COMP2001: Artificial Intelligence Methods – Lab Exercise 3

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 26th February 2024. The contents of this exercise will be/was covered in **lectures 5 and 6** on Evolutionary Algorithms (part I and part II respectively).

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 (multi point/population based) metaheuristics for solving combinatorial optimisation problems.
- Students should understand and be able to implement components of evolutionary algorithms including crossover, mutation, local search, parent selection, and population replacement.
- Students should be able to critically evaluate the behaviours of different evolutionary algorithms and adapt their settings to achieve desirable performance.

Objectives

The purpose of this lab exercise is to gain experience in implementing some population-based metaheuristics that were covered in lectures 5 and 6. This lab exercise, in addition to importing the files, is split into six sections with the tasks 3 and 6 being optional for eager students. Note that completion of the optional tasks may take more than the four hours allocated.

- COMP2001 framework background for populations of solutions. **[REQUIRED].**
- Implementation of a memetic algorithm and local search investigation. **[REQUIRED].**
- Implementation and evaluation of more sophisticated components of memetic algorithms. **[OPTIONAL].**
- COMP2001 framework background for memes and memeplexes. **[REQUIRED].**
- Implementation of a multi-meme memetic algorithm with investigation of allele frequencies. **[REQUIRED].**
- Extension of MMA with k-DBHC with co-evolved ‘k’. **[OPTIONAL].**

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 **population** folder into your IDE so that it is a direct subfolder of "com.aim.metaheuristics".

There are two exercise runners as part of lab exercise 3: 3a corresponding to Memetic Algorithms, and 3b corresponding to Multi-meme Memetic Algorithms. Place the "Exercise3XTestFrameConfig" and "Exercise3XRunner" classes within the "runner" package alongside the ones from exercises 0-2.

Task 1 – Framework Background for Populations [REQUIRED]

A memetic algorithm is one example of a population-based search method. The COMP2001 Framework allows multiple solutions to be stored in memory, the difference compared to single point methods is that we have 16 memory addresses to manage for a population size of 8. 8 addresses for the current population of solutions, and another 8 for the offspring population. Typically, we do not require backup solutions hence the 8 + 8 memory indices are sufficient. Special methods are needed for applying heuristics to the correct solutions in memory and these are given in the **implementation hints** section.

Below is an illustration of a population-based search method using the COMP2001 Framework. Notice that at the start of each iteration (generation), the current population should be stored in the first POPULATION_SIZE indices of the solution memory. The second POPULATION_SIZE indices should be used for creating and mutating/improving the offspring. At the end of each iteration, POPULATION_SIZE solutions should be selected and moved to the first POPULATION_SIZE indices according to the replacement scheme.

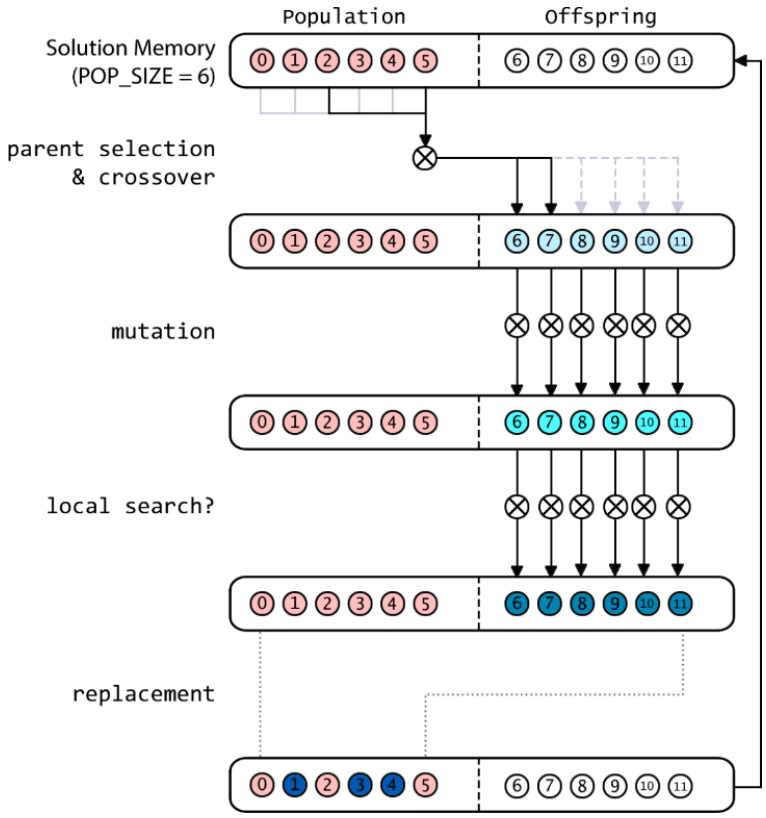


*Figure 1 – Population based Search illustrating a Memetic Algorithm in the COMP2001 Framework. Note that both the current population and the offspring share the same contiguous memory with indices [0..POPULATION_SIZE-1] storing the current population, and indices [0..POPULATION_SIZE-1] storing the current population, and indices [POPULATION_SIZE..POPULATION_SIZE*2-1] storing the offspring (children).*

