ASSESSED

G52AIM Lab 4 – Local Search in Memetic Algorithms

This is the fourth assessed lab session and will account for 1/5th (including the report) of the module's total coursework mark (5% of the entire module).

1 Objectives

1.1 Lab Exercise

Implement a Memetic Algorithm including:

- The Memetic Algorithm itself,
- Tournament Selection for parent selection,
- Uniform Crossover for the crossover step,
- Bit Mutation for the mutation step, and
- Transgenerational Replacement with Elitism for the replacement scheme.

Note that DBHC used for the local search step has been updated to work across multiple solution memory indices and is supplied in a JAR file.

1.2 Report

- To be announced at the start of the lab.

2 Implementation [50 marks]

2.1 Tasks

You are given 5 Classes, `MemeticAlgorithm`, `UniformCrossover`, `BitMutation`, `TransGenerationalReplacementWithElitistReplacement`, and `TournamentSelection`, which you will need to complete to implement the Memetic Algorithm.

Remember your implementation should perform a single iteration/generation of the respective heuristic method, hence no `while(terminationCriterionNotMet)` loop is required. Running of your solutions is handled by the `Lab_04_Runner` Class that is provided for you and the experimental parameters can be configured in `Lab04TestFrameConfig`.

The configuration which you are asked to use is as follows:

- Population size = 8
- Mutation rate = 1 / chromosome_length
- Crossover rate = 1.0;
- Parent selection = tournament selection with tournament size = 3 for both parents.

- Local search operator = DBHC accepting IE moves.
- Crossover operator = uniform crossover.
- Replacement scheme = trans-generational replacement with elitist replacement.

2.1.1 Memetic Algorithms

Memetic Algorithms (MA's) were covered in the lecture "Evolutionary Algorithms II". Some components may have been covered in the previous lecture "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 the local search.

```
1  | INPUT: PopulationSize, MaxGenerations
2  | generateInitialPopulation();
3  | FOR 0 -> MaxGenerations
4  |     FOR 0 -> PopulationSize / 2
5  |         select parents using tournament selection
6  |         apply crossover to generate offspring
7  |         apply mutation to offspring
8  |     ENDFOR
9  |     do population replacement
10 | ENDFOR
11 | return s_best;
```

2.1.2 Uniform Crossover

Within Uniform Crossover, the values of each bit are swapped between parents with a probability 0.5.

```
1 | INPUTS: p1, p2, c1, c2
2 | memory[c1] = copyOf(p1), memory[c2] = copyOf(p2);
2 | FOR 0 -> chromosome_length
3 |     IF random < 0.5 THEN
4 |         swap(c1,c2,j) // swap the j'th bit between offspring
5 |     ENDIF
6 | ENDFOR
```

2.1.3 Bit Mutation

Unlike in the single point based search methods from previous labs where a bit flip was applied to one random variable, the mutation operator is applied to each allele (variable) with a probability of $1/chromosomeLength$ to flip the bit value.

```
1 | INPUT: s
2 | FOR 0 -> chromosome_length
3 |     IF random < 1 / chromosome_length THEN
4 |         bitFlip(s,j) // flip the jth bit of solution s
5 |     ENDIF
6 | ENDFOR
```

2.1.4 Trans-Generational Replacement with Elitism

In trans-generational replacement, the current population will be replaced with the offspring (Fig 1 pt. 1). If however the best solution is not contained in the offspring (Fig 1 pt. 2), then the worst solution in the offspring (dark red) is replaced with the best solution in the current population (green).

```
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
```

In the G52AIM Framework, a population replacement scheme should return an array of integers corresponding to the memory locations of the solutions to carry forward to the next generation. The framework then uses these to copy the solutions for the next generation for you. For example, assuming a population size of 4, the population replacement scheme could return $[7,1,6,3]$.

