COMP2001: Artificial Intelligence Methods – Lab Exercise 2

Recommended Timescales

This lab exercise is designed to take no longer **four** hours and it is recommended that you start and complete this exercise within the **two weeks** commencing 12th February 2024. The contents of this exercise will be/was covered in **lectures 3 and 4**.

Getting Support

The computing exercises have been designed to facilitate flexible studying and can be worked through in your own time or during timetabled lab hours. It is recommended that you attend the lab even if you have completed the exercises in your own time to discuss your solutions and understanding with one of the lab support team and/or your peers. It is entirely possible that you might make a mistake in your solution which leads to a false observation and understanding.

Learning Outcomes

By completing this lab exercise, you should meet the following learning outcomes:

- Students should gain experience implementing some (single point-based) metaheuristics for solving combinatorial optimisation problems.
- Students should understand the parameter setting issues and the impact that poor parameter choices have on the performance of such metaheuristics.
- Students should be able to perform statistical tests to compare the performance of different methods and configurations given a null hypothesis.

Objectives

The purpose of this lab exercise is to gain experience in implementing some single point-based (local search) metaheuristics that were covered in the lecture "Metaheuristics". As a stretch goal, there is an optional exercise to implement one of the acceptance strategies covered in the lecture "Move Acceptance" as a component of a single point-based stochastic local search metaheuristic.

This lab exercise, in addition to importing the files, is split into four sections with the last being optional for eager students. Note that completion of the optional tasks may take more than the four hours allocated.

- COMP2001 framework background including backup solutions. **[REQUIRED].**
- Implementation and parameter tuning of Iterated Local Search (ILS). **[REQUIRED].**
- Implementation and parameter tuning of Simulated Annealing (SA) using two different cooling schedules – Geometric Cooling and Lundy and Mees's Cooling. **[REQUIRED].**
- Implementation and parameter investigation of Late Acceptance (LA). **[OPTIONAL].**

Note - The implementation of ILS has parameterised local search and mutation heuristics and these heuristics are re-used from completion of previous lab exercises; ensure that you have completed lab exercises 0 and 1 prior to this exercise.

Task 0 – Importing lab exercise files

Download the source code for this lab exercise from the Moodle page. You should find various files related to the metaheuristics, the exercise runner configurations, and the exercise runners themselves. Drag the **metaheuristics** folder into your IDE so that it is a direct subfolder of "com.aim". Notice that each metaheuristic is in its own package and these packages are in a parent "singlepoint" package; the focus of this exercise is on single point-based metaheuristics.

There are three exercise runners as part of lab exercise 2: 2a corresponding to ILS, 2b corresponding to SA, and (optionally) 2c corresponding to LA. Place the "Exercise2XTestFrameConfig" and "Exercise2XRunner" classes within the "runner" package alongside the ones from exercise 0 and 1.

Task 1 – Framework Background

For a single point-based search method, the COMP2001 Framework maintains two solutions in memory:

- The solution that we are operating upon (the current solution), and
- A backup of the solution that we started the current iteration with (the backup solution) so that we can reuse it later if we "reject" any modification(s) made to the current solution.

When a `SATHeuristic` is invoked on a problem, it modifies the solution in the current solution index (`CURRENT_SOLUTION_INDEX`) leaving the candidate solution (s') in the `CURRENT_SOLUTION_INDEX` and does not modify the backup solution in the `BACKUP_SOLUTION_INDEX` within the intermediate solution memory state. The move should then either be accepted ($s_{i+1} \leftarrow s_i'$) or rejected ($s_{i+1} \leftarrow s_i$) using the respective move acceptance criterion.