Rather than having number of evaluations as the termination criterion, it is common for the termination criterion of population-based methods to be defined as the maximum number of generations. For the Memetic Algorithm, we therefore use number of generations as the termination criterion rather than a "time limit"/evaluation count as we did in all previous lab exercises.

Task 2 – Memetic Algorithms [REQUIRED]

Implementation

Memetic Algorithms (MA's) were covered in lecture 5 "Evolutionary Algorithms". Below is the pseudocode for a **Genetic Algorithm**. The difference between GA's and MA's is the inclusion of a local search step. It is entirely up to you to decide which place (or places) to apply local search.

Like the previous labs, you only need to implement the main loop of the GA (lines 3-7). The outermost loop and initialisation are handled by `Exercise3aRunner.java`.

Memetic Algorithm

Below is the pseudocode for a Genetic Algorithm. You should implement this and include a local search step to form a Memetic Algorithm, but it is up to you to decide where this should be. You can run a series of experiments to determine the best location.

```
s[0-N) = generateInitialPopulation()    // handled by the lab exercise runner.
REPEAT max_generation TIMES:            // handled by the lab exercise runner.
       REPEAT population_size / 2 TIMES:
              (p1, p2) <- Select two unique parents
              (c1, c2) <- crossover(p1, p2)
              Apply mutation to children (c1, c2)
       DO population_replacement
return s*                               // handled by the lab exercise runner.
```

Random Selection

In the COMP2001 framework, the current population (which contains the parent solutions) are stored in memory indices 0 through to the population size (exclusive). A parent can be selected an identified by specifying the memory index in which it is stored.

In random selection, we arbitrarily select a random parent from the current population:

```
INPUT: CURRENT_POPULATION
p <- random ∈ CURRENT_POPULATION
return p
```

Fittest Selection

In the COMP2001 framework, the current population (which contains the parent solutions) are stored in memory indices 0 through to the population size (exclusive). A parent can be selected an identified by specifying the memory index in which it is stored.

In fittest selection, we select the parent which has the best objective value:

```
INPUT: CURRENT_POPULATION
p_best <- min(CURRENT_POPULATION)
return p_best
```

Uniform Crossover

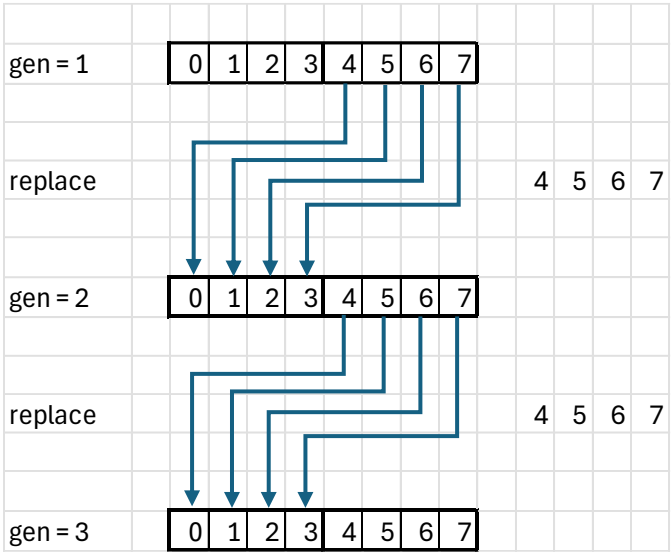
Crossover operators are generally used to create offspring from two parent solutions. There are POPULATION_SIZE parent solutions in the current population, and space for POPULATION_SIZE offspring solutions in the next POPULATION_SIZE memory indices (see Figure 1).

