COMP2001: Artificial Intelligence Methods – Lab Exercise 4

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 18th March 2024. The contents of this exercise will be/was covered in **lectures 7 and 8** on Hyper-heuristics (part I and part II respectively).

Note that there is no lab on 29th March due to the holidays, but you can access support in the normal way via the Moodle forums.

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 using the HyFlex Framework API.
- Students should understand, and be able to implement, the components of selection hyper-heuristics.
- Students should be able to critically evaluate heuristic selection methods.

Objectives

The purpose of this lab exercise is to gain experience with using the HyFlex Framework to implement a reinforcement learning-based hyper-heuristic which selects pairs of mutation and local search heuristics to form an iterated local search hyper-heuristic approach for solving the cross-domain search problem.

- HyFlex Introduction. **[OPTIONAL] – Covered in lectures.**
- Creating a project with HyFlex. **[REQUIRED].**
- Implementation of a reinforcement learning-based selection hyper-heuristic with analysis of heuristic selection of mutation-local search heuristic pairings. **[REQUIRED].**

Task 0 – HyFlex Introduction [OPTIONAL – From lectures]

An introduction and overview of the HyFlex framework and CHeSC 2011 competition was given in lecture 7 on hyper-heuristics part 1. It is important that you understand the concepts before attempting this exercise, hence, you should revisit that lecture if needed.

You can find an updated version of the CHeSC 2011 webpages at CHeSC: Cross-Domain Heuristic Search Competition (nott.ac.uk) (https://www.cs.nott.ac.uk/~pszwj1/chesc2011/) which contains:

- A tutorial: https://www.cs.nott.ac.uk/~pszwj1/chesc2011/hyflex_tutorial.html
- JAR file downloads: https://www.cs.nott.ac.uk/~pszwj1/chesc2011/hyflex_download.html
- Benchmarking tools: https://www.cs.nott.ac.uk/~pszwj1/chesc2011/benchmarking.html
- And description of the framework: https://www.cs.nott.ac.uk/~pszwj1/chesc2011/hyflex_description.html

An additional HyFlex tutorial worksheet is provided on Moodle for additional reading.

Task 1 – Creating a project for HyFlex [REQUIRED]

You may choose to include the libraries and source code into the existing lab exercise project or create a new project solely for HyFlex. The latter may be better and act as a starting point for the implementation part of the project coursework.

If you have forgotten how to set up a Java project in IntelliJ and/or how to include the libraries, you should refer to the instructions from lab exercise 0.

Included in this exercise are:

- Two JAR files containing the HyFlex Framework (chesc.jar and chesc-ps.jar); you need to import these as libraries into your project.
- A new test frame for HyFlex and cross-domain search and a utilities class which contains some potentially useful methods. These should go into a package **com.aim**.
- An exercise 4 specific runner and configuration class which should go in **com.aim.runners**.
- The hyper-heuristic to be implemented along with supporting classes which should go into **com.aim.hyperheuristics**.

Task 2 – A reinforcement learning – iterated local search based hyper-heuristic for cross-domain search [REQUIRED]

Implementation

You are given 3 Classes, `RLILS_AM_HH`, `RouletteWheelSelection`, and `HeuristicPair`. You will need to complete the implementations for `RLILS_AM_HH` and `RouletteWheelSelection` to implement the Reinforcement Learning based Iterated Local Search Hyper-heuristic.

Now that we are using HyFlex, your hyper-heuristic will be called a **single** time. You must implement the while loop containing the termination condition. Execution of your solution is handled by the `LabExercise4Runner` Class that is provided for you and the experimental parameters can be configured in `Lab4TestFrameConfig`.

Everything for the hyper-heuristic can be implemented within the `solve(ProblemDomain problem)` method of `RLILS_AM_HH.java` however you should implement Roulette Wheel

Selection within `RouletteWheelSelection.java`. We have also specified five additional methods which we ask that you implement to perform the respective operations. This has the advantage of making RWS easier to implement and debug during the lab! These methods are as follows:

```
1. int getScore( int[] hs );
2. int getTotalScore();
3. void incrementScore( int[] hs );
4. void decrementScore( int[] hs );
5. int[] performRouletteWheelSelection();
```

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

- `Roulette Wheel Selection:`
    - `Default score = 5`
    - `Lower_bound   = 1`
    - `Upper_bound   = 10`
- `Target problems and instances:`
    - `MAX-SAT (3,11)`
    - `TSP(0,6)`
    - `0-1 Bin Packing (7,11)`
- `Run time = 30 nominal (competition) seconds.`
- **`Trials per instance = 5.`**

You may want to change the number of trials per instance in HyFlexTestFrame.java such that the total runtime is 15 minutes instead of 45 minutes as if the default 15 trials was used!

