4M17 Practical Optimisation, 2018
Assignment 2: Optimisation Algorithm Performance Comparison Study
Candidate number: 5552B

Introduction

This report compares the performance of the metaheuristic optimisation algorithms Tabu Search and Simulated Annealing (SA) on the 5-dimensional Schwefel function (5D SF). The 2D SF is used to illustrate the behaviour of the algorithms. The effects of parameters on algorithm performance are also investigated. The code used for Tabu Search was custom-written for this investigation and the package is titled `tabusearch`.

A slightly modified version of the SF provided in the 4M17 coursework handout is used in this report. Here the d-dimensional SF is defined as follows:

$$f(x) = 418.9829d - \sum_{i=1}^{d} x_i \sin(|\sqrt{x_i}|), \tag{1}$$

where the feasible region is the hypercube defined by $|x_i| \leq 500$ for $i \in (1, ..., d)$. A surface plot of the 2D SF using (1) is shown in Fig. 1. The constant offset term $418.9829d$ conveniently ensures that the global minimum takes value $f(x_*) = 0$ at $x_* = (420.9867, ..., 420.9867)$ in any dimension.

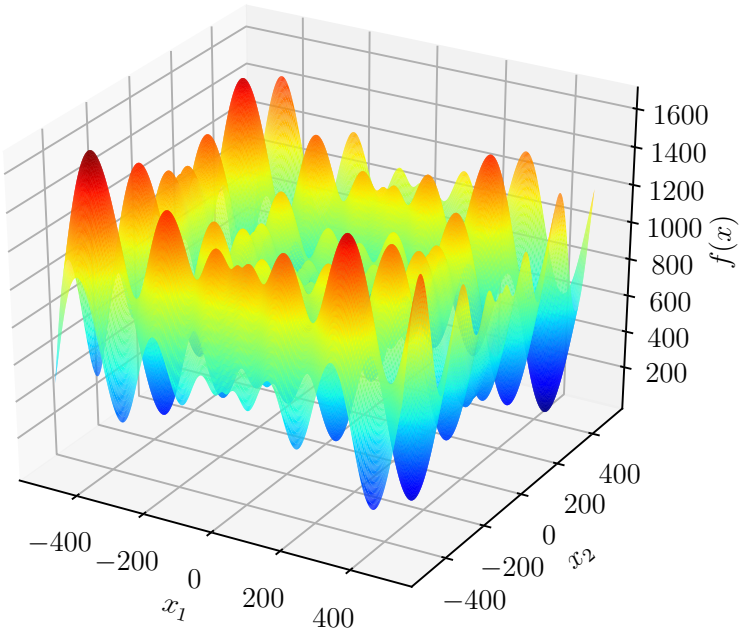


Figure 1: Surface plot for the 2D SF.

Tabu Search

Implementation details of Java package tabusearch

The structure of `tabusearch` is mostly as described in the 4M17 Tabu Search handout. This section details some of the additions in the Java package `tabusearch`.

Dividing the input space into a grid

LTM in `tabusearch` is implemented by converting the input points x visited during the search into 'grid coordinates'. This is done by dividing the feasible region into a grid of regular segments and finding the segment that the input x lies within. A `HashSet` is used to store the grid coordinates that have already been visited during the search. By using a set, a duplicate grid position will not be stored in the LTM when the algorithm attempts to add it to the set.

Generation of a new diversified starting point

When diversification is triggered, the new starting point is generated uniformly over the grid segments not yet in the LTM by rejection sampling. If the entire grid is stored in the LTM (which is highly unlikely unless the grid segments are too large or in low dimensions), then the sample is drawn uniformly over the entire feasible region. A more elaborate method might sample from the grid segment with the smallest count of visits so far, but since the LTM will very rarely fill up, the two methods will be identical in practice.

Updating the LTM efficiently

It becomes increasingly less likely that the search will move from one grid segment to another as the search progresses because the step size decreases in a geometric series. This is exploited in `tabusearch` in a principled manner to provide some improvement in efficiency. This is achieved by varying the number of Tabu iterations per update to the LTM over the course of the search, as follows:

- The ratio between the LTM grid segment size and the Tabu step size is computed each time the step size is reduced.
- This ratio is rounded to an integer to give the number of Tabu steps between grid segments.
- The integer is divided by 2 to determine the number of iterations per update to the LTM. The division by 2 in this step accounts for the fact that the step size can effectively double when pattern moves are being performed.

The effect of this is that the LTM is updated each time a new local search is triggered, but the rate of subsequent updates decreases proportionally as the step size is reduced.

Step size limit

The 'double' data type used to store information in `tabusearch` has a numerical accuracy of 15-16 decimal digits. It was found that if the step size shrinks below this accuracy threshold, all of the incremented positions in a Tabu step can incorrectly lie within the short-term memory (STM) in the double representation. This forces the algorithm to stop since there are no non-tabu moves available. To guard against this, `tabusearch` takes a step size limit parameter: if the new step size after step-size reduction (SSR) would be below this limit, the step size is not reduced.

Verbose printing

One can choose to print the occurrence of search events and the evolution of the counter to the console during a run of the algorithm. This is useful for debugging and illuminates how the algorithm behaves. An example is shown in Listing 1. One can observe the effect of the STM pushing the search out of local minima, causing the counter to increase. As expected, the counter will reset if the threshold for SSR is reached, or if a new optimal objective is found during a local search.

Listing 1: Verbose printing with `tabusearch` demonstrating the evolution of the search. The counter limits for intensifying, diversifying and SSR were 10, 15 and 25, respectively.

```
Starting a Tabu Search. Printing counter evolution and other search events.

Initial search: 0 0 1 2 3 4 5 6 7 8 9
Intensiying search: 10 11 12 13 0 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
Diversifying after 0 rejected samples: 15 16 17 0 0 1 2 3 0 1 2 3 4 5 6 7 8 9 10 11 0 1 2
    3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
Reducing step size: 0 1 2 3 4 5 6 7 8 9
Intensiying search: 10 11 12 13 14
Diversifying after 0 rejected samples: 15 16 17 18 19 20 21 22 23 24
Reducing step size: 0 1 2 3 4 5 6 7 8 9
Intensiying search: 10 11 12 13 14
Diversifying after 2 rejected samples: 15 16 17 18 0 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
    15 16 17 18 19 20 21 22 23 24
Reducing step size: 0 0 1 2 3 4 5 6 7 8 9
```

Illustration of Tabu Search for 2D SF

Fig. 2 illustrates a particular run of `tabusearch` on the 2D SF. In this case the search successfully locates the global minimum at $x_* = (420.9867, 420.9867)$. One can see the trace of the starting search at the top of the plot where the step size is greatest. The intensification stage taking the search to the average of the M best solutions (the medium-term memory or MTM) is also evident from the large cluster of samples around the two good solutions in the top corners. What look like single random points are actually diversified searches from later in the search where the step size has become very small.

Tabu Search can loosely be characterised by two phases, 'exploration' followed by 'fine-tuning'. In the exploration phase, the step size is still fairly large, and the search improves the objective function through diversification. The searches after SSR (which return to the best solution) are unlikely to contribute much information in this phase, since best solution in the early stages is unlikely to be the best found in the entire search.

In contrast, in the fine-tuning phase the step size becomes much smaller than the length scale over which the function changes non-negligibly, and the search cannot hope to improve the objective through diversification. The search can only improve the solution in the best local minimum discovered in the search by making marginal perturbations after intensification and SSR.

Fig. 3 shows the evolution of the function value against the number of function evaluations[1] for the search illustrated in Fig. 2. The positive jumps correspond to a randomly sampled diversified solution.

[1]The function value versus Tabu iteration was mapped to function value versus number of evaluations by zero-holding - a method used throughout this report.

After the global optimum is located at around 750 function evaluations, intensification and SSR will always bring the search back to that location, explaining the subsequent step-like spikes. The gradient after the diversification spikes become shallower as the search progresses due to the step size decreasing.

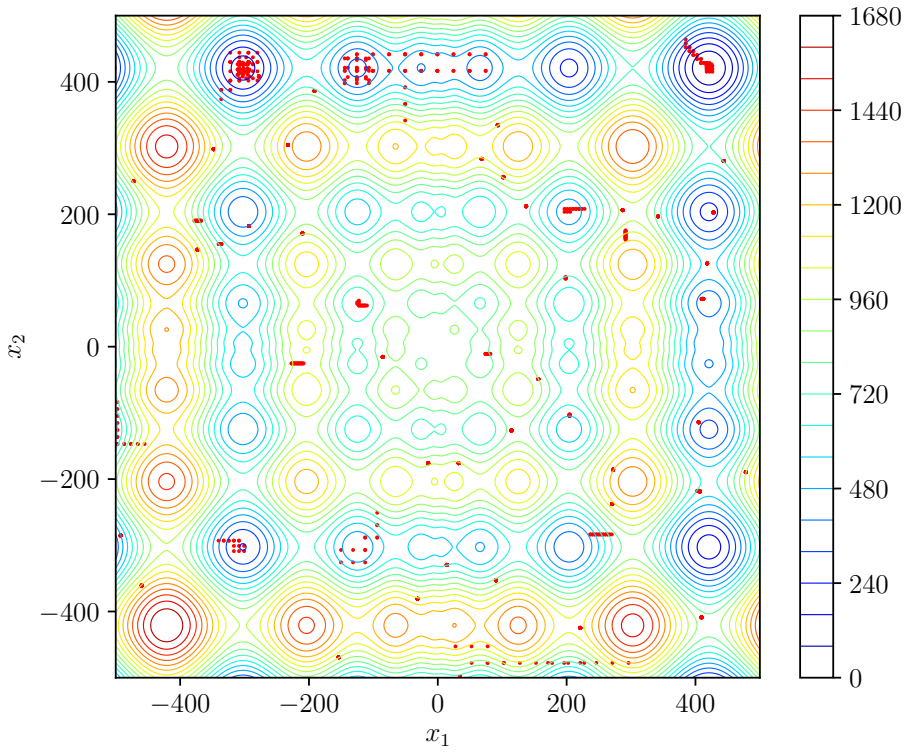


Figure 2: Schwefel function contour plot and a Tabu Search path search that successfully locates the optimum.

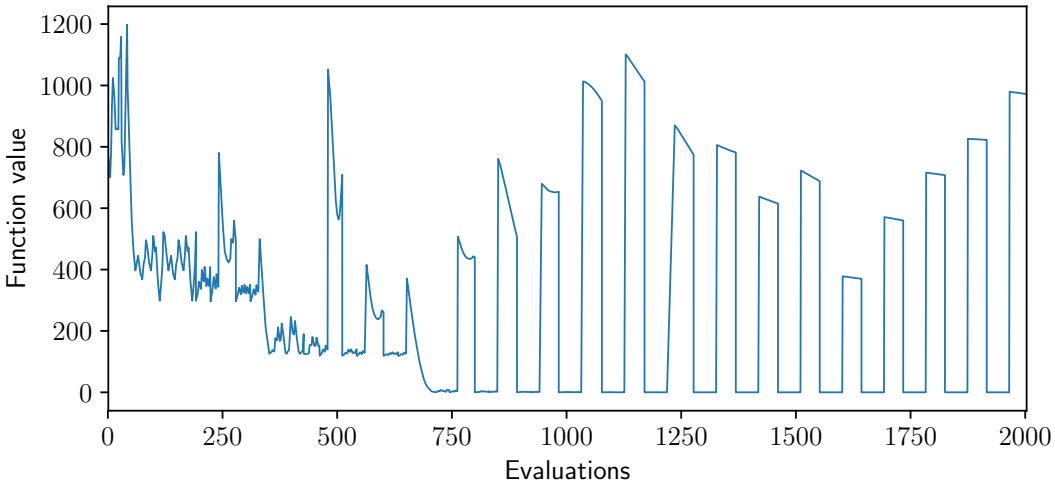


Figure 3: Evolution of the objective values for the single Tabu run illustrated in Fig. 2.