In UXO, the idea is to copy the two parents (p1, p2) into two offspring indices (c1,c2) whereby each bit in the bitstring is exchanged between the offspring with a probability of 50%.

```
INPUTS: p1, p2, c1, c2
memory[c1] = copyOf(p1)
memory[c2] = copyOf(p2)
FOR j ∈ [0, chromosome_length]:
      IF random ∈ [0,1] < 0.5 THEN
            exchange(c1,c2, j)
```

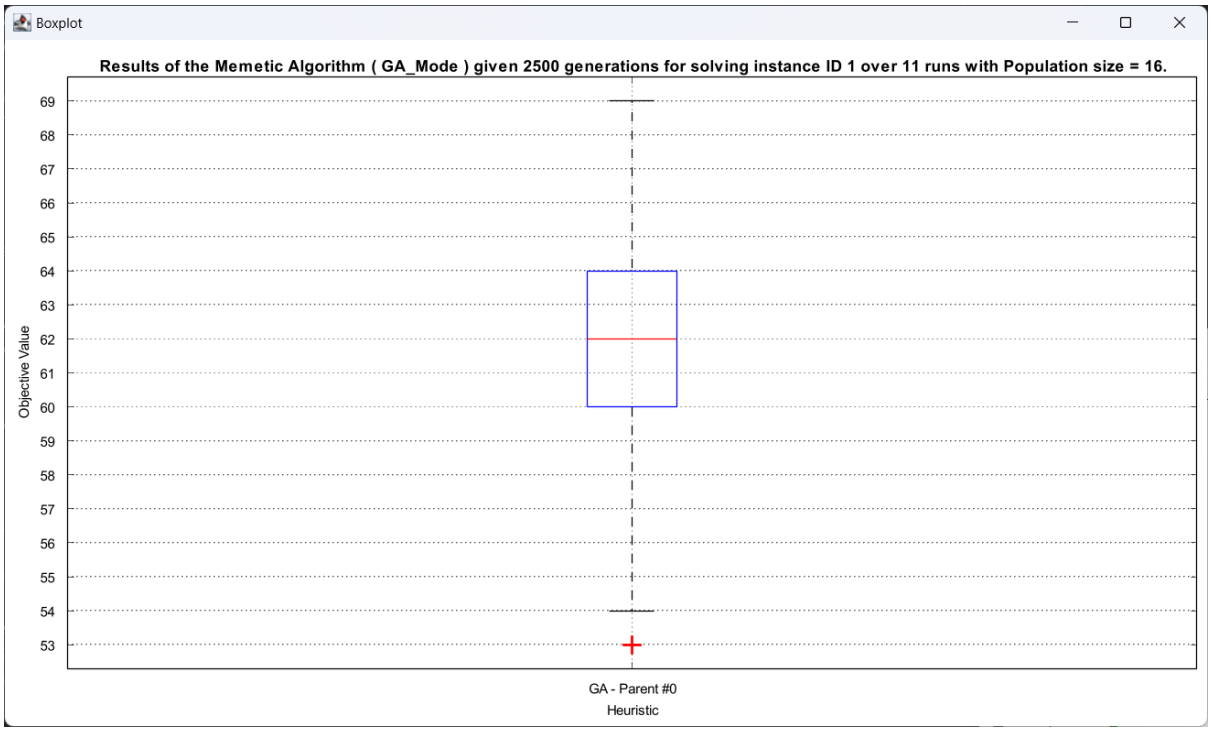
Basic Replacement

In the COMP2001 framework between each generation, the current population is replaced by a subset of the union of the current population and offspring populations by specifying the indices of the solutions in memory that we wish to carry forward. In a basic replacement strategy, we wish to set the current population in the next generation as the offspring population in the current generation. For example, if the population size was 4 and we wanted to carry forward all offspring solutions, then we would preserve $s \in [4,5,6,7]$ by specifying an integer array containing 4, 5, 6, and 7.




```
No pseudocode given. This should be trivial from the definition.
```

Expected output (GA Mode – Population Size = 16, P1 selection as random, P2 selection as fittest)



Experimentation

At a minimum, you should complete the below three exercises and discuss your findings in the Moodle discussion forum.

Question 1

A Memetic Algorithm is an extension of a Genetic Algorithm with the inclusion of a local search step; you should compare the performance of your implementation using GA mode and MA mode. Assuming that the number of iterations is equivalent in terms of nominal runtime, discuss on the Moodle forum why do you think the GA/MA performed better than the other?

Question 2

A Memetic Algorithm is an extension of a Genetic Algorithm with the inclusion of a local search step. You should experiment with placing the local search operator at different points within the memetic algorithm (applied once per offspring) and find the best placement for the local search step. Discuss on the Moodle forum why you think that the placement of the local search step that you found to be the best is where it is in the EA process.

Question 3

Experiment with increasing and decreasing the population size from the default setting of 8 and the number of generations. What is the effect on the results? For example, try the following:

- Set the population size = 4 and double the run time (set to GA(5000))
- Set the population size = 16 and half the run time (set to GA(1250))

Task 3 – Advanced Methods for MAs [OPTIONAL]

Implementation

The components in task 2 were deliberately trivial to implement with the focus being on learning how the framework can handle each of selection, crossover, and replacement. The strategies implemented are not necessarily performant and in task 3 we will explore some more effective (but more complex to implement) strategies for selection and replacement.

