ENGINEERING TRIPOS PART IIB

4M17 Coursework Assignment 2

Candidate 5606G

January 5, 2022

Contents

1 Introduction

This report explores the performance of the Simulated Annealing (SA) and Tabu Search (TS) algorithms in minimising the 6D Eggholder function, given in eq. 1, where d = 6. This problem is hard to solve as we have many local maxima and minima, with a global optimum on a constraint boundary. Our problem is made easier by the fact all variables are continuous and on the same scale, all constraints are inequalities and our feasible region is not disjoint. We used the 2D Eggholder function (d=2) first in order to visualise how the algorithm is operating and also varied the parameters of each algorithm to see what effect they have on the performance.

$$\begin{aligned} &\text{Minimise} \quad f(\mathbf{x}) = \sum_{i=1}^{n-1} \left[-(x_{i+1} + 47) \sin\left(\sqrt{|x_{i+1} + 0.5x_i + 47|}\right) - x_i \sin\left(\sqrt{|x_i - x_{i+1} - 47|}\right) \right] \\ &\text{subject to} \quad -512 \leq x_i \leq 512 \end{aligned} \tag{1}$$

The surface and contour plots of the Eggholder function are shown in fig. 1, and have the minimum point marked as a red dot (x = [512, 404.2752], f(x) = -959.6394526997). We will use this contour to display the success of algorithms for the 2D problem in exploring the whole space and to see which minima they converge to. We have written all the code to conduct and display a search of both methods and their results. The contour plot will be displayed twice, first showing all archived solutions and then showing the entire path i.e. all accepted solutions.

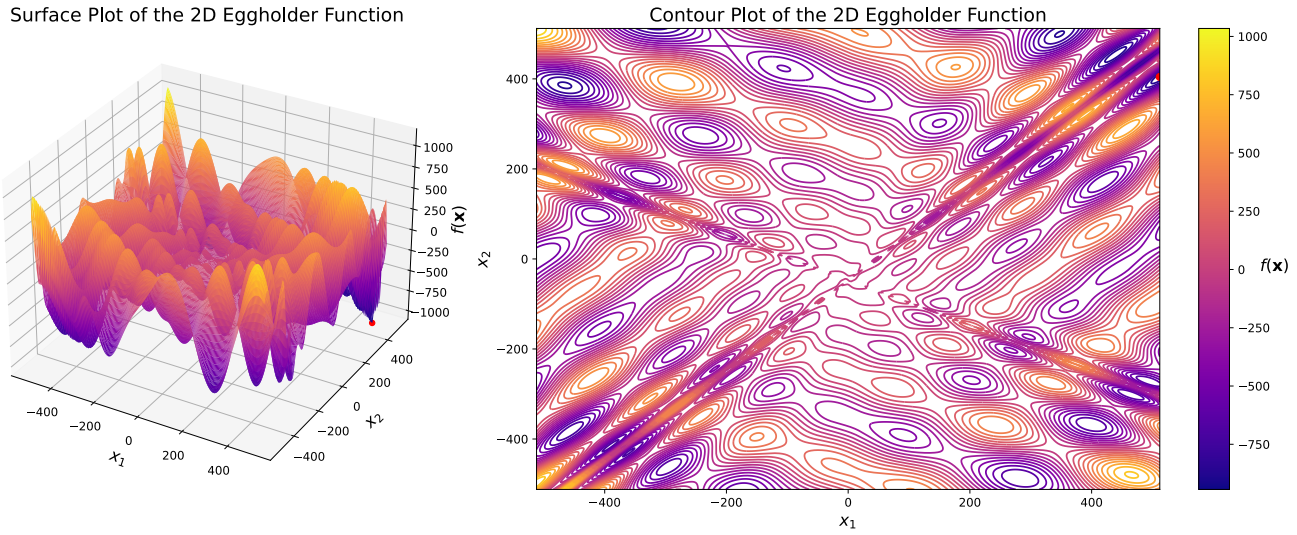

Figure 1: 2D Eggholder Surface and Contour Plot with Marked Minimum

2 Simulated Annealing (SA)

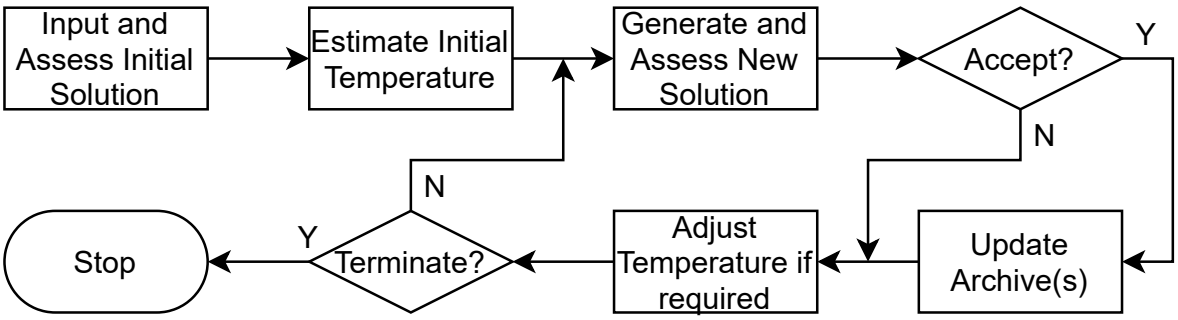

Figure 2: Simulated Annealing Algorithm Structure

The structure of the SA search method, as detailed in lecture handouts, is shown in fig. 2. In our implementation, the initial solution is chosen by randomly selecting a starting point over the whole feasible space from a uniform distribution, and evaluating the cost at this point. The initial temperature (T_0) was determined by two different methods. Initially, we would take 1000 random steps and accept all of them. We then take all the steps that increased the objective function and calculate the mean increase, $\bar{\delta f}^+$. We calculate T_0 according to eq. 3, in which χ_0 is the average probability of a solution that increases f, which we set to 0.8 [Scott Kirkpatrick 1984].

A new solution would be generated by taking a random step from the previous accepted solution. In this report we investigate 3 methods of taking a random step. The first was the basic method, where new solutions were found according to eq. 2. In this, $\mathbf{u}$ is a vector of random numbers in the range [-1,1] and $\mathbf{C}$ is a constant diagonal matrix to define the maximum allowed variation in each input variable of $\mathbf{x}$. We initially set each diagonal term to 25. The other methods are detailed later in the report.

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mathbf{C}\mathbf{u} \quad (2) \qquad T_0 = -\frac{\bar{\delta f}^+}{\ln \chi_0} \quad (3) \qquad p = \exp\left(-\frac{\delta f}{T}\right) \quad (4) \qquad T_{k+1} = \alpha T_k \quad (5)$$

If the step decreased the objective function, we would always accept the change. Otherwise, we would accept the change with probability according to eq. 4, where $\delta f = f(\mathbf{x}_{i+1}) - f(\mathbf{x}_i)$ and T is the temperature at the time. This is where the analogy to annealing of metals comes in, as atoms are constantly moving and are trying to move to a structure of least energy. They will always accept a new stable structure that decreases the overall energy, but they will also sometimes accept changes increasing the overall energy if the temperature is high enough to allow the new structure.

If we accepted the change, we would add the new point to the path, and add the point to the archive if it satisfied the archiving criteria detailed in lectures. For this problem, we set $d_{min} = 60$ and $d_{sim} = 6$. Our maximum archive length $L = 25$. Each set of accepted points at a particular temperature was said to be a Markov Chain, as the probability of the next point only depended on the state of the current point. The temperature for the k$^{\text{th}}$ Markov Chain, T_k, would be changed to T_{k+1} after either η_{min} acceptances were made, or the chain had reached the maximum length L_k. We set $L_k = 500, \eta_{min} = 0.6L_k$. We changed the temperature according to eq. 5, where we initially set $\alpha = 0.95$ [S. Kirkpatrick, Gelatt, and Vecchi 1983]. We also included a restart structure, where we would set the current point to our best achieved point so far after 10000 function evaluations. We terminated the search when we reached 15000 function evaluations or had a Markov Chain with no acceptances. Figs. 3 and 4 show the result from a run using these initial settings. We see that we get good exploration of the surface and reach close to the optimum value in the end. We also see the restart occurring close to the end of the run, which then makes the algorithm evaluate points closer to our minimum.

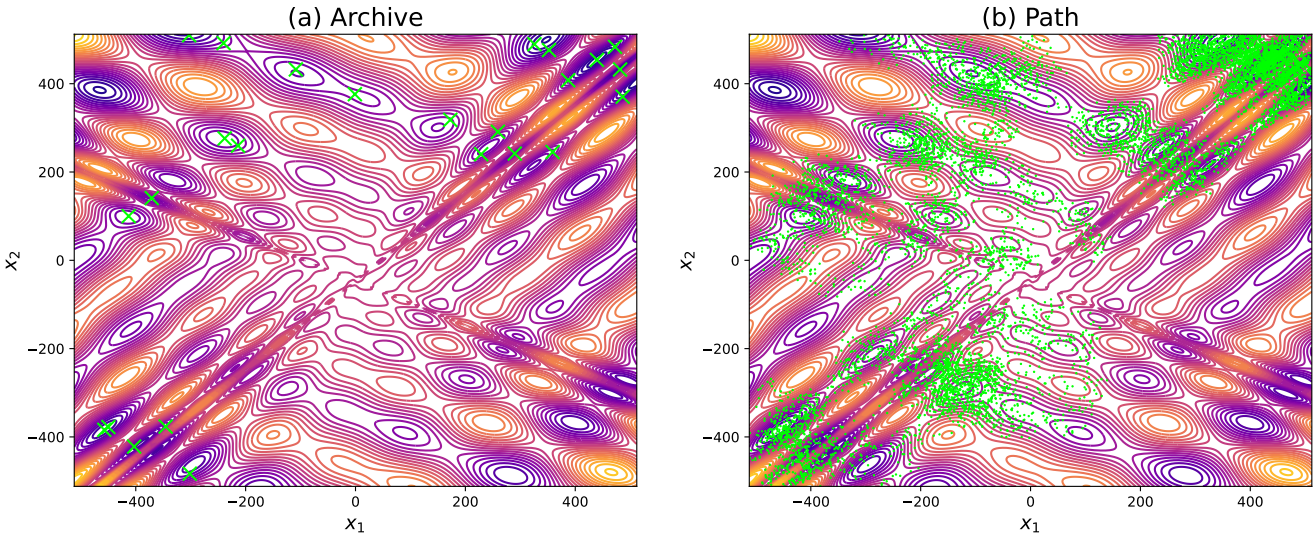

Figure 3: SA Initial Run on 2D Eggholder Function

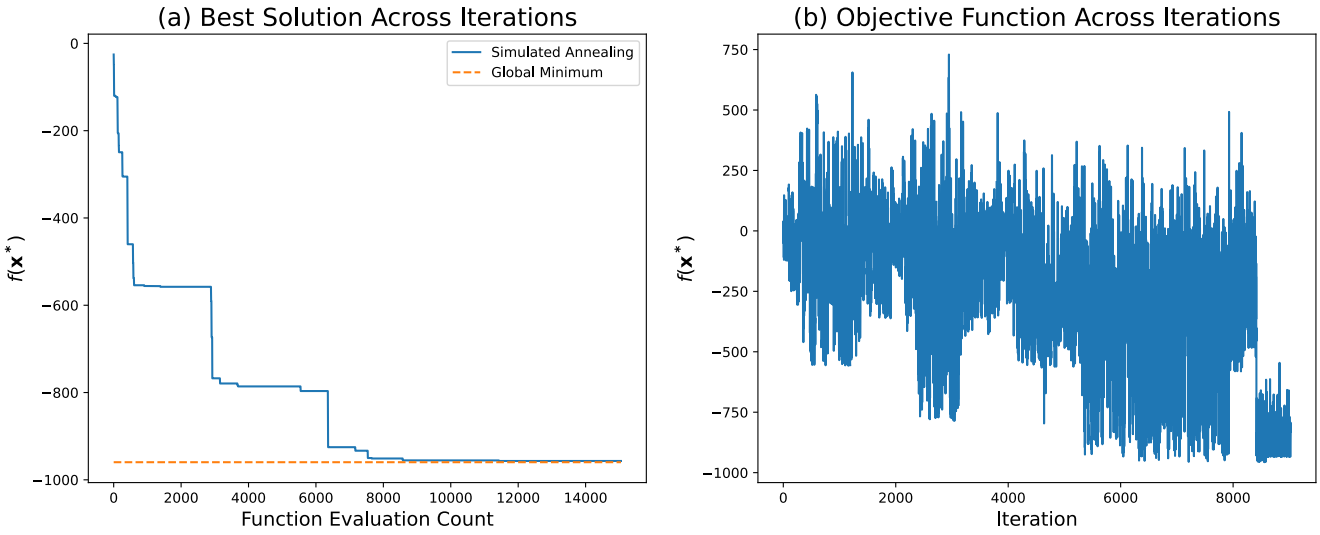

Figure 4: SA Progress of Initial Run

2.1 Initial Temperature

We altered the parameters and methods of our SA algorithm in various ways to judge the differences and see how we could maximise performance. We first analysed two methods of setting the initial temperature. The first was the Kirkpatrick method used in the initial run, which calculated an initial temperature of 474.9 for this initial run. We implemented a second method, which would first take 1000 random steps, accept all of them and find the standard deviation (σ_0) of the cost function in this set. Then $T_0 = \sigma_0$ [White 2008]. The results for a run using this method are shown in fig. 5. In this run we also don't get as much exploration as we'd like. The initial temperature calculated was 306.2, which explains why it got less coverage.

