# **Hill Climbing Algorithm**

The Hill Climbing Algorithm compares its neighbouring values and chooses the best node to improve the function value. If that neighbour is above the current state (e.g. has a higher function value than now) then move to that point, otherwise stop.

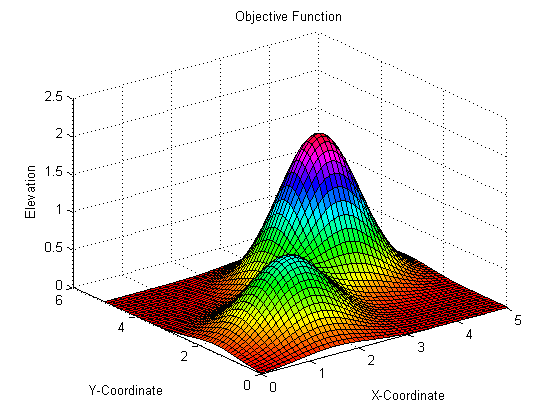
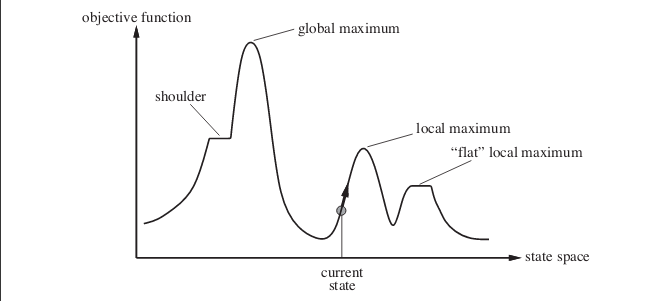


Figure 1: State Space diagram for Hill Climbing

Hill Climbing Algorithm is very slow due to the fact it is moving through the search space at random. This algorithm is like a blind person in a 2D searching space (e.g. a Heatmap, a 2D graph in 3D perspective) stepping in one direction at random and saying, “am I higher than I was?”, so if that direction is higher than the current one then it proceeds there, if not it steps back and try to step somewhere else randomly again.

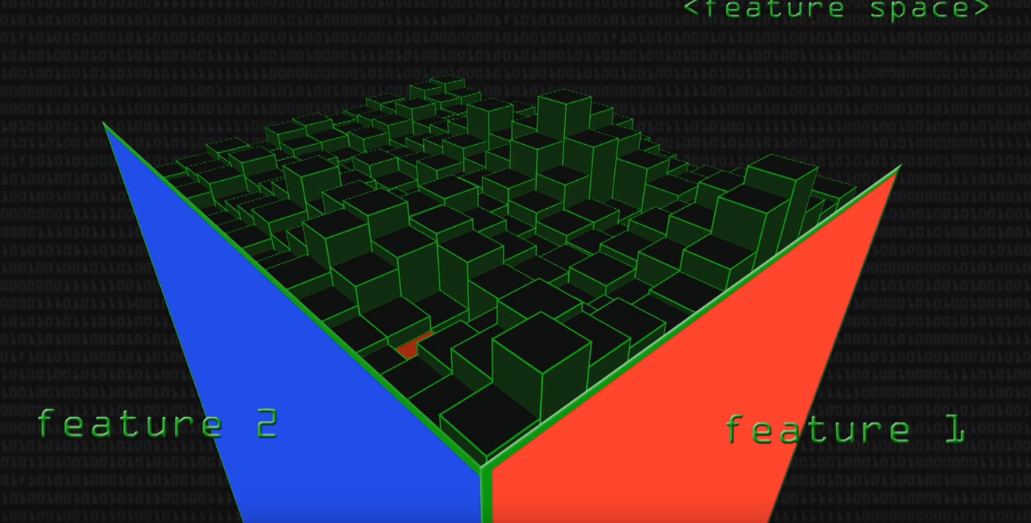


Figure 2: 2D Space Diagram in 3D perspective

In simpler words, what hill-climbing says is to find the neighbour that has the largest function value and eventually, very gradually will climb up the mountain. Although a problem occurs. For example, if the algorithm climbs up a mountain and there is no higher than that around it, everything around it is worse and it is going down, then it stays there. This does not mean that is the highest mountain on the searching space, but it cannot see it. As can be seen in Figure 3 the algorithm can see only what is underneath, therefore it stuck to a point called “Local Maximum” or “Local Minimum” which is sort of a trap and this happens to evolution very often.