The parameters used to generate this run are given in Table 1. The parameter values were tuned by visualising the path as in Fig. 2. This meant that the ability of the algorithm to find the global minimum

could be clearly observed. This privilege is not readily available above 3 dimensions, so in general finding the 'best' algorithm parameters for a high-dimensional problem is a challenge. In the following section the performance on the 5D SF is investigated. The parameters were held fixed as in Table 1 unless otherwise stated.

Parameter	Value
Seed	40
Intensify threshold	10
Diversify threshold	15
SSR threshold	25
Starting step size	25
Step size limit	10^{-13}
Step reduce factor	0.75
Evaluation limit	10,000
STM size	8
MTM size	4
LTM grid segment size	1000

Table 1: Parameters used for the run in Fig. 2.

Tabu Search for the 5D SF

Fig. 4 was generated by using `tabusearch` with the 5D SF, and shows the evolutions of the best solution found versus the number of function evaluations for 30 different random seeds. Each run was permitted 10,000 function evaluations, but only the first 2,000 are plotted as minimal change occurs after this point. It is clear from the figure that none of the 30 runs are able to locate the global optimum solution where objective takes value zero. The 5D SF is much more difficult to optimise than the 2D SF because the feasible region size grows exponentially with the number of dimensions.

Fig. 5 plots the average across all 30 random seeds for the experiment illustrated in Fig. 4. This figure provides greater insight into the performance of the algorithm by averaging out random variations in the performance of individual runs. In this section, the dependence of algorithm performance on certain parameters is investigated. The performance measures used are the average best objective value, $\bar{f}(x_*)$, and the standard deviation of the best objective value, σ_f, measured after 10,000 evaluations. These measure the algorithm's average ability to reduce the objective function and the reliability of that reduction, respectively. The relative importance of these two performance measures will depend on the application and can result in a trade-off. 50 runs with different random seeds were used for each experiment.

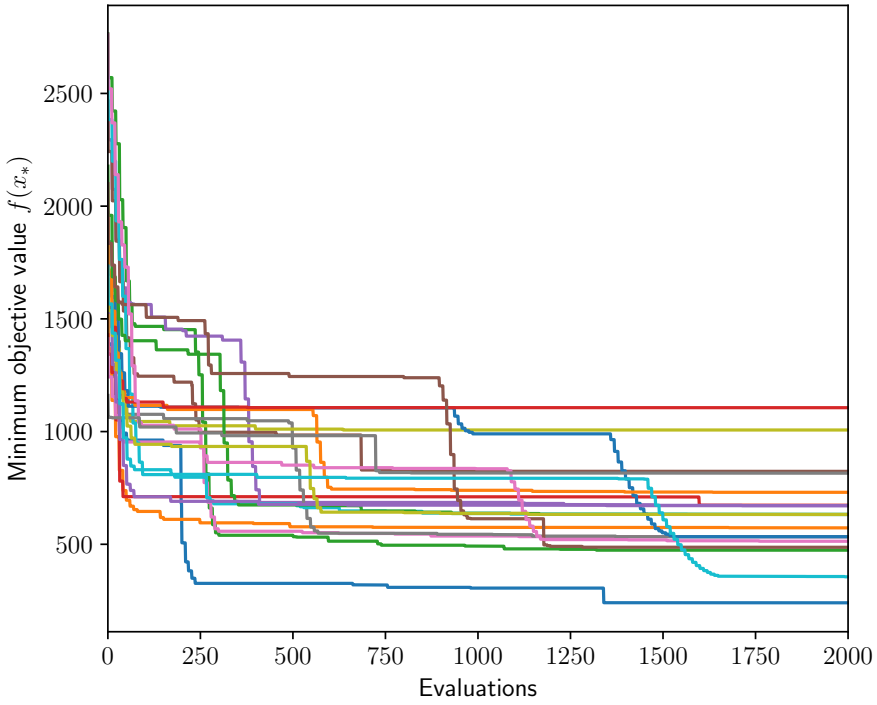


Figure 4: Evolutions of the of the best solution found versus the number of function evaluations for 30 different random seeds.

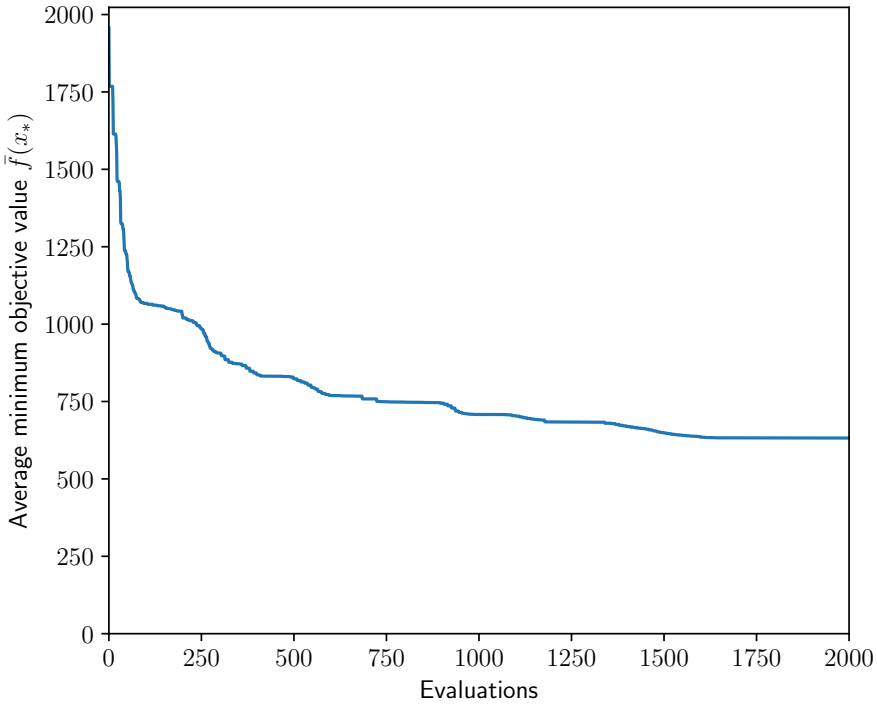
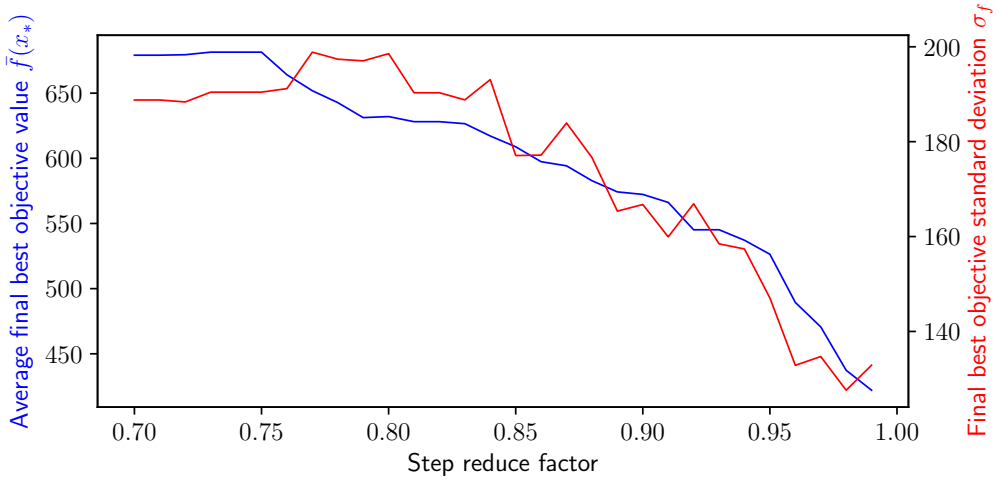


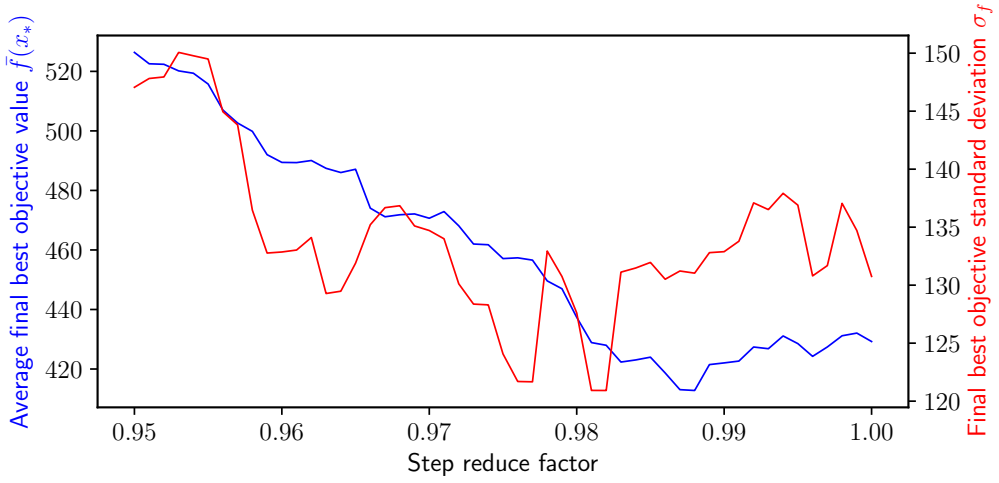
Figure 5: Results from Fig. 4 averaged across all random seeds.

Step reduce factor

Fig. 6a shows the effect of the step reduce factor (SRF) on the performance measures. It is clear that a greater step reduce factor generally results in both a smaller average best solution and greater reliability in that solution. Fig. 6b shows a more finely sampled plot for the larger SRFs, which shows that the optimal performance for this problem occurs between 0.98 and 0.99. Larger SRFs may perform better because the step size stays large for longer, meaning the search behaviour is more 'shotgun'-like and the exploratory phase lasts longer. This allows more of the feasible region to be explored before the fine-tuning phase hones in on the best solution found. The performance starts to decrease as the SRF approaches a value of 1, because after 10,000 evaluations the step size will still be close to the starting step size, and the fine-tuning phase will not occur. For the rest of the investigation, an SRF of 0.985 is used.



(a) Range 0.70 to 0.99



(b) Range 0.950 to 1.000

Figure 6: Effect of the step reduce factor on the performance measures.

Starting step size

Fig. 7 shows the effect of the starting step size (SSS) on the performance measures. This shows that larger SSSs generally result in better performance. However, the performance begins to plateau after the SSS reaches roughly 175. A smaller SSS can be safer from premature algorithm termination because large step sizes may result in the the search becoming 'stuck' in a corner of the feasible region hypercube, where all possible Tabu increments are either infeasible or tabu (i.e. in the STM). However, this issue was noticed for the 2D SF but not the 5D SF. This is because each extra dimension provides an extra route out of a corner, so a large SSS is permissible in higher dimensions. $\bar{f}(x_*)$ appears to start to increase as the SSS approaches 400. This could be for the same reason that SRFs very close to 1 reduce performance by eliminating the fine-tuning phase when there is a budget on the number of evaluations.

With a large SSS and large SRF (see previous section) there will essentially be a 'shotgun' search phase before the exploration and fine-tuning phases, where the feasible region is sampled very coarsely due to the large step size. In the subsequent sections, an SSS of 175 is used. In practice, the appropriate choice for the starting step size and step reduce factor will depend on the characteristic scale of the objective function (how rapidly it changes) and any budget on the number of evaluations.

