HyFlex Tutorial Worksheet

1 Introduction to HyFlex and CHeSC 2011

1.1 HyFlex Introduction

In the COMP2001 Framework, a single problem (MAX-SAT) is implemented. This problem is represented as a binary string and access to the underlying representation was given to you in the form of a bitflip() method which you then used to implement different search techniques from low-level heuristics through to single-point and population-based metaheuristics.

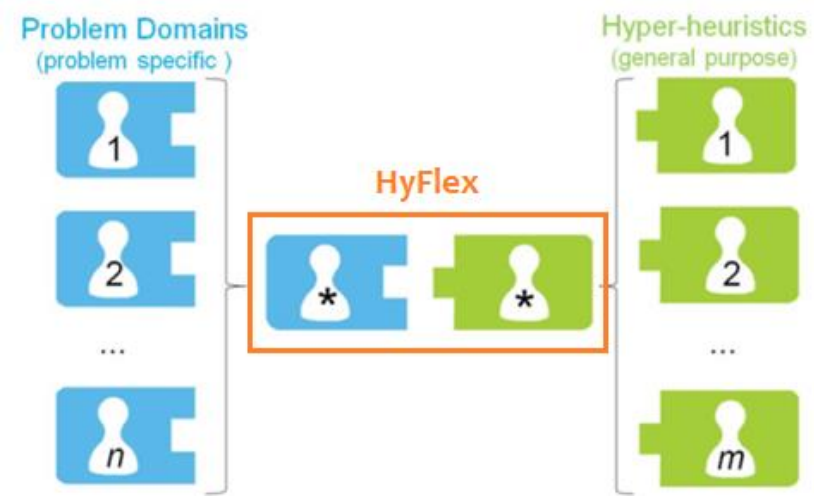
Question: How would you go about solving a different problem (such as TSP)?

When solving a different problem, we may require a different representation. In the case of TSP this can be a permutation. This in turn leads to another issue. The heuristics/operators that we designed for solving MAX-SAT problems will not work for TSP as we assumed binary representation!

Question: How might we be able to generalise the search framework so that we can use the same search algorithms to solve different problems?

This is where the HyFlex Framework comes in. HyFlex provides a software interface to separate problem specific details from the higher-level optimisation methods and their components. For example, when we implemented Iterated Local Search, we needed an operator to perturb the solution, and an operator to locally improve the solution. Using the HyFlex interface, we can find a pool of low-level mutation heuristics to perturb the solution, and a pool of low-level heuristics to locally improve the solution.

HyFlex is a software framework which enables the implementation and comparison of iterative general-purpose heuristic search algorithms (e.g. hyper-heuristics) across different domains without requiring modification.



The HyFlex framework provides an implementation of six different combinatorial optimisation problems: Bin Packing, Flow Shop Scheduling, MAX-SAT, Personnel Scheduling, Travelling Salesman Problem, and Vehicle Routing Problem. For each problem domain, there are a set of problem instances, a solution initialisation method, and several low-level heuristics of various "types".

Problem Domain	Number of Instances	Instance IDs
Bin Packing	12	$[0 \rightarrow 11]$
Flow Shop	12	$[0 \rightarrow 11]$
MAX-SAT	12	$[0 \rightarrow 11]$
Personnel Scheduling	12	$[0 \rightarrow 11]$
Travelling Salesman	10	$[0 \rightarrow 9]$
Vehicle Routing	10	$[0 \rightarrow 9]$

Table 1. HyFlex problem domains, indicating initialisation, the total number of low-level heuristics and the number of heuristics per type.

Domain	*Initialisation*	*Total*	*Mut.*	*R&R*	*Xover.*	*LS.*
Max-SAT	Random bit-string	9	4	1	2	2
Bin Packing	Randomised first-fit heuristic [15]	8	3	2	1	2
Flow Shop	Randomised NEH procedure [17]	15	5	2	3	4
Pers. Sched.	Randomised hill climbing heuristic	12	1	3	3	4
TSP	Random permutation	15	5	1	3	6
VRP	Randomised constructive heuristic	12	4	2	2	4

You will see how each of the components can be accessed and used later in the tutorial. Crucially, to maintain generalisability, HyFlex uses the concept of a *domain barrier* which restricts the flow of information between the domain itself, and the search procedure. We can only see the objective function values of solutions, types of heuristics, and various other information about the search procedure. We cannot find out specifics about the objective function, solution representation, or the actual heuristics. That is to say, all we know about DBHC is that it is a local search heuristic.