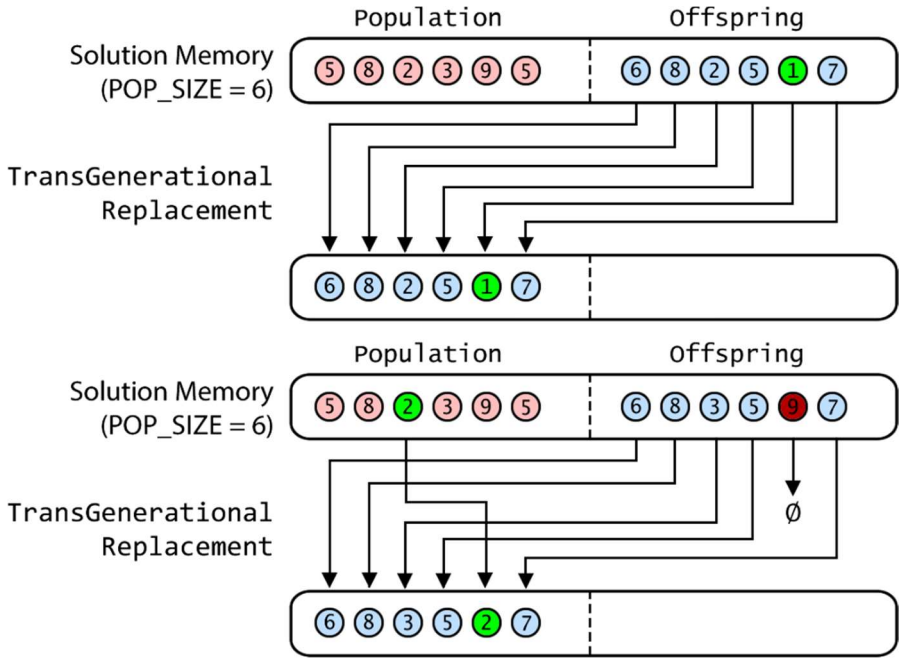

Figure 1 - Illustration of Trans-Generational Replacement with elitist best replacement. The circles represent solutions in memory and the numbers contained in them represent their objective values. TGReplacement will return [6,7,8,9,10,11] in the first example and [6,7,8,9,2,11] in the second.

2.1.5 Tournament Selection

In tournament selection, the best solution from a specified random number of solutions is chosen.

```
1 | INPUT: parent_pop, tournament_size
2 | solutions = getUniqueRandomSolutions(tournament_size);
3 | bestSolution = getBestSolution(solutions);
4 | index = indexOf(bestSolution);
5 | return index;
```

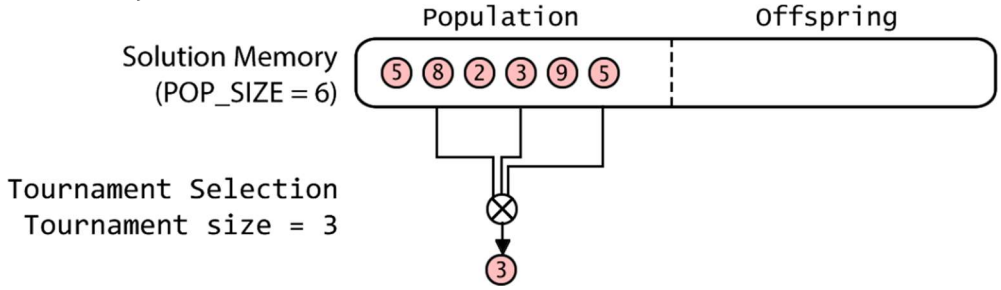

Figure 2 - Illustration of Tournament Selection. Note that in this example, tournament selection will return the memory index 3.

2.1.6 Importing the required files

Within the Lab 04 source files archive is one folder named after the package which you must copy the containing source files to. Within "`com.g52aim.lab04`" are 7 classes. `MemeticAlgorithm.java`, `Lab_04_Runner.java`, `Lab04TestFrameConfig.java`, `TransGenerationalReplacementWith-ElitistReplacement.java`, `TournamentSelection.java`, `BitMutation.java`, and `UniformXO.java`. Copy these files into the package "`com.g52aim.lab04`".