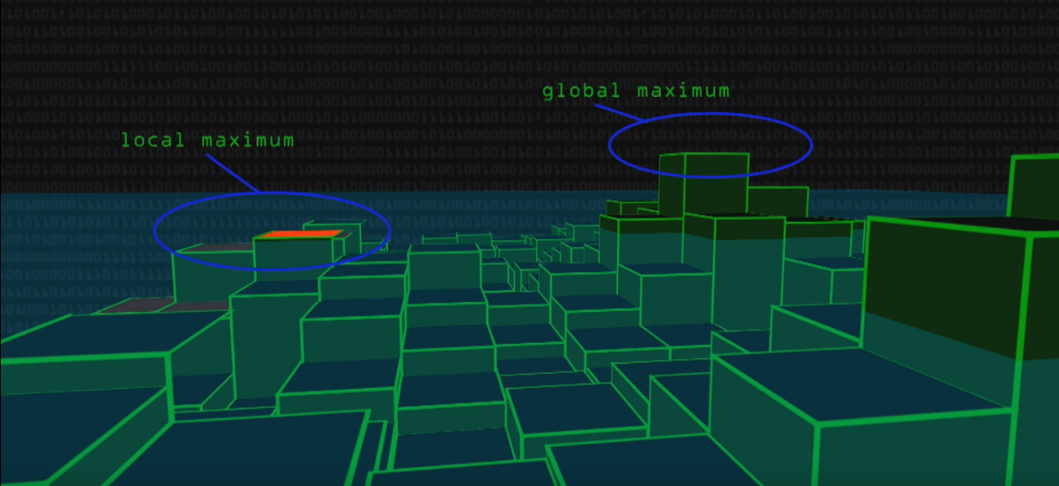


Figure 3: Local and Global Maximum

## **Algorithm for Simple Hill-Climbing:**

**Step 1:** Evaluate the original condition. Stop or restore progress if it is a target condition. Alternatively, make initial state as a current state.

**Step 2:** Loop until the state of the solution is reached or no new operators are available that can be added to the current state.

1. Choose a state not yet added to the current state and add it to create a new one.
2. Perform these in order to assess the new state.
   1. If the current condition is a goal, then stop and return.
   2. If it is better than the current condition, then make is a current state and proceed.
   3. If the current status is not getting better, then repeat the loop until a solution is found.

**Step 3:** Exit

Figures 4-7 present the plots and pictures of the results of hill climbing algorithm:

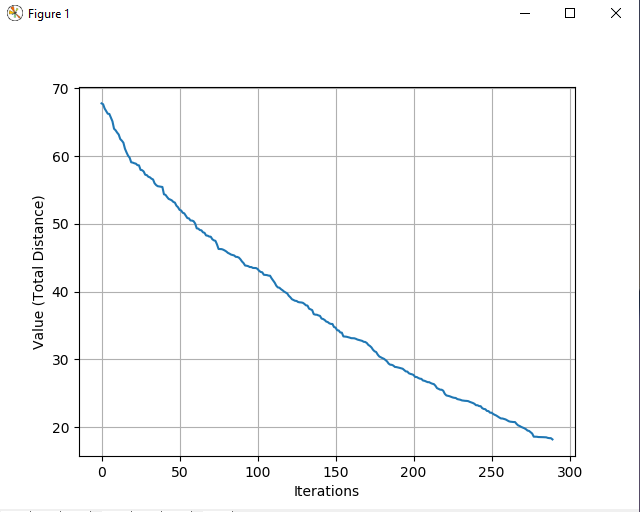


Figure 4: Hill Climbing, 1 Iteration - 100 Colours Plot

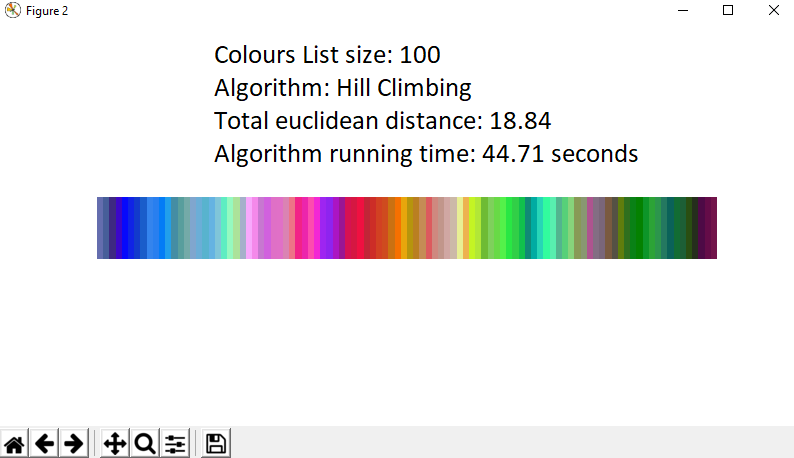


Figure 5: Hill Climbing, 1 Iteration - 100 Colours Figure

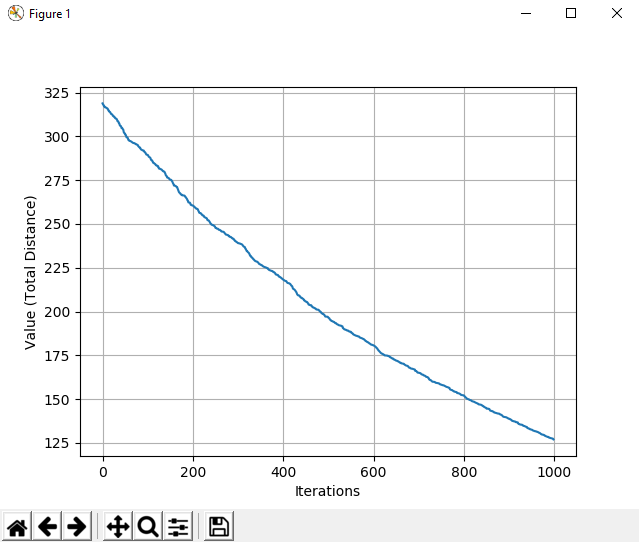


Figure 6: Hill Climbing, 1 Iteration - 500 Colours Plot

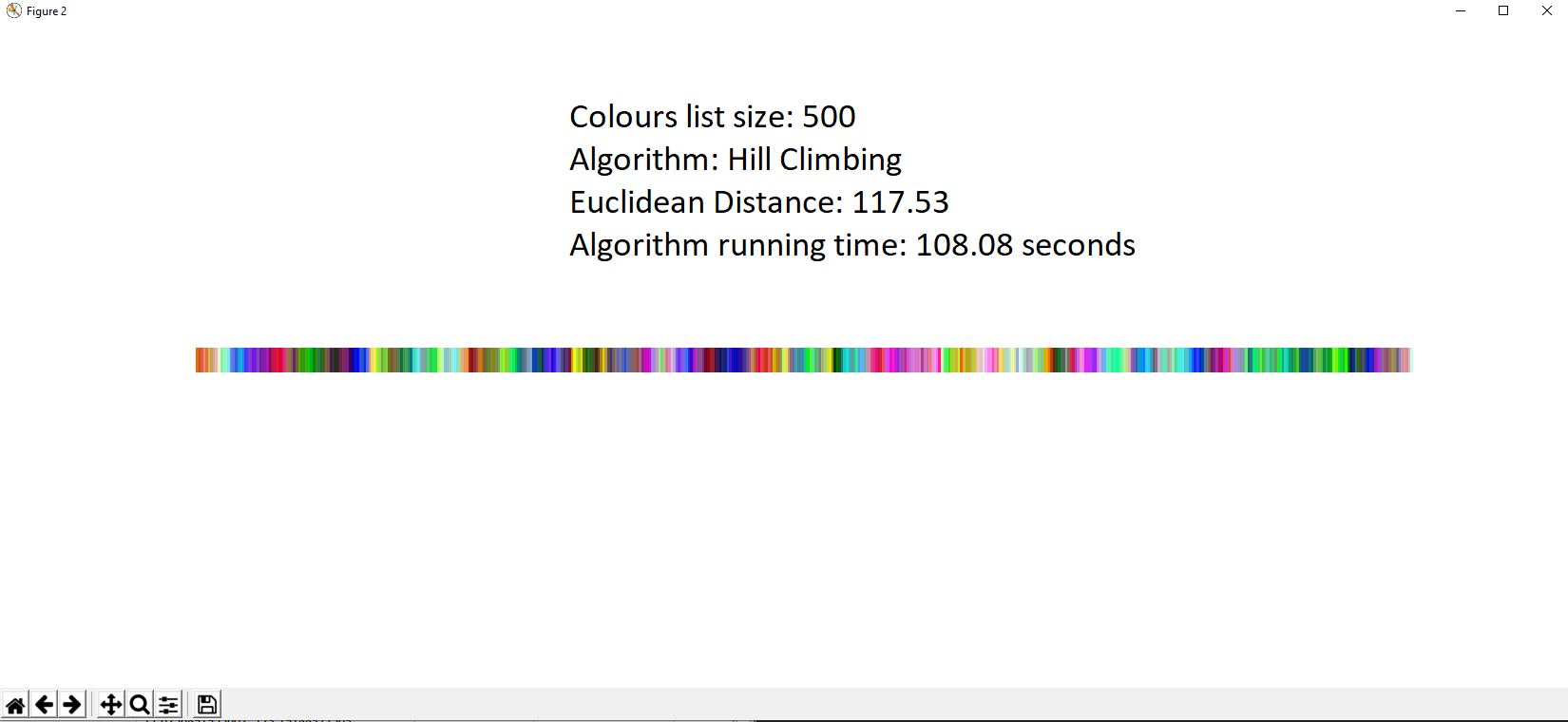


Figure 7: Hill Climbing, 1 Iteration - 500 Colours Figure

# **Multi-Start Hill Climbing Algorithm**

The principles of Multi-Start or Random Start Hill Climbing Algorithm are the same as the normal Hill Climbing Algorithm. The only difference is that it runs multiple times and it has a fixed number of how many times it will restart in order to fix the problem that Hill Climbing has. The multiple tries attempt to find a good starting place which is one of the best advantages of MHC (Multi-Start Hill Climbing algorithm). This means that after several iterations it will eventually start from a location that will reach the global optimum.

Both Hill Climbing and MHC algorithms can undo earlier choices (if next neighbour is less than the current state then stop), thus, it is not classified arrogant. It only seems sometimes greedy as it seeks at every move for the best optimization.

## **Algorithm for Multi-Start (Random-Restart) Hill Climbing**

Multi-Start Hill-Climbing is a very effective algorithm built on top of the normal Hill-Climbing which it repeatedly runs the Hill-Climbing algorithm with a random starting location χ0. The best local minimum or local maximum (χm) is kept for that run. This replaces the stored state if a new run delivers a better χm than the stored state. On many occasions, MHC is an algorithm that is surprisingly effective. It turns out that spending more CPU time exploring space is often easier, than carefully optimizing it from an initial condition.