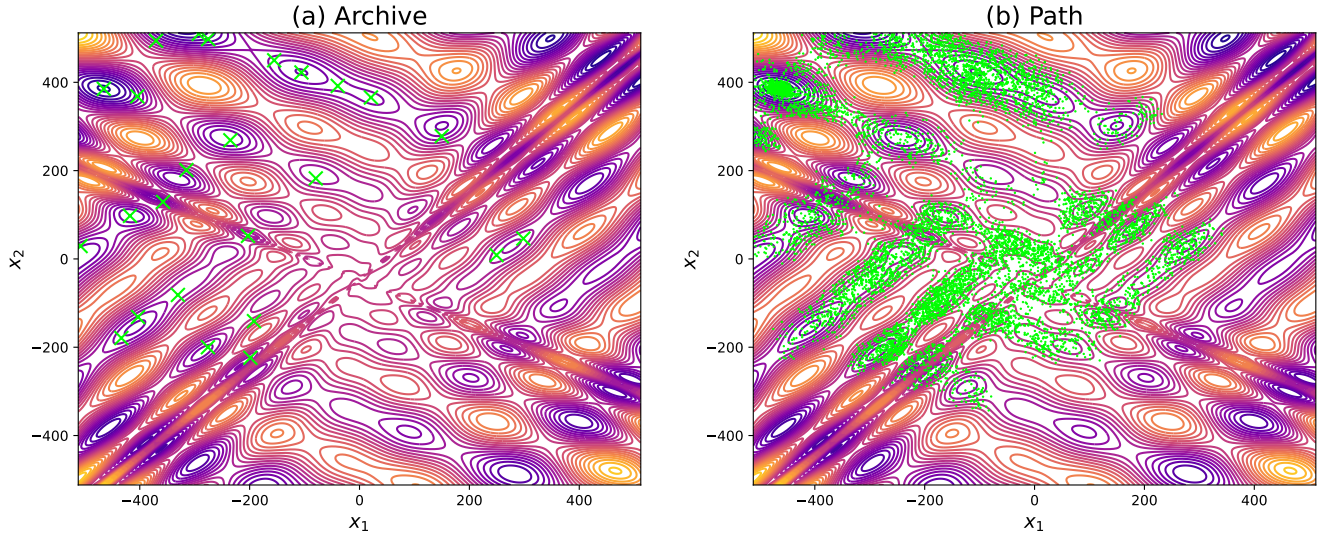

Figure 5: SA Initial Run on 2D Eggholder Function

We then evaluated our different methods each over 60 different random seeds for the 2D and 6D case, which are shown in fig 6. For the 2D case, the Kirkpatrick method gave an average starting temperature of 505.6, compared to 324.8 in the White Method. The average minimised cost function over runs is shown in the titles as f_av. We see the Kirkpatrick method does better in the 2D case. The White method does better in the 6D case but by a smaller proportion. This could be because the White method reaches a lower temperature in the later stages, which allows it to better converge to minima in valleys it has found. We investigated this by running the same analysis, but this time with $\alpha = 0.9$ for all runs. The results are shown in fig. 7. The Kirkpatrick method does better in both the 2D and 6D case, and improves on itself in the 6D case. The method to use will depend on the setting of α.

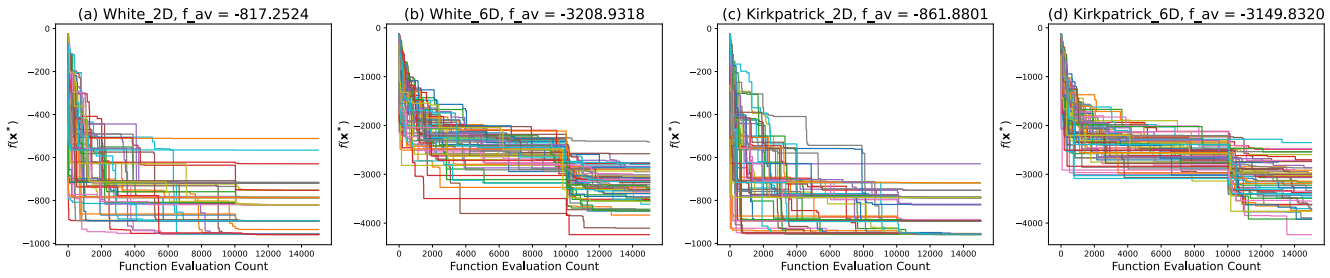

Figure 6: Temperature Initialisation Methods Comparison

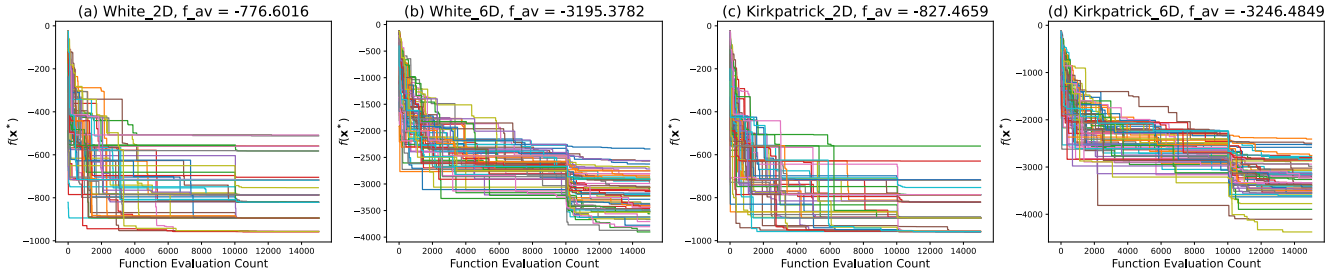

Figure 7: Temperature Initialisation Methods Comparison, $\alpha = 0.9$

2.2 Varying α

This leads on to looking at how the setting of α affects our results. Ideally we want to start at high temperatures to allow our algorithm to explore more of the feasible region, but we want it to become low enough that by the end of the run, we are able to go as deep as possible into valleys we have found. To investigate this, we ran both initial temperature estimation methods for varying values of α, conducting 100 runs for each case. The results are shown in Figs. 8 and 9. We find that as before, both methods do similarly well but the Kirkpatrick method prefers slightly smaller values for α. Since they end up giving similar results, for the rest of the report we will use the Kirkpatrick method with $\alpha = 0.9$.

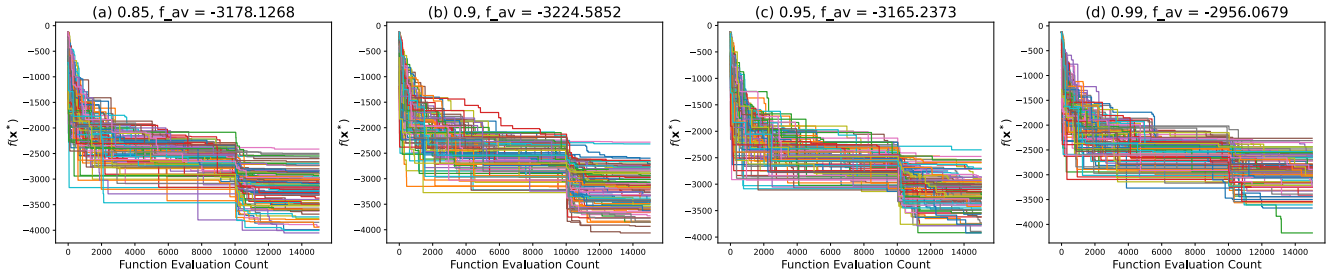

Figure 8: Comparing the Effect of α Setting on Performance for the Kirkpatrick Method

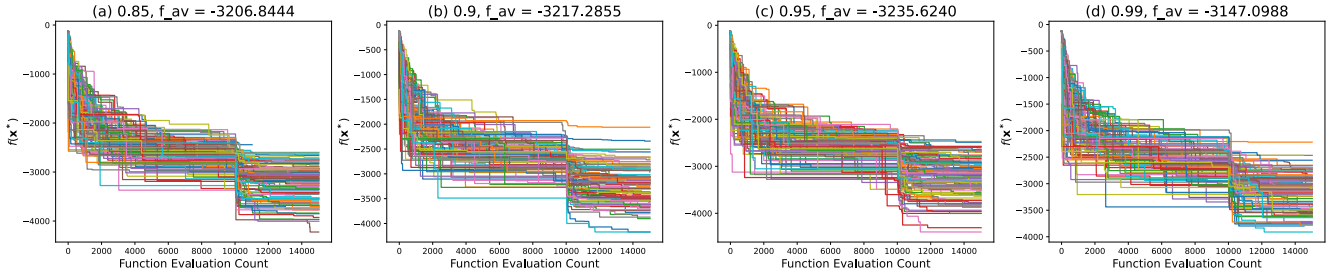

Figure 9: Comparing the Effect of α Setting on Performance for the White Method

2.3 Unconstrained

Since we were told the best solution for this problem is on a boundary, we next investigated using an unconstrained problem, replacing our inequality constraints with penalty functions detailed in lectures. We used the method detailed in eq. 6, where $\mathbf{w}$ is a weight vector and $\mathbf{c}$ represented the magnitude of constraint violations in each input variable. We set $\mathbf{w}$ to 1×10^6 for every input variable. We thought that this would allow our algorithm to investigate points closer to our boundaries as more points along the boundaries would be accepted. Fig. 10 shows how the algorithm search spread for the 2D case. We see that the algorithm accepted some points with $x_1 \leq -512$, but all the archived points were still in the feasible region.

$$f_A(\mathbf{x}) = f(\mathbf{x}) + \frac{1}{T}\mathbf{w}^T\mathbf{c}_V(\mathbf{x}) \tag{6}$$

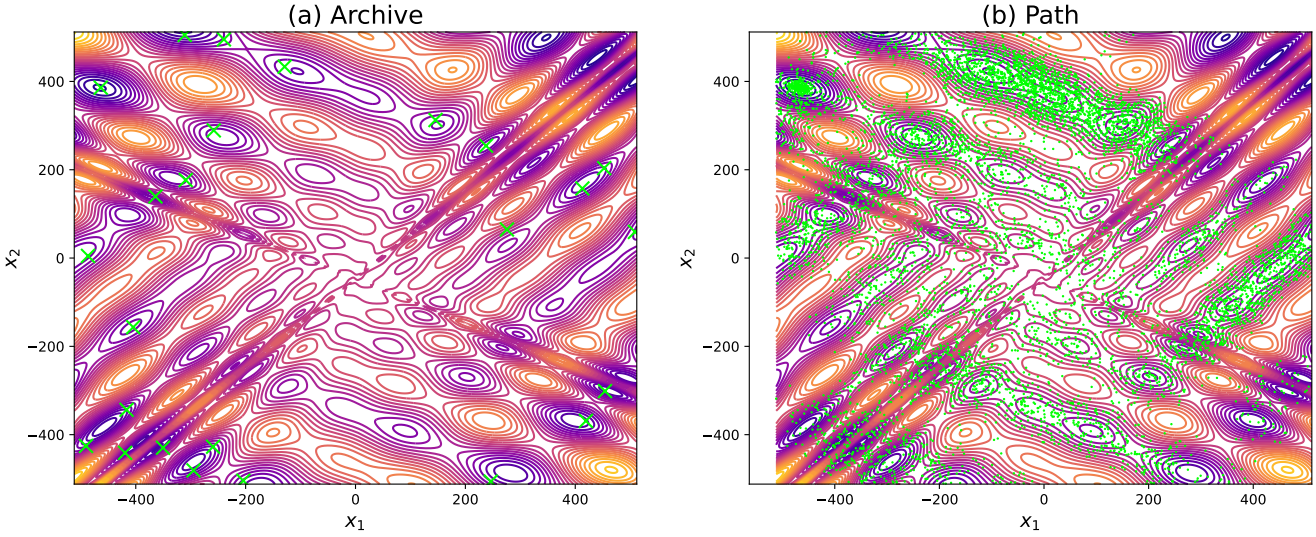

Figure 10: Contour Plot for an Unconstrained Search

Fig. 11 shows the results for a constrained vs unconstrained method. This is with the basic step method, the Kirkpatrick initial temperature evaluation method, and $\alpha = 0.9$. The titles show the penalty weights $\mathbf{w}$ or if the plot was for the constrained cost function. We can see from this result that using an unconstrained but highly penalised method of weights 1×10^6 was the most effective, although all plots performed similarly. This could be because the algorithm was allowed to accept points slightly farther out, so the search could be maintained more on the boundary regions. For future runs we decided to adopt the unconstrained cost function with a weight of 1×10^6.

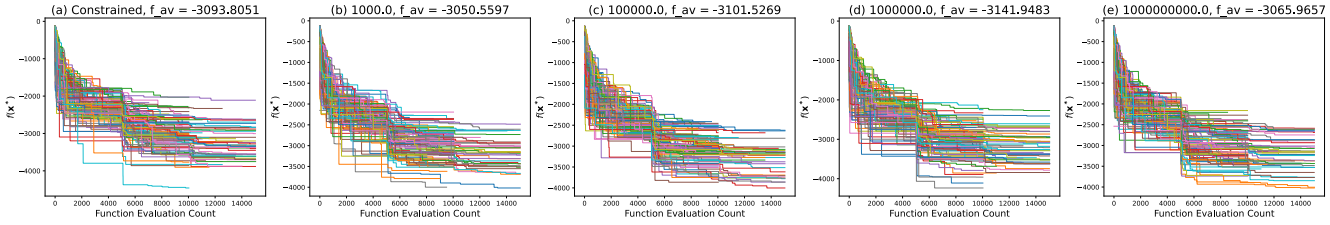

Figure 11: Comparing A Constrained Cost Function vs. Using Penalty Functions

2.4 Random Step Methods

We next looked at three different methods in which we could take random steps. The main goal would be to look at how different step sizes affected performance, and how the methods to vary step size throughout the run performed.

2.4.1 Basic

We first looked at the basic method, which we described in the initial run. Since this method used a constant maximum magnitude change throughout the run, we investigated it by trialling different step sizes to see how it performed. Larger step sizes meant we would be able to explore more of the feasible region, but wouldn't be able to descend down a particular valley as easily, essentially a comparison of breadth vs. depth. We looked at the performance of different step sizes and produced the results in fig. 12. We see that the best performance was best for a maximum step of 50. This could be because the feasible region in our 6D case is so large, that the algorithm being able to move rapidly between spaces proves to be the more successful to an extent, and moving deeper into valleys found wasn't as important.