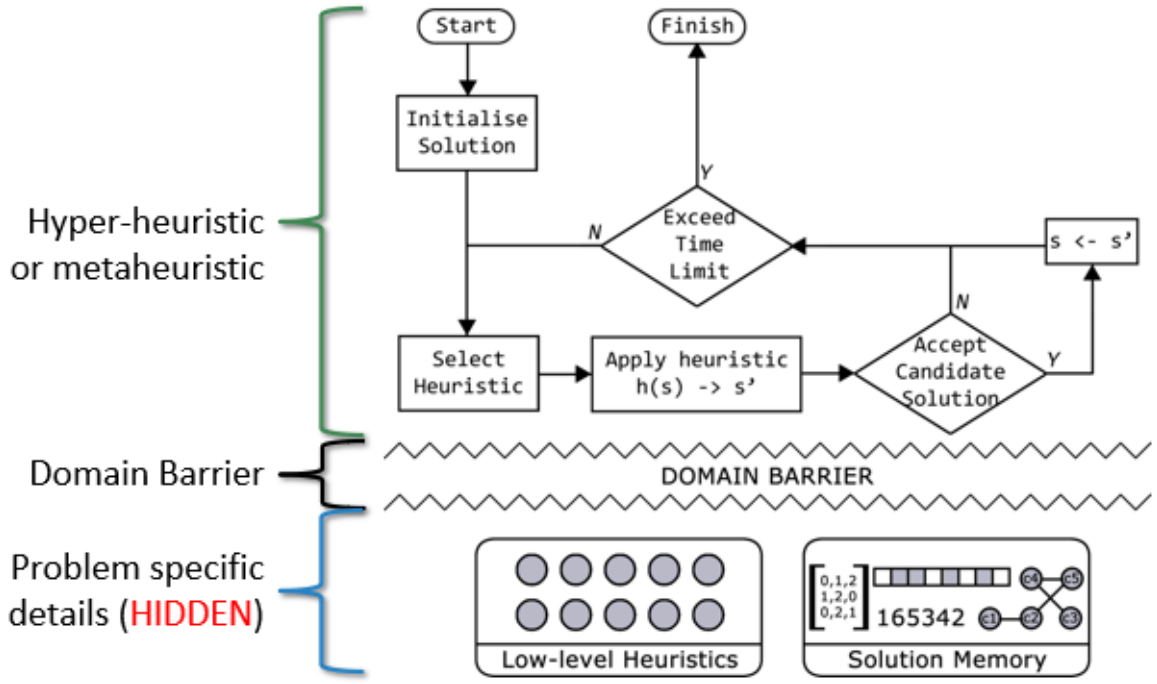


Figure 1 - Illustration of the concept of the domain barrier.

1.2 The Cross-domain Heuristic Search Challenge (CHeSC) 2011

The aim of the cross-domain heuristic search challenge (CHeSC) 2011 https://www.cs.nott.ac.uk/~pszwj1/chesc2011/ was to develop a single hyper-heuristic which could be used to effectively solve a variety of problems across different problem domains.

Each hyper-heuristic (HH) was given 10 minutes of execution time per trial on the challenge machine to solve 30 different problem instances composed of 5 instances each from 6 domains. The HH is evaluated 31 times for each instance (with different initial solutions) to assess a reasonable sample size of the HH's performance.

Given that each HH is allowed 10 minutes to solve a given problem on the competition machine, it is not fair to say that 10 minutes is given across all machines; otherwise, we'd just run some form of randomised search on a supercomputer! To facilitate a fair benchmark, a tool (https://www.cs.nott.ac.uk/~pszwj1/chesc2011/benchmarking.html) is provided to estimate a relative amount of time that a machine needs relative to the desktop technologies of 2011. **You should run this tool from a command line/terminal window and note your run time. This could be different for the university computers and your personal device(s) so make sure you repeat this. When running the benchmark, you should not have any workload-heavy processes running as these will affect the reported time.**

Caveat: This may have given a more accurate estimation when the CPU architectures of 2011 did not vary as much as they do now. Technologies such as Intel Turbo Boost, AMD Precision Boost Overdrive, hybrid architectures mixing performance and efficiency CPU cores, significantly increased memory speeds, increased cache sizes, etc… all add new variables which may or may not be reflected by the benchmark tool.

It is important to note that when it comes to evaluating a hyper-heuristic, you should **not** run experiments in parallel due to resource contention adversely affecting the runtime of your HH. For the same reason, running experiments on a cloud-based cluster is not recommended.