Tournament Selection

In tournament selection, a parameter, `tournament_size`, is used to determine the size of a subset of parent solutions that are to be selected at random. The best solution is then chosen from this subset of parent solutions and its index returned for the selection procedure. If the best solution is tied with at least one other solution, then a solution is chosen randomly from the subset of best solutions from the subset of tournament selected solutions.

For example, given the following four parent solutions and a tournament size of two, selecting at random solutions [0] and [2], the chosen solution would be [2] since the objective value of 88 is the best in the subset { 88, 100 }.

0(100)	1(89)	2(88)	3(91)
		^	

Transgenerational Replacement with Elitism

This replacement scheme is very similar to the basic replacement implemented in task 2 but with one key difference. If the best solution in `CURRENT_POPULATION` "union" `OFFSPRING_POPULATION` exists in the `CURRENT_POPULATION`, then the set of solutions to be carried over to the next generation should be equal to the `OFFSPRING_POPULATION` less the worst solution in that population and including the best solution.

```
1 | INPUT: current_pop, offspring_pop
2 | fitnesses <- evaluate( current_pop ∪ offspring_pop );
3 | best <- min(fitnesses);
4 | next_pop <- indicesOf( offspring_pop );
5 | IF best ∉ offspring_pop THEN
6 |     next_pop.replace( worst, best );
7 | ENDIF
8 | OUTPUT: next_pop; // return the indices of the next population
```

Below is an illustration where the best solution resides first in the offspring population, and then in the current population demonstrating which solutions are carried over into the next generation.

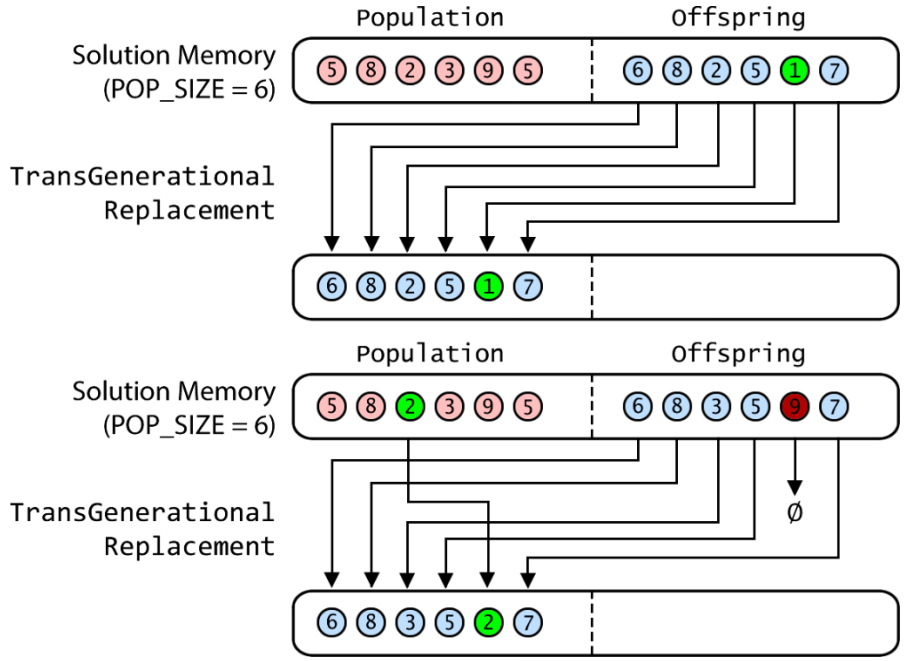
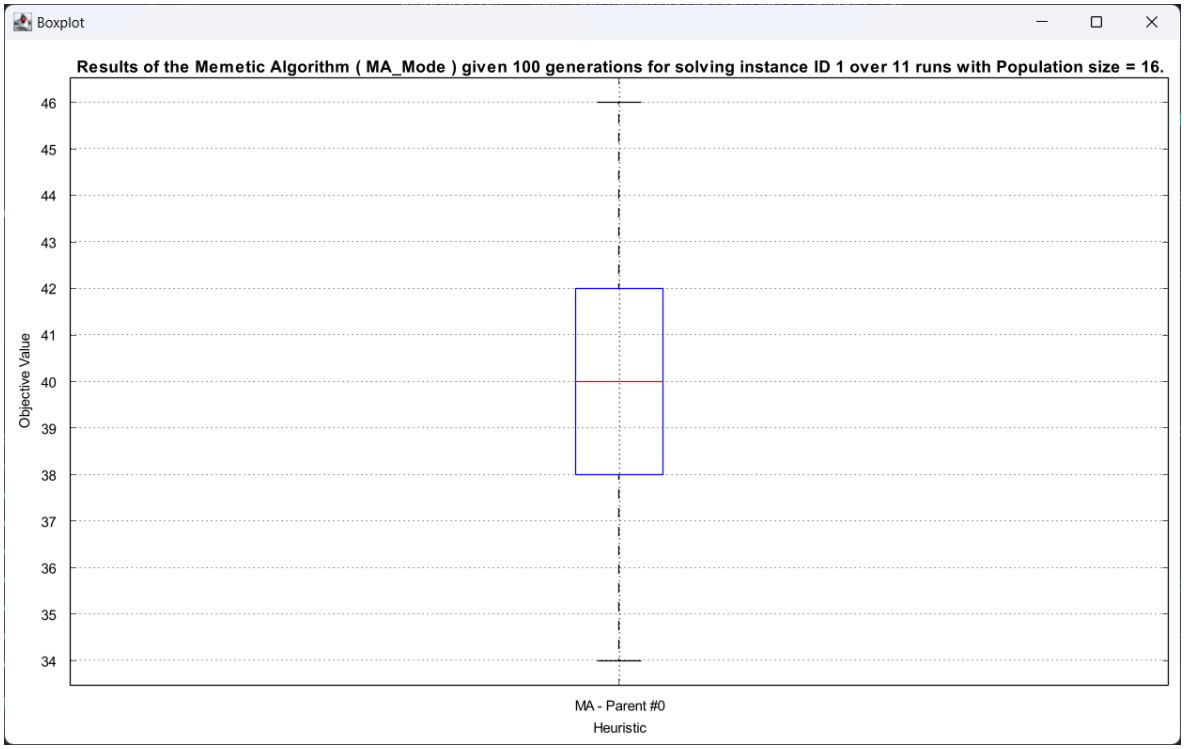