There are also two other methods which we have supplied which you can use when debugging your code. These are `printHeuristicIds()` and `printHeuristicScores()` and will print out (into the console) the heuristic IDs and their associated scores respectively.

Heuristic Selection – Roulette Wheel Selection (AKA Proportionate Selection)

Roulette Wheel Selection (RWS), also known as Proportionate Selection, is a reinforcement learning based heuristic selection method. The basic idea behind RWS is that each heuristic has associated with it a score which is bounded between a lower and upper bound. If the application of a heuristic results in an improving move `(f(s') < f(s))` then the score for that heuristic is incremented such that the probability of selecting it again is increased. In contrast, if the application of a heuristic results in a worsening move `(f(s') > f(s))`, then its score is reduced such that the probability of selecting it again in decreased. The scores for each heuristic are then used directly as "proportions" or probability weights which **can be** calculated as shown in Equation (1). When it comes to selecting a heuristic using RWS, a random number is generated and used to select a heuristic. An illustration is given below of this process.

$$P(h_i) = \frac{score(h_i)}{\sum_{j=0}^{N-1} score(h_j)} \qquad (1)$$

Given a set of heuristic IDs and scores:

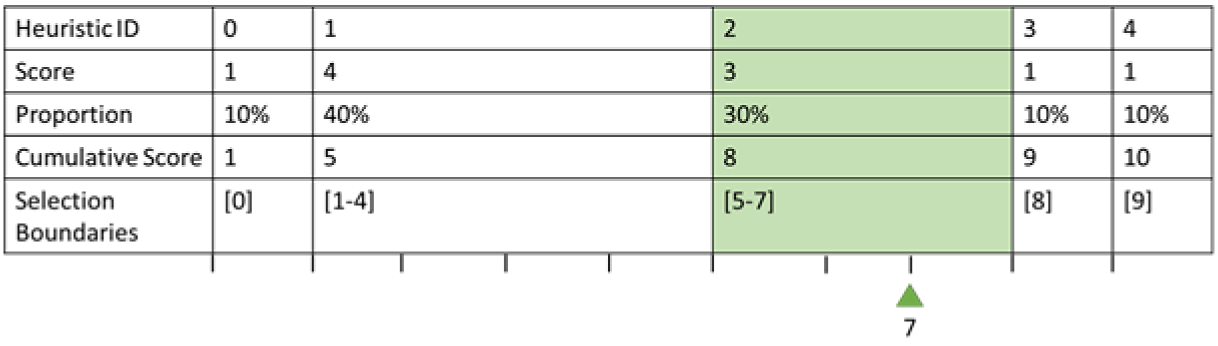
Heuristic ID	0	1	2	3	4
Score	1	4	3	1	1
Proportion	10%	40%	30%	10%	10%

Generate a random number, `random_value`, between 0 (inclusive) and the sum of scores (exclusive).

$$random\_value \leftarrow 7$$

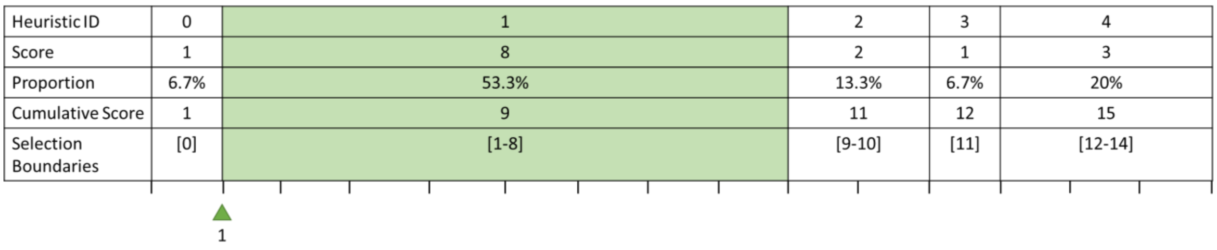
Then select the heuristic by choosing the first heuristic to have a cumulative score, as calculated in Equation (2), greater than the $random\_value$.

$$cumulative\_score(h_i) = \sum_{j=0}^{i-1} score(h_j) \tag{2}$$



Heuristic ID	0	1	2	3	4
Score	1	4	3	1	1
Proportion	10%	40%	30%	10%	10%
Cumulative Score	1	5	8	9	10
Selection Boundaries	[0]	[1-4]	[5-7]	[8]	[9]

Here we choose the heuristic as heuristic ID 2 because $cumulative\_score(h_2) = 8, 8 > 7$, and $cumulative\_score(h_1) = 5$, and $5 \ngtr 7$.