1.2.1 CHeSC 2011 Performance Comparison

The competition element of CHeSC 2011 sought to find the best HH in terms of overall performance. To compare each of the HH's, a Formula 1 (as in the racing tournament) style ranking method was used to assign scores to each hyper-heuristic based on their relative performance to each other for each instance, and the sum of scores were taken across all 30 instances to arrive at the final score. The performance of each HH is determined from its median result across the 31 trials on each instance. If more than one HH has the same rank, then the average score is assigned as seen in the below example for the median 15.

	HH1	HH2	HH3	HH4	HH5	HH6	HH7	HH8	HH9	HH10
Median	2	5	15	19	34	10	8	11	15	99
Rank	1	2	6-	8 (not 7)	9	4	3	5	6-	10
Score	10	8	(3+2)/2	1	0	5	6	4	(3+2)/2	0

Table 1 - Example scoring for a single instance 'A'.

	HH1	HH2	HH3	HH4	HH5	HH6	HH7	HH8	HH9	HH10
Score(A)	10	8	2.5	1	0	5	6	4	2.5	0
Score(B)	8	5	6	1	0	5	4	3	0	10
Score(C)	3.9	3.9	3.9	3.9	3.9	3.9	3.9	3.9	3.9	3.9
F1 Score	**21.9**	**16.9**	**12.4**	**5.9**	**3.9**	**13.9**	**13.9**	**10.9**	**6.4**	**13.9**

Puzzle

Given the F1 scores for instance 'C', what can you deduce about the results of the 10 different hyper-heuristics? *Hint: They performed the same is not strictly correct!*

2 HyFlex Framework API

2.1 API Overview

In this section, we provide some useful HyFlex API methods and typical usage from within the code. You should read and understand each of these methods, relating them to the example hyper-heuristics on the HyFlex/CHeSC webpages for further understanding.

Note that the full API specification for `HyperHeuristic` and `ProblemDomain` can be found at HyperHeuristic (nott.ac.uk) and ProblemDomain (nott.ac.uk) respectively.

HH Initialisation – to do before solving a problem using the hyper-heuristic

The solver seed is set in the constructor of the `HyperHeuristic` Object	`long seed = -1568485354L;` `HyperHeuristic hh = new HH(seed);`
The time limit can be set using the `setTimeLimit(long ms)` method	`long timeLimit = 10 * 60 * 1000; //10 minutes` `hh.setTimeLimit(timeLimit);`

The problem domain and instance to be solved can be set using the `loadProblemDomain(ProblemDomain problem)` method	`int instanceID = 0;`
	`ProblemDomain problem = new SAT(seed);`
	`problem.loadInstance(instanceID);`
	`hh.loadProblemDomain(problem);`

The hyper-heuristic is then executed using the `run()` method, **not the solve method!**	`hh.run();`

Methods required for initialising problems and hyper-heuristics

Other useful methods:

The amount of time elapsed can be got using the `getElapsedTime()` method	`long elapsed = getElapsedTime();`
You can check if the amount of time has expired using the `hasTimeExpired()` method	`boolean hasTimeExpired = hasTimeExpired();`
The time limit given can be retrieved using `getTimeLimit()`	`long timeLimit = getTimeLimit();`
The objective function value of the best solution found during the search process can be found using the `getBestSolutionValue()` method	`double bestFitness = getBestSolutionValue();`
Random number generator is a protected field member and can be accessed simply by typing `rng`	`double random = rng.nextDouble();`

Useful methods for the HyperHeuristic Object

If accessing methods within the HyperHeuristic class or a Class extending HyperHeuristic, there is no need to precede the method call with an instance of HyperHeuristic. E.g. hh.method();

Problem initialisation methods

We can set the number of memory locations by calling `setMemorySize(int size)` with the size of the memory to create	`int memorySize = 2;` `problem.setMemorySize(memorySize);`
We can initialise a new solution by calling `initialiseSolution(int index)` with the index of the solution memory for where to put it	`int currentSolutionIndex = 0;` `problem.initialiseSolution(currentSolutionIndex);`

Problem domain initialisation methods

Manipulating low-level heuristic components

Low-level heuristics have specific parameter settings known as intensity of mutation, and depth of search. These can be manipulated using the following methods:

Low-level heuristic Manipulation methods

We can set the intensity of mutation by calling `setIntensityOfMutation(int iom)`

```
double intensityOfMutation = 0.20d;
problem.setIntensityOfMutation(intensityOfMutation);
```

We can set the depth of search by calling `setDepthOfSearch(int dos)`

```
double depthOfSearch = 0.20d;
problem.setDepthOfSearch(depthOfSearch);
```

We can make use of heuristics that use these parameters by calling their respective `getHeuristicsThatUse…()` methods

```
int[] IOM = problem.getHeuristicsThatUseIntensityOfMutation();
int[] DOS = problem.getHeuristicsThatUseDepthOfSearch();
```

The values for depth of search and intensity of mutation must be in the range [0,1]
By default, these are set to 0.2

Useful methods when searching

Applying a heuristic can be done in two ways depending on the type of heuristic.