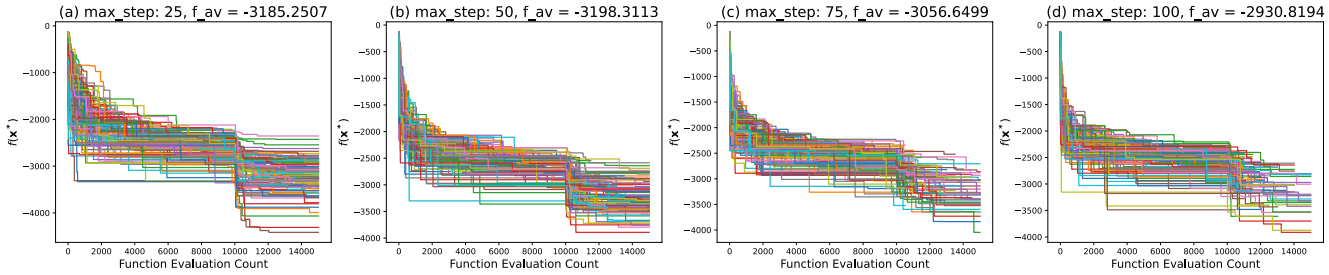

Figure 12: Comparison of runs using different step size allowances

2.4.2 Vanderbilt and Parks

We also had two other methods of being able to produce random steps for SA. The first was the Vanderbilt method [Vanderbilt et al. 1984], described in eq. 7, where $\mathbf{u}$ is a vector of random numbers in the range $[-\sqrt{3}, \sqrt{3}]$ and $\mathbf{Q}$ is a matrix controlling the magnitude of step sizes. We used a covariance matrix $\mathbf{S}$ and found $\mathbf{Q}$ by the Cholesky decomposition of $\mathbf{S}$. We would update $\mathbf{S}$ after every Markov Chain using eq. 8 to include the covariance of the local area. Here, $\mathbf{X}$ is the covariance of the path followed by the algorithm. We set $\alpha = 0.1, \omega = 2.1$ as suggested in the lectures.

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mathbf{Q}\mathbf{u} \tag{7}$$

$$\mathbf{S}_{j+1} = (1-\alpha)\mathbf{S}_j + \alpha\omega\mathbf{X} \tag{8}$$

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \mathbf{D}\mathbf{u} \tag{9}$$

$$\mathbf{D}_{j+1} = (1-\alpha)\mathbf{D}_j + \alpha\omega\mathbf{R} \tag{10}$$

The second method was the Parks method [Parks 2017], which would produce random steps according to eq. 9, where $\mathbf{u}$ and $\mathbf{D}$ were the same as in the basic method, except we would update $\mathbf{D}$ after every successful step using eq. 10. Here, $\mathbf{R}$ is a matrix to represent successful changes made to $\mathbf{x}$. α and ω had the same setting as the Vanderbilt method. We compared these methods in fig 13, where a (*) in the title represented using a constrained formulation. We see that the Parks method performs better and the constrained method performs best. The constrained method performing better seemed to be because the unconstrained method allowed our step-size controlling matrices to grow too large so they couldn't give small enough steps to converge to the minima in potential valleys found. The Parks method outperformed the Vanderbilt method because it seemed to be able to adapt to smaller values in the step-size matrix better.

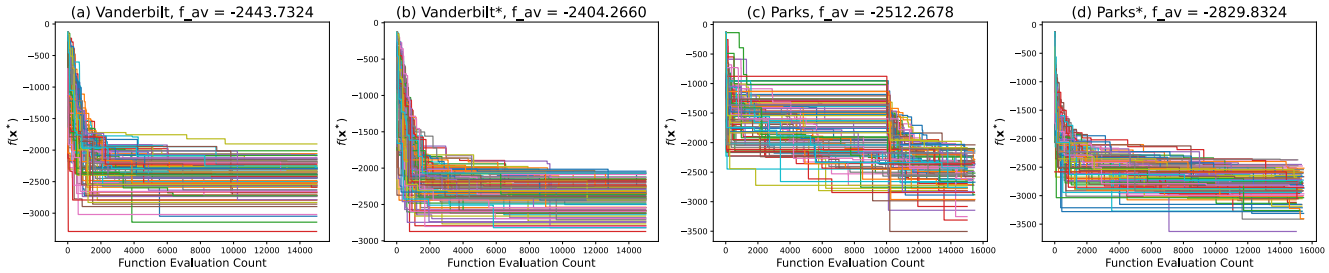

Figure 13: Comparison of runs using different Step Methods

After deducing that the Parks method proved better for the length of runs we could use, we next analysed how the initial step size affected this method, as the ability to change throughout the run changes the effect of this setting compared to using the basic step. The results of this analysis are shown in fig. 14, where we see that all initial steps perform similarly. Surprisingly, we see this method performs significantly worse than our basic step method, which suggests that these methods do not adapt sufficiently well to the topography of the problem. From these investigations, we conclude that the basic step method should be used going forward.

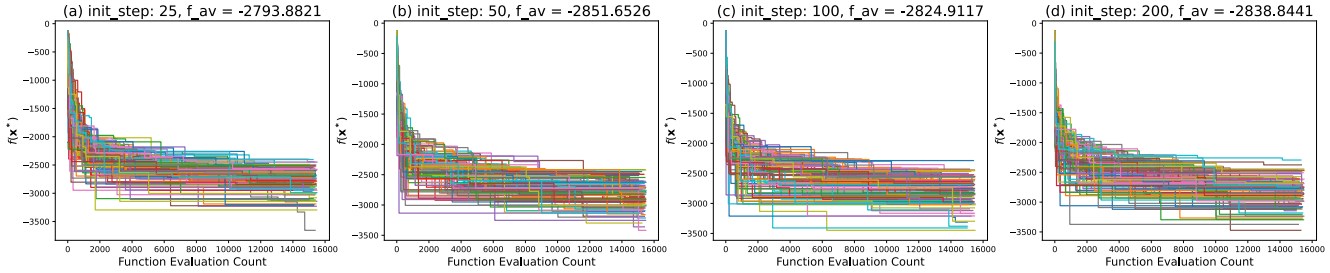

Figure 14: Comparison of Different Initial Step Sizes for the Parks Method

2.5 Markov Chain (MC) Length

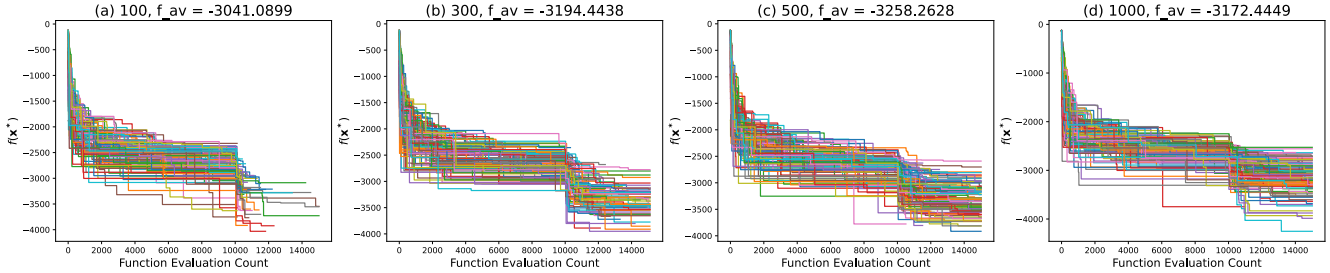

Figure 15: Comparison of Different Markov Chain Lengths

We next looked at the effect of changing our maximum MC length. We investigated this by comparing runs over different lengths and our results are shown in fig. 15, using the settings concluded from previous investigations i.e. the Kirkpatrick temperature initialisation method, $\alpha = 0.9$, an unconstrained formulation and the basic random step method. We found that a Markov Chain of 500 performed the best, although again the variation in performance between lengths of 300 to 1000 wasn't very significant. We would expect that the optimal Markov Chain Length and optimal setting for α to be dependent on each other, as they both affect the final temperature we can reach and how quickly we vary the temperature, which has a big impact on the final performance. For this reason, we compared different pairings of the two variables in fig. 16. As expected, the run with a longer MC length performed better with a smaller α and vice versa. We see our best performance so far in (c), so we compared larger MC lengths with $\alpha = 0.8$.

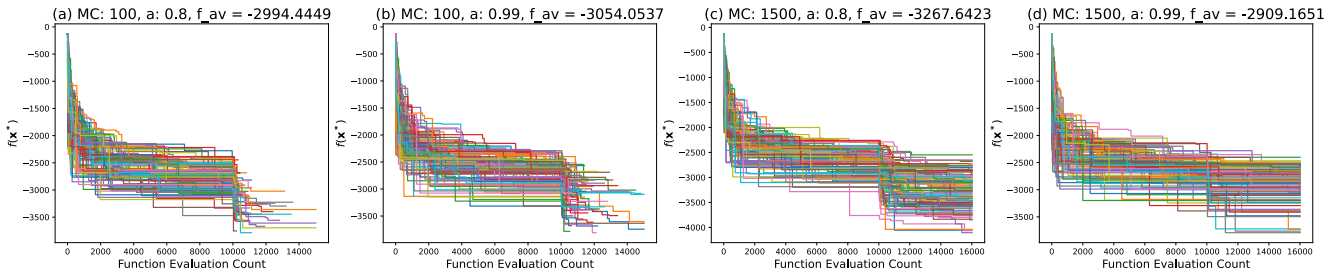

Figure 16: Comparison of Different MC Length and α Setting Pairs

2.6 SA remarks

We conclude the best settings for our SA run were the Fitzpatrick temperature initialisation method, $\alpha = 0.9$, an unconstrained formulation with our basic random-step method and an MC length of 500.

3 Tabu Search

These are the steps for the Tabu Search algorithm as written in the lectures.

Local Search:

1. Choose a starting point $\mathbf{x}$ and evaluate $f(\mathbf{x})$. Choose a step size δ to take in each input variable.
2. Record the current point as a *base point*.
3. For each input variable in turn:
   - increase by the step size, $x_i + \delta$ while fixing all other input variable values. Evaluate $f(\mathbf{x})$
   - decrease by the step size, $x_i - \delta$ while fixing all other input variable values. Evaluate $f(\mathbf{x})$
4. of all steps evaluated in step 3, make the best allowed (i.e. *non-tabu*) move.
5. If step 4 reduced $f(\mathbf{x})$, perform a pattern move by repeating the step vector. If this reduces $f(\mathbf{x})$, keep the move. Otherwise discard it.
6. Go to Step 2 (end of 1 iteration).

Tabu Search:

1. Set COUNTER = 0
2. Test Convergence.
3. If converged, stop. Otherwise perform a local search iteration.
4. If a new best solution has been found, go to step 1. If not, increment COUNTER
5. If COUNTER = INTENSIFY, intensify the search.
6. If COUNTER = DIVERSIFY, diversify the search.
7. If COUNTER = REDUCE, reduce the step size and go to step 1.
8. Go to step 2.

Whether a move is allowed is decided by the Short-Term Memory (STM), which is a list of the previous N successful points visited. Points in this list are tabu, so cannot be revisited. The search intensification is determined by the Medium-Term Memory (MTM), which is a list of the M best solutions found so far in the run. When we intensify, we take our new base point as the average point between all points in the MTM. Search diversification is determined by the Long-Term Memory (LTM). In the LTM, the whole feasible space is divided into sectors with a count of accepted solutions for each sector. Any newly accepted points increment the count for the sector they are in. On diversification, a new base point is chosen randomly in the sector with the lowest count from a uniform distribution over the sector space. On reduction, the step size is reduced by a constant factor α and the best found solution so far is chosen as the new base point. Initially we used a setting of $\alpha = 0.9$. Because we were working in problems with less than 12 input variables, we used the parameter values suggested from lectures, i.e. N = 7, M = 4, INTENSIFY = 10, DIVERSIFY = 15, REDUCE = 25. The results for these initial settings for the 2D problem are shown in figs. 17 and 18.

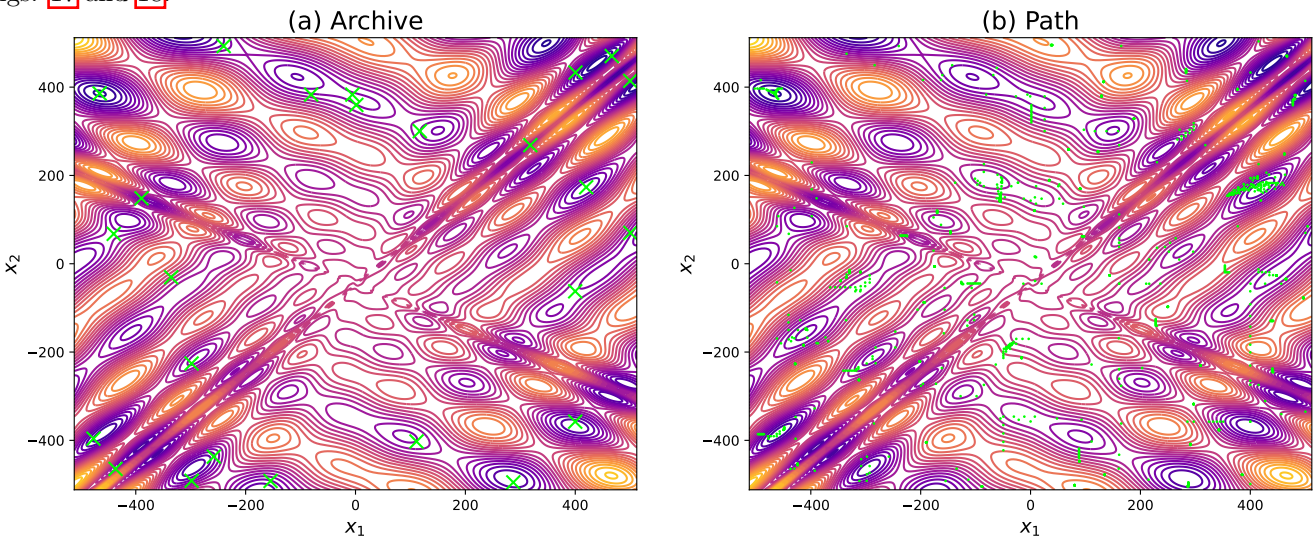

Figure 17: Contour Plot of Initial TS Run for the 2D Problem

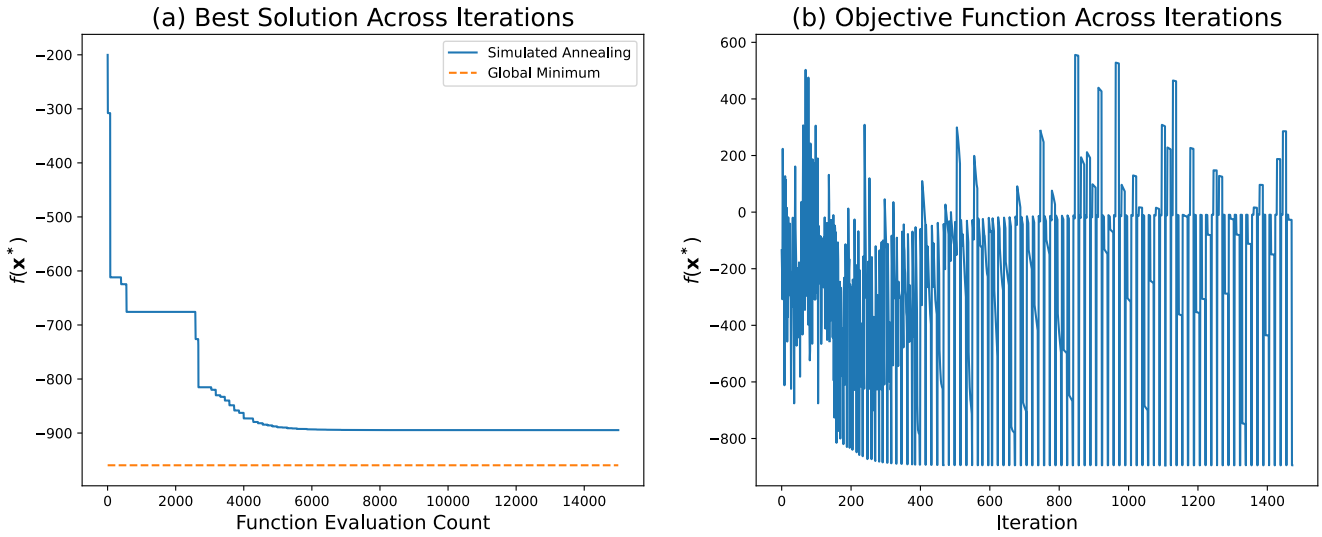

Figure 18: Progress of Initial TS Run for 2D Problem

We see that the algorithm gets good coverage across the feasible region, but does not reach the optimal value. Fig. 18(b) shows how the intensify and diversify steps cause the algorithm to spike in function evaluations, causing the high frequency of large step jumps.