There is also a JAR file which you should import and contains the implementation of DBHC. When invoking DBHC, you should pass the memory index of the solution you want to apply it to.

2.1.7 Problem with program hanging

On some student's computers, the program hangs before displaying boxplots/progress plots. Within the `ExperimentalSettings` Class, there are two variables ***`ENABLE_GRAPHS`***, and ***`ENABLE_PARALLEL_EXECUTION`***; Try setting one or both to `false` if you are experiencing these problems. They can be changed back to `true` on the lab machines if you wish to analyse these algorithms later. **Note**: ***`ENABLE_PARALLEL_EXECUTION`*** is not used in this lab's test framework.

2.2 Marking Criterion

1. A correct implementation of Memetic Algorithm **[15 marks]**.
2. A correct implementation of Tournament Selection **[10 marks]**.
3. A correct implementation of Bit Mutation **[5 marks]**.
4. A correct implementation of Uniform Crossover **[10 marks]**.
5. A correct implementation of Trans-generational Replacement with Elitism **[10 marks]**.

3 Report [50 marks]

To be announced at the start of the lab via a "report exercise sheet" on Moodle.

4 Framework Background

A memetic algorithm is one example of a population-based search method. The G52AIM Framework allows multiple solutions to be stored in memory, the difference being this time that we have 16 memory addresses to manage. Special methods should therefore be used for applying various heuristics such that they are applied to the correct solutions in memory and are given in the **implementation hints** section. Below is an illustration of a population based search method using the G52AIM Framework.

> 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 number of **generations**. For the Memetic Algorithm, we therefore use number of generations as the termination criterion rather than a "time limit"/evaluation count.

- At the start of each iteration (generation), the current population should be stored in the first POP_SIZE indices of the solution memory.

- The second POP_SIZE indices should be used for creating and mutating/improving the offspring.
- At the end of each iteration, POP_SIZE solutions should be selected and moved to the first POP_SIZE indices according to the replacement scheme.