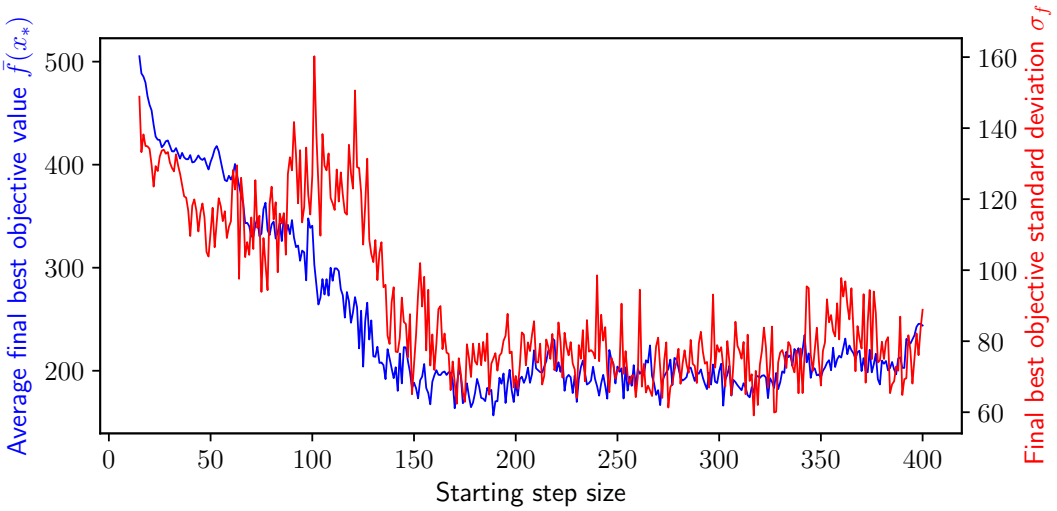


Figure 7: Effect of the starting step factor on the performance measures.

Counter thresholds

The effects on performance of the counter thresholds were investigated by varying the intensify threshold (IT) while fixing the diversify threshold (DT) and SSR thresholds (SSRT) to be greater by 5 and 15, respectively. This produced the results shown in Fig. 8, which surprisingly show that $\bar{f}(x_*)$ is smallest when IT $= 1$. In this case, if the first move after SSR (going back to the best solution) does not improve the best objective, then the search instantly intensifies, and will readily diversify after a few more iterations. This prevents the search from wasting too much of its computational budget in the crucial exploratory by phase exploring the same best local minimum (which is unlikely to be the best overall in the search). However, if reliability is of greater importance, then a value of 6 or 7 for the IT produces slightly smaller standard deviation σ_f, and indeed may be more generally applicable to other objective functions.

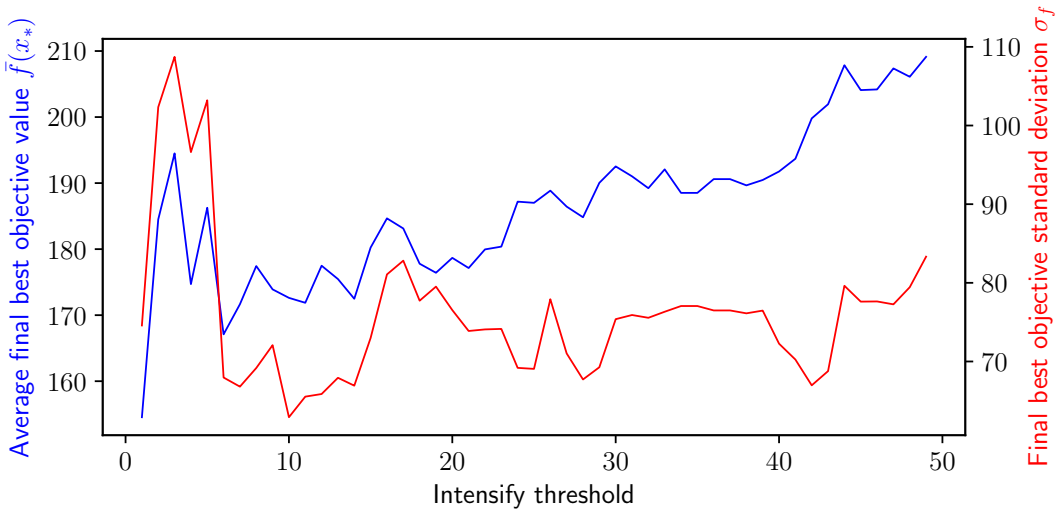


Figure 8: Effect of the counter thresholds on the performance measures.

Simulated Annealing

The second part of this experiment is based on SA's performance on the 5D SF. SA was implemented using MATLAB code found online, which was modified significantly for the purposes of the investigation (see the Appendix for both versions). The unchanged attributes of the algorithm are a) that temperature decrements follow the exponential cooling scheme ($T_{k+1} = \alpha_l T_k$), and b) the temperature is reduced if L trial samples or η_{min} accepted samples occur at a single T_k. Modified attributes of the algorithm are as follows:

- Termination occurs when the total number of function evaluations exceeds 10,000. Previously the criterion was based on the number of consecutive rejections exceeding a threshold or the temperature T reaching a small enough value T_{min}.
- The search restarts at the best solution if no improvement is made to that solution after a restart limit on function evaluations is reached. If the restart limit is set too small then the search can get stuck in local minima. If it is too large then the search may not sufficiently explore the best local minimum it has found. 1,500 was chosen as a compromise of these two factors.
- Trial solution generation: The Parks [1990] method of $x_{i+1} = x_i + Du$ is used, where $D = \text{diag}(d)$ where d is a constant and u is a uniformly sampled vector whose elements lie in $[-1, 1]$. This modification allows the search to move anywhere within a hypercube whose sides have length $2d$. Previously, perturbation was by a random amount along a single axis, which was observed to limit exploration potential somewhat (based on 2D visualisations).
- Initial temperature: The White [1984] method of evaluating the standard deviation σ of a set of random samples from $f(x)$ is used to determine the starting temperature T_0 each time SA is run. 100 samples are used so as not to use too much of the computational budget, and typical values of T_0 for the 5D SF range from 200-400.
- Solutions are rejected if they do not lie in the feasible region and some information is stored and returned depending on which user-chosen experiments are run.

Illustration of Simulated Annealing with the 2D SF

Fig. 9 illustrates a particular run of SA on the 2D SF, which successfully locates the global minimum in the top right corner of the plot, and Fig. 10 plots the evolution of the function value. The phases of an SA search can be loosely characterised as 'melting' and 'freezing'. Melting occurs early in the search when the temperature is high. During melting, moves with a positive δf have a high probability of being accepted and the search resembles a random walk around the feasible region without any constraints. This explains the disperse samples in Fig. 9 and the white noise appearance in Fig. 10 for the early stages of the search. The behaviour of the algorithm changes as it transitions to the 'freezing' phase at the end of the search. In this period, the temperature has been reduced significantly by the Annealing Schedule, and upward moves with a positive δf have a low probability of being accepted. The search steps necessarily become smaller and the search therefore tends to converge towards its closest local minimum (as seen by the cluster in Fig. 9). This final local minimum may not be the best minimum found in the search, although the resetting procedure makes it more likely to be so. The parameters used for this run are given in Table 2.

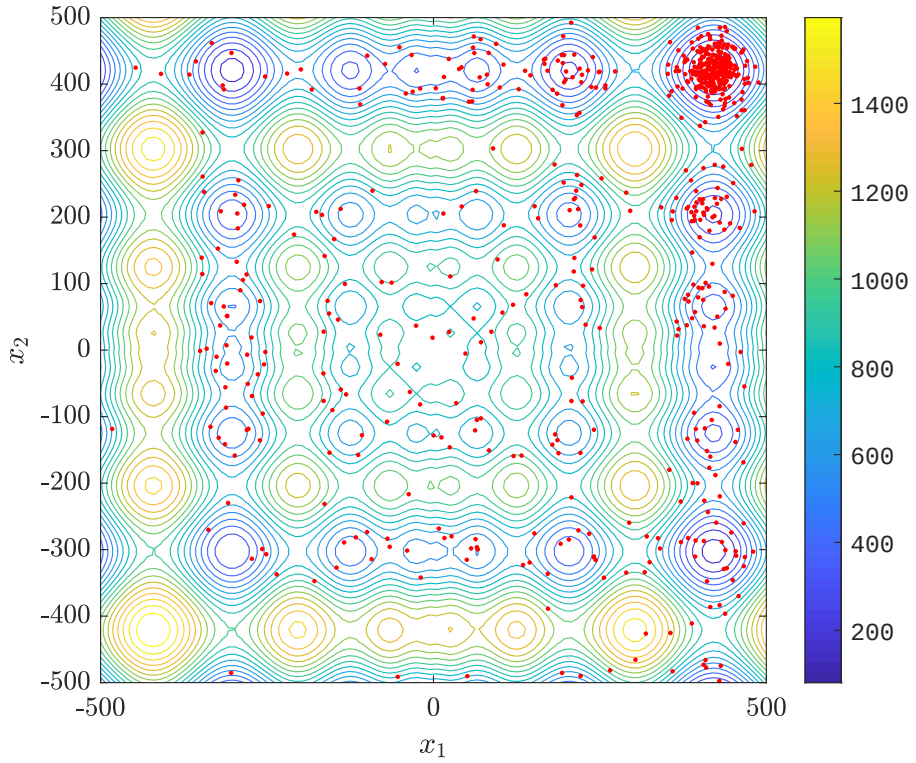


Figure 9: 2D SF contour plot and an SA path search that successfully locates the optimum.

Parameter	Value
d	200
α	0.95
L	300
ν_{min}	20

Table 2: Parameters used for the run in Fig. 9.

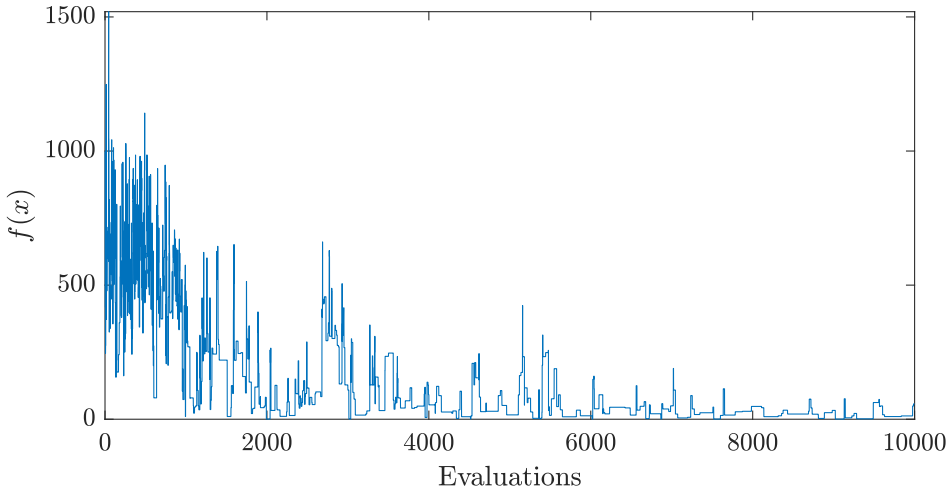


Figure 10: Evolution of the objective values for the single SA run illustrated in Fig. 9.

Simulated Annealing for the 5D SF

Fig. 11 plots the evolutions of the best objective found for 30 SA runs with different random seeds for the 5D SF. Fig. 12 plots the average of Fig. 11 across all random seeds - compared with Fig. 5, SA appears to still make progress on reducing the objective after 10,000 iterations, unlike Tabu Search. This could be because SA uses more function evaluations on average than Tabu Search due to the large numbers of rejected samples, which slows down progress.