Figures 8-11 illustrate the plots and pictures of the results of MHC algorithm:

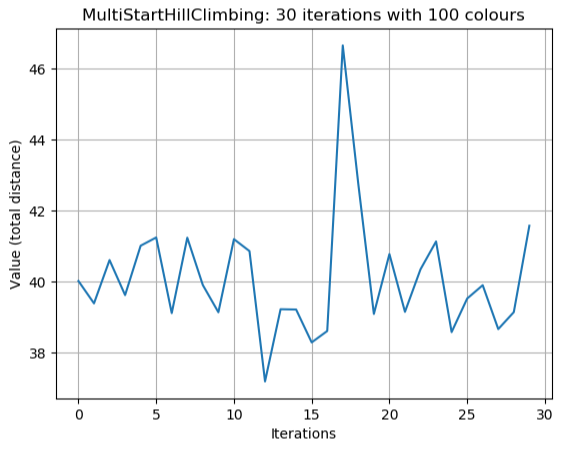


Figure 8: Multi-Start Hill Climbing, 30 Iterations - 100 Colours Plot

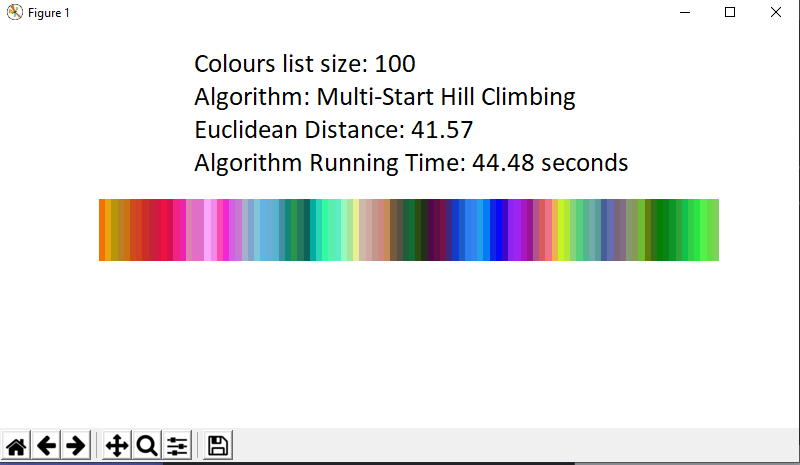


Figure 9: Multi-Start Hill Climbing, 30 Iterations - 100 Colours Figure

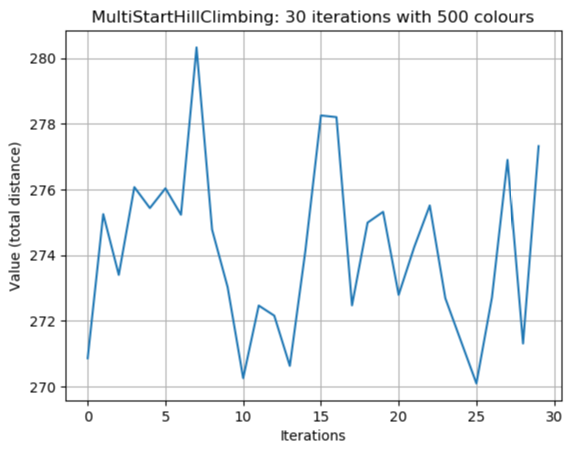


Figure 10: Multi-Start Hill Climbing, 30 Iterations - 500 Colours Plot

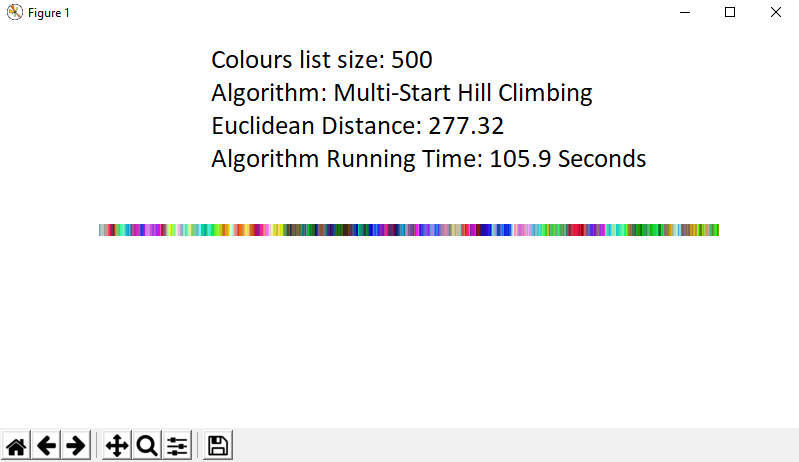


Figure 11: Multi-Start Hill Climbing, 30 Iterations - 100 Colours Figure

# **Greedy Algorithm**

Both Hill-Climbing and Greedy algorithms are heuristics that can be used for problems of optimization. Greedy is an algorithmic method which constructs a piece by piece approach, often selecting the next element that provides the most apparent and immediate benefit. Therefore, the problems where the locally ideal alternative often contributes to a global solution are best suited to Greedy.

The 0-1 knapsack is a good example of an optimization problem. There is a knapsack with a certain weight limit and a bunch of things to place in a knapsack. Every element has its own weight and price. The goal is to optimize the weight of the items in the bag, while retaining the weight below the limit. A greedy algorithm will select high density objects and place them in until the knapsack is full. For example, a diamond has a high value and a low weight compared to a brick, so the diamond is going in the bag knapsack first.

Below, it can be found an example of a greedy algorithm failing. We assume that there is a knapsack with capacity of 100 and there are 2 available items to choose: Diamonds and Gold Coins.

|  |  |  |  |
| --- | --- | --- | --- |
| **Item** | **Value** | **Weight** | **Density** |
| 1x Diamond | 1000 | 90 | 11.1 |
| 5x Gold Coins | 210 | 20 | 10.5 |

The greedy algorithm will choose the diamond first with value 1000 and weight 90 and that’s it. However, the best solution relies on the 5 Gold Coins with total value 1050.

Figures 12-15 demonstrate the plots and pictures of the results of the Greedy algorithm for this project:

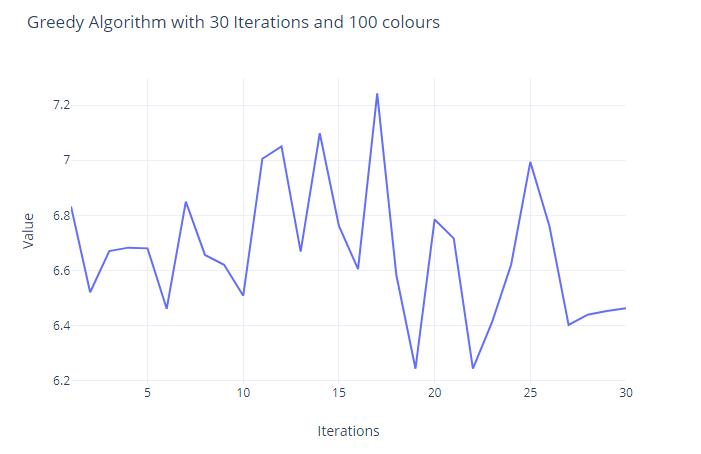


Figure 12: Greedy, 30 Iterations - 100 Colours Plot

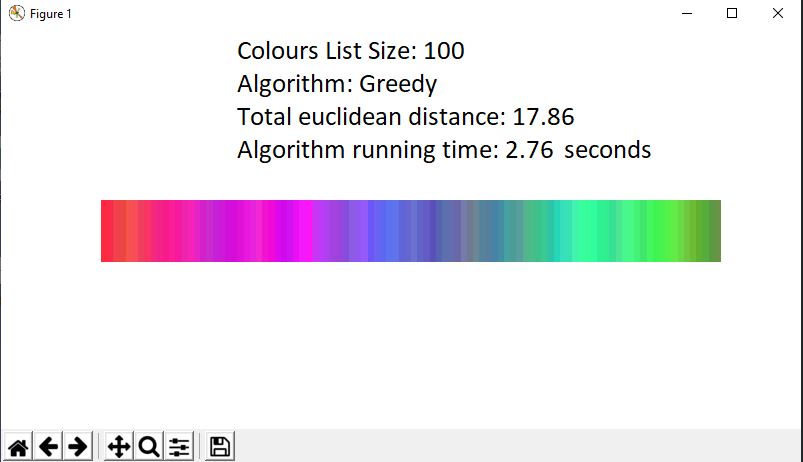


Figure 13: Greedy, 30 Iterations - 100 Colours Figure

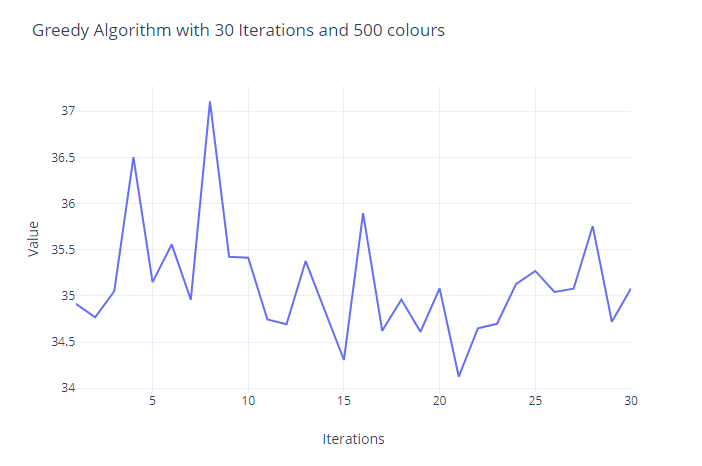


Figure 14: Greedy, 30 Iterations - 500 Colours Plot

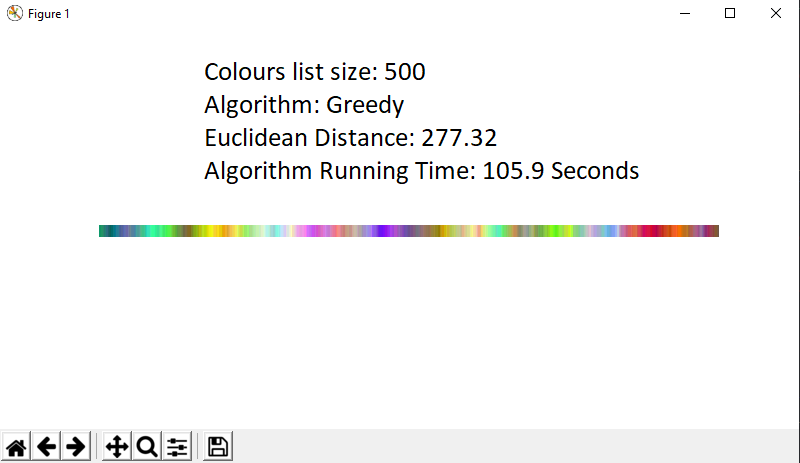


Figure 15: Greedy, 30 Iterations - 500 Colour Figure

# **Comparing algorithm performance**

Figures 16 and 17 compare the performance of the Greedy and MHC algorithms in terms of seconds and distance values. The time taken as well as the distance value for the Greedy algorithm are both ways less than the Multi-Start Hill Climbing algorithm. Therefore, for this occasion the Greedy algorithm performs better than MHC.[[1]](#footnote-1)

Figure : Greedy VS MHC (seconds)

Figure : Greedy VS MHC (Values)

1. More detailed information about the runs for each algorithm can be found in the Excel File [↑](#footnote-ref-1)