Figure 2 - Illustration of Trans-Generational Replacement with elitist best replacement. The circles represent solutions in memory and the numbers contained in them represent their objective values. TGReplacement will return [6,7,8,9,10,11] in the first example and [6,7,8,9,2,11] in the second.

Expected Output

Using tournament selection with a tournament size of 3 for selection of both parent 1 and parent 2, uniform crossover, local search immediately after crossover (enable MA mode), using trans-generational replacement with elitism as the population replacement method, and setting a population size of 16, yields the following output.

Note that you will need to **modify the Exercise3aTestFrameConfig class** to ensure that the correct mechanisms are being used.



Suggested Experimentation

Feel free to experiment with the different parameters and mechanisms used within the Memetic Algorithm. We suggest that you try some of the following and post on the Moodle forum any additional experiments that you have performed that have interesting outcomes.

Question 4

By default, the tournament size is set as a constant equal to 3. Can you find either a constant value for the tournament size or a variable tournament size that depends on the population size that outperforms the default configuration?

Question 5 [Recommended only for the ultra-curious]

In the trans-generational replacement with elitism mechanism, the best solution replaces the worst in the offspring population iff the offspring does not contain the best solution from the union of the current and offspring populations. Can you implement a trans-generational replacement with k-best elitism mechanism that replaces the k-best solutions in the offspring population and does this outperform that from question 4?

Note you will need to create a new class and edit the `Exercise3aRunner` and `Exercise3aTestFrameConfig` classes to enable performing experiments with this alternative selection mechanism.

== End of content from lecture 5 (see final page for implementation hints) ==

Task 4 – Additional Background for Multimeme Memetic Algorithms [REQUIRED]

A multi-meme memetic algorithm extends the standard memetic algorithm by introducing memeplexes to each solution in the population. Within the framework, each memeplex is referenced within each solution as illustrated below with each memeplex containing a set of memes. These memes are stored in an array and can be referenced by their meme index. In the below example meme m_1 is in meme index 0, and meme m_2 is in meme index 1.

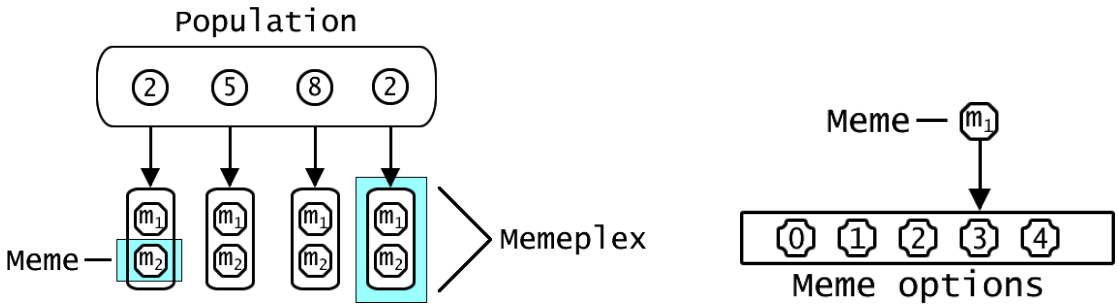


Figure 3 - Representation of the solution memory with memes and memeplexes (left) and association of memes with meme options (right).