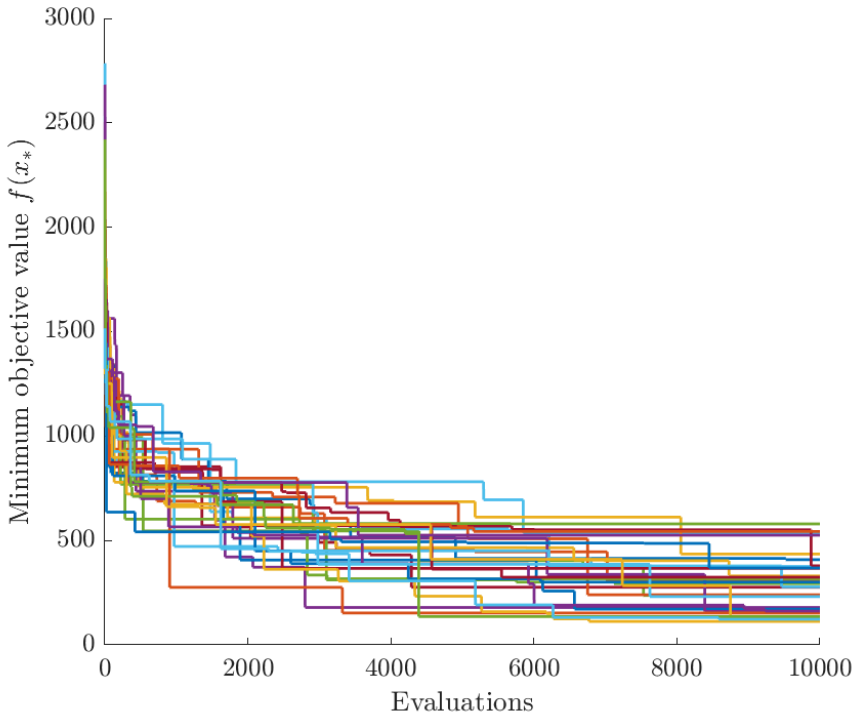


Figure 11: Evolution of the best objective found versus the number of function evaluations for 30 different random seeds for the 5D SF.

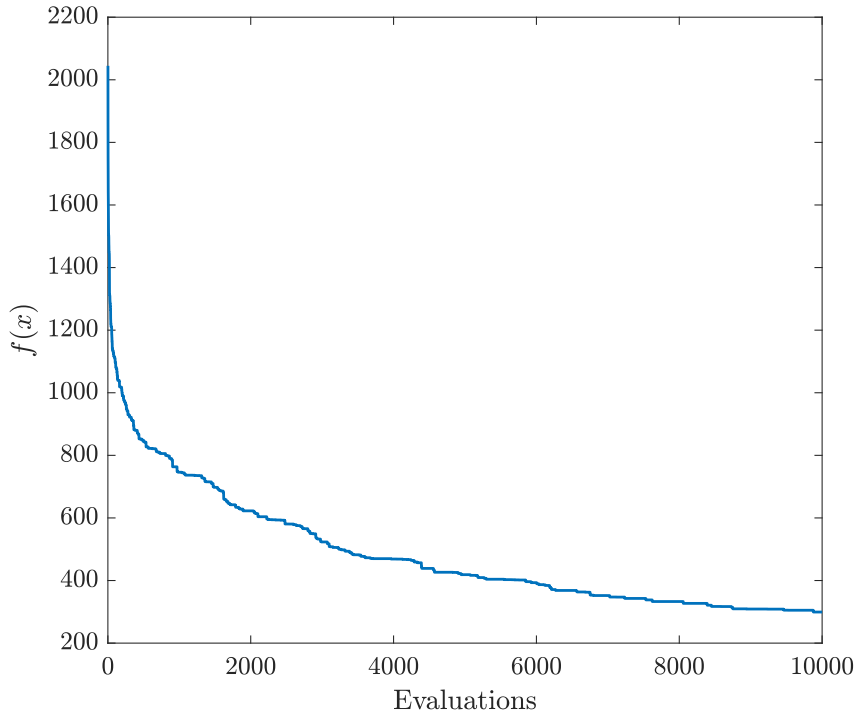


Figure 12: Results from Fig. 11 averaged across all random seeds.

In the following trials, the parameters were held as in Table 2 unless otherwise stated. The same performance measures of the algorithm are used as for Tabu Search, and are computed using 25 random seeds.

Accepted sample limit η_{min}

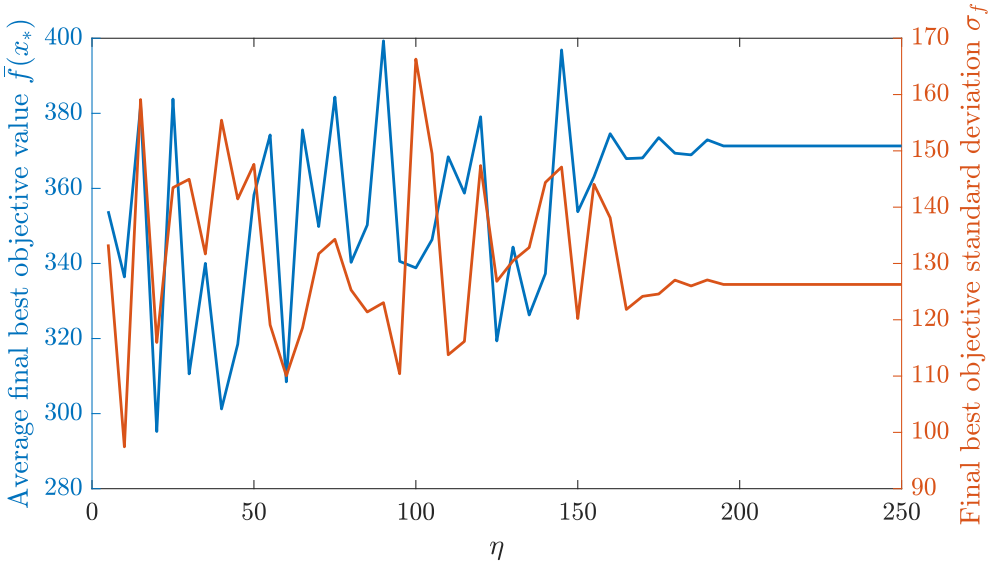


Figure 13: Effect of η_{min} on the performance measures.

Fig. 13 shows that the optimal value for the accepted sample limit η_{min} is roughly 20. With $L = 300$, this surprisingly disagrees with the 4M17 handout's suggestion of $\eta_{\text{min}} \approx 0.6L$. It can be seen that performance gets worse as η_{min} increases in terms of average best objective $\bar{f}(x_*)$. As η_{min} increases, the Annealing Schedule will progress more slowly. Eventually, η_{min} will become redundant because L becomes a stronger threshold, explaining the plateau in the figure. η_{min} causes the Annealing Schedule to speed up in the 'melting' phase when many solutions are accepted. Thus it is expected that if η_{min} becomes too large then the search will spend a greater proportion of time in the melting phase, and perhaps not explore local minima sufficiently in the freezing phase. For the rest of this investigation $\eta_{\text{min}} = 20$ is used.

Cooling factor α

The cooling factor α controls the ratio between the duration of the melting phase and the duration of the freezing phase. Fig. 14 shows a clear optimal value for the cooling factor α of 0.95 (as suggested in Kirkpatrick et al. [1982]) in terms of both performance measures. α is therefore kept at 0.95 for the rest of this report.

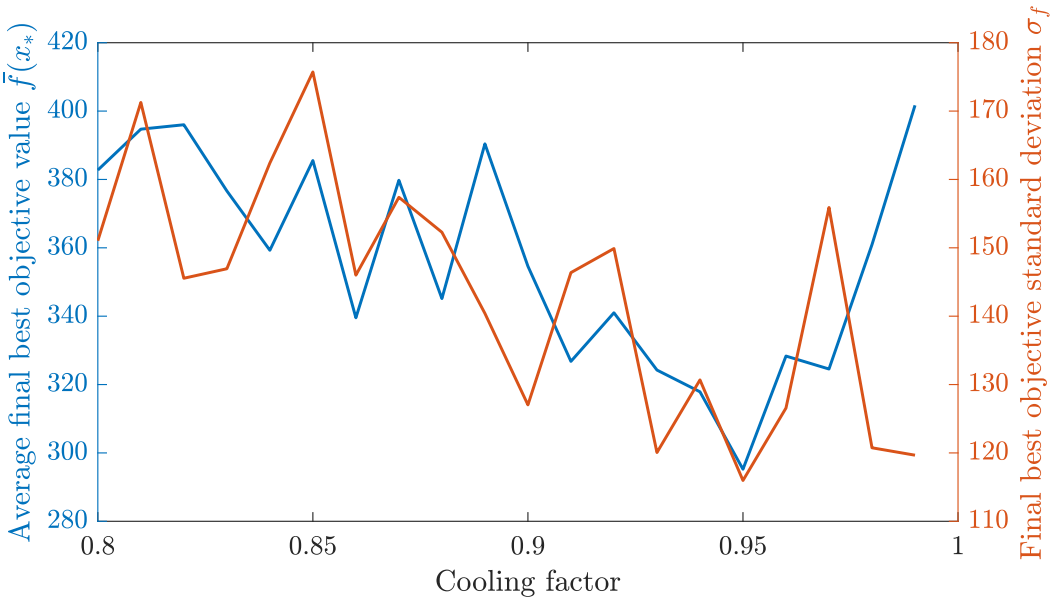


Figure 14: Effect of the cooling factor on the performance measures.

Generation parameter d

Fig. 15 shows that $d = 200$ (as chosen initially) produces the optimal behaviour in terms of the performance measures. The value of d should be chosen according to some characteristic length scale of the objective function, if possible. Since d controls the size of the hypercube of possible moves, if d is too small then the search will not explore the feasible region sufficiently, and the search is likely to simply fall into one of the closest local minima.

If d is too large, then the samples will be very disperse in the melting phase. However, in the freezing phase, the search will be able to 'hop' between local minima, which may be undesirable. Furthermore, when the search has 'frozen' into a local minimum, many samples will be rejected (due to many positive δf moves). The optimal value of $d = 200$ agrees with the size of the 2D SF minima (see the contour plot of Fig. 9), which may well generalise to the 5D SF.

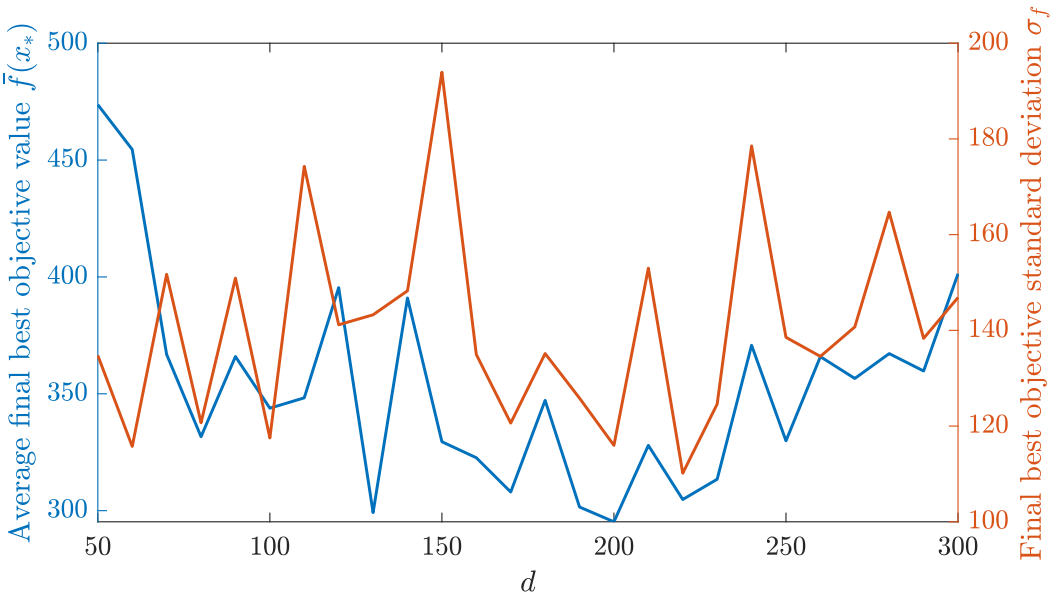


Figure 15: Effect of the generation parameter d on the performance measures.