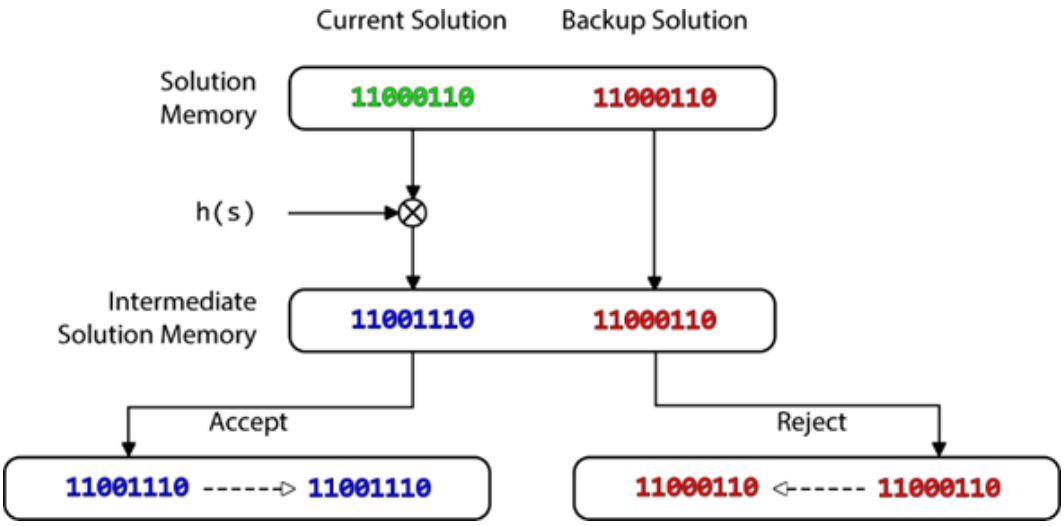


Figure 1 - Single Point-based Search in the COMP2001 Framework. The dashed arrow represents a solution being copied between solution memory indices using the copySolution(int source_memory_index, int destination_memory_index) method of the SAT problem.

Question 1: In Java, deep copying of objects is expensive as we need to create a copy of all classes, member variables/fields, and construct a new Object to contain these. The COMP2001 framework performs a deep copy of a solution when using the copySolution(int indexFrom, int indexTo) API method. Q1a) How might we reduce the runtime of an algorithm that uses a bit flip neighbourhood operator? Q1b) How might we reduce the runtime of an algorithm that uses other neighbourhood operators where we cannot know exactly what modifications were made to the incumbent solution?

Task 2 – Iterated Local Search [REQUIRED]

Implementation

Iterated Local Search (ILS) was covered in the lecture on metaheuristics. Below is the pseudocode from the lecture, adapted to embed intensity of mutation (IOM) and depth of search (DOS) parameters. **Remember that the test framework handles solution initialisation and the outer loop of the search method**; hence, only the lines within the REPEAT-UNTIL block need to be implemented in the `runMainLoop()` method.

You will need to think about what the line `s <- s'` means, and how the framework is designed in order to correctly implement this step (HINT: the order of instructions in the pseudocode and when implemented in the COMP2001 framework is not the same).

For the acceptance criterion, you can use either; accept only improving moves, or accept improving or equal moves. We suggest that you think about how ILS works and think about whether strict acceptance is necessary when deciding which criterion to use.

Below is the pseudocode for Iterated Local Search embedding the parameters IOM and DOS:

```
s_0 = generatedInitialSolution()        // handled by the lab exercise runner.
s = localSearch(s_0)                    // optional step, not used.
REPEAT                                  // handled by the lab exercise runner.
    s' <- s

    REPEAT iom TIMES:                   // apply iom times mutation.
        s' <- mutate(s')

    REPEAT dos TIMES:                   // apply dos times local search.
        s' <- localSearch(s')

    s <- acceptance(s, s', memory)      // decide which solution to keep as the
                                        // incumbent solution.
UNTIL( termination conditions are met ) // outer loop handled by framework.
return s*                               // handled by framework.
```

IMPORTANT: Note that there are SATHeuristic objects initialised as fields in the IteratedLocalSearch class; these contain the mutation and local search low-level heuristics and reuse the implementations of `RandomBitFlipHeuristic` from lab exercise 0, and `DavissBitHC` from lab exercise 1. Ensure that these implementations are completed (and correct; ask us in the labs!) before comparing your results with the expected outputs. Additionally, ensure that DBHC is implemented to accept bit flips that are strictly improving only.

Experimentation

Within the test frame configuration 2a, there are two parameters that you should change:

1. Depth of search (`iDepthOfSearch`)
2. Intensity of mutation (`iIntensityOfMutation`)

You can change these settings by modifying them within the `Lab2aTestFrameConfig` Class. You should investigate the effects of changing these settings where each setting should be from $\{0,1,2,3\}$. As the same as the previous lab, you can save any plots produced by your experiments for easier comparison across different runs.