Each meme points to a specific setting/option which symbolises its genetic code. Each individual meme has a specific number of different options, which in this case is 5. These are integer values starting at 0 and ending at the number of options **exclusive**. You may want to create a mapping of these integer values to each option's characteristic, for example local search operator.

In a multi-meme memetic algorithm, an individual is associated with a memeplex containing a set of memes. The memeplex is intrinsic to each individual and can be accessed using the `getMeme(int solutionIndex, int memeIndex)` method on the `SAT` Object where `solutionIndex` is the memory index where the associated solution along with the associated meme are stored, and `memeIndex` is the index of the meme in the memeplex. In the COMP2001 Framework, a meme is encoded as an integer value as shown in the table below. In the example, the memeplex contains 3 memes. A meme option is the discrete integer value that a meme is assigned to. Memes are represented by a `Meme` Object. Within this object, you can do two things; get the current meme option and set the meme option. For example, if we get the meme from solution index 0 and meme index 2 ($M_{0,2}$), then `getMemeOption()` will return 5.

Solution Memory	Genetic representation	Memeplex
[0] (parent 1)	$s_i(P_1) = 1100101101$	meme[0] = 2
		meme [1] = 0
		meme [2] = 5
[1] (parent 2)	$s_i(P_2) = 1100101101$	meme [0] = 3
		meme [1] = 1
		meme [2] = 7
[2] (child 1)	$s_i(C_1) = 1100101101$	meme [0] = 1
		meme [1] = 0
		meme [2] = 3
[3] (child 2)	$s_i(C_2) = 1100101101$	meme [0] = 5
		meme [1] = 6
		meme [2] = 9

Task 5 – Multi-meme Memetic Algorithms [REQUIRED]

Implementation

You are given 2 Classes, `MultiMeme.java and SimpleInheritanceMethod.java`, which you will need to complete to implement the Multimeme Memetic Algorithm. The multimeme code reuses your implementations of crossover, mutation, local search, population replacement, and tournament selection from the previous tasks. If you did not complete the optional task 3, then you must use set the OPERATOR_MODE in the Exercise3bTestFrameConfig to BASIC, otherwise you should set it to ADVANCED to make use of the more advanced selection and replacement schemes.

Within `MutiMeme.java`, we have specified three methods which we ask that you implement and call from runMainLoop() to perform the respective operations. This has the advantage of making your code neater and easier to implement. These methods are as follows:

- `applyMutationForChildDependentOnMeme( int childIndex, int memeIndex );`
- `applyLocalSearchForChildDependentOnMeme( int childIndex, int memeIndex );`
- `performMutationOfMemeplex( int solutionIndex );`

You are asked to use the following configuration which is set automatically depending on how you have configured the OPERATOR_MODE:

Configuration	Did not do optional task 3	Completed optional task 3
Population size	16	
Parent selection	P1: Random P2: Fittest	P1, P2: Tournament selection with tournament size = 3
Crossover operator	1PTX (Implementation given)	
Replacement	Basic	Trans-generational replacement with elitism
Meme "0"	Meme in index 0 is for the intensity of mutation setting. Intensity of mutation ∈ {0,1,2,3,4}.	
Meme "1"	Meme index 1 is for the local search operator. Local search operator ∈ {$DBHC\_OI$,$DBHC\_IE$, $SDHC\_OI$, $SDHC\_IE$}.	

MultiMeme

Multi-meme Memetic Algorithms (MMA's) were covered in the lecture "Evolutionary Algorithms II". Below is the pseudocode for a Multimeme Memetic Algorithm embedding memeplexes for deciding which local search operator to apply and the intensity of mutation as described in section 2.1.2.

```
1 | INPUT: PopulationSize, MaxGenerations, InnovationRate
2 | generateInitialPopulation();
3 | FOR 0 -> MaxGenerations
4 |     FOR 0 -> PopulationSize / 2
5 |         select parents using tournament selection
6 |         apply crossover to generate offspring
7 |         inherit memeplex using simple inheritance method
8 |         mutate the memes within each memeplex of each child with
          ⏎       probability dependent on the innovation rate
9 |         apply mutation to offspring with intensity of mutation set for
          ⏎       each solution dependent on its meme option
```

```
10 |         apply local search to offspring with choice of operator
           ⏎       dependent on each solution's meme option
11 |     ENDFOR
12 |     do population replacement
13 | ENDFOR
14 | return s_best;
```

Code 1 - Pseudocode for the Multimeme Memetic Algorithm. Greyed-out pseudocode has already been implemented for you.