Performance comparison of Tabu Search and Simulated Annealing

Using the optimal values obtained in the analyses of Tabu Search and SA and using 30 random seeds, the data series in Fig. 16 were generated. The dotted lines indicated the $\pm 1\sigma$ values. 20,0000 evaluations were permitted to allow the algorithms greater opportunity to converge. It is clear that Tabu Search outperforms SA in terms of the average best solution obtained at all evaluations. The standard deviation lines show that both algorithms have similar reliability.

There are some important similarities and differences between the two algorithms. The generation parameter d for SA plays a similar role to the SSS of Tabu Search. However, Tabu Search explicitly reduces the step size through the SRF, while SA does this implicitly through the Annealing Schedule, which makes larger moves less likely as the temperature decreases. This is a less efficient way of performing SSR because many samples will be rejected.

Comparing the 2D searches shown in Fig. 2 and Fig. 9, it is clear that SA performs a more disperse sampling of the feasible region than Tabu Search. However, the lack of a 'memory' element in SA (apart from resetting) means that it is not able to make implicit use of this information. Further, SA is not guaranteed to freeze into the best local minimum it found, while Tabu Search is virtually guaranteed to through the intensification and SSR steps.

There are several ways of augmenting SA to improve performance, such as making the generation matrix D and the cooling factor α vary based on information from the search. This could make SA compete more strongly with Tabu Search, and so Fig. 16 is not a definitive proof that Tabu Search is a stronger metaheuristic than SA.

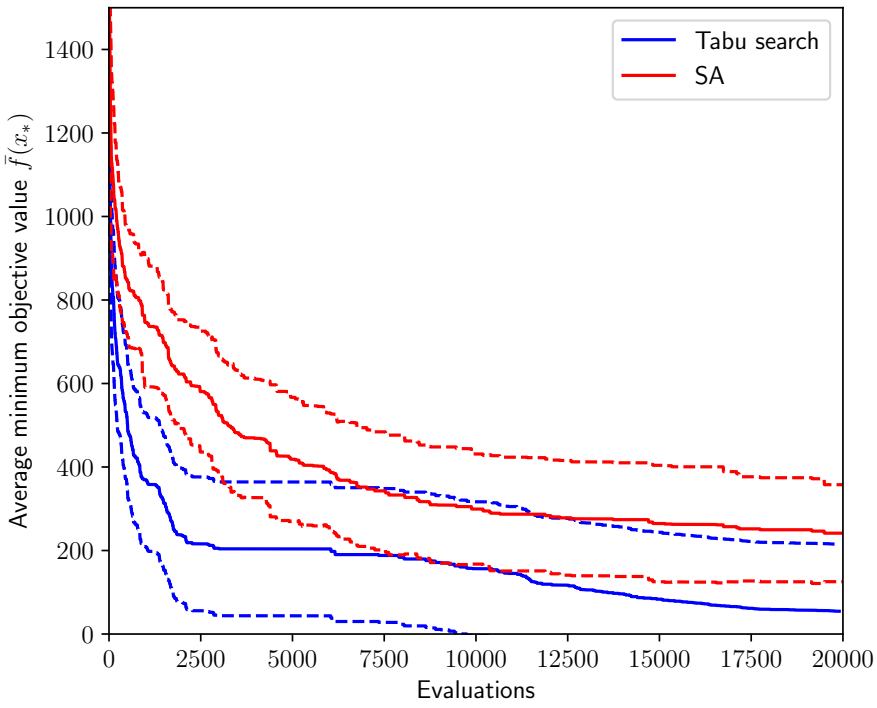


Figure 16: Performance comparison between Tabu Search and SA over 20,000 function evaluations. Dotted lines indicated $\pm 1\sigma$ values.

Conclusions

In this investigation, it was found that Tabu Search performs optimally on the 5D SF when the step reduce factor is close to 1 and the starting step size is large. This makes the search more 'shotgun'-like in the early phases and which provides the search with more information about the function, which the memory elements make use of later in the search. Further, the counter thresholds triggering new moves should be small so that the search does not waste time on poor local minima early in the search.

Simulated Annealing performs best when the accepted sample limit is small, suggesting that an Annealing Schedule that progresses faster in the early stages can be appropriate. A cooling factor of 0.95 performed optimally, as suggested by Kirkpatrick et al. [1982]. The scale over which new solutions are generated should be tuned to the characteristic length scale of the objective function, if possible.

After tuning parameters, Tabu Search performed better than SA. It is suggested that SA performs step size reduction implicitly, and this makes it use solution samples less efficiently than Tabu Search (which does so explicitly). The lack of memory elements in SA also means that it is not guaranteed to hone in on the best local minima it discovers in its search, unlike for Tabu Search. Finally, the variant of SA used in this investigation is not the most sophisticated, so certain augmentations are likely to improve its performance.

References

For the references mentioned in this report, see the reference sections of the relevant 4M17 handouts.

Appendix

Java implementation of Tabu Search

Main program

```
package tabusearch;

import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.stream.Collectors;

import org.json.simple.JSONObject;

// Perform the Tabu Search algorithm on the Schwefel Function

public class TSMain {

   // Main method
   @SuppressWarnings("unchecked")
   public static void main(String[] args) throws CloneNotSupportedException, IOException {

      // Setting up the objective function to minimise
      Function myFunc = new Schwef(); // Using polymorphism to allow for the objective
          function to be easily changed here
      final int dim = 5; // Input dimension
      myFunc.setDim(dim); // Set static dimension variable of the the SchwefelFunction
          class

      // Setting up the classes
      Tabu.myFunc = myFunc;
      Tabu.dim = dim;
      // Default algorithm parameters to be defined (may be varied later in experiments)
      Tabu.verbose = false; // True: print progress events of the search.
      Tabu.seed = 50; // Set the rng seed
      Tabu.intensifyThresh = 1; // Counter limit to intensify search using MTM
      Tabu.diversifyThresh = 6; // Counter limit to diversify search using long-term
          memory (LTM)
      Tabu.ssrThresh = 16; // Counter limit to perform step-size reduction
      Tabu.startingStepSize = 175; // Starting step size for the Tabu local search
      Tabu.stepLimit = 1E-13; // Lower limit on the step size (>1E-13 to prevent double
          precision issues)
      Tabu.stepReduceFactor = 0.985; // Constant factor to reduce stepSize by after
          step-size reduction
```

```
Tabu.evalLimit = 20000; // Convergence criterion - stop when numEvals reaches
    evalLimit
Tabu.stmSize = 8; // Short-term memory (STM) size
MTM.mtmSize = 4; // Medium-term memory (MTM) size
Tabu.constraint = 500.0; // Upper limit on variable magnitude
LTM.setSegSize(100.0); // Long-term memory (LTM) segment size for grid


// EXPERIMENTS FOR PYTHON USING JSON FILES

// Choose whether to save the data
boolean saveData = true;
// Choose the filename of saveData = true
String jsonFilename = "TSjson.json";

// Choose which experiment to do
boolean doExperiment1 = false;
boolean doExperiment2 = true;
boolean doExperimetn3 = false;

JSONObject jsonObj = new JSONObject();
System.out.println("Starting experiment.");

// Store the search path followed and the evolution of the function value for a
    single run
if (doExperiment1 == true) {
   Tabu tabuObj = new Tabu();
   tabuObj.doTabuSearch();
   List<String> tabuPath =
       tabuObj.globSearchHist.stream().map(Point::getStringx).collect(Collectors.toList());
   List<String> fEvolution =
       tabuObj.globSearchHist.stream().map(Point::getStringFval).collect(Collectors.toList());
   jsonObj.put("tabu_path", tabuPath);
   jsonObj.put("f_evolution", fEvolution);
   jsonObj.put("num_eval_evolution", tabuObj.numEvalEvolution);
}

// Store the (zero held) evolution of the function value and the minimum value found
    versus number of function evaluations
// for a number of different seeds (to be averaged in Python)
if (doExperiment2 == true) {
   int nTabuRuns = 25; // Number of iterations to average over with different random
       seeds
   // Nested array of zero-held function evolutions
   ArrayList<LinkedList<Double>> allZeroHeldHistories = new
       ArrayList<LinkedList<Double>>(nTabuRuns);
   ArrayList<LinkedList<Double>> allZeroHeldMinVals = new
       ArrayList<LinkedList<Double>>(nTabuRuns);
   Tabu.seed = 1; // Initial seed
   for (int k = 0; k < nTabuRuns; k++) {
```

```
      Tabu tabuObj = new Tabu();
      tabuObj.doTabuSearch();
      allZeroHeldHistories.add(tabuObj.globfEvolZeroHold);
      allZeroHeldMinVals.add(tabuObj.globMinValZeroHold);
      Tabu.seed+=1;
   }
   jsonObj.put("all_zero_held_evols", allZeroHeldHistories);
   jsonObj.put("all_zero_held_minvals", allZeroHeldMinVals);
}

// Vary a Tabu search parameter and store lists of the best objectives found over
    many random seeds
// (to be averaged in Python)
if (doExperimetn3 == true) {
   int nTabuRuns = 50; // Number of iterations to average over with different random
       seeds
   ArrayList<ArrayList<Double>> bestSolutions = new
       ArrayList<ArrayList<Double>>(nTabuRuns);
   ArrayList<Double> parameterValues = new ArrayList<Double>(nTabuRuns);
   ArrayList<Double> currentBestMinVals; // Min vals for the current parameter value
   for (int l = 1; l <50; l++) { // Range of parameter values
      currentBestMinVals = new ArrayList<Double>(nTabuRuns);
      Tabu.seed = 1; // Initial seed
      Tabu.intensifyThresh = l;
      Tabu.diversifyThresh = Tabu.intensifyThresh + 5;
      Tabu.ssrThresh = Tabu.intensifyThresh + 15;

      for (int k = 0; k < nTabuRuns; k++) {
         Tabu tabuObj = new Tabu();
         tabuObj.doTabuSearch();
         System.out.println(tabuObj.bestSolution.fval);
         currentBestMinVals.add(tabuObj.bestSolution.fval);
         Tabu.seed+=1;
      }
      parameterValues.add((double) Tabu.intensifyThresh);
      bestSolutions.add(currentBestMinVals);
   }
   jsonObj.put("parameter", parameterValues);
   jsonObj.put("best_solus", bestSolutions);
}

System.out.println("\nExperiment completed.");

if (saveData == true) {
   // Save the search data to a .json file for analysis with Python
   System.out.println("Saving the data to a .json file.");

   String workingdir = System.getProperty("user.dir");
   String parentdir = workingdir.substring(0,workingdir.lastIndexOf('\\'));
   String jsondir = parentdir + "\\json\\";
```

```
        try (FileWriter file = new FileWriter(jsondir + jsonFilename)) {
            file.write(jsonObj.toJSONString());
        }

        System.out.println("Json file saved.");
    }
  }
}
```