Question 2

Perform some experimentation to find a parameter configuration for Iterated Local Search (ILS) that outperforms a default configuration of (depth_of_search = 0; intensity_of_mutation = 1). Report your configuration in the Moodle forum and discuss why you think your configuration performed better than the default. Alternatively, come to chat with us in the lab to check your understanding.

Question 3

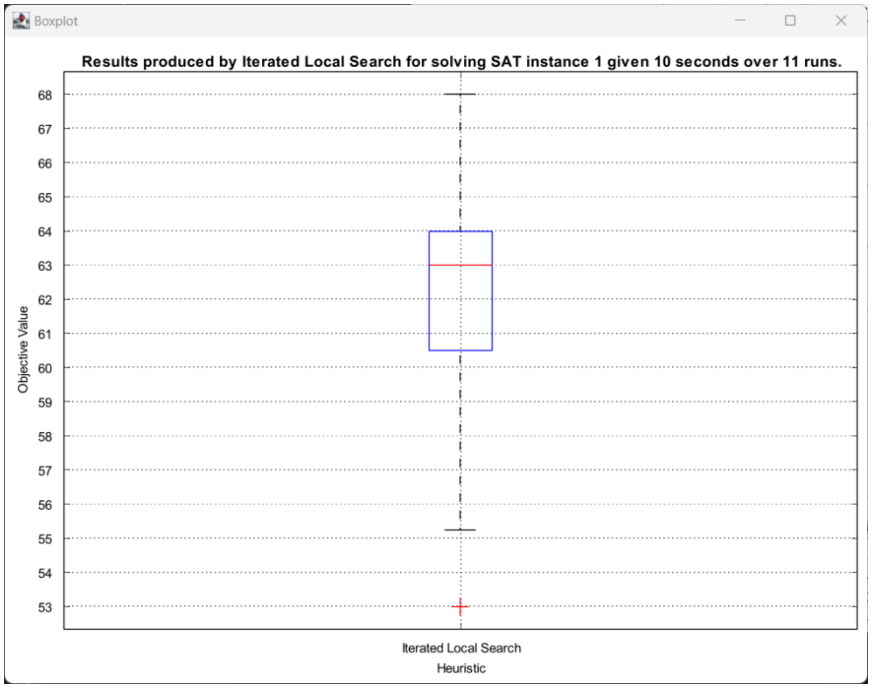
Update your implementation of DBHC from lab 1 to accept **non-worsening** moves. Perform some experimentation to find the best configuration for intensity of mutation and depth of search. Report your configuration in the Moodle forum and critically evaluate from the perspective of parameter tuning how you found the best parameter configuration. How might this scale with hundreds or thousands of configurations?

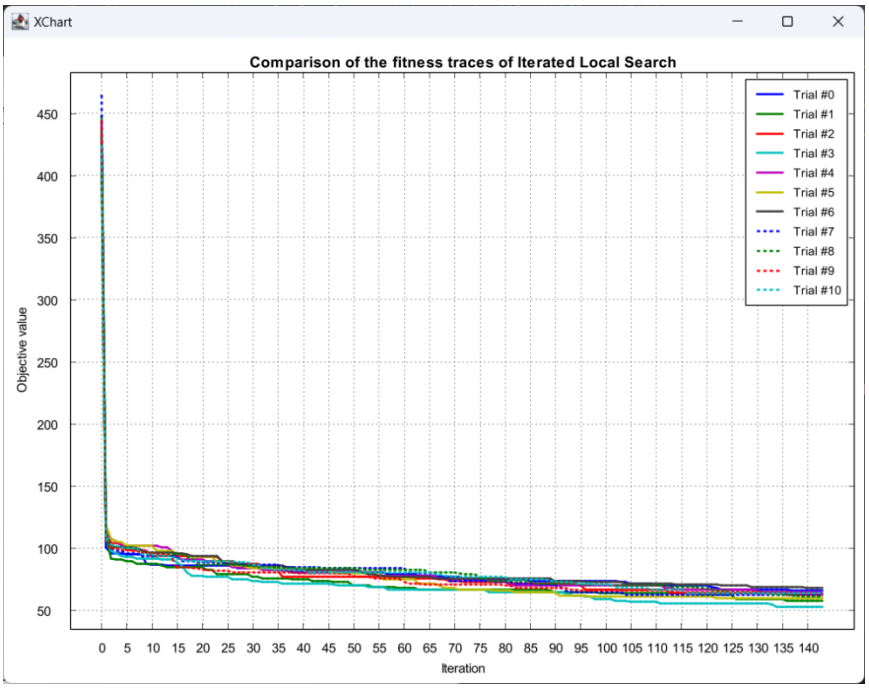
Question 4

In the Moodle forums, or in the labs, discuss why you think accepting "non-worsening" or "improving or equal" cost moves for the acceptance of moves in ILS would result in better performance. You should think critically about how ILS operates and the search space when using each of the acceptance criteria.