Useful problem domain methods for search process

LS, MTN, RR are applied using `applyHeuristic(int heuristicID, int sourceIndex, int destinationIndex);`

```
int[] MTN = problem.getHeuristicsOfType(HeuristicType.MUTATION);
int heuristic = MTN[rng.nextInt(MTN.length)];
problem.applyHeuristic(heuristic, 0, 1);
```

XO heuristics are applied using `applyHeuristic(int heuristicID, int sourceIndex1, int sourceIndex2, int destinationIndex)`

```
//assume best solution is stored in memory index 2
int[] XO = problem.getHeuristicsOfType(HeuristicType.CROSSOVER);
int heuristic = XO[rng.nextInt(XO.length)];
problem.applyHeuristic(heuristic, 0, 2, 1);
```

Heuristics can be segregated by their types using `getHeuristicsOfType(HeuristicType type)`

```
int[] LS = problem.getHeuristicsOfType(HeuristicType.LOCAL_SEARCH);
int[] RR = problem.getHeuristicsOfType(HeuristicType.RUIN_RECREATE);
```

Useful methods when searching

Useful problem domain methods for search process

Solutions can be copied between memory indices using the `copySolution(int sourceIndex, int destinationIndex)` method

```
problem.copySolution(1, 0);
```

The total number of low-level heuristics can be found using `getNumberOfHeuristics()`

```
int numHeuristics = problem.getNumberOfHeuristics();
```

The objective function value of solutions can be calculated using `getFunctionValue(int solutionIndex)`

```
double candidateFitness = problem.getFunctionValue(1);
```

The objective function value of the current best solution found can be retrieved using `getBestSolutionValue()`

```
double bestFitness = problem.getBestSolutionValue();
```

A few other methods exist but for the scope of the labs are not required…

2.2 Examples and Exercise to Familiarise yourselves with HyFlex/Hyper-heuristics

A hyper-heuristic is formed of two key components: a heuristic selection method, and a move acceptance criterion. These are depicted in the following figure as "Select Heuristic" and "Accept Candidate Solution" respectively. Hyper-heuristics are high-level search methodologies meaning that they can also be used to solve different problems without modification.

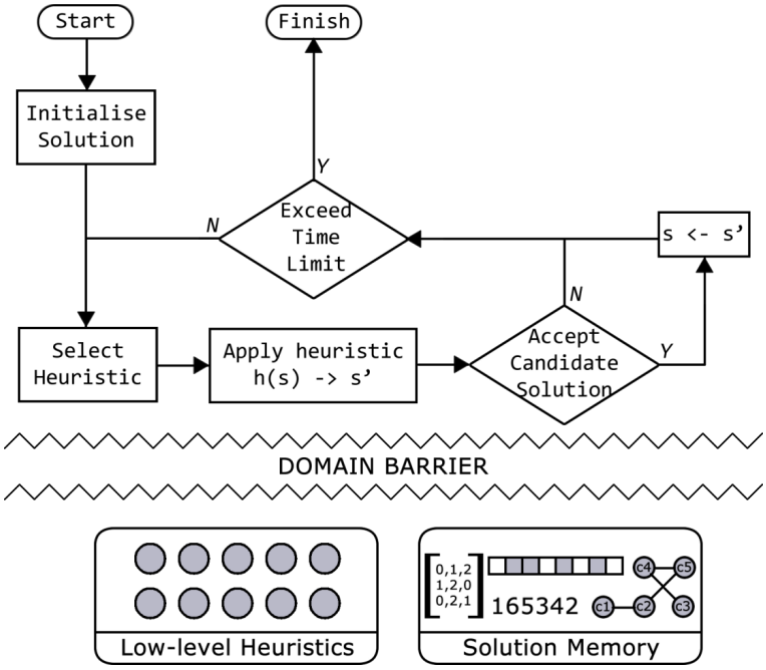


Figure 2 – Single point-based Selection Hyper-heuristic.