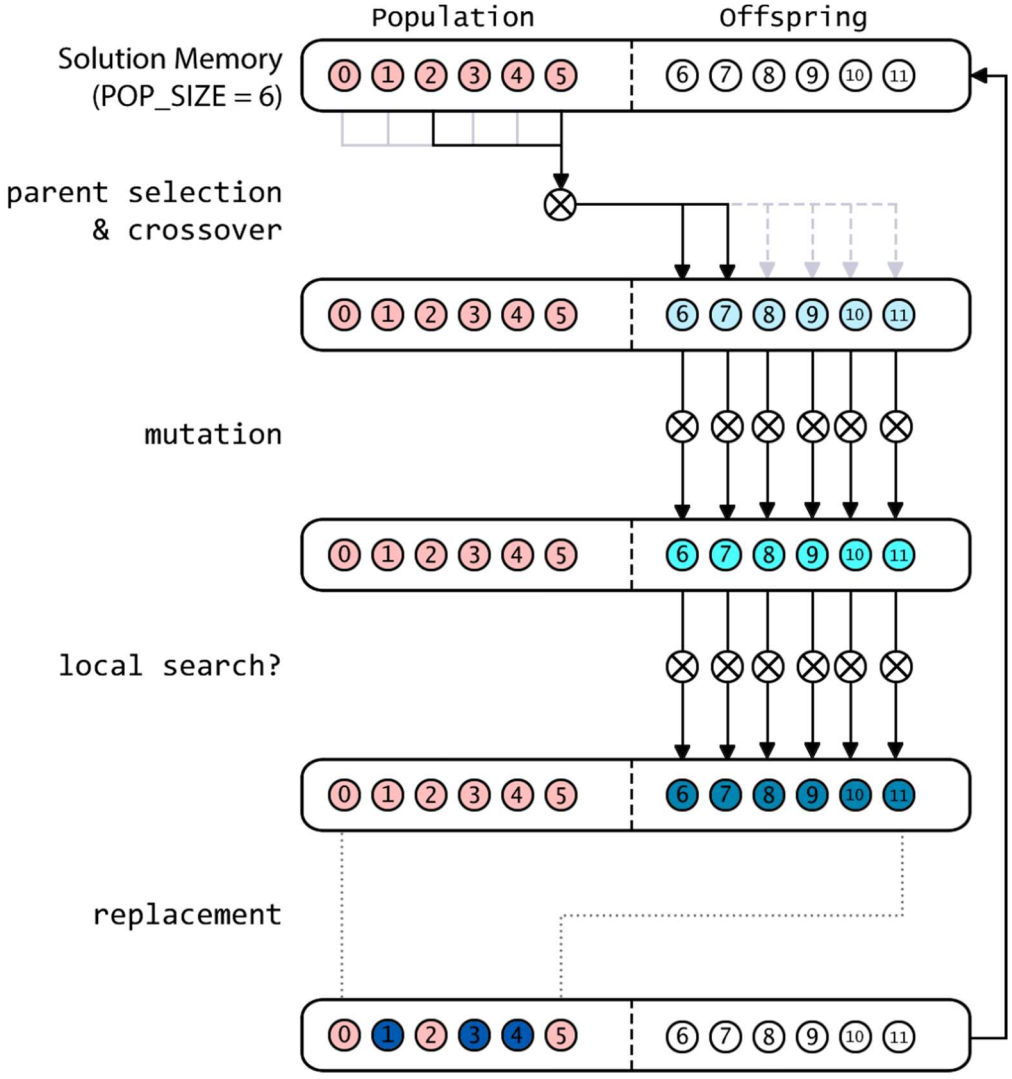

Figure 3 – Population based Search illustrating a Memetic Algorithm in the G52AIM Framework. Note that both the current population and the offspring share the same contiguous memory with indices $[0..POP\_SIZE-1]$ *storing the* ***current population****, and indices* $[POP\_SIZE..POP\_SIZE*2-1]$ *storing the* ***offspring*** *of the current population.*

5 Implementation Hints

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

See the G52AIM Framework API for framework specific questions on Moodle!

6 SUBMISSION

As with all future labs, each coursework will comprise a two-part submission:

(i) [50%] your implementation (code) which needs to be submitted within the computing hours at the lab which will be enforced by attendance sheets. If any extenuating circumstances are preventing you from attending the lab, then please contact me.
(ii) [50%] a brief report which needs to be submitted by Monday, 3pm following each computing session.

IMPORTANT: No late submissions are allowed. Hence any late submission will receive a mark of 0. It is fine just to return the (i) code and not the report for a partial mark. However, if (i) code is not returned within the computing hours, then the report will not be marked yielding a mark of 0 for that computing exercise.

6.1 IMPLEMENTATION SUBMISSION

Deadline: 01/03/2018 – 17:00

You should submit a single **ZIP folder** called **[username]-lab04-implementation.zip** including:

1. `TransGenerationalReplacementWithElitistReplacement.java`
2. `BitMutation.java`
3. `UniformCrossover.java`
4. `MemeticAlgorithm.java`
5. `TournamentSelection.java`

…to Moodle under Submission area CW4a. **Reminder** late submissions or solutions completed outside of the lab will receive a mark of 0 and you will not be able to submit the report section of the coursework.

6.2 Report Submission

Deadline: Tuesday 06/03/2018 – 15:00

You should submit a single PDF file called **[username]-lab04-report.pdf** to Moodle under Submission area CW4b.

7 Outcomes

1. You should know how to implement a Memetic Algorithm.
2. You should be able to implement different components of an Evolutionary Algorithm including population replacement, crossover, and parent selection.
3. You should have an insight into the possible advantages of population based search methods.

IMPORTANT INFORMATION

The labs are assessed exam style sessions. Hence, you are not allowed to complete the coursework outside of the lab session or discuss your solutions with other students during the lab. (You can, of course, ask questions to the lab helpers though!). You should not prepare code in advance of the labs and use it as part of your answer(s). Template code will be provided before the lab including the test framework to run your implementations, and you will be given part completed files where you should provide your answers.