Tabu.java

```
package tabusearch;

import java.util.Arrays;
import java.util.LinkedList;
import java.util.Random;

// Class with static fields for holding important search information
// and static methods for performing typical processes of Tabu search.
public class Tabu {



  // Inner class for performing a local search (with access to all enclosing class
     members)
  class LocalSearch {



    // Inner class for the short-term memory (STM)
    class STM {
      public int stmSize = Tabu.stmSize; // STM size
      private LinkedList<Point> stmList = new LinkedList<Point>(); // LinkedList
          variable

      public void tryAddToSTM(Point currentPoint) throws CloneNotSupportedException {
        // If the STM is not yet full
        if (stmList.size() < stmSize) {
          stmList.add(currentPoint.clone()); // Add the current point to the STM
        }
        else {
          stmList.offerFirst(currentPoint.clone());
          stmList.removeLast();
        } // Replace with the current point in the STM on a first in, first out basis
      }
    }
```

```
// LocalSearch members
    ------------------------------------------------------------------------------------------

public LinkedList<Point> localSearchHist = new LinkedList<Point>(); // List to store
    history of points in the local search
// Local history of objective function value zero-held with # of function evaluations
public LinkedList<Double> localfEvolZeroHold = new LinkedList<Double>();
STM stmObj = new STM();
private double stepSize; // Step size
private Point currentPoint; // Point object corresponding to the current position of
    the local search
private int nEvalsLastIter; // Number of function evaluations during the previous
    iteration
private int num_its = 0; // Total number of local search iterations

// Constructor for creating a LocalSearch object
public LocalSearch(double stepSize) {
   this.stepSize = stepSize;
}

// Generate a linked list of points around 'currentPoint' to test
// This is done by incrementing and decrementing each variable of the current point
// The testList object is reused by reference to avoid frequent expensive object
    creation
private void updateTestList(LinkedList<double[]> testList, Point currentPoint,
    double stepSize) {
   testList.clear(); // Remove all elements from the list

   // Populate the list
   for (int i = 0; i < currentPoint.x.length; i++) {
      double[] xTemp1 = currentPoint.x.clone();
      xTemp1[i] += stepSize;
      testList.add(xTemp1);

      double[] xTemp2 = currentPoint.x.clone();
      xTemp2[i] -= stepSize;
      testList.add(xTemp2);
   }
}

// Generate list to store the permitted non-tabu moves that are within the feasible
    region
// The validList object is reused by reference to avoid frequent expensive object
    creation
private void updateValidList(LinkedList<double[]> testList,LinkedList<Point>
    validList, STM STMObj) {
   validList.clear(); // Remove all elements from the list
```

```
   for (double[] testEl : testList) {
      boolean validCheck1 = true; // Feasible region check
      for (int i = 0; i < testEl.length; i++) {
         if (Math.abs(testEl[i]) > constraint) {
            validCheck1 = false;
         } // "If any of the variables violate the constraints"
      }

      boolean validCheck2 = true; // Non-tabu check
      if (validCheck1 == true) {
         for (Point stmEl : STMObj.stmList) {
            if (Arrays.equals(testEl,stmEl.x)) {
               validCheck2 = false;
            } // "If the test point is in the STM"
         }
      }

      if (validCheck1==true && validCheck2==true) {
         validList.add(new Point(testEl,myFunc));
      }
   }
}

// Make the best allowed move of the current and return true of the objective
    function was reduced
private boolean makeBestAllowedMove(LinkedList<Point> validList) throws
    CloneNotSupportedException {
   boolean functionReduced = false;
   double prevFval = currentPoint.fval;

   if (validList.peekFirst() == null) {
      // This is very unlikely to occur unless the input space is 1D
      // and the search has reached the feasible region boundary
      // or if the stepSize is below double precision (~1E-16)
      System.out.println("Error: There are no valid, non-tabu moves in the current
          local search.");
      System.exit(0);
   }
   else {
      // Find the element with minimum function value
      double min = validList.peekFirst().fval;
      int minloc = 0;
      for (int i = 0; i < validList.size(); i++) {
         Point p = validList.get(i);
         if (p.fval < min) {
            min = p.fval;
            minloc = i;
         }
      }
```

```
      currentPoint = validList.get(minloc).clone(); // Move the current point to the
          best valid position

      // If the function is reduced by the step, see if it is the new local minimum
          value
      if (currentPoint.fval < prevFval) {
         functionReduced = true;
      }
   }

   return functionReduced;
}


// Attempt a pattern move and execute it if it improves upon the minimum of the
    local search
private void attemptPatternMove(double[] xBase, Point currentPoint, STM stmObj)
    throws CloneNotSupportedException {
   double[] xCurrent = currentPoint.x;
   double[] xPattern = new double[xCurrent.length];

   // Perform a pattern move by repeating the vector from the last base point
   for (int i = 0; i < xPattern.length; i++) {
      xPattern[i] = xCurrent[i] + (xCurrent[i] - xBase[i]);
   }

   // Check whether the pattern move is still in the feasible region
   boolean validCheck = true; // Feasible region check
   for (int i = 0; i < xPattern.length; i++) {
      if (Math.abs(xPattern[i]) > constraint) {
         validCheck = false;
      }
   }

   if (validCheck == true) {
      // If the objective function is reduced, store the intermediate Tabu move and
          retain the pattern move
      double fPattern = myFunc.f(xPattern);
      if (fPattern < currentPoint.fval) {
         storePoint(currentPoint);
         currentPoint = new Point();
         currentPoint.x = xPattern.clone();
         currentPoint.fval = fPattern;
      }
   }
}

// Attempt to store a clone of the current Point object in the various memory objects
private void storePoint(Point currentPoint) throws CloneNotSupportedException {
   // Update best solution if the best objective has been reduced
```

```
    checkBestSolution(currentPoint);

    // Zero hold the function evolution and best solution values by
    // the number of evaluations since the last point was stored
    nEvalsLastIter = numEvals - numEvalsLagged;
    for (int i = 0; i < nEvalsLastIter; i++) {
       localfEvolZeroHold.add(currentPoint.fval); // Add the current point to the
          local search history
       globMinValZeroHold.add(bestSolution.fval); // Add the current optimal solution
    }

    localSearchHist.add(currentPoint.clone()); // Add the current point to the local
       search history
    // See if the point should be stored in the STM, the MTM or the global minimum
    stmObj.tryAddToSTM(currentPoint);
    mtmObj.tryAddToMTM(currentPoint);
    // Attempt to store current point in LTM after an integer number of iterations,
    // where the integer is the number of Tabu steps that fit within each grid
       segment.
    if (num_its % ltmUpdateRate == 0) {
       ltmObj.storeInLTM(currentPoint.x);
    }
    numEvalEvolution.add(numEvals); // Record number of objective function evaluations

    numEvalsLagged = numEvals; // Number of evaluations since previous stored point
}

// Perform a local search starting at startingPoint and ending when the counter
    threshold
// has been reached
public void doLocalSearch(Point startingPoint) throws CloneNotSupportedException {

    // Initialisation
    currentPoint = startingPoint.clone(); // Generate the initial point
    storePoint(currentPoint);

    LinkedList<double[]> testList = new LinkedList<double[]>(); // List of
       coordinates to check for validity
    LinkedList<Point> validList = new LinkedList<Point>(); // List of points
       corresponding to valid Tabu moves
    int counterThresh; // Counter limit for this local search

    if (searchType.matches("initialise|ssr")) {
       counterThresh = intensifyThresh;
    }
    else if (searchType.equals("intensify")) {
       counterThresh = diversifyThresh;
    }
    else {
       counterThresh = ssrThresh;
```

```
        }

        // Begin local search
        while (counter < counterThresh) {
            if (verbose == true) {
                System.out.print(counter + " ");
            }

            double[] xBase = currentPoint.x.clone(); // Update the base point
            updateTestList(testList,currentPoint,stepSize);
            updateValidList(testList,validList,stmObj);
            boolean functionReduced = makeBestAllowedMove(validList); // Move currentPoint
                by best allowed move

            // Reset the counter if a new global minimum value has been found
            if (currentPoint.fval < bestSolution.fval) {
                counter = 0;
            }
            else {
                counter += 1; // Increment the counter
            }

            storePoint(currentPoint);

            // If the objective function was reduced, attempt a pattern move
            if (functionReduced == true) {
                attemptPatternMove(xBase, currentPoint, stmObj);
            }

            num_its++;
            if (num_its > 1000000) {
                System.out.println("Error: Excessive number of local search iterations
                    reached without triggering counter threshold.");
                System.exit(0);
            }

        }

        if (searchType.matches("initialise|ssr")) {
            searchType = "intensify";
        }
        else if (searchType.equals("intensify")) {
            searchType = "diversify";
        }
        else {
            searchType = "ssr";
        }
    }
}
```

```
//// Tabu members
    ------------------------------------------------------------------------------------------

// Shared static variables
public static boolean verbose; // True: print progress events of the search.
public static int evalLimit; // Convergence criterion on the number of objective
    function evaluations permitted
public static int dim; // Input dimension
public static double constraint; // Upper limit on variable magnitude
public static Function myFunc; // Function to minimise
public static double startingStepSize; // Starting step size for the Tabu local search
public static double stepLimit; // Lower limit on the step size
public static double stepReduceFactor; // Constant factor to reduce the step size by
    after step-size reduction
public static long seed; // Rng seed
public static int stmSize; // Short-term memory (STM) size
public static int intensifyThresh; // Counter threshold to intensify search
public static int diversifyThresh; // Counter threshold to diversify search
public static int ssrThresh; // Counter threshold to perform step-size reduction
public static Random generator; // Random generator (shared across the package)
public static int numEvals; // Counter for number of objective function evaluations

// Instance variables
private int ltmUpdateRate; // Local search iterations per update to the LTM
public Point bestSolution; // Current Point with the minimum associate objective
    function value
private Point startingPoint = new Point(); // Force the local search to begin from
    this point
public double stepSize; // Variable step size
private MTM mtmObj = new MTM();
private LTM ltmObj = new LTM();
private int numEvalsLagged = 0; // Number of function evaluations up to the previous
    Tabu iteration
private String searchType; // Type of local search to conduct: "initial", "intensify",
    "diversify" or "ssr"
public LinkedList<Point> globSearchHist = new LinkedList<Point>(); // Object to store
    the entire Tabu search history as a linked list
public LinkedList<Double> globfEvolZeroHold= new LinkedList<Double>(); // History zero
    held with # of function evaluations
public LinkedList<Integer> numEvalEvolution = new LinkedList<Integer>(); // Evolution
    of the number of function evaluations
public LinkedList<Double> globMinValZeroHold = new LinkedList<Double>(); // History of
    optimal solution found so far (zero held)
private int counter; // Counter for number of iterations without improvement to
    minimum value found

// Generate a random input point
public static Point genRandomPoint() {
```

```
   double[] x = new double[dim];
   for (int i = 0; i < x.length; i++) {
      x[i] = generator.nextDouble() * (2*constraint) - constraint;
   }

   return new Point(x, myFunc);
}

// Check whether this point corresponds the best solution found so far and store it if
    so
private void checkBestSolution(Point currentPoint) throws CloneNotSupportedException {
   if (bestSolution == null) {
      bestSolution = currentPoint.clone();
   }
   else if (currentPoint.fval < bestSolution.fval) {
      bestSolution = currentPoint.clone();
   }
}

// Perform a complete Tabu search
public void doTabuSearch() throws CloneNotSupportedException {
   // Initialisation
   generator = new Random(seed);
   numEvals = 0;
   counter = 0;
   stepSize = startingStepSize;
   ltmUpdateRate = (int) Math.ceil(LTM.getSegSize()/(2*stepSize));
   searchType = "initialise";
   if (verbose == true) {
      System.out.println("\n\nStarting a Tabu search. Printing counter evolution and
          other search events.");
   }

   startingPoint = genRandomPoint();

   while (numEvals < evalLimit) {
      if (searchType.equals("initialise") && verbose == true) {
         System.out.print("Initial search: ");
      }

      // Intensify search
      if (searchType.equals("intensify")) {
         startingPoint = MTM.findMTMAvg(mtmObj);
         if (verbose == true) {
            System.out.print("\nIntensiying search: ");
         }
      } // Intensify: set starting point to the meant point of the MTM

      // Diversify search
      else if (searchType.equals("diversify")) {
```

```
            startingPoint = ltmObj.genDiversifiedPoint();
        } // Diversify: generate a random point whose grid coordinates are not in the LTM

        // Reduce step size
        else if (searchType.equals("ssr")) {
            startingPoint = bestSolution; // Restart from the minimum point found so far
            // If the step size limit will not be violated, reduce the step size
            if (stepSize*stepReduceFactor > stepLimit) {
                stepSize = stepSize*stepReduceFactor; // Reduce the increment size by a
                    constant factor
            }
            counter = 0; // Reset counter
            ltmUpdateRate = (int) Math.ceil(LTM.getSegSize()/(2*stepSize));
            if (verbose == true) {
                System.out.print("\nReducing step size: ");
            }
        }

        // Perform a local search from startingPoint
        LocalSearch LSObj = new LocalSearch(stepSize);
        LSObj.doLocalSearch(startingPoint);
        globSearchHist.addAll(LSObj.localSearchHist); // Append the local search history
            of points to the global search history
        globfEvolZeroHold.addAll(LSObj.localfEvolZeroHold); // Append zero-held objective
            history
    }
  }
}
```