3.1 Step Sizes

Our first investigation involved looking at how the algorithm performed with varying initial step sizes. We conducted 100 runs each for a range of different step sizes, and have plotted the results in fig. 19. From this, we can see that the largest initial step size (500) performed the best, which makes sense as the algorithm wants to search around as large an area as possible at the start before reducing the step size. We couldn't use a larger step size as the algorithm would barely be able to find steps in the feasible region, so using an initial step size of 500 seemed optimal.

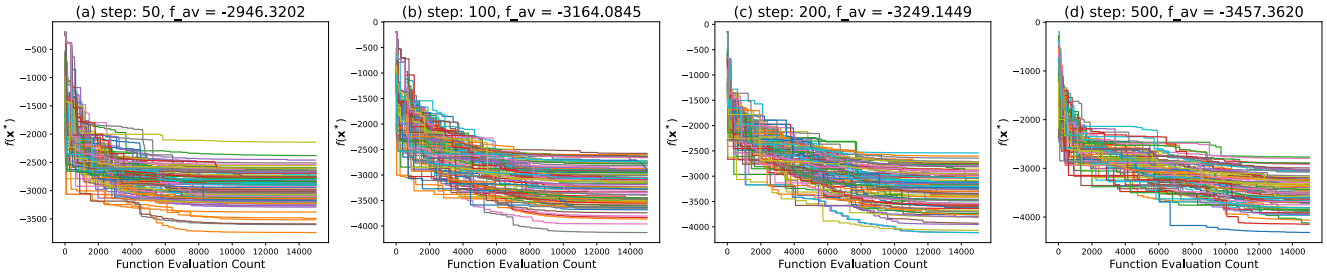

Figure 19: Comparing Performance of the TS algorithm using varying initial step sizes.

3.2 Varying α

We next wanted to look at how varying alpha affected our performance, as this would determine how quickly we reduce the step size in order to converge to optimums. We analysed this by using an initial step size of 500 and varying α, conducting 100 runs for each case. We plotted the results in fig. 20. From these plots, we can see that $\alpha = 0.9$ performed the best, which would suggest that this value balances the breadth wanted at the start of runs with the depth wanted near the end well. Similar to SA, we would expect the optimal value of α to be dependent on the initial step size. We would expect larger initial step sizes to favour smaller α so that they can reach smaller steps at the end of runs. Since our largest step size of 500 performed best at $\alpha = 0.9$, it would make sense to look at the performance of a smaller initial step size on larger α. We did this for an initial step size of 100 and plotted the results in fig. 21. We can see that this time the algorithm performs better at a higher α, however using an initial step size of 500 with $\alpha = 0.9$ still gives the best results, so that is what we continue with in further analysis.

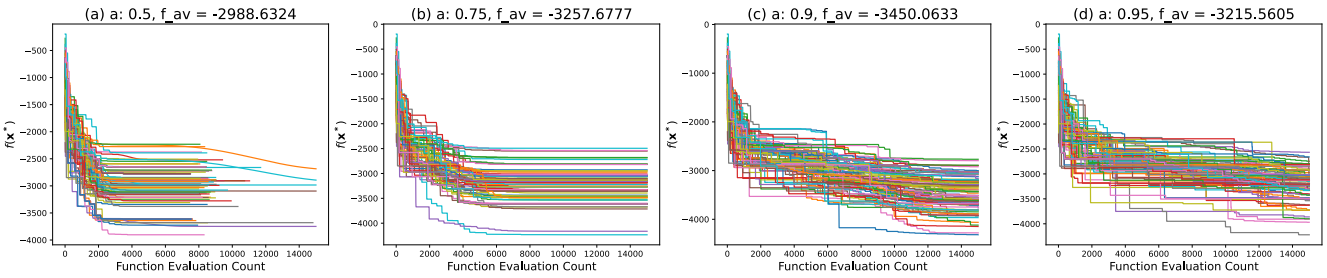

Figure 20: Varying α on with initial step size of 500.

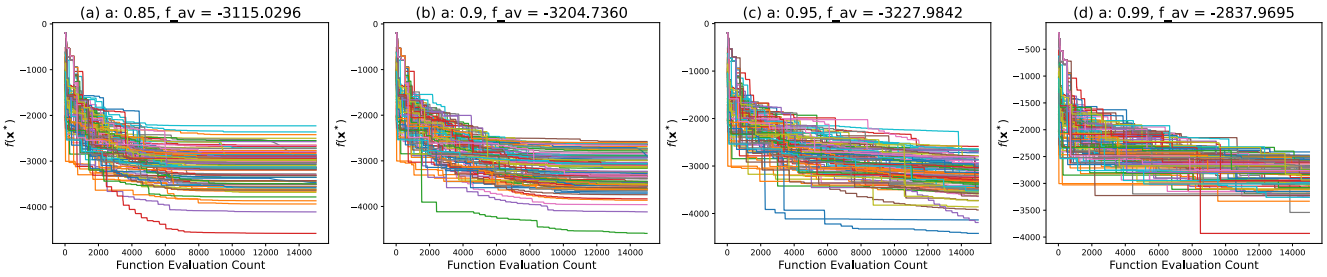

Figure 21: Varying α with initial step size of 100.

3.3 Local Search Method

The next parameter we analysed was the local search method. Until this point, we had been using the plain method described in the initial run, where we would evaluate steps in every direction before choosing a direction to go to. This can be problematic for problems of higher input variables, as evaluating all these possibilities at every step can be expensive. To analyse this, we implemented three other methods of evaluating the local search steps. The first was the random subset method, which involves evaluating a random subset of the non-tabu moves available at one time rather than all of them. For this, we define the proportion of the moves we shall analyse. For the 6D problem, we had a maximum of 12 possible moves at each step. To analyse this method, we varied the size, P, of the random subset. Our results are in fig. 22. We got the best results at $P = 6$, so the algorithm was evaluating half the number of points at every step than the simple method. This saving in evaluation allowed the search to progress further to reach more optimum values.

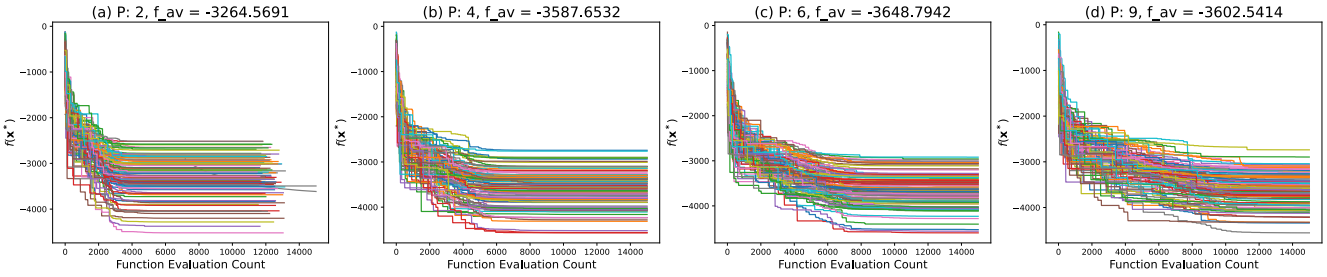

Figure 22: Comparison of Random Subset Method Performance with Varying setting of P

The next method we evaluated was the successive random subset (SRS) method, which works similarly to the random subset method, except for one difference. In this method, if a subset gave no improvement, another subset was taken from the list of non-tabu moves and again evaluated. This would repeat until an improvement was found or all non-tabu moves had been evaluated. We analysed this method similarly to the random subset method, our results displayed in fig. 23. This plot shows our best setting is at $P = 4$, and the algorithm performance is very similar to the random subset method performance.

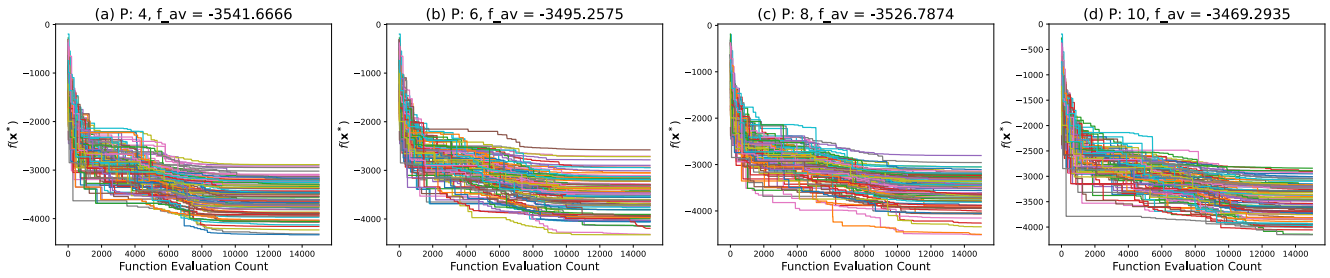

Figure 23: Comparison of the SRS Method Performance with Varying setting of P

Finally, we evaluated the variable prioritisation method (VP). The first step of this method was to evaluate the function in all directions from the base point. We would then calculate the sensitivity of the cost function to change in each input variable from eq. 11, i.e. a first order gradient approximation. We would order our input variables by sensitivity and then restrict our search to only changing the 50% of input variables that were most sensitive for a number of iterations. After that, we would repeat the process from step 1. We analysed this method by varying the number of iterations in cycle, and have plotted our results in fig. 24. We found that this method performed significantly worse than the others evaluated, and the change in cycle lengths was almost completely insignificant in performance. We suspect that this is because the surface of the problem is so variable that limiting the algorithm to half the search directions for even small periods causes large missed opportunities in descent. Because the other methods performed better, we decided to continue with the random subset method for future analysis due to simplicity.

$$s_i = \left\| \frac{f(\mathbf{x} + \delta_i) - f(\mathbf{x} - \delta_i)}{2\delta_i} \right\| \tag{11}$$

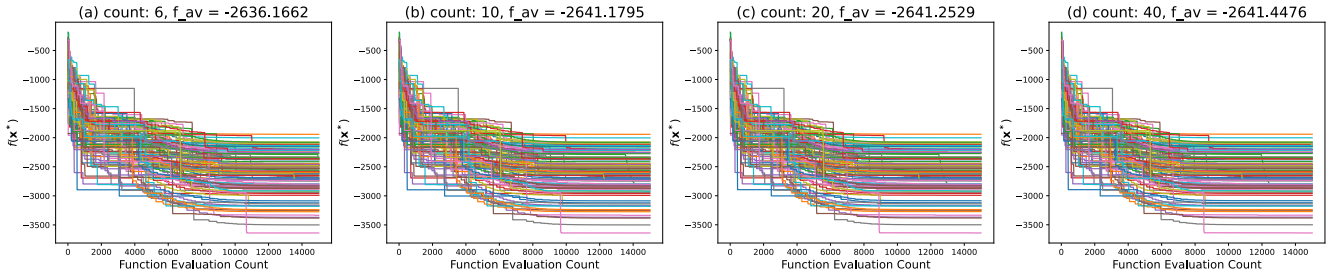

Figure 24: Comparison of the VP Method Performance with Varying sizes of Iteration Cycles (count)

3.4 Unconstrained

Similarly to our evaluation in the SA algorithm, since we are told that the optimal solution is on a boundary, we also compared our algorithm performance across the constrained and unconstrained formulations with varying weights on the penalty functions. Our results are displayed in fig. 25. We see that there is in-fact little variation between our results. This could be because the TS algorithm is more capable of moving directly along a boundary anyway if it is beneficial, as each step only ever changes the value in one input variable of x. Because of this and due to simplicity, we continue with our constrained formulation for future analysis.

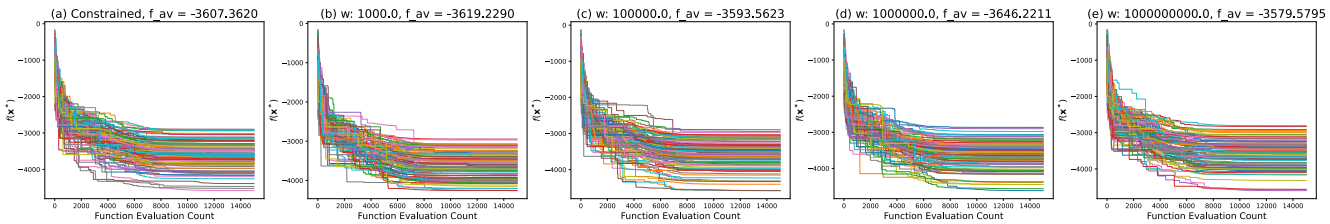

Figure 25: Comparing our Constrained and Unconstrained Formulations with Varying Penalty Weights

3.5 M and N

The next investigation was on our STM and LTM sizes. It made sense that increasing the size of these could improve our performance as we would be less susceptible to returning to previously evaluated positions, and perhaps intensify our search using more information. We tried 4 different pairings of the values of these sizes and the results are seen in fig. 26. This shows that there was little change in the algorithm performance with varying LTM and MTM sizes, which suggests that enough information is already held within the smallest configuration, as we have been using so far. Due to the slight improvement in our results, we decided to continue with $N = 20, M = 15$ for future analysis.