Expected output

When setting `intensity of mutation` and `depth of search` both equal to 2, you should get the following graphs as the output for a correct implementation of ILS (assuming a correct implementation of DBHC accepting only improving bit flips). **Please discuss with us in the labs if you did not get this output so that we can check your implementation and give feedback.**





Task 3 – Simulated Annealing [REQUIRED]

Implementation

There are three Classes in the package …metaheuristics.singlepoint.simulatedannealing: `SimulatedAnnealing`, `GeometricCooling`, and `LundyAndMeesCooling`. In each of these classes, you will need to complete the implementation of Simulated Annealing, and the two respective cooling schedules. Remember that your implementation should perform a **single iteration** of the respective heuristic method, hence no `while(terminationCriterionNotMet)` loop is required. Running of your solutions is handled by the `Exercise2bRunner` Class that is provided for you and the experimental

parameters can be configured in `Exercise2bTestFrameConfig`. Note also that `Exercise2bRunner` executes the experiments in parallel to reduce the time you need to wait (which is ok to do given an evaluation-based termination criterion such as that used within this framework). Do not be concerned that experiments complete quicker than expected. Your computer may slow down during the experiments due to the high CPU usage and parallelisation. This is normal and is nothing to worry about. Depending on your exact implementation and reflection to the answer of question 1, these experiments could finish fairly quickly or could take much longer. Discuss your solution if you think it is taking too long.

Simulated Annealing (SA) was covered in the lecture on *Move Acceptance in Local Search Metaheuristics and Parameter Setting Issues*. Below is the pseudocode from the lecture. Note that updating of the cooling schedule is purposely abstract. This allows us to use different cooling schedules without having to reimplement the main body of SA. Remember that the test framework handles solution initialisation and the outer loop of the search method. Hence only lines 7 to 14 need to be implemented in the `runMainLoop()` method. For the acceptance criterion, you **can use either accept only improving moves, or accept improving or equal moves however unlike previous algorithms, this will not make a difference for Simulated Annealing using the Boltzmann distribution equation** ($P(\Delta,Temp) = e^{-\Delta/Temp}$).

```
INPUTS: { T_0, "any other parameters of the cooling schedule" }
s_0 = generateInitialSolution()
Temp ← T_0
s* ← s_0
s ← s_0
REPEAT
    s' ← perturb(s)
    Δ = f(s') - f(s)
    r ← random ∈ [0,1)
    IF Δ < 0 || r < P(Δ,Temp) THEN
        s ← s'
    FI

    s* ← updateBest()
    Temp ← updateTemperature()
UNTIL ( termination conditions are met )
return s*
```

You should re-implement the random bit flip heuristic move operator within the implementation of Simulated Annealing. This has the advantage that you know which bit has been flipped in the solution. Remember that at the end of each main loop, in the case of the pseudocode that is after line 15, the solutions in the CURRENT and BACKUP solution indices must be the same in accordance with the COMP2001 Framework specification. (See Figure 1). It is up to you *how* this is done, but efficient implementations will result in faster executions!

IMPORTANT: When generating a random number, whilst there are multiple methods to generate random numbers in the range $[0,1]$, please **only use the nextDouble() method of the random**

number generator. Usage of any other method is likely to result in an output different to the expected output (which is fine, but you cannot then verify your implementations likely correctness).

Geometric Cooling

You should implement the `advanceTemperature()` method to set the '*currentTemperature*' by applying the Geometric Cooling as shown in the below pseudocode. **Make sure that you also set the value of α "appropriately" from within the constructor. You may want to run some experiments once implemented to find a reasonable setting.**

Lundy and Mees's Cooling

You should implement the `advanceTemperature()` method to set the '*currentTemperature*' by applying the Lundy and Mees's cooling as shown in the below pseudocode. **Make sure that you also set the value of β "appropriately" from within the constructor. You may want to run some experiments once implemented to find a reasonable setting.**

Experimentation

At a minimum, you should complete the below three exercises and discuss your findings in the Moodle discussion forum. Please use a sensible name for the topic so that everyone knows which exercise you are referring to. I.e. "Lab#2: Exercise#5" and so on.