LTM.java

```
package tabusearch;

import java.util.HashSet;
import java.util.Set;

// Class for the long-term memory (LTM) grid for search diversification. This divides the
    feasible region into a grid and
// stores the grid segments visited during the Tabu search in a HashSet. In
    diversification, when a new random starting
// point is generated it must not lie in one of the grid segments already visited.
public class LTM {

  public Set<LTMGridPoint> ltmSet = new HashSet<LTMGridPoint>(); // Set of grid
      coordinates already visited
  private static double segSize; // Segment size for the LTM grid
  public boolean ltmFull = false; // True if the LTM is full
```

```
public static void setSegSize(double segSizeVal) {
   segSize = segSizeVal;

   if (Tabu.constraint % segSize != 0.0) {
      System.out.println(
            "Warning: It is recommended that the constraint be divisible by the LTM grid
               segment size.");
   }
}

public static double getSegSize() {
   return segSize;
}

// Convert current point into 'grid' coordinates and try to add it to the LTM set
public boolean storeInLTM(double[] currentPos) {
   // Convert current point to grid coordinates
   double factor = Tabu.constraint / segSize; // Half of the number of grid segments
       for each variable
   LTMGridPoint gridPoint = new LTMGridPoint();

   for (int i = 0; i < Tabu.dim; i++) {
      gridPoint.gridPos[i] = (int) Math.ceil(currentPos[i] / Tabu.constraint * factor);
   }

   // Attempt to add new grid coordinates to the set. Will not add if the set
   // already contains this grid position.
   boolean addingToLTM = ltmSet.add(gridPoint);

   if (addingToLTM == true && ltmFull == false) {
      if (ltmSet.size() == (int) Math.pow(2 * factor, Tabu.dim) && Tabu.constraint %
          segSize == 0) {
         ltmFull = true;
         if (Tabu.verbose == true) {
            System.out.println("(LTM set now full) ");
         }
      } // "If a grid point was added to the LTM and it is now full"
   }

   return addingToLTM;
}

// Generate a random input point
public Point genDiversifiedPoint() {
   Point p = Tabu.genRandomPoint();
   int num_attempts = 0;

   if (ltmFull == false) {
      while (storeInLTM(p.x) == false) {
         p = Tabu.genRandomPoint();
```

```
            num_attempts++;
         } // "While the generated point is already in LTM, generate a new point and try
            again."
      }

      if (Tabu.verbose == true) {
         System.out.print("\nDiversifying after " + num_attempts + " rejected samples: ");
      }
      return p;
   }
}
```

MTM.java

```
package tabusearch;

import java.util.LinkedList;

//Class for the medium-term memory (MTM) storing the 'mtmSize' Points with the least
   function
//values visited so far
public class MTM {

   public static int mtmSize; // Size of the medium-term memory
   public double maxMTMVal; // Greatest function value in the MTM
   public int maxMTMLoc; // Index into the MTM list of the greatest function value point
   private LinkedList<Point> mtmList = new LinkedList<Point>(); // LinkedList for the MTM

   // Update the greatest value in the MTM and its location
   public void setMaxMTMPoint() {
      if (mtmList.size() > 0) {
         maxMTMVal = mtmList.getFirst().fval;
         maxMTMLoc = 0;
         for (int i = 1; i < mtmList.size(); i++) {
            Point p = mtmList.get(i);
            if (p.fval > maxMTMVal) {
               maxMTMVal = p.fval;
               maxMTMLoc = i;
            }
         }
      }
   }

   public void tryAddToMTM(Point currentPoint) throws CloneNotSupportedException {
      // Only call this if the current value is lower than the greatest in the MTM
      // Replace the corresponding Point in MTM with the currentPoint
      if (mtmList.size() < mtmSize) {
         mtmList.addLast(currentPoint.clone());
```

```
            setMaxMTMPoint();
        } // "If the MTM list is not yet full, add the current point"
        else if (currentPoint.fval < maxMTMVal) {
            mtmList.remove(maxMTMLoc);
            mtmList.addLast(currentPoint.clone());
            setMaxMTMPoint();
        } // If the MTM is full, replace the point corresponding to the maximum function
            value in the MTM with the current point
    }

    // Sum and average the coordinates of the locations in the MTM and return the
    // corresponding Point object
    public static Point findMTMAvg(MTM mtmObj) {
        double[] avgLoc = new double[Tabu.dim];
        for (int i = 0; i < Tabu.dim; i++) {
            double avgSum = 0;
            for (Point mtmEl : mtmObj.mtmList) {
                avgSum += mtmEl.x[i];
            }
            avgLoc[i] = avgSum / mtmSize;
        }

        return new Point(avgLoc, Tabu.myFunc);
    }
}
```

Point.java

```
package tabusearch;

import java.util.Arrays;

// Object storing an input location and its associated function value.
// N.B. The Func.Schwef.dim field should be set before an object of this class can be
    created,
// otherwise an error will occur.

public class Point implements Cloneable {

    public double[] x; // Position
    public double fval; // Associated function value

    // Constructor to initialise the object
    public Point(double[] x, Function myFunc) {
        this.x = x;
        fval = myFunc.f(x);
    }
```

```
   public Point() {
      // Empty constructor
   }

   // Return a string of the x array
   public String getStringx() {
      return Arrays.toString(x);
   }

   // Return a string of function value
   public String getStringFval() {
      return String.valueOf(fval);
   }

   @Override
   // Create and return a deep clone
   protected Point clone() throws CloneNotSupportedException {
      Point p = new Point();
      p.x = this.x.clone();
      p.fval = this.fval;
      return p;
   }

}
```

LTMGridPoint.java

```
package tabusearch;

import java.util.Arrays;

// Class for long-term memory (LTM) grid points.
// This is necessary because int[] is a reference type and so Object.equals must be
    overridden to compare the arrays with 'equals()'.
// This allows the 'add' method of HashSet to behave as expected when adding grid points
    to the LTM.
public class LTMGridPoint {
   int[] gridPos = new int[Tabu.dim];

   @Override
   public int hashCode() {
      final int prime = 31;
      int result = 1;
      result = prime * result + Arrays.hashCode(gridPos);
      return result;
   }

   @Override
```

```
   public boolean equals(Object obj) {
      if (this == obj)
         return true;
      if (obj == null)
         return false;
      if (getClass() != obj.getClass())
         return false;
      LTMGridPoint other = (LTMGridPoint) obj;
      if (!Arrays.equals(gridPos, other.gridPos))
         return false;
      return true;
   }

}
```

Function.java

```
package tabusearch;

// Interface for functions to be minimised by Tabu Search.
// All classes which implement this interface must have all of the methods stated in the
    interface.

public interface Function {

   public void setDim(int dimension);
   public double f(double[] x);

}
```

Schwef.java

```
package tabusearch;

import java.lang.Math;

// Class containing functions which we seek to minimise.

public class Schwef implements Function {

   private int dim; // Input dimension

   // Setter for the dimension variable 'dim'
   public void setDim(int dim) {
      this.dim = dim;
```

```
        if (dim <= 0) {
            System.out.println("Error: Input dimension must be a positive integer.");
            System.exit(0);
        }
    }

    // Method to evaluate and return the Schwefel function (dim must be defined).
    public double f(double[] x) {
        Tabu.numEvals++; // Counter for the number of function evaluations

        double funcval = 418.9829*dim;

        if (x.length == dim) {
            for (Double xi : x) {
                funcval = funcval - xi * Math.sin(Math.sqrt(Math.abs(xi)));
            }
        } else if (dim == 0) {
            System.out.println("Error: You must initialise the input dimension before
                attempting evaluation.");
            System.exit(0);
        } else {
            System.out.println("Error: Dimension of input point must match the defined input
                dimension.");
            System.exit(0);
        }

        return funcval;
    }

}
```