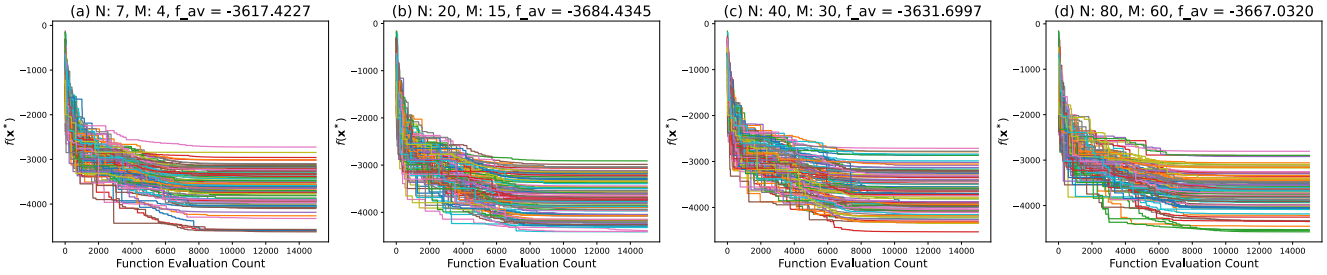

Figure 26: Comparing the effect of different Memory Sizes on Performance

3.6 Intensify, Diversify and Reduce

We also wanted to look at changing the parameters associated with when our algorithm intensified, diversified and reduced the step size. These parameters have a very large effect on the travel of the algorithm, as the intensify and diversify steps both cause large jumps in the space being explored and the reduce step has a large impact on how precisely the algorithm explores the area it is in. We again tried four different settings of this group of parameters and compared the results, as shown in fig. 27. Here, there is again not much variance but we see a slight jump in performance for the setting (c), which uses larger counts than we were using previously. We also see that our runs continue to make progress for much longer than we see for (a) or (b). This suggests that allowing the algorithm to explore further in each area before taking an intensification, diversification or reduction step gives it a better chance of finding optimums. Given this, we continue with the settings described in (c) for future analysis.

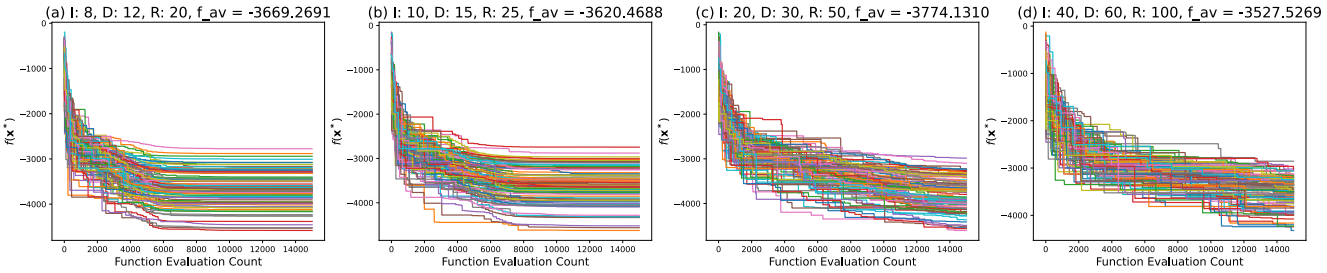

Figure 27: Evaluating the Change of Counts to Intensify, Diversify and Reduce on Performance

3.7 Diversification Sector Splits

Our final investigation involved changing the amount of sectors that we split our feasible region into, for use in the LTM and diversification steps. Our logic was that more splits would allow a more detailed exploration of the entire feasible space, so we compared splitting each input variable into 2 sections vs 3 sections, and show the results in fig. 28. We see that the performance is very similar, with a slight advantage in setting (a). This could be because splitting our feasible region too finely might make our algorithm too spread out, and not give it enough time to explore regions it has already been to as it is dragged to a new region too often. With each input variable split in half, we have $2^6 = 32$ sectors, which gives an average of $\frac{15000}{32} = 468.75$ evaluations per sector. Variables split into thirds gives $3^6 = 729$ sectors, which is an average of $\frac{15000}{729} = 20.56$ evaluations per sector. The sizes of the sectors will decrease proportionally as well, but the algorithm has to populate many more sectors in the latter scenario.

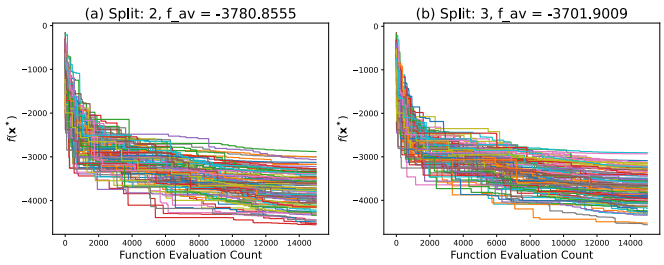

Figure 28: Evaluating the Effect of the Amount of Sectors in the LTM on Performance

3.8 TS Remarks

We have explored a wide range of the choices available to us in the implementation of the Tabu Search Algorithm. We have concluded that for this problem, one of the most optimal settings to use is the random subset local search method, with a subset proportion 6/12, an initial step size of 500, splitting the feasible space of each variable in half when creating sectors, a constrained formulation, $\alpha = 0.9$, $N = 20, M = 15$, and the intensification, diversification and reduction counts to 20, 30 and 50 respectively. This set up is the one used in fig. 28(a).

4 Conclusions

This report has shown that the settings of parameters in each of these search algorithms has huge effects on performance, even with small changes. Overall, we see that the Tabu Search Method performs significantly better than the Simulated Annealing Algorithm for this problem, as our best formulation for the SA algorithm gave an average minimised solution of -3258 from fig. 15(c), compared to -3780 from fig. 28(a) in the TS algorithm. We think this is mainly because the TS algorithm is much more mobile in its search and spreads over the feasible region more, especially thanks to the diversification step ensuring a somewhat even spread of exploration over the space. This could also be because even though the region is not disjoint, the very large peaks that occur throughout the space can almost simulate a disjoint nature to the problem, for which the TS algorithm is known to be better at handling.

References

Kirkpatrick, S., C. D. Gelatt, and M. P. Vecchi (1983). "Optimization by simulated annealing". In: *Science (New York, N.Y.)* 220.4598, pp. 671–680. ISSN: 0036-8075. DOI: 10.1126/SCIENCE.220.4598.671. URL: https://pubmed.ncbi.nlm.nih.gov/17813860/.

Kirkpatrick, Scott (Mar. 1984). "Optimization by simulated annealing: Quantitative studies". In: *Journal of Statistical Physics 1984 34:5* 34.5, pp. 975–986. ISSN: 1572-9613. DOI: 10.1007/BF01009452. URL: https://link.springer.com/article/10.1007/BF01009452.

Parks, Geoffrey Thomas (2017). "An Intelligent Stochastic Optimization Routine for Nuclear Fuel Cycle Design". In: *http://dx.doi.org/10.13182/NT90-A34350* 89.2, pp. 233–246. ISSN: 00295450. DOI: 10.13182/NT90-A34350. URL: https://www.tandfonline.com/doi/abs/10.13182/NT90-A34350.

Vanderbilt, David et al. (1984). "A Monte Carlo Simulated Annealing Approach to Optimization over Continuous Variables". In: *JCoPh* 56.2, pp. 259–271. ISSN: 0021-9991. DOI: 10.1016/0021-9991(84)90095-0. URL: https://ui.adsabs.harvard.edu/abs/1984JCoPh..56..259V/abstract.

White, Steve R. (May 2008). "Concepts of scale in simulated annealing". In: *AIP Conference Proceedings* 122.1, p. 261. ISSN: 0094-243X. DOI: 10.1063/1.34823. URL: https://aip.scitation.org/doi/abs/10.1063/1.34823.

Appendices

A Eggholder.py

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import LinearLocator
from matplotlib import cm
from tqdm import tqdm
import matplotlib.gridspec as gridspec
from sys import path_importer_cache
import os

def eggholder(x):
    '''Evaluates Eggholder function for d-length vector x.'''
    assert feasible_check(x),"x out of bounds: {}".format(x)

    d = len(x)
    fx = 0
    for i in range(0, d-1):
        fx += -1*(x[i+1] + 47)*np.sin(np.sqrt(np.abs(x[i+1] + 0.5*x[i] + 47)))\
              -x[i]*np.sin(np.sqrt(np.abs(x[i] - x[i+1] - 47)))
    return fx

def eggholder_unconstrained(x, w_val=1e6, T=np.inf):
    '''Evaluates Eggholder function for d-length vector x.'''
    # assert feasible_check(x),"x out of bounds: {}".format(x)

    d = len(x)
    w = np.ones(d) * 1e6
    c = violations(x, d)
    fx = 0
    for i in range(0, d-1):
        fx += -1*(x[i+1] + 47)*np.sin(np.sqrt(np.abs(x[i+1] + 0.5*x[i] + 47)))\
              -x[i]*np.sin(np.sqrt(np.abs(x[i] - x[i+1] - 47)))
    fx += (1/T) * w@c
    return fx

def violations(x, d):
    c = np.zeros(d)
    for i in range(d):
        c[i] = np.max([0, (x[i] - 512), (-x[i] - 512)])
    return c


def plot_eggholder2D(archive=[], path=[], run_name = 'initial'):
    '''Plot the Eggholder Function for 2D vector x.
    inputs:
      - archive    {f(x): [x]}
      - path       {f(x): [x]}
    '''
    N = 1000 # for plotting
    x_range = np.linspace(-512,512,N)
    y_range = np.linspace(-512,512,N)
```

```
xgrid, ygrid = np.meshgrid(x_range, y_range)
zvals = np.zeros((N,N))

# fill zvals according to whether file exists yet or not.
if os.path.exists('zvals{}.npy'.format(N)):
    zvals = np.load('zvals{}.npy'.format(N))
else:
    # create zvals array to represent the surface.
    for i in tqdm(range(N)):
        for j in range(N):
            zvals[j,i] = eggholder([x_range[i],y_range[j]])
    np.save('zvals{}.npy'.format(N), zvals)

# form arrays of points to plot on the surface
A = len(archive)
P = len(path)

x_archive = np.zeros(A)
y_archive = np.zeros(A)
z_archive = np.zeros(A)
x_path = np.zeros(P)
y_path = np.zeros(P)
z_path = np.zeros(P)

for index, key in enumerate(archive):
    z_archive[index] = key
    x_archive[index] = archive[key][0]
    y_archive[index] = archive[key][1]

for index, key in enumerate(path):
    z_path[index] = key
    x_path[index] = path[key][0]
    y_path[index] = path[key][1]

if path == []:
    # plot the surface of the 2D eggholder function

    fig = plt.figure(figsize=(12,9))
    ax = fig.add_subplot(1,1,1,projection='3d')
    surf = ax.plot_surface(xgrid, ygrid, zvals, rstride=5, cstride=5,
                           linewidth=0, cmap=cm.plasma)

    ax.set_title('Surface Plot of the 2D Eggholder Function', fontsize=16)
    ax.set_xlabel('$x_1$', fontsize=14)
    ax.set_ylabel('$x_2$', fontsize=14)
    ax.set_zlabel('$f(\mathbf{x})$', fontsize=14)
    ax.scatter(x_archive, y_archive, z_archive, color = 'r')
    plt.tight_layout()

# plot the archived solutions on the contour plot

gs = gridspec.GridSpec(1,2)
fig2 = plt.figure(figsize=(16, 6), dpi=80)
ax1 = fig2.add_subplot(gs[0,0])
ax2 = fig2.add_subplot(gs[0,1])
```

```
    # fig2 = plt.figure(figsize=(12,9))
    # ax = fig2.add_subplot(1,1,1)
    ax1.contour(xgrid, ygrid, zvals, 30, cmap=cm.plasma, zorder = -1)
    if archive != []:
        ax1.set_title('(a) Archive', fontsize=18)
        ax1.scatter(x_archive, y_archive, zorder = 1, s=100, marker = 'x',
                    color = 'lime')
    else:
        ax1.set_title('Contour Plot of the 2D Eggholder Function', fontsize=16)
    ax1.set_xlabel('$x_1$', fontsize=14)
    ax1.set_ylabel('$x_2$', fontsize=14)
    cbar = fig2.colorbar(surf, aspect=15)
    cbar.set_label('$f(\mathbf{x})$', rotation=0, fontsize=14)

    z = [min(i) for i in zvals]

#     plot the path solutions on the contour plot

    if path != []:

        ax2.contour(xgrid, ygrid, zvals, 30, cmap=cm.plasma, zorder = -1)
        ax2.scatter(x_path, y_path, zorder = 1, s=0.8, marker = 'o', color = 'lime')

        ax2.set_title('(b) Path', fontsize=18)
        ax2.set_xlabel('$x_1$', fontsize=14)
        ax2.set_ylabel('$x_2$', fontsize=14)
        cbar = fig2.colorbar(CS, aspect=15)
        cbar.set_label('$f(\mathbf{x})$', rotation=0, fontsize=14)

        fig4 = plt.figure(figsize=(12,9))
        ax = fig4.add_subplot(1, 1, 1)
        plt.plot(z_path)
        ax.set_xlabel('Iteration', fontsize=14)
        ax.set_ylabel('$f(\mathbf{x})$', fontsize=14)
        ax.set_title('Objective Function value over Iterations', fontsize=16)

    fig2.savefig('Figures/contours_{}.pdf'.format(run_name), format = 'pdf',
                 transparent=True, bbox_inches='tight')
    plt.tight_layout()
    plt.show()
    return(z_path)

def plot_progress(fx_progress, z_path, d, run_name, f_evals_count_list = []):
    '''Plot the progress of best solution found across number of iterations'''

    gs = gridspec.GridSpec(1,2)
    fig2 = plt.figure(figsize=(16, 6), dpi=80)
    ax1 = fig2.add_subplot(gs[0,0])
    ax2 = fig2.add_subplot(gs[0,1])

    if f_evals_count_list == []:
        ax1.plot(fx_progress, label='Simulated Annealing')
    else:
        ax1.plot(f_evals_count_list,fx_progress, label='Simulated Annealing')
    ax1.set_title('(a) Best Solution Across Iterations', fontsize=18)
    ax1.set_xlabel('Function Evaluation Count', fontsize=14)
```