For each exercise, if applicable, you should save both the boxplot and progress plots as these will help you to discuss your observation(s) in the Moodle discussion forum!

Question 5

Update the Exercise2btestFrameConfig to use the geometric cooling schedule. Find the best configuration for α and report your results (boxplot and progress plot) into the Moodle discussion forum. How does your value of α compare to that suggested in the lecture and how does Simulated Annealing appear to behave when looking at your progress plot?

Question 6

Update the Exercise2btestFrameConfig to use the Lundy and Mees cooling schedule. Find the best configuration for β and report your results (boxplot and progress plot) into the Moodle discussion forum. How does your value of β compare to that suggested in the lecture and how does Simulated Annealing appear to behave when looking at your progress plot?

Question 7

Compare your best variant of Simulated Annealing to your results obtained when using Iterated Local Search from the previous lab. Why do you think algorithm A performed better than algorithm B? Do you think that the better performing algorithm is guaranteed to perform better for all MAX-SAT problem instances and why?

Expected Output

The parameter configurations (α and β) for Geometric Cooling and Lundy and Mees's Cooling are initially set to 0.5 and the expected plots for these are given below.

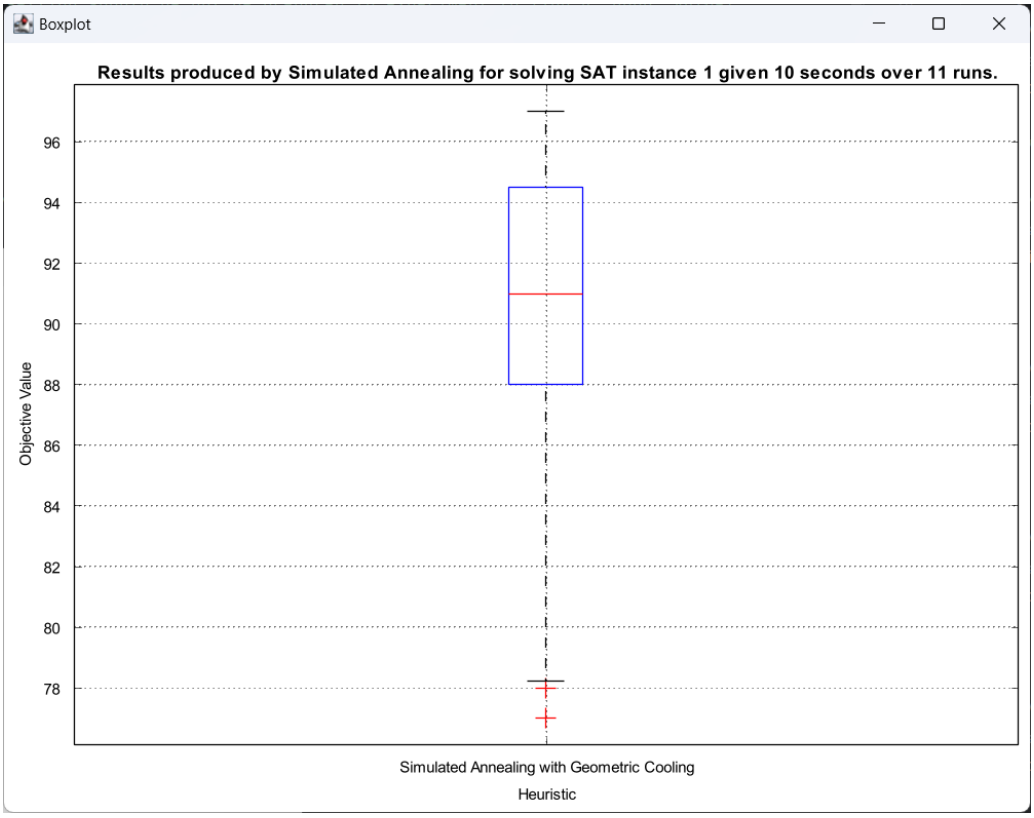


Figure 2 - Boxplot results for SA using Geometric Cooling with alpha = 0.5.

