
3	Impact	3300	3208.57	151.96	0	33	40
4	Effort	4770	4365.85	73.29	32	27	14

6 Appendix

6.1 Link to Github Repository

The full code and dataset used in this project are available on GitHub at: `https://github.com/Annie0619/multivariate_assignment`.

6.2 Simulated Annealing Parameter Grid Results

Table 6: Simulated Annealing Results Across Parameter Grid.

Iterations	Start Temp	Cooling Rate	Best Score	Final Temp	Final Score	Mean Score	Score SD
5000	1	0.85	2.80	0.00	2.80	2.66	0.23
10000	1	0.85	2.89	0.00	2.89	2.76	0.23
15000	1	0.85	2.95	0.00	2.95	2.83	0.17
5000	5	0.85	2.79	0.00	2.79	2.71	0.20
10000	5	0.85	2.86	0.00	2.86	2.79	0.18
15000	5	0.85	2.86	0.00	2.86	2.82	0.14
5000	10	0.85	2.87	0.00	2.87	2.76	0.22
10000	10	0.85	2.92	0.00	2.92	2.84	0.16
15000	10	0.85	2.94	0.00	2.94	2.88	0.12
5000	1	0.90	2.83	0.00	2.83	2.71	0.20
10000	1	0.90	2.88	0.00	2.88	2.78	0.20
15000	1	0.90	2.94	0.00	2.94	2.85	0.17
5000	5	0.90	2.89	0.00	2.89	2.76	0.23
10000	5	0.90	2.90	0.00	2.90	2.82	0.17
15000	5	0.90	2.95	0.00	2.95	2.86	0.13
5000	10	0.90	2.88	0.00	2.88	2.75	0.22
10000	10	0.90	2.92	0.00	2.92	2.83	0.16
15000	10	0.90	2.95	0.00	2.95	2.87	0.12

Iterations	Start Temp	Cooling Rate	Best Score	Final Temp	Final Score	Mean Score	Score SD
5000	1	0.95	2.81	0.03	2.81	2.68	0.22
10000	1	0.95	2.92	0.00	2.92	2.79	0.22
15000	1	0.95	2.96	0.00	2.96	2.84	0.16
5000	5	0.95	2.88	0.02	2.88	2.76	0.21
10000	5	0.95	2.93	0.00	2.93	2.82	0.19
15000	5	0.95	2.95	0.00	2.95	2.87	0.14
5000	10	0.95	2.87	0.03	2.87	2.73	0.23
10000	10	0.95	2.92	0.00	2.92	2.80	0.17
15000	10	0.95	2.94	0.00	2.94	2.86	0.14
5000	1	0.99	2.85	0.05	2.85	2.74	0.22
10000	1	0.99	2.92	0.00	2.92	2.82	0.17
15000	1	0.99	2.96	0.00	2.96	2.86	0.12
5000	5	0.99	2.90	0.07	2.90	2.76	0.21
10000	5	0.99	2.94	0.00	2.94	2.84	0.14
15000	5	0.99	2.95	0.00	2.95	2.87	0.12
5000	10	0.99	2.90	0.08	2.90	2.74	0.23
10000	10	0.99	2.95	0.00	2.95	2.83	0.15
15000	10	0.99	2.97	0.00	2.97	2.87	0.13

6.3 MILP Plant Solutions

Table 7: The results after applying the MILP algorithm to the optimal Garden Grid. The table shows how much of each plant and size should be bought to maximise impact within a set budget.

	Scientific_Name	size	Cost	Impact	solution	Total_Cost	Total_Impact
1	Dimorphotheca ecklonis	Large	75.00	87.41	9.00	675.00	786.69
2	Arctotis stoechadifolia	Medium	45.00	59.92	9.00	405.00	539.28
3	Osteospermum spp.	Large	75.00	100.00	5.00	375.00	500.00
4	Gazania rigens	Large	60.00	80.52	6.00	360.00	483.12
5	Dianthus spp.	Large	80.00	74.29	6.00	480.00	445.74
6	Ixia spp.	Large	75.00	96.83	4.00	300.00	387.32
7	Anemone coronaria	Large	95.00	74.25	5.00	475.00	371.25
8	Ranunculus asiaticus	Large	75.00	73.01	3.00	225.00	219.03
9	Penstemon spp.	Medium	65.00	58.24	3.00	195.00	174.72
10	Phormium tenax	Medium	100.00	69.52	2.00	200.00	139.04
11	Sparaxis tricolor	Large	70.00	68.57	2.00	140.00	137.14
12	Festuca glauca	Large	80.00	91.78	1.00	80.00	91.78
13	Gaillardia × grandiflora	Small	30.00	22.65	4.00	120.00	90.60
14	Rosa floribunda	Small	65.00	29.14	3.00	195.00	87.42
15	Stachys byzantina	Large	75.00	85.33	1.00	75.00	85.33
16	Achillea millefolium	Large	85.00	79.33	1.00	85.00	79.33
17	Helichrysum petiolare	Medium	60.00	69.46	1.00	60.00	69.46
18	Rosa hybrid teas	Small	70.00	31.24	2.00	140.00	62.48
19	Ranunculus asiaticus	Medium	50.00	54.20	1.00	50.00	54.20
20	Miscanthus sinensis	Medium	90.00	54.08	1.00	90.00	54.08
21	Watsonia spp.	Medium	75.00	47.68	1.00	75.00	47.68
22	Pennisetum setaceum ‘Rubrum’	Small	45.00	47.01	1.00	45.00	47.01
23	Rosa ‘Iceberg’	Small	75.00	31.03	1.00	75.00	31.03
24	Rosa chinensis ‘Mutabilis’	Small	80.00	23.66	1.00	80.00	23.66