Mutation of each meme within each memeplex should be performed separately such that it is possible to mutate the meme in meme index 0 but not the meme in meme index 1 even though they are from the same memeplex.

Within the MMA that you are asked to implement, there will be two memes in each memeplex.

Meme 0

The first meme (meme index 0) will represent the intensity of mutation setting and has 5 possible options which are mapped to intensity of mutation settings as $iom \leftarrow option$. That is, the intensity of mutation setting equals the meme option.

The mutation operator is the same as BitMutation from task 2 but where the MUTATION_RATE was set as $1/chromosome\_length$, the rate of mutation depends on an intensity of mutation setting equal to $x/chromsome\_length$ where x is the integer intensity of mutation setting. There is a method `setMutationRate(int intensityOfMutation)` which should be called on the mutation heuristic to set the aforementioned parameter setting.

Meme 1

The second meme (meme index 1) will represent the local search operator to use. The option values should be mapped as:

- `map | 0 -> DBHC accepting only improving moves.`
- `    | 1 -> DBHC accepting non-worsening moves.`
- `    | 2 -> SDHC accepting only improving moves.`
- `    | 3 -> SDHC accepting non-worsening moves.`

Simple Inheritance Method

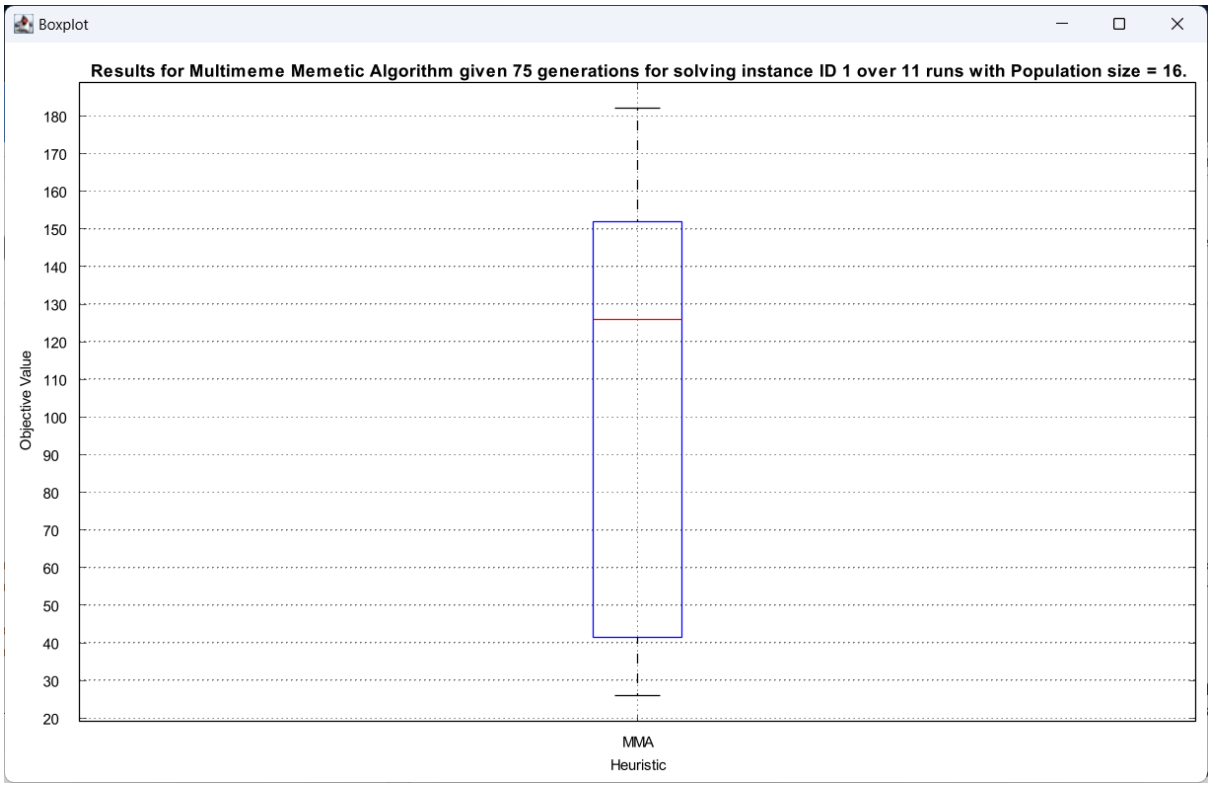
Within the simple inheritance method (SIM), all memes (the memeplex) of each offspring are inherited from the best parent where the best parent is the parent with the best objective value. In the event of a tie, the parent from which the memes are inherited is chosen randomly. Below is the **pseudocode** for SIM. Information relating to the actual methods can be found both in the API document and in the sections "Framework Background" and "Implementation Hints".

```
 1 | INPUT: parent1, parent2, child1, child2
 2 | IF f(parent1) == f(parent2) THEN
 3 |     inherit = random ∈ {parent1, parent2}
 4 |     child1.memeplex <- inherit.memeplex
 5 |     child2.memeplex <- inherit.memeplex
 6 | ELSEIF f(parent1) < f(parent2) THEN
 7 |     child1.memeplex <- parent1.memeplex
 8 |     child2.memeplex <- parent1.memeplex
 9 | ELSE
10 |     child1.memeplex <- parent2.memeplex
11 |     child2.memeplex <- parent2.memeplex
12 | ENDIF
13 | return;
```