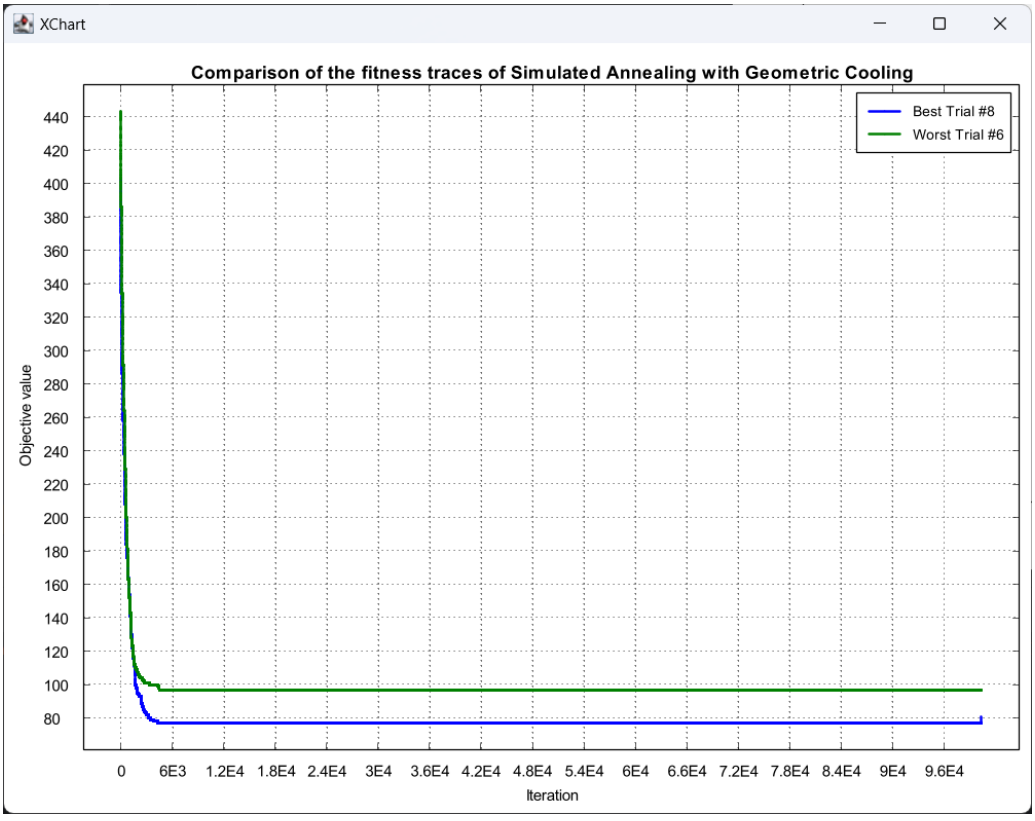


Figure 3 - Progress plot for SA using Geometric Cooling with alpha = 0.5.