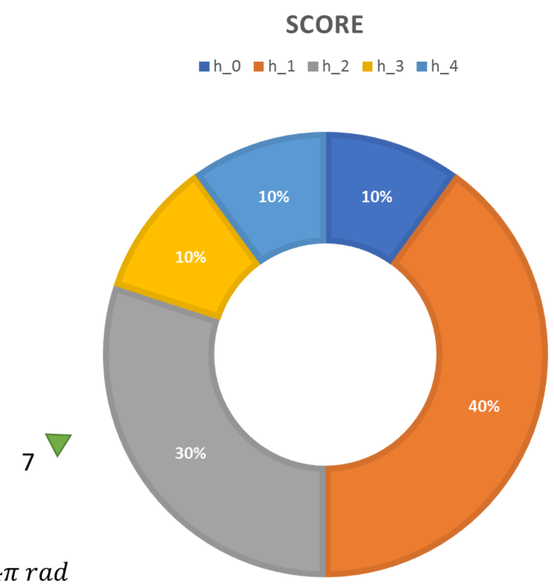
Heuristic ID	0	1	2	3	4
Score	1	8	2	1	3
Proportion	6.7%	53.3%	13.3%	6.7%	20%
Cumulative Score	1	9	11	12	15
Selection Boundaries	[0]	[1-8]	[9-10]	[11]	[12-14]

Here we choose the heuristic as heuristic ID 1 because $cumulative\_score(h_1) = 9$, 9 >1, and $cumulative\_score(h_0) = 1$, and $1 \ngtr 1$.

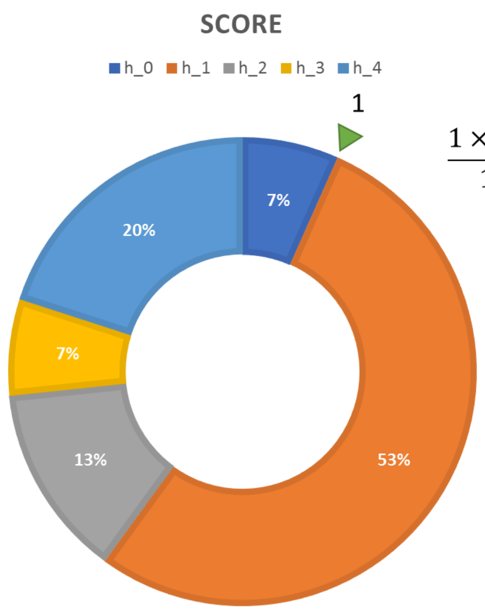
Another way to visualise this is as a roulette wheel where each heuristic has an associated segment, and its size is proportioned based on its score. The following two examples are the same as the above two but displayed using this method.

```
INPUT: heuristic_ids, heuristic_scores.
total_score = sum(heuristic_scores);
value = random ∈ $[0, total\_score]$;
WHILE cumulative_score < value DO
    h <- selectNextHeuristicFromHeuristicIDs;
    chosen_heuristic <- h;
    cumulative_score += score(h);
END_WHILE
return chosen_heuristic;
```

Code 2 - Pseudocode for selection part of Roulette Wheel Selection.



$$\frac{7 \times 360}{10} = 252° = 1.4\pi\ rad$$



$$\frac{1 \times 360}{15} = 24° = 0.13\pi\ rad$$

Move Acceptance – All Moves

For the move acceptance component, you will implement All Moves acceptance. As its name suggests, all moves are accepted and never rejected. The idea behind such a method is that if the heuristic selection method chooses the "best" heuristic at each iteration of the search, then we should not need a move acceptance criterion that filters (rejects) candidate solutions; hence, all moves acceptance accepts all moves.