Code 2 - Pseudocode for Simple Inheritance Method.

Expected Output

Below is the expected output for the MMA using the default configuration with the BASIC operator mode for solving MAX-SAT instance #1:



Experimentation

When the experiments are running, the output includes some information which tells us how many times each meme option (allele) was present in the parent population. You should run the experiments on all three problem instances (1, 7, and 9) by updating INSTANCE_ID in Exercise3bTestFrameConfig.java. For each, you should record the number of times each meme has been present in the population over all 11 trials (i.e. "MEME 0:" and below) before completing the following questions.

Question 6: What do the allele frequency values tell us over the different problem instances tested when using the simple inheritance method?

Question 7: Are the ratios of the allele frequency values as expected?

- If yes, why?
- If no, what in the test frame configuration could have caused the discrepancy? Update and re-run any experiments to test your theory is correct. Does updating this/these lead to an improvement in overall performance?

Task 6 – Extended Local Search Heuristic Set for MMAs [OPTIONAL]

As part of the optional exercises from lab exercise 1, you were asked to implement some variants of the local search heuristics; namely k-DBHC and Shallowest Descent. This task gets you to replace the standard DBHC used in task 5 with the parameterised k-DBHC. You should include at least three versions of k-DBHC in your MMA each having a different setting for 'k'. You may choose to implement k-DBHC as accepting only non-worsening moves, or accepting only improving moves, or implement two versions as given in task 5. Below is some guidance for how you can do that but since the optional exercises are designed to promote curiosity and autonomy, there will not be any exact instructions to follow, and as such there is no expected output. After completing any implementation and modification of any framework runners/configurations, you should re-run any experimentation performed as part of task 5 and draw conclusions as to if the parameterised version of k-DBHC is useful and if introducing more choice has meant the overall performance of the MMA has improved or worsened.

Implementation Guidance

For this task you will need to create a new class which extends PopulationHeuristic and provide a modified version of KBitDavissHC from lab exercise 1 that applies itself to a specified solution index.

You should not change any interfaces in the framework; hence it will be easier to define values for 'k' in the constructor and create multiple versions of the heuristic which can then be applied from within your MMA.

You may need to modify the Exercise3bTestFrameConfig to allow the framework to encode the correct number of options per meme depending on how many versions of k-DBHC you have created and encoded.

Question 8: Does the MMA using k-DBHC in place of DBHC perform better or worse (or neither conclusively) than the MMA from task 5? Why might this be the case?

> Prompt 1: Which values of 'k' did you use?
>
> Prompt 2: Which acceptance of bitflips did you use in k-DBHC and how does this compare to that in task 5?
>
> Prompt 3: Will adding more (or using less) versions of 'k' for k-DBHC improve the performance further? Why/Why not?

Implementation Hints / Q&A

How do I perform a bit flip on a specific solution in memory?

`problem.bitFlip(i, solutionIndex)` is used to flip the *i^th* bit of the solution in memory index *solutionIndex*.

How do I swap/exchange the values of a specific bit of two solutions?

`problem.exchangeBits(a, b, i)` is used to exchange the *i^th* bit between solutions in memory indices *a* and *b*.

How do I apply the various heuristics within the MA?

Mutation and local search operators are both `PopulationHeuristic's` and are invoked by calling `the applyHeuristic(int index)` method where index is the index in solution memory of the solution to apply the heuristic to.

Crossover operators are invoked by calling the `applyHeuristic(p1, p2, c1, c2)` method where `p1` is the memory index of the first parent, `p2` is the memory index of the second parent, `c1` is the memory index of the first child, and `c2` is the memory index of the second child.

The replacement scheme is performed by invoking the `doReplacement(problem, POP_SIZE)` method.

How do I run the algorithm in GA/MA mode?

There is a configuration in the `Exercise3aTestFrameConfig` class which allows you to run your algorithm in MA mode or GA mode. The framework replaces the hill climbing heuristic with a "NOOP" when in GA mode.

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!