```
    ax1.set_ylabel('$f(\mathbf{x^*})$', fontsize=14)

    if d == 2:
        if f_evals_count_list == []:
            ax1.plot(np.ones(len(fx_progress))*-959.639453, '—', label='Global Minimum')
        else:
            ax1.plot(f_evals_count_list, np.ones(len(fx_progress))*-959.639453,
                    '—', label='Global Minimum')
        ax1.legend()

    ax2.plot(z_path)
    ax2.set_title('(b) Objective Function Across Iterations', fontsize=18)
    ax2.set_xlabel('Iteration', fontsize=14)
    ax2.set_ylabel('$f(\mathbf{x^*})$', fontsize=14)

    fig2.savefig('Figures/progress_{}.pdf'.format(run_name), format = 'pdf',
            transparent=True, bbox_inches='tight')
    plt.tight_layout()
    plt.show()

def plot_progress_multseed(fx_progress_list, d, title=None):
    '''Plot the progress of best solution found across number of iterations'''
    plt.figure()
    max_length = max(len(fx_progress_list[i]) for i in range(len(fx_progress_list)))
    for seed in range(len(fx_progress_list)):
        plt.plot(fx_progress_list[seed], label=seed)
    plt.xlabel('Function Evaluation Count', fontsize=14)
    plt.ylabel('$f(\mathbf{x^*})$', fontsize=14)

    if d == 2:
        plt.plot(np.ones(max_length)*-959.639453, '—', label='Global Minimum')

    if title == None:
        plt.title('Best Soln Across Iterations for Various Seeds', fontsize=14)
        title = 'General'
    else:
        plt.title(title, fontsize=14)
    plt.savefig('Figures/{}_multseed.pdf'.format(title), format = 'pdf',
            transparent=True, bbox_inches='tight')
    plt.show()

def plot_progress_multvar(fx_progress_dict, d, filename = 'General', count=False,
            f_evals_count_dict = None):
    '''Plot the progress of best solution found across number of iterations'''

    alphabet = ['a','b','c','d','e']
    num_graphs = len(fx_progress_dict)

    gs = gridspec.GridSpec(1,num_graphs)
    fig = plt.figure(figsize=(7*num_graphs, 5), dpi=80)

    for index, value in enumerate([*fx_progress_dict]):
        progress_list = fx_progress_dict[value]
        if count:
            f_evals_count_list = f_evals_count_dict[value]
        max_length = max(len(progress_list[i]) for i in range(len(progress_list)))
```

```
        ax = fig.add_subplot(gs[0,index])
        final_vals = []
        for seed in range(len(progress_list)):
            if count:
                ax.plot(f_evals_count_list[seed], progress_list[seed], label = seed)
            else:
                ax.plot(progress_list[seed], label = seed)
            final_vals.append(progress_list[seed][-1])
            if d == 2:  ax.plot(np.ones(15500)*-959.639453, '--', label='Global Minimum')
        ax.set_xlabel('Function Evaluation Count', fontsize=14)
        ax.set_ylabel('$f(\mathbf{x^*})$', fontsize=14)
        f_av = np.mean(final_vals)
        ax.set_title('({}) {}, f_av = {:.4f}'.format(alphabet[index], value, f_av))

    plt.savefig('Figures/{}.pdf'.format(filename), format = 'pdf',
            transparent=True, bbox_inches='tight')
    plt.show()

def feasible_check(x):
    '''Checks if the vector x is in the feasible region.
    Returns True if feasible and False if not.
    '''
    if np.any(np.abs(x) > 512):
        return False
    else:
        return True
```

B init_archive.py

```
import numpy as np
import eggholder as egg


def init_x(n,d):
    '''Chooses starting point x by randomly choosing d
    possible starting points and picking the one that minimises
    f most.

    inputs
        d(int): dimension of x
        n(int): number of compared starting values

    returns the best x of the randomly chosen starting values
    '''
    xbest = np.zeros(d)
    fxbest = egg.eggholder(xbest)
    for i in range(n):
        x_try = np.random.rand(d)*1024-512
        fx_try = egg.eggholder(x_try)
        if fx_try < fxbest:
            xbest = x_try
            fxbest = fx_try
    return xbest

L = 25 # max number of archived solutions

def update_archive(x,fx,archive):
    '''Updates the archive if necessary.
    inputs
        x:          trial point
        fx:         cost function evaluated at x
        archive:    current set of archived solutions

    returns updated set of archived solutions and a boolean on whether
    the archive set has changed or not.
    '''
    min_val_found = False
    archive_updated = False

    listed_archive = [*archive]
    best_archived = min(listed_archive)
    worst_archived = max(listed_archive)
    l = len(listed_archive)
    assert l <= 25

    if fx < best_archived:
        min_val_found = True
    # dissimilarity parameters
    D_min = 60
    D_sim = 6

    # check if not dissimilar to any previously archived solution
```

```
close_archived_point = None
for fx0 in archive:
    D = np.linalg.norm(archive[fx0]-x)
    if D < D_min:
        close_archived_point = fx0
        if D < D_sim and fx < fx0: # meets archive criterion (4)
            del archive[fx0]
            archive[fx] = x
            archive_updated = True
            return archive, archive_updated

if close_archived_point == None:
    if l == 25: # archive criterion (2) satisfied
        del archive[worst_archived]
    archive[fx] = x # archive criterion (1) satisfied
    archive_updated = True
    return archive, archive_updated
elif min_val_found: # archive criterion (3) satisfied
    del archive[close_archived_point]
    archive[fx] = x
    archive_updated = True
    return archive, archive_updated
else:
    return archive, archive_updated
```

1 C: Simulated Annealing Code

1.1 Importing Libraries

[ ]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import LinearLocator
from matplotlib import cm
from tqdm import tqdm
import matplotlib.gridspec as gridspec
from sys import path_importer_cache
import os
import eggholder as egg
import init_archive as inar
```

1.2 Initial Temperature and Random Step Functions

[ ]:
```
def init_temp_kirkpatrick(x0, fx0):
    """Estimates the initial temperature by conducting an initial search
    of M steps and accepting all of them. Then we take all steps that increased
    the objective function and calculated the mean increase df_pos. Then␣
 ↪calculate
    T0 = -(df_pos)/ln(chi_0) [Kirkpatrick, 1984].

    inputs:
        x0: starting value of x
        fx0: function evaluated at x0

    returns estimated initial temperature T0
    """
    M = 1000
    chi_0 = 0.8

    f_array = []
    for i in range(M):
        x = random_step_basic(x0,d)
        fx = egg.eggholder(x)
        if fx > fx0:
            f_array.append(fx-fx0)
        x0 = x.copy()
        fx0 = fx.copy()
    df_pos = np.mean(f_array)
    T0 = -df_pos/np.log(chi_0)
    return T0

def init_temp_white(x0):
```

```
    """Estimates the initial temperature by conducting an initial search
    of M steps, accepting all random steps, and finding the standard deviation
    sigma of f across the steps. Then T0 = sigma [White, 1984]

    inputs:
        x0: starting value x

    returns estimated initial temperature T0
    """
    M = 1000

    f_array = []
    for i in range(M):
        x = random_step_basic(x0,d)
        f_array.append(egg.eggholder(x))
        x0 = x.copy()
    T0 = np.std(f_array)
    return T0

def random_step_basic(x0, d, c=25, constrained = True):
    '''Taking a random step from point x0.

    inputs
        x0: point x we are taking the random step from
        d: dimension of x
        c: max change allowed of each variable.
        constrained: boolean for if using constraints or penalty functions.

    returns new point x.
    '''
    x = np.zeros(d)
    for i in range(d):
        new = x0[i] + np.random.uniform(-c, c)
        if constrained:
            while abs(new) > 512:
                new = x0[i] + np.random.uniform(-c, c)
        x[i] = new
    return x

def vanderbilt_recal(path, S0):
    '''Recalculating S and Q for taking random steps in the vanderbilt method.
    inputs
        path:   {fx:x}                  dict of all points accepted
        S0:     (d,d)-length matrix     previous value of covariance matrix S

    returns recalculated Q and S.
    '''
```

```
    alpha = 0.1
    omega = 2.1
    if len(path) == 1:
        X = np.eye(d)
    else:
        path_values = np.array([*path.values()])
        X = np.cov(path_values.T, bias = True)

    S = (1-alpha)*S0 + alpha*omega*X
    try:
        Q = np.linalg.cholesky(S)
        return Q, S
    except np.linalg.LinAlgError:
        print("Cholesky decomposition failed.")
        raise ValueError


def random_step_vanderbilt(x0, d, Q, constrained = False):
    '''Taking a random step from point x0 using the vanderbilt method.

    inputs
        x0:             d-length vector         point x we are taking the random␣
↪step from
        d:              int                     dimension of x
        Q:              (d,d)-length matrix     cholesky decomposition of the␣
↪estimated covariance
        constrained:    boolean                 using constraints or penalty␣
↪functions

    returns new point x, new covariance matrix S and a boolean on whether the
    Cholesky decomposition has failed.
    '''

    u = (np.random.rand(d)*2-1)*np.sqrt(3)
    x_new = x0 + Q@u
    if constrained:
        count = 0
        count_limit = 5e3
        while not egg.feasible_check(x_new) and count < count_limit:
            u = np.random.rand(d)*np.sqrt(3)
            x_new = x0 + Q@u
            count += 1
        if count >= count_limit:
            raise RuntimeError
    return x_new
```

```
def parks_recal(u, D):
    '''Recalculating the D matrix for taking random steps using the Parks method.
    D is a diagonal matrix defining the max change for each dimension of x.
    inputs
        u:      d-length vector         previous random step taken
        D:      (d,d)-length matrix     Controls max change in each dimension

    returns new calculated D and R.
    '''
    alpha = 0.1
    omega = 2.1

    R = np.abs(np.diag((D@u)))
    D_new = (1 - alpha) * D + alpha * omega * R

    return D_new, R


def random_step_parks(x0, d, D, constrained = False):
    '''Taking a random step from point x0 using the third method from lectures,␣
 ↪proposed by
    GT Parks in 1990.

    inputs
        x0:     d-length vector         point x we are taking the random step␣
 ↪from
        d:      int                     dimension of x
        D:      (d,d)-length matrix     Matrix controlling max change in each␣
 ↪dimension

    returns new point x and random step variable u.
    '''
    u  = np.random.rand(d)*2-1
    x_new = x0 + D.dot(u)

    if constrained:
        count_limit = 5e3
        count = 0
        while not egg.feasible_check(x_new) and count < count_limit:
            u  = np.random.rand(d)*2-1
            x_new = x0 + D.dot(u)
            count += 1
        if count >= count_limit:
            raise RuntimeError

    return x_new, u
```

```
def init_SA(d, seed, temp_method = 'Kirkpatrick'):
    '''Create starting parameters for SA run.'''
    x0 = inar.init_x(1, d)
    fx0 = egg.eggholder(x0)
    T0 = np.inf

    if temp_method == 'Kirkpatrick':
        T0 = init_temp_kirkpatrick(x0, fx0.copy())
    else:
        T0 = init_temp_white(x0)

    path = {fx0:x0.copy()}
    archive = {fx0:x0}
    np.random.seed(seed)

    return x0, fx0, T0, path, archive
```

1.3 Main Simulated Annealing Function (2D)

This is the initial implementation of the simulated annealing function for a 2D vector space, to check it is working properly and visualise the progression more easily.

[ ]:
```
def SA_run(d = 2, step_method = 1, seed = 1, L_k = 500, c = 25, alpha = 0.99,
        run_name = '1', temp_method = 'Kirkpatrick', constrained = True, w_val =␣
↪1e6):
    '''Conduct a run of Simulated Annealing under specified conditions.
    inputs
        d:              int     dimension of datapoints.
        step_method:    int     method used to take random steps
        seed:           float   random seed for different runs
        L_k:            int     length of Markov Chain
        c:              float   max step size for step method 1.
        alpha:          float   temperature change ratio
        run_name:       str     identifier for the run if required
        temp_method:    str     choose which method to use in initialising␣
↪temperature
        constrained:    bool    decide whether to use active constraints or␣
↪penalty functions

    returns the progress in best solution over iterations for the specified␣
↪setting.
    '''
    x0, fx0, T0, path, archive = init_SA(d, seed, temp_method)

    current_best = fx0.copy() # this will be used to track restarts.
    x_best = x0.copy()
```

```
    new_best = fx0.copy() # this will be used to track restarts.
    x_old = x0.copy()
    fx_old = fx0.copy()

    # variables for different step methods.
    S = np.eye(d) * c
    D = np.eye(d) * c
    R = np.eye(d)

    f_evals_count = 0
    fx_progress = []
    T = T0
    restart_count = 0 # controls when to restart to the current best solution.
    while f_evals_count < 15000:
        if step_method == 2:
            try:
                Q, S = vanderbilt_recal(path, S)
            except ValueError:
                break
        n_min = 0.6*L_k # number of acceptances
        trial_count = 0 # controls number of iterations before adjusting the␣
↪temperature
        acceptance_count = 0 # controls number of iterations before adjusting␣
↪the temperature

        while trial_count < L_k and acceptance_count < n_min:
            fx_progress.append(current_best)
            trial_count += 1

            # generate a new solution
            if restart_count > 10000:
                restart_count = 0
                fx_old = current_best.copy()
                x_old = archive[fx_old].copy()
                break
            else:
                if step_method == 1:
                    x_new = random_step_basic(x_old,d, c, constrained)
                elif step_method == 2:
                    try:
                        x_new = random_step_vanderbilt(x_old, d, Q, constrained)
                    except RuntimeError:
                        print('failed to find feasible point.')
                        break
                else:
                    try:
                        x_new, u = random_step_parks(x_old, d, D, constrained)
```

```
                except RuntimeError:
                    print('failed to find feasible point.')
                    break
            if constrained:
                fx_new = egg.eggholder(x_new)
            else:
                fx_new = egg.eggholder_unconstrained(x_new, w_val, T)
            f_evals_count += 1

        # assess the solution and choose whether to accept.
        accept = False
        del_f = fx_new - fx_old
        if del_f < 0:
            accept = True
        elif step_method == 3:
            d_bar = 0
            for i in range(len(x0)):
                d_bar += R[i,i]**2
            d_bar = np.sqrt(d_bar)
            p = np.exp(-del_f/(T*d_bar))
            accept = np.random.rand() < p
        else:
            p = np.exp(-del_f/T)
            accept = np.random.rand() < p

        # if accepting, try archiving.
        if accept:
            acceptance_count += 1
            archive, updated = inar.update_archive(x_new, fx_new, archive)
            if updated:
                new_best = min([*archive]).copy()
            path[fx_new] = x_new.copy()
            x_old = x_new.copy()

            if step_method == 3:
                D, R = parks_recal(u, D)

        if new_best == current_best:
            restart_count += 1
        elif new_best < current_best:
            current_best = new_best.copy()
            x_best = archive[current_best].copy()
        else:
            print('failed')
            raise ValueError
    # adjust the temperature
    T *= alpha
```

```
        if acceptance_count == 0:
            break

    fx_star = min([*archive])
    print("\nBest solution: \n", fx_star, archive[fx_star])
    print(len(path))
    print(len(archive))
    if step_method == 2:
        print('Q: ',Q)
    elif step_method == 3:
        print('D: ',D)
    z = []
    if d==2:
        # Draw contour plot and evolution of best solution found across␣
 ↪iterations
        z = egg.plot_eggholder2D(archive, path, run_name)
    egg.plot_progress(fx_progress, z, d, run_name)
    return fx_progress
```