Simulated Annealing code

Original `anneal.m` file

Source available here (PDF version only)

```
function [minimum,fval] = anneal(loss, parent, options)
% ANNEAL Minimizes a function with the method of simulated annealing
% (Kirkpatrick et al., 1983)
%
%  ANNEAL takes three input parameters, in this order:
%
%  LOSS is a function handle (anonymous function or inline) with a loss
%  function, which may be of any type, and needn't be continuous. It does,
%  however, need to return a single value.
%
```

```
%  PARENT is a vector with initial guess parameters. You must input an
%  initial guess.
%
%  OPTIONS is a structure with settings for the simulated annealing. If no
%  OPTIONS structure is provided, ANNEAL uses a default structure. OPTIONS
%  can contain any or all of the following fields (missing fields are
%  filled with default values):
%
%      Verbosity: Controls output to the screen.
%                 0 suppresses all output
%                 1 gives final report only [default]
%                 2 gives temperature changes and final report
%      Generator: Generates a new solution from an old one.
%                 Any function handle that takes a solution as input and
%                 gives a valid solution (i.e. some point in the solution
%                 space) as output.
%                 The default function generates a row vector which
%                 slightly differs from the input vector in one element:
%                 @(x) (x+(randperm(length(x))==length(x))*randn/100)
%                 Other examples of possible solution generators:
%                 @(x) (rand(3,1)): Picks a random point in the unit cube
%                 @(x) (ceil([9 5].*rand(2,1))): Picks a point in a 9-by-5
%                                                discrete grid
%                 Note that if you use the default generator, ANNEAL only
%                 works on row vectors. For loss functions that operate on
%                 column vectors, use this generator instead of the
%                 default:
%                 @(x) (x(:)'+(randperm(length(x))==length(x))*randn/100)'
%       InitTemp: The initial temperature, can be any positive number.
%                 Default is 1.
%       StopTemp: Temperature at which to stop, can be any positive number
%                 smaller than InitTemp.
%                 Default is 1e-8.
%        StopVal: Value at which to stop immediately, can be any output of
%                 LOSS that is sufficiently low for you.
%                 Default is -Inf.
%      CoolSched: Generates a new temperature from the previous one.
%                 Any function handle that takes a scalar as input and
%                 returns a smaller but positive scalar as output.
%                 Default is @(T) (.8*T)
%     MaxConsRej: Maximum number of consecutive rejections, can be any
%                 positive number.
%                 Default is 1000.
%       MaxTries: Maximum number of tries within one temperature, can be
%                 any positive number.
%                 Default is 300.
%     MaxSuccess: Maximum number of successes within one temperature, can
%                 be any positive number.
%                 Default is 20.
%
```

```
%
%  Usage:
%    [MINIMUM,FVAL] = ANNEAL(LOSS,NEWSOL,[OPTIONS]);
%         MINIMUM is the solution which generated the smallest encountered
%         value when input into LOSS.
%         FVAL is the value of the LOSS function evaluated at MINIMUM.
%    OPTIONS = ANNEAL();
%         OPTIONS is the default options structure.
%
%
%  Example:
%    The so-called "six-hump camelback" function has several local minima
%    in the range -3<=x<=3 and -2<=y<=2. It has two global minima, namely
%    f(-0.0898,0.7126) = f(0.0898,-0.7126) = -1.0316. We can define and
%    minimise it as follows:
%         camel = @(x,y)(4-2.1*x.^2+x.^4/3).*x.^2+x.*y+4*(y.^2-1).*y.^2;
%         loss = @(p)camel(p(1),p(2));
%         [x f] = ANNEAL(loss,[0 0])
%    We get output:
%              Initial temperature:    1
%              Final temperature:      3.21388e-007
%              Consecutive rejections:  1027
%              Number of function calls: 6220
%              Total final loss:       -1.03163
%              x =
%                 -0.0899   0.7127
%              f =
%                 -1.0316
%    Which reasonably approximates the analytical global minimum (note
%    that due to randomness, your results will likely not be exactly the
%    same).

%  Reference:
%    Kirkpatrick, S., Gelatt, C.D., & Vecchi, M.P. (1983). Optimization by
%    Simulated Annealing. _Science, 220_, 671-680.

%   joachim.vandekerckhove@psy.kuleuven.be
%   $Revision: v5 $ $Date: 2006/04/26 12:54:04 $

def = struct(...
        'CoolSched',@(T) (.8*T),...
        'Generator',@(x) (x+(randperm(length(x))==length(x))*randn/100),...
        'InitTemp',1,...
        'MaxConsRej',1000,...
        'MaxSuccess',20,...
        'MaxTries',300,...
        'StopTemp',1e-8,...
        'StopVal',-Inf,...
        'Verbosity',1);
```

```
% Check input
if ~nargin %user wants default options, give it and stop
    minimum = def;
    return
elseif nargin<2, %user gave only objective function, throw error
    error('MATLAB:anneal:noParent','You need to input a first guess.');
elseif nargin<3, %user gave no options structure, use default
    options=def;
else %user gave all input, check if options structure is complete
    if ~isstruct(options)
        error('MATLAB:anneal:badOptions',...
            'Input argument ''options'' is not a structure')
    end
    fs = {'CoolSched','Generator','InitTemp','MaxConsRej',...
        'MaxSuccess','MaxTries','StopTemp','StopVal','Verbosity'};
    for nm=1:length(fs)
        if ~isfield(options,fs{nm}), options.(fs{nm}) = def.(fs{nm}); end
    end
end

% main settings
newsol = options.Generator;   % neighborhood space function
Tinit = options.InitTemp;     % initial temp
minT = options.StopTemp;      % stopping temp
cool = options.CoolSched;     % annealing schedule
minF = options.StopVal;
max_consec_rejections = options.MaxConsRej;
max_try = options.MaxTries;
max_success = options.MaxSuccess;
report = options.Verbosity;
k = 1;                        % boltzmann constant

% counters etc
itry = 0;
success = 0;
finished = 0;
consec = 0;
T = Tinit;
initenergy = loss(parent);
oldenergy = initenergy;
total = 0;
if report==2, fprintf(1,'\n T = %7.5f, loss = %10.5f\n',T,oldenergy); end

while ~finished;
    itry = itry+1; % just an iteration counter
    current = parent;

    % % Stop / decrement T criteria
    if itry >= max_try || success >= max_success;
        if T < minT || consec >= max_consec_rejections;
```

```
            finished = 1;
            total = total + itry;
            break;
        else
            T = cool(T); % decrease T according to cooling schedule
            if report==2, % output
                fprintf(1,' T = %7.5f, loss = %10.5f\n',T,oldenergy);
            end
            total = total + itry;
            itry = 1;
            success = 1;
        end
    end

    newparam = newsol(current);
    newenergy = loss(newparam);

    if (newenergy < minF),
        parent = newparam;
        oldenergy = newenergy;
        break
    end

    if (oldenergy-newenergy > 1e-6)
        parent = newparam;
        oldenergy = newenergy;
        success = success+1;
        consec = 0;
    else
        if (rand < exp( (oldenergy-newenergy)/(k*T) ));
            parent = newparam;
            oldenergy = newenergy;
            success = success+1;
        else
            consec = consec+1;
        end
    end
end

minimum = parent;
fval = oldenergy;

if report;
    fprintf(1, '\n Initial temperature:  \t%g\n', Tinit);
    fprintf(1, ' Final temperature:    \t%g\n', T);
    fprintf(1, ' Consecutive rejections: \t%i\n', consec);
    fprintf(1, ' Number of function calls:\t%i\n', total);
    fprintf(1, ' Total final loss:     \t%g\n', fval);
end
```

Modified `mod_anneal.m` file

```
function [bestVal,path,funcEvol,minvalEvol] = mod_anneal(loss, parent, options)
%  My modified version of anneal.m

%  Reference:
%    Kirkpatrick, S., Gelatt, C.D., & Vecchi, M.P. (1983). Optimization by
%    Simulated Annealing. _Science, 220_, 671-680.

%   joachim.vandekerckhove@psy.kuleuven.be
%   $Revision: v5 $ $Date: 2006/04/26 12:54:04 $

% default values
def = struct(...
        'CoolSched',@(T) (.8*T),...
        'Generator',@(x) (x+(randperm(length(x))==length(x))*randn/100),...
        'InitTemp',1,...
        'MaxConsRej',1000,...
        'MaxSuccess',20,...
        'MaxTries',300,...
        'StopTemp',1e-8,...
        'StopVal',-Inf,...
        'Verbosity',1);

% Check input
if ~nargin %user wants default options, give it and stop
    minimum = def;
    return
elseif nargin<2, %user gave only objective function, throw error
    error('MATLAB:anneal:noParent','You need to input a first guess.');
elseif nargin<3, %user gave no options structure, use default
    options=def;
else %user gave all input, check if options structure is complete
    if ~isstruct(options)
        error('MATLAB:anneal:badOptions',...
            'Input argument ''options'' is not a structure')
    end
    fs = {'CoolSched','Generator','InitTemp','MaxConsRej',...
        'MaxSuccess','MaxTries','StopTemp','StopVal','Verbosity'};
    for nm=1:length(fs)
        if ~isfield(options,fs{nm}), options.(fs{nm}) = def.(fs{nm}); end
    end
end

% main settings
experiment = options.Experiment; % 1: output search path & funcval evolution
newsol = options.Generator;   % neighborhood space function
cool = options.CoolSched;     % annealing schedule
```

```
max_try = options.MaxTries;   % max no of tries at one temp
max_success = options.MaxSuccess;% max no of accepted samples within one temp
report = options.Verbosity;
k = 1;                        % boltzmann constant

% counters etc
itry = 0;
success = 0;
finished = 0;
num_evals = 0;
noimprovement = 0; % number of samples since the last best solution
restart_limit = 1500;
consec = 0;
total = 0; % total number of function evaluations
num_eval_limit = 20000; % convergence limit on 'total'
initenergy = loss(parent); total = total + 1; % (one function evaluation here)
oldenergy = initenergy;
total = 0; % counter for total
if report==2, fprintf(1,'\n T = %7.5f, loss = %10.5f\n',T,oldenergy); end

% Variables for experiments
path = [];
funcEvol = [];
minvalEvol = [];

% Determine initial temperature using White [1984]
dim = length(parent);
n_white_samples = 100;
white_samples = zeros(n_white_samples,1);
for i = 1:n_white_samples
    white_samples(i) = loss(1000*rand(1,dim) - 500);
    total = total + 1;
end
Tinit = std(white_samples,1);
T = Tinit;
%disp(Tinit)

% Determine initial temperature using Kirkpatrick [1984]

while ~finished
    num_evals = num_evals + 1;
    itry = itry+1; % counter for number of func evals at current temperature
    current = parent;

    % Schedule update: terminate or adjust temperature
    if itry >= max_try || success >= max_success
        % Stop if # evaluations exceeds limit
        if total >= num_eval_limit
            finished = 1;
            total = total + itry;
```

```
        break;
    % Decrement T
    else
        T = cool(T); % decrease T according to cooling schedule
        if report==2 % output
            fprintf(1,' T = %7.5f, loss = %10.5f\n',T,oldenergy);
        end
        total = total + itry;
        itry = 1;
        success = 1;
    end
end

% Restart search from the best solution if there has been no
% improvement
if noimprovement > restart_limit
    newparam = bestPos;
    newenergy = minvalEvol(end);
    itry = itry+1;
    noimprovement = 0;
    % Store reset function value depending on experiments
    if ~isempty(find(experiment,2))
        funcEvol = [funcEvol newenergy];
    end
else
    % Reject samples not in the feasible region, reject it
    isfeasible = false;
    while isfeasible == false
        newparam = newsol(current);
        if ~all(abs(newparam)<500)
            isfeasible = false;
        else
            isfeasible = true;
        end
    end
    newenergy = loss(newparam);
end

% Try to update best solution
if ~isempty(find(experiment,3))
    if isempty(minvalEvol) || newenergy < minvalEvol(end)
        minvalEvol = [minvalEvol newenergy]; % Update best objective
        bestPos = newparam;
    else
        minvalEvol = [minvalEvol minvalEvol(end)]; % Repeat best objective
        noimprovement = noimprovement + 1;
    end
end

% Update the solution if the objective is reduced (by a non-negligible
```

```
    % amount)
    if (oldenergy-newenergy > 1e-6)
        parent = newparam;

        % Store new param depending on experiments
        if ~isempty(find(experiment,1))
            path = [path parent.'];
        end
        if ~isempty(find(experiment,2))
            funcEvol = [funcEvol newenergy];
        end
        oldenergy = newenergy;
        success = success+1; % Increment accepted solution counter
        consec = 0; % Reset consecutive rejection counter
    % If the objective is increased, store trial solution randomly
    else
        % Storing the trial
        if (rand < exp( (oldenergy-newenergy)/(k*T) ))
            parent = newparam;

            % Store new param depending on experiments
            if ~isempty(find(experiment,1))
                path = [path parent.'];
            end
            if ~isempty(find(experiment,2))
                funcEvol = [funcEvol newenergy];
            end
            oldenergy = newenergy;
            success = success+1; % Increment accepted solution counter
        % Rejecting the trial
        else
            consec = consec+1; % Increment rejected solution counter
            funcEvol = [funcEvol oldenergy];
        end
    end
end

bestVal = minvalEvol(end);

if report
    fprintf(1, '\n Initial temperature:  \t%g\n', Tinit);
    fprintf(1, ' Final temperature:    \t%g\n', T);
    fprintf(1, ' Consecutive rejections: \t%i\n', consec);
    fprintf(1, ' Number of function calls:\t%i\n', total);
end
```