Experimentation

HyFlex uses time as its termination criterion. This means that for a fair comparison, your machine will need to be benchmarked using the HyFlex benchmarking tool which can be found at https://www.cs.nott.ac.uk/~pszwj1/chesc2011/benchmarking.html . Note that Mac OS X/OS X/macOS does not appear to be supported by this tool and you will need to use a Linux or

Windows machine for your experiments. Alternatively, set your machine time as 10 actual minutes but consider that any results you compare against must be as a result of running those algorithms on your own machine.

You should download and run the benchmark tool to work out how long your computer should run for to obtain 10 nominal competition minutes. You should then set the `MILLISECONDS_IN_TEN_MINUTES` variable in `HyFlexTestFrame.java` to this value. The test frame(s) will use this value to figure out the run time for 60 nominal seconds. **You should do this for every computer that you use since their speeds will be different**. Moreover, the tool is not suitable for cloud computing services or even running experiments concurrently on the same machine!

The proposed reinforcement learning mechanism uses a concept of having an upper and lower bound for the "scores" of each heuristic pair. You should investigate the effects of changing these values in any way you see fit subject to $0 \leq S_{lower} \leq S_{higher} \leq \text{Integer.MAX\_VALUE}$, $S_{higher} > 0$, and observe the impact that this has on solving problem instances from different domains. For each set of experiments, you should set the default score to be:

$$S_{default} = \left\lfloor \frac{S_{lower} + S_{upper}}{2} \right\rfloor$$

You should compare your configurations by applying the formula 1 ranking mechanism to each hyper-heuristic configuration and declaring the best as the one that has the highest F1 score. Report your configurations and F1 scores for each on the Moodle forum for peer-feedback.

Expected output

No expected output is given since any slight variation in your implementation and your computer specification has the potential to significantly affect the output, even if your implementation is correct! For reference only, the median results I obtained with the default configuration of DEFAULT_SCORE = 5, LOWER_BOUND = 1, and UPPER_BOUND = 10, were as follows:

Domain	Instance	Median Best Cost
SAT	3	4
SAT	11	10
TSP	0	49305.22
TSP	6	59769.03
BP	7	7.489×10^{-2}
BP	11	6.534×10^{-2}

Note that when the hyper-heuristic is run, as output there will be a CSV file created in the project root called `RL-ILS_HH.csv`. This file will contain the results of the experiments including the objective values of the best solutions found for each problem domain, instance, and trial. **You should create a copy of this file before changing the algorithmic parameters so that you can compare the performance of the RL-ILS hyper-heuristic across different sets of parameters.**

Implementation Hints / Q&A

A `LinkedHashMap` has been initialised in `RouletteWheelSelection.java` for you. Being “linked”, at least in Java, ensures that the heuristic IDs (the key set) remain ordered in the way that they were added. The use of a Map allows for a convenient way to store the **heuristics (as keys)** and **heuristic scores (as values)**.

The Javadocs for some of the additional data structures can be found at:

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/LinkedHashMap.html

In particular, you should pay attention to the methods:

- `put(Object o)`
- `get(Object o)`
- `values()`
- `keySet()`

Below are some methods which may come in useful when implementing Roulette Wheel Selection:

How do I access the heuristics within the HeuristicPair Record Class?

See an explanation of Java Records at Java 14 Record Keyword | Baeldung (https://www.baeldung.com/java-record-keyword).

How do I get the list of heuristic IDs?

Being stored in a (Linked)HashMap, you can retrieve the heuristics as a Set of HeuristicPair. You can get this by calling the `keySet()`method on the HashMap.

How do I get the score for each heuristic ID?

Since the scores are stored as values in a Map, the score of a heuristic can be retrieved by calling the `get( HeuristicPair hs )` method on the HashMap.

How do I set the score of a heuristic?

Because we have chosen a Map data structure, you can easily update the score of a heuristic with ID `id` by calling the `put( HeuristicPair hs, Integer score )` method on the HashMap. It doesn’t matter if the entry already exists in the Map since the definition of a Map is to **replace** an element if the key already exists.

I cannot iterate over the heuristic IDs using a for loop!

Being a Set of Integers, you cannot use a standard for loop to iterate over the contents. The most synonymous way that you can do this is to use a foreach loop which is written as follows:

```
for(int i = 0; i < array.length; i++) {  // standard for-loop
    Element e = array[i];
    …
}
```

```
Goes to:
```

```
for(Element e : array) {                // foreach loop
    …
}
```

In our case, the “array” will be replaced by the `keyset` (feel free to use the `entrySet` if you know how this works!) and the type of e will be `Integer`.