[ ]:
```
d = 2
SA_run(d,1,1,500,50,0.9,'unconstrained','Kirkpatrick',False, 1e6)
```

1.4 Large Runs

1.4.1 Varying Just the Seed

[ ]:
```
d=6

fx_progress_list = []
for j in tqdm(range(100)):
    fx_progress_list.append(SA_run(d, 1, j, 500, 25, 0.9, 'unconstrained',␣
 ↪'Kirkpatrick', False, 1e9))

egg.plot_progress_multseed(fx_progress_list, d, title = '1 billion')
```

1.4.2 Temperature Initialisation Methods

[ ]:
```
fx_progress_dict = {}
alpha = 0.9
for method in ['White', 'Kirkpatrick']:
    for d in [2,6]:
        graph_name = '{}_{}D'.format(method, d)
        print(graph_name)
        progress_list = []
        for i in range(60):
            progress_list.append(SA_run(d, 1, i, 500, 25, alpha,␣
 ↪'temp_comparisons', method, True))
```

```
        fx_progress_dict[graph_name] = progress_list

egg.plot_progress_multvar(fx_progress_dict, d,␣
 ↪filename='Temp_init_comparisons_alpha={}'.format(alpha))
```

1.4.3 Alpha Variation

[ ]:
```
fx_progress_dict = {}
for alpha in [0.85, 0.9, 0.95, 0.99]:
    graph_name = alpha
    print(graph_name)
    progress_list = []
    for i in tqdm(range(100)):
        progress_list.append(SA_run(d, 1, i, 500, 25, alpha, 'temp_comparisons',␣
 ↪'Kirkpatrick', True))
        fx_progress_dict[graph_name] = progress_list

egg.plot_progress_multvar(fx_progress_dict, d,␣
 ↪filename='Alpha_comparisons_Kirkpatrick')
```

1.4.4 Investigating Unconstrained

[ ]:
```
d = 6
fx_progress_dict = {}
for w_val in [0, 1e3, 1e5, 1e6, 1e9]:
    graph_name = w_val
    if w_val == 0:
        constrained = True
        graph_name = 'Constrained'
    else:
        constrained = False
    print(graph_name)
    progress_list = []
    for i in tqdm(range(100)):
        progress_list.append(SA_run(d, 1, i, 500, 25, 0.9,␣
 ↪'unconstrained_comparisons', 'Kirkpatrick', constrained, w_val))
        fx_progress_dict[graph_name] = progress_list

egg.plot_progress_multvar(fx_progress_dict, d,␣
 ↪filename='Unconstrained_Comparison_2')
```

1.4.5 Investigating Step Size

[ ]:
```
d = 6
fx_progress_dict = {}
for step_size in [25, 50, 75, 100]:
    graph_name = 'max_step: {}'.format(step_size)
    print(graph_name)
    progress_list = []
    for i in tqdm(range(100)):
        progress_list.append(SA_run(d, 1, i, 500, step_size, 0.9,␣
 ↪'unconstrained_comparisons', 'Kirkpatrick', False, 1e6))
        fx_progress_dict[graph_name] = progress_list

egg.plot_progress_multvar(fx_progress_dict, d, filename='Step_Size_Comparison')
```

1.4.6 Investigating Markov Chain Length

[ ]:
```
d = 6
fx_progress_dict = {}
for chain_length in [100, 300, 500, 1000]:
    graph_name = chain_length
    print(graph_name)
    progress_list = []
    for i in tqdm(range(100)):
        progress_list.append(SA_run(d, 1, i, chain_length, 50, 0.9,␣
 ↪'unconstrained_comparisons', 'Kirkpatrick', False, 1e5))
        fx_progress_dict[graph_name] = progress_list

egg.plot_progress_multvar(fx_progress_dict, d,␣
 ↪filename='Markov_Chain_Length_Comparison')
```

1.4.7 Investigating Random Step Methods

[ ]:
```
d = 6
constrained = False
fx_progress_dict = {}
for index,step_method in enumerate(['Vanderbilt', 'Parks']):
    for w_val in [1e5, np.inf]:
        title_edit = ''
        if w_val == np.inf:
            title_edit = '*'
            if step_method == 'Parks':
                constrained = True
        graph_name = '{}{}'.format(step_method, title_edit)
        print(graph_name)
        progress_list = []
        for i in tqdm(range(100)):
```

```
            progress_list.append(SA_run(d, (index+2), i, 500, 100, 0.9,␣
 ↪'unconstrained_comparisons', 'Kirkpatrick', constrained, w_val))
            fx_progress_dict[graph_name] = progress_list

egg.plot_progress_multvar(fx_progress_dict, d,␣
 ↪filename='Step_Methods_Comparison_2')
```

1.4.8 Investigating Parks Method

[ ]:
```
d = 6
fx_progress_dict = {}
for step_size in [25, 50, 100, 200]:
    graph_name = 'init_step: {}'.format(step_size)
    print(graph_name)
    progress_list = []
    for i in tqdm(range(100)):
        progress_list.append(SA_run(d, 3, i, 500, step_size, 0.95,␣
 ↪'unconstrained_comparisons', 'White', True))
        fx_progress_dict[graph_name] = progress_list

egg.plot_progress_multvar(fx_progress_dict, d,␣
 ↪filename='Parks_Step_size_constrained_white')
```

1.4.9 MC Length with Alpha

[ ]:
```
d = 6
fx_progress_dict = {}
for chain_length in [100, 1500]:
    for alpha in [0.8, 0.99]:
        graph_name = 'MC: {}, a: {}'.format(chain_length, alpha)
        print(graph_name)
        progress_list = []
        for i in tqdm(range(100)):
            progress_list.append(SA_run(d, 1, i, chain_length, 50, alpha,␣
 ↪'unconstrained_comparisons', 'Kirkpatrick', False, 1e5))
            fx_progress_dict[graph_name] = progress_list

egg.plot_progress_multvar(fx_progress_dict, d, filename='MC_alpha_cross_2')
```

1 D: Tabu Search Code

1.1 Importing Libraries

[ ]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.ticker import LinearLocator
from matplotlib import cm
from tqdm import tqdm
import matplotlib.gridspec as gridspec
from sys import path_importer_cache
import os
import eggholder as egg
import init_archive as inar
import random
import pickle
```

1.2 Tabu Search Functions

[ ]:
```
def in_array(array, list_of_arrays):
    '''Check if an array (x) is in a list of arrays.'''
    return any(np.array_equal(array, a) for a in list_of_arrays)

def poss_steps(d, step):
    '''returns a vector of possible steps to make.'''
    poss_steps = []
    for i in range(d):
        new = np.zeros(d)
        new[i] = step
        poss_steps.append(new.copy())
        new[i] = -step
        poss_steps.append(new.copy())
    return poss_steps

def search(base, step, STM, x_best, fx_best, constrained, f_evals_count, w_val):
    '''evaluates a step in the given step direction.'''
    try:
        x_new = base + step
        if not in_array(x_new, STM):
            if constrained:
                fx_new = egg.eggholder(x_new)
            else:
                fx_new = egg.eggholder_unconstrained(x_new, w_val)
            f_evals_count += 1
            if fx_best == None:
                x_best = x_new.copy()
```

```
                fx_best = fx_new.copy()
            if fx_new < fx_best:
                x_best = x_new.copy()
                fx_best = fx_new.copy()
    except AssertionError:
        print("not in bounds.")
        pass

    return x_best, fx_best, f_evals_count

def pattern(x_best, fx_best, base, constrained, w_val, f_evals_count):
    '''perform a pattern move in the specified direction.'''
    try:
        pattern_move = 2*x_best[1] - base
        if constrained:
            f_pattern = egg.eggholder(pattern_move)
        else:
            f_pattern = egg.eggholder_unconstrained(pattern_move, w_val)
        f_evals_count += 1
        if f_pattern < fx_best:
            fx_best  = f_pattern
            x_best = pattern_move
    except AssertionError:
        print("not in bounds.")
        pass

    return x_best, fx_best, f_evals_count

def local_search_basic(base, f_base, d, STM, steps, f_evals_count, constrained,␣
↪w_val):
    '''Perform a local search step for the tabu search function.
    inputs
        base:           d-length vector     current point x
        f_base:         float               function evaluated at x
        d:              int                 dimension of x
        STM:            {x:fx}              dictionary of recently explored␣
↪locations
        steps:          list                a list of +/- delta for each␣
↪dimension of x.
        f_evals_count:  int                 count of total function evaluations
        constrained:    bool                controls if we use penalty function␣
↪or not
        w_val:          int                 penalty value

    returns best step x0, fx0 from the local search
    and the updated count of total cost function evaluations.
    '''
```

```
    x_best = None
    fx_best = None

    while len(steps) > 0:
        step = steps.pop()
        x_best, fx_best, f_evals_count = search(base, step, STM, x_best, fx_best,
                                                constrained,␣
↪f_evals_count, w_val)
    if fx_best != None:
        if fx_best < f_base:
            x_best, fx_best, f_evals_count = pattern(x_best, fx_best, base,␣
↪constrained, w_val, f_evals_count)

    if fx_best == None:
        x_best = base
        fx_best = f_base
    return x_best, fx_best, f_evals_count

def random_subset(base, f_base, d, P, STM, steps, f_evals_count, constrained,␣
↪w_val):
    '''Perform a local search  using the random subset method for the tabu␣
↪search function.
    inputs
        base:           d-length vector     current point x
        f_base:         float               function evaluated at x
        d:              int                 dimension of x
        P:              int                 the max number of possible non-tabu␣
↪moves to evaluate
        STM:            {x:fx}              dictionary of recently explored␣
↪locations
        steps:          list                a list of +/- delta for each␣
↪dimension of x.
        f_evals_count   int                 count of total function evaluations
        constrained:    bool                controls if we use penalty function␣
↪or not
        w_val:          int                 penalty value

    returns best step x0, fx0 from the local search
    and the updated count of total cost function evaluations.
    '''
    x_best = None
    fx_best = None

    for i in range(P):
        step = steps.pop()
```

```
        x_best, fx_best, f_evals_count = search(base, step, STM, x_best, fx_best,
                                                                constrained,␣
↪f_evals_count, w_val)
    if fx_best != None:
        if fx_best < f_base:
            x_best, fx_best, f_evals_count = pattern(x_best, fx_best, base,␣
↪constrained, w_val, f_evals_count)
    if fx_best == None:
        x_best = base
        fx_best = f_base
    return x_best, fx_best, f_evals_count


def successive_rs(base, f_base, d, P, STM, steps, f_evals_count, constrained,␣
↪w_val):
    '''Perform a local search  using the successive random subset method for the␣
↪tabu search function.
    inputs
        base:           d-length vector     current point x
        f_base:         float               function evaluated at x
        d:              int                 dimension of x
        P:              int                 the max number of possible non-tabu␣
↪moves to evaluate
        STM:            {x:fx}              dictionary of recently explored␣
↪locations
        steps:          list                a list of +/- delta for each␣
↪dimension of x.
        f_evals_count   int                 count of total function evaluations
        constrained:    bool                controls if we use penalty function␣
↪or not
        w_val:          int                 penalty value

    returns best step x0, fx0 from the local search
    and the updated count of total cost function evaluations.
    '''
    x_best = None
    fx_best =  None
    improved = False

    while not improved and len(steps) > 0:
        if len(steps) > P:
            subset = [steps.pop() for _ in range(P)]
        else:
            subset = steps.copy()
            steps = []
```

```
        for step in subset:
            x_best, fx_best, f_evals_count = search(base, step, STM, x_best,␣
↪fx_best,
                                                   constrained,␣
↪f_evals_count, w_val)
            if fx_best != None:
                if fx_best < f_base:
                    improved = True

    if improved:
        x_best, fx_best, f_evals_count = pattern(x_best, fx_best, base,␣
↪constrained, w_val, f_evals_count)
    if fx_best == None:
        x_best = base
        fx_best = f_base
    return x_best, fx_best, f_evals_count

def sensitivity(base, d, delta, f_evals_count, w_val):
    '''Find the d/2 most sensitive directions.'''
    s = np.zeros(d)
    for i in range(d):
        f_plus = egg.eggholder_unconstrained(base+delta, w_val)
        f_minus = egg.eggholder_unconstrained(base-delta, w_val)
        s[i] = np.abs((f_plus - f_minus)/(2*delta))
        f_evals_count += 2
    j = d//2
    most_sensitive_directions = sorted(range(len(s)), key=lambda i: s[i])[-j:]
    return most_sensitive_directions, f_evals_count
```

1.2.1 MTM and STM Functions

[ ]:
```
def update_MTM(x0, fx0, M, MTM):
    '''Updates the Medium-Term Memory (MTM) if Necessary.
    inputs
        MTM:        {fx:x}              current set of points in MTM
        x0:         d-length vector     new trial point
        fx0:        float               evaluation at new point
        M:          int                 max size of MTM

    returns updated MTM.
    '''
    if len(MTM) < M:
        MTM[fx0] = x0
    else:
        MTM_list = [*MTM]
        worst = min(MTM_list)
        if fx0 < worst:
```

```
            del MTM[worst]
            MTM[fx0] = x0

    return MTM

def update_counter(x0, fx0, best_sln, counter):
    '''update the counter and best solution.'''
    if fx0 < best_sln[0]:
        best_sln = [fx0,x0]
        counter = 0
    else:
        counter += 1

    return counter, best_sln

def update_STM(x0, N, STM):
    '''Update the Short-Term Memory (STM).'''
    STM.append(x0)
    while len(STM) > N:
        STM.pop()
    return STM
```

1.3 Long-Term Memory (LTM) Functions

[ ]:
```
def index_to_increase(sector, d, split):
    '''Used in the initialisation of the LTM. Essentially for counting up in␣
 ↪base-n, where
    n=split, as this is how all the sectors are divided.