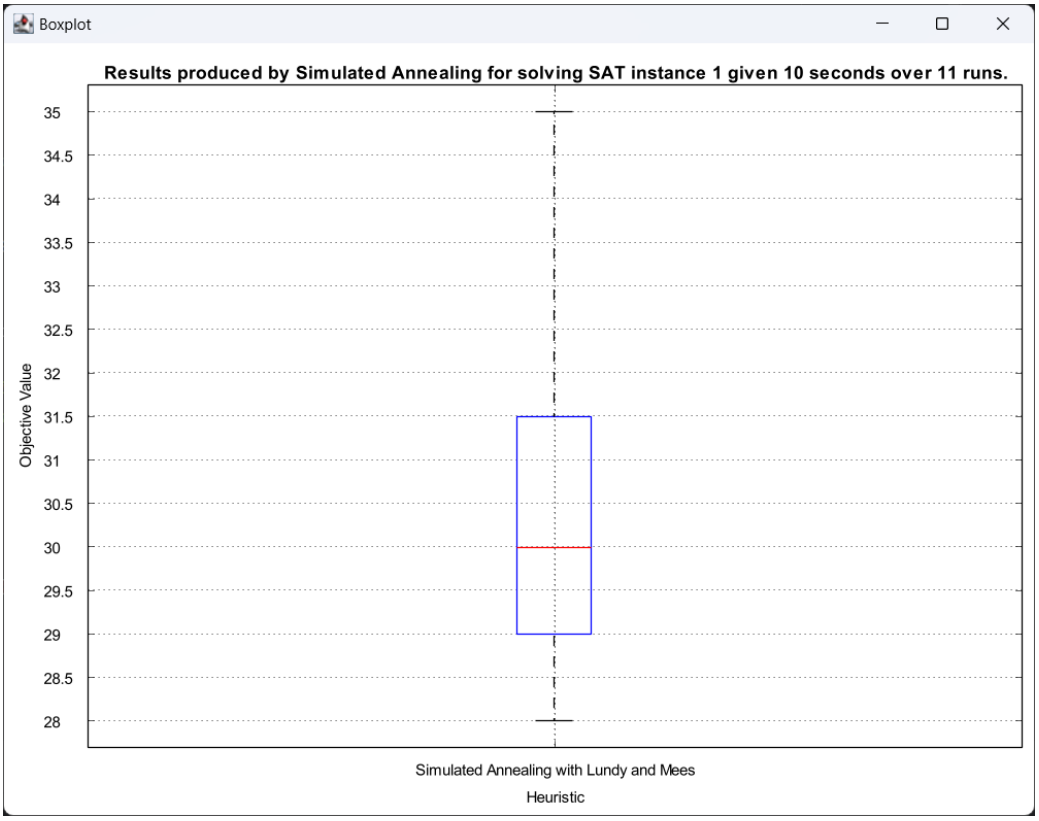


Figure 4 - Boxplot results for SA using Lundy and Mees's Cooling with beta = 0.5.

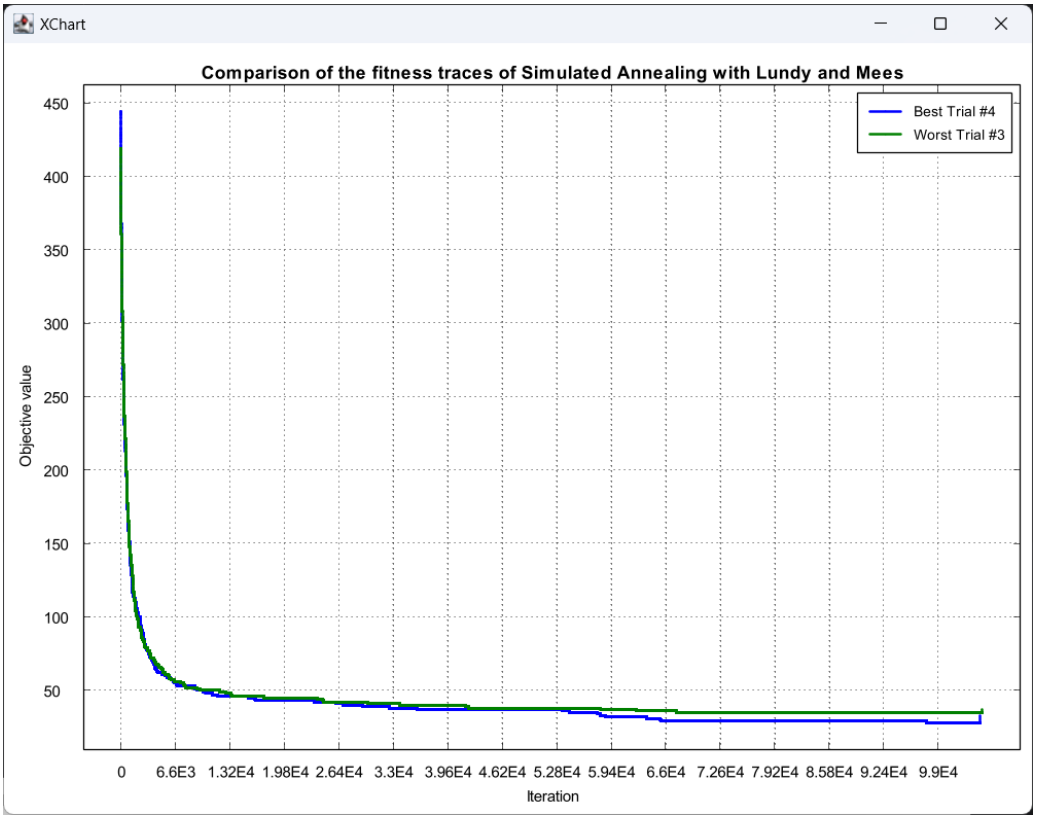


Figure 5 - Progress plot for SA using Lundy and Mees's Cooling with beta = 0.5.