    inputs
        sector:     d-length list       each element is in range(0,split). This␣
 ↪is the most
                                        recently appended sector to the LTM␣
 ↪initialisation.
        d:          int                 dimension of x
        split:      int                 number of sections that each variable is␣
 ↪split into

    returns the index that should be increased for the next sector to append to␣
 ↪the LTM.
    If all sectors have been appended (all values in sector = (split-1)), then␣
 ↪return -1.
    '''
    for i in range(d):
        if sector[i] != split-1:
            return i
```

```
    return -1

def init_LTM(d, split, sector, LTM):
    '''Creates an LTM with a key for each sector according to the specified␣
 ↪dimension
    and split.

    inputs
        d:          int             dimension of x
        split:      int             number of sections that each variable is␣
 ↪split into.
        sector:     d-length list   The most recently appended sector
        LTM         {sector: count} All counts initialised to 0. Once filled␣
 ↪with all
                                    sectors this is returned.

    returns the initialised LTM with a key for every sector needed.
    '''
    i = index_to_increase(sector,d,split)
    if i == -1:
        return(LTM)
    elif i > 0:
        for j in range(i):
            sector[j] = 0
    sector[i] += 1
    LTM[tuple(sector)] = 0
    return init_LTM(d, split, sector, LTM)

def update_LTM(x0, d, LTM, split):
    '''Find the sector that the new point is in and increment the count
    for that sector. returns the updated LTM.'''
    location = [0]*d
    sector_width = 1024/split
    for i in range(d):
        for j in range(split):
            if x0[i] >= -512 + j*sector_width and x0[i] < (-512 +␣
 ↪(j+1)*sector_width):
                location[i] = j
    location = tuple(location)
    LTM[location] += 1
    return LTM

def diversify_x(LTM, d, split):
    '''Carries out a diversification step and returns a randomly found
    x within the least populated sector.'''
    sector_width = 1024/split
    values = [*LTM.values()]
```

```
    min_index = values.index(min(values))
    keys = [*LTM.keys()]
    sector = keys[min_index]
    new_x = np.zeros(d)
    for i, j in enumerate(sector):
        range_min = -512 + j*sector_width
        range_max = -512 + (j+1)*sector_width
        new_x[i] = np.random.uniform(range_min, range_max)

    return new_x
```

1.4 Main TS Code

[ ]:
```
def TS_run(d, seed = 1, search_method = 0, init_step_size = 500, P = 6, split =␣
↪2, N=7, M=4, run_name = 'initial',
           intensify = 10, diversify = 15, reduce = 25, prioritise= 6,␣
↪constrained=True, w_val = 1e6, alpha = 0.9):
    '''Conduct a run of Tabu Search under specified conditions.
    inputs
        d:              int     dimension of datapoints.
        seed:           float   random seed for different runs
        search_method:  int     choose which local search method to use.
        init_step_size: int     starting step size to take
        P:              int     the number of steps to evaluate in each cycle␣
↪for rs.
        split:          int     the number of splits per variable for creating␣
↪sectors.
        N:              int     size of STM
        M:              int     size of MTM
        run_name:       str     identifier for the run if required
        intensify:      int     value of counter at which to intensify search
        diversify:      int     value of counter at which to diversify search
        reduce:         int     value of counter at which to reduce search
        prioritise:     int     number of iterations in variable prioritisation␣
↪before reevaluating
        constrained:    bool    controls whether we are using a constrained␣
↪formulation or not
        w_val           int     weights on the penalty function for the␣
↪unconstrained formulation
        alpha           float   the ratio to reduce step size by on reduce steps.

    returns the progress in best solution over iterations for the specified␣
↪setting.
    '''
    np.random.seed(seed)
    x0 = inar.init_x(1, d)
```

```
    fx0 = egg.eggholder(x0)
    counter = 0
    step_size = init_step_size
    f_evals_count = 0
    STM = []
    MTM = {fx0: x0.copy()}
    first_sector = [0]*d
    LTM = init_LTM(d, split, first_sector, {tuple(first_sector):0})
    prioritisation_count = 0
    directions = [0]

    path = {fx0:x0.copy()}
    archive = {fx0:x0.copy()}
    fx_progress = []
    f_evals_count_list = []

    best_sln = [fx0, x0]
    while step_size > 1e-4 and f_evals_count < 15000:
        steps = poss_steps(d, step_size)
        if search_method == 0:
            random.shuffle(steps)
            x0, fx0, f_evals_count = local_search_basic(x0.copy(), fx0.copy(),␣
↪d, STM, steps,
                                                        f_evals_count,␣
↪constrained, w_val)
        elif search_method == 1:
            random.shuffle(steps)
            x0, fx0, f_evals_count = random_subset(x0.copy(), fx0.copy(), d, P,␣
↪STM, steps,
                                                   f_evals_count, constrained,␣
↪w_val)
        elif search_method == 2:
            random.shuffle(steps)
            x0, fx0, f_evals_count = successive_rs(x0.copy(), fx0.copy(), d, P,␣
↪STM, steps,
                                                   f_evals_count, constrained,␣
↪w_val)
        else:
            if prioritisation_count >= prioritise or f_evals_count == 0:
                directions, f_evals_count = sensitivity(x0.copy(), d, step_size,
                                                        f_evals_count, w_val)
                prioritisation_count = 0
            prioritisation_count += 1
            chosen_steps = []
            for direction in directions:
                chosen_steps.append(steps[2*direction])
```

```
                chosen_steps.append(steps[2*direction + 1])
            random.shuffle(chosen_steps)
            x0, fx0, f_evals_count = local_search_basic(x0.copy(), fx0.copy(),␣
↪d, STM, chosen_steps,
                                                        f_evals_count,␣
↪constrained, w_val)

        archive, updated = inar.update_archive(x0, fx0, archive)
        path[fx0] = x0.copy()
        counter, best_sln = update_counter(x0, fx0, best_sln, counter)
        fx_progress.append(best_sln[0])
        f_evals_count_list.append(f_evals_count)
        STM = update_STM(x0.copy(), N, STM)
        MTM = update_MTM(x0.copy(), fx0, M, MTM)
        LTM = update_LTM(x0.copy(), d, LTM, split)

        if counter == intensify:
            vals = np.array([*MTM.values()])
            x0 = np.mean(vals, axis=0)
        elif counter == diversify:
            xnew = diversify_x(LTM, d, split)
            x0 = xnew

        elif counter == reduce:
            step_size *= alpha
            counter = 0
            fx0 = best_sln[0]
            x0 = best_sln[1]
            continue

        if constrained:
            fx0 = egg.eggholder(x0)
        else:
            fx0 = egg.eggholder_unconstrained(x0, w_val)
        f_evals_count += 1

    fx_star = min([*archive])
    print("\nBest solution: \n", fx_star, archive[fx_star])
    print(len(path))
    print(len(archive))

    z = []
    if d==2:
        # Draw contour plot and evolution of best solution found across␣
↪iterations
        z = egg.plot_eggholder2D(archive, path, run_name)
    egg.plot_progress(fx_progress, z, d, run_name, f_evals_count_list)
```

```
    return fx_progress, f_evals_count_list
```

[ ]:
```
TS_run(6, 0, 0, 500, prioritise=20,constrained=True)
```

1.5 Large Runs

1.5.1 Changing Alpha

[ ]:
```
fx_progress_dict = {}
f_evals_count_dict = {}

d=6
for alpha in [0.85, 0.9, 0.95, 0.99]:
    graph_name = 'a: {}'.format(alpha)
    print(graph_name)
    progress_list = []
    f_evals_count_list = []
    for i in tqdm(range(100)):
        f_progress, f_evals = TS_run(d, i, 0, 100, constrained=True, alpha=alpha)
        progress_list.append(f_progress)
        f_evals_count_list.append(f_evals)
    fx_progress_dict[graph_name] = progress_list
    f_evals_count_dict[graph_name] = f_evals_count_list


egg.plot_progress_multvar(fx_progress_dict, d, 'TS_alpha_step100', True,␣
 ↪f_evals_count_dict)
```

1.5.2 Changing Step Size

[ ]:
```
fx_progress_dict = {}
f_evals_count_dict = {}

d=6
for init_step in [50, 100, 200, 500]:
    graph_name = 'step: {}'.format(init_step)
    print(graph_name)
    progress_list = []
    f_evals_count_list = []
    for i in tqdm(range(100)):
        f_progress, f_evals = TS_run(d, i, 0, init_step, constrained=True)
        progress_list.append(f_progress)
        f_evals_count_list.append(f_evals)
    fx_progress_dict[graph_name] = progress_list
    f_evals_count_dict[graph_name] = f_evals_count_list
```

```
egg.plot_progress_multvar(fx_progress_dict, d, 'TS_step_size', True,␣
 ↪f_evals_count_dict)
```

1.5.3 Random Subset Method

[ ]:
```
fx_progress_dict = {}
f_evals_count_dict = {}

d=6
for p in [2, 4, 6, 9]:
    graph_name = 'P: {}'.format(p)
    print(graph_name)
    progress_list = []
    f_evals_count_list = []
    for i in tqdm(range(100)):
        f_progress, f_evals = TS_run(d, i, 1, P = p)
        progress_list.append(f_progress)
        f_evals_count_list.append(f_evals)
    fx_progress_dict[graph_name] = progress_list
    f_evals_count_dict[graph_name] = f_evals_count_list

egg.plot_progress_multvar(fx_progress_dict, d, 'RS', True, f_evals_count_dict)
```

1.5.4 Successive RS

[ ]:
```
fx_progress_dict = {}
f_evals_count_dict = {}

d=6
for p in [4, 6, 8, 10]:
    graph_name = 'P: {}'.format(p)
    print(graph_name)
    progress_list = []
    f_evals_count_list = []
    for i in tqdm(range(100)):
        f_progress, f_evals = TS_run(d, i, 2, P = p)
        progress_list.append(f_progress)
        f_evals_count_list.append(f_evals)
    fx_progress_dict[graph_name] = progress_list
    f_evals_count_dict[graph_name] = f_evals_count_list

egg.plot_progress_multvar(fx_progress_dict, d, 'successive_RS', True,␣
 ↪f_evals_count_dict)
```

1.5.5 Variable Prioritisation

[ ]:
```
fx_progress_dict = {}
f_evals_count_dict = {}

d=6
for count in [6, 10, 20, 40]:
    graph_name = 'count: {}'.format(count)
    print(graph_name)
    progress_list = []
    f_evals_count_list = []
    for i in tqdm(range(100)):
        f_progress, f_evals = TS_run(d, i, 3, prioritise=count)
        progress_list.append(f_progress)
        f_evals_count_list.append(f_evals)
    fx_progress_dict[graph_name] = progress_list
    f_evals_count_dict[graph_name] = f_evals_count_list

egg.plot_progress_multvar(fx_progress_dict, d, 'var_prioritisation', True,␣
 ↪f_evals_count_dict)
```

1.5.6 Constrained Analysis

[ ]:
```
fx_progress_dict = {}
f_evals_count_dict = {}

d=6
constrained = False
for w in [0, 1e3, 1e5, 1e6, 1e9]:
    graph_name = 'w: {}'.format(w)
    if w == 0:
        constrained = True
        graph_name = 'Constrained'
    print(graph_name)
    progress_list = []
    f_evals_count_list = []
    for i in tqdm(range(100)):
        f_progress, f_evals = TS_run(d, i, 1, constrained=constrained, w_val = w)
        progress_list.append(f_progress)
        f_evals_count_list.append(f_evals)
    fx_progress_dict[graph_name] = progress_list
    f_evals_count_dict[graph_name] = f_evals_count_list

egg.plot_progress_multvar(fx_progress_dict, d, 'unconstrained', True,␣
 ↪f_evals_count_dict)
```

1.5.7 M and N

[ ]:
```
fx_progress_dict = {}
f_evals_count_dict = {}

N_vals = [7,20,40, 80]
M_vals = [4,15,30, 60]

d=6
for index,M in enumerate(M_vals):
    N = N_vals[index]
    graph_name = 'N: {}, M: {}'.format(N, M)
    print(graph_name)
    progress_list = []
    f_evals_count_list = []
    for i in tqdm(range(100)):
        f_progress, f_evals = TS_run(d, i, 1, N=N, M=M)
        progress_list.append(f_progress)
        f_evals_count_list.append(f_evals)
    fx_progress_dict[graph_name] = progress_list
    f_evals_count_dict[graph_name] = f_evals_count_list

egg.plot_progress_multvar(fx_progress_dict, d, 'NM2', True, f_evals_count_dict)
```

1.5.8 Parameter Variance

[ ]:
```
fx_progress_dict = {}
f_evals_count_dict = {}

intensify_vals = [8, 10, 20, 40]
diversify_vals = [12, 15, 30, 60]
reduce_vals = [20, 25, 50, 100]

d=6
for index,I in enumerate(intensify_vals):
    D = diversify_vals[index]
    R = reduce_vals[index]
    graph_name = 'I: {}, D: {}, R: {}'.format(I, D, R)
    print(graph_name)
    progress_list = []
    f_evals_count_list = []
    for i in tqdm(range(100)):
        f_progress, f_evals = TS_run(d, i, 1, intensify=I, diversify=D, reduce=R)
        progress_list.append(f_progress)
        f_evals_count_list.append(f_evals)
    fx_progress_dict[graph_name] = progress_list
    f_evals_count_dict[graph_name] = f_evals_count_list
```

```
egg.plot_progress_multvar(fx_progress_dict, d, 'IDR', True, f_evals_count_dict)
```

1.5.9 Split

[ ]:
```
fx_progress_dict = {}
f_evals_count_dict = {}

d=6
for split in [2,3]:
    graph_name = 'Split: {}'.format(split)
    print(graph_name)
    progress_list = []
    f_evals_count_list = []
    for i in tqdm(range(100)):
        f_progress, f_evals = TS_run(d, i, 1,split=split, intensify=20,␣
 ↪diversify=30, reduce=50)
        progress_list.append(f_progress)
        f_evals_count_list.append(f_evals)
    fx_progress_dict[graph_name] = progress_list
    f_evals_count_dict[graph_name] = f_evals_count_list

egg.plot_progress_multvar(fx_progress_dict, d, 'Split', True, f_evals_count_dict)
```