Task 4 – Late Acceptance [OPTIONAL]

Note that this task is optional but may be interesting if you are eager to go further. Late Acceptance is/was covered in lecture 4 "Move Acceptance".

Implementation

A template is provided for you in …`metaheuristics.singlepoint.LateAcceptance.java`. Your task is to implement Late Acceptance (LA) with a list of a length specified by the `length` parameter in the constructor.

A description of LA is given as the header to the apply heuristic method and a more detailed description is/will be given in lecture 4. You should implement LA as a stochastic single point-based local search metaheuristic. That is, that it embeds a randomised perturbation heuristic for exploring the neighbourhood of solutions, and each incumbent solution is accepted or rejected in accordance with LA.

Experimentation

Your task in this exercise is to perform some form of parameter tuning methodology for the list length of LA to try to outperform ILS or SA using the better configuration that you obtained in tasks 3 and 4.

Question 8

By default, the list length is set to $L = 1$. What move acceptance is LA equivalent to with this configuration?

Question 9

Run a series of experiments on a MAX-SAT instance of your choice with at least three different list lengths. After each experiment you will want to save the results of each trial by copying the output from the console and saving them in an appropriately labelled dataset for analysis. After which, you should perform some statistical comparison and report your best list length along with any statistical measures that evidence your hypothesis.

Implementation Hints / Q&A

Where do I find the mutation and local search heuristics?

> Both mutation and local search heuristics are created within `Exercise2xRunner` and are passed into the constructor of the `IteratedLocalSearch` Class. They are then stored as member variables named `oMutationHeuristic` and `oLocalSearchHeuristic` respectively.

How do I use different/multiple heuristics within a single search method?

A heuristic h is applied to a solution by calling `h.applyHeuristic(problem)`. Remember `SATHeuristics` within this framework modify the solution in the `CURRENT_SOLUTION_INDEX` **only**. Hence, multiple `applyHeuristic` calls can be made one after another with each heuristic receiving the candidate solution generated by the previous heuristic.

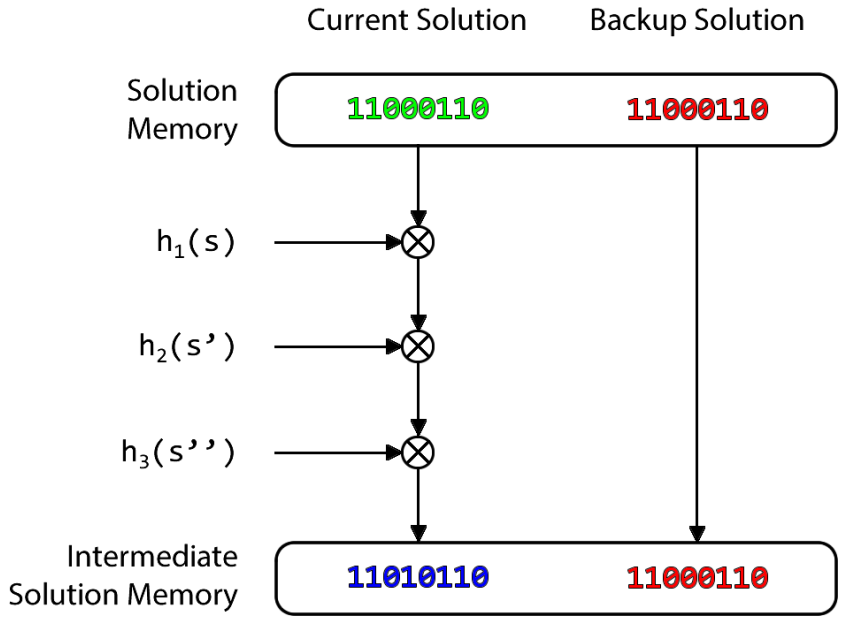


Figure 6 - Illustration of chaining multiple heuristics in the COMP2001 Framework.

How do I: perform a bit flip; get the number of variables; evaluate the solution; etc.

See the COMP2001 Framework API for framework specific questions on Moodle!

How do I use Euler's number in Java?

Java's Math API includes a constant `E` which is Euler's number. The Math API actually contains a function `exp(double a)` which allows you to calculate e to the power of `a`.

E.g. `Math.exp(0) = 1`.