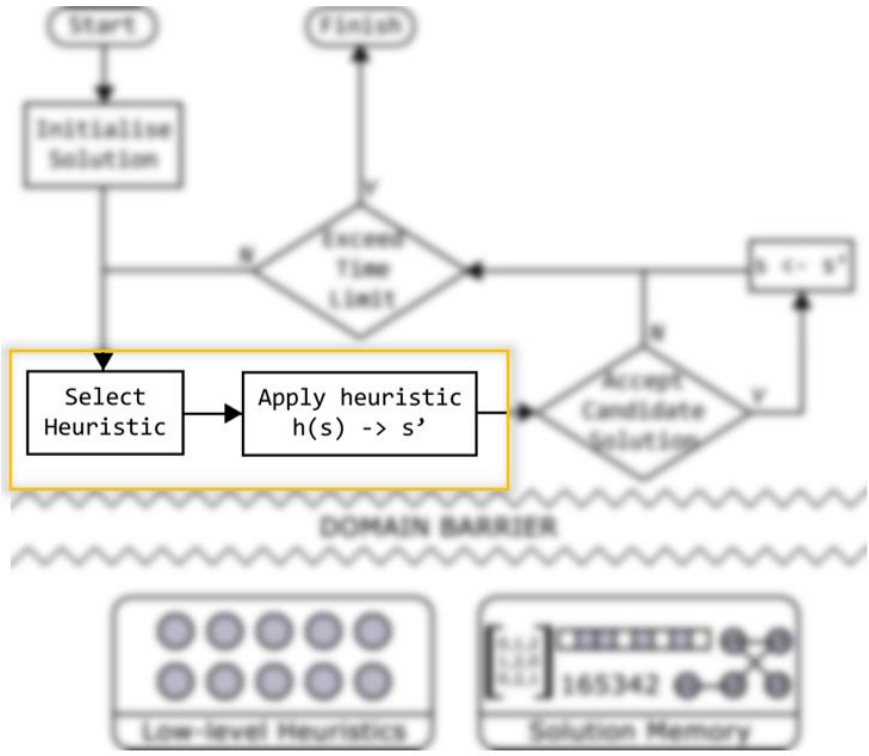
Below is the pseudocode for a single point-based selection hyper-heuristic utilising only a **single heuristic** for each iteration. Note that the problem domain is responsible for tracking the best solution found so far; hence, you do not need to implement lines 11-13!

```
 1 | s <- generateInitialSolution();
 2 | WHILE ( termination_criterion_not_satisfied ) DO
 3 |     h <- heuristicSelection( heuristic_set );
 4 |     s' <- applyHeuristic(h, s);
 5 |     accept <- moveAcceptance(s, s', …);
 6 |     IF accept THEN
 7 |         s <- s';
 8 |     ELSE
 9 |         s <- s;
10 |     END_IF
11 |     IF f(s') < f(s_{best}) THEN
12 |         s_{best} <- s';
13 |     ENDIF
14 | END_WHILE
15 | return s_{best};
```

Code 1 - Pseudocode for a Selection Hyper-heuristic.

2.2.1 Heuristic Selection

Heuristic Selection: Given the current search state, need to decide a (sequence of) heuristic(s) to apply in the current iteration. Notice here that in general, a metaheuristic uses a predefined sequence of heuristics, whereas in a hyper-heuristic this sequence is learnt.



In HyFlex, we have a set of low-level heuristics addressed by indices. The only information we have about each heuristic is their "heuristic type". These types are one of:

- Mutation
- Local Search
- Crossover
- Ruin Recreate

We need to select a (sequence) of heuristic(s) using a predefined mechanism (metaheuristic) or learning mechanism (hyper-heuristic). We can do this by generating a sequence of heuristic IDs such as:

[0], [2], [1], … Simple Random?

[0,1], [2,3], [0,1], … Iterated Local Search?

…where even IDs **in this example** are mutation operators, and odd IDs local search operators.

We can "apply" each heuristic by calling the applyHeuristic(…) method on the instance of the problem domain object (see "useful methods when searching" in the earlier API overview). When we do this, HyFlex creates a copy of the solution in the first specified solution index and stores it in the second specified index before applying the heuristic mapped to by the heuristic index.

```
int n = problem.getNumberOfHeuristics();
int h = rng.nextInt(n);
problem.applyHeuristic(h, 0, 1);
```

Heuristic ID	Solution index "from"	Solution index "to"

Applying a sequence of heuristics takes a bit more thought but it is possible by manipulating the solution indices that we are applying the heuristic to and where we are storing the result. Essentially, the very first heuristic in the sequence is applied as before going from one index to another, but all subsequent heuristics are applied to the same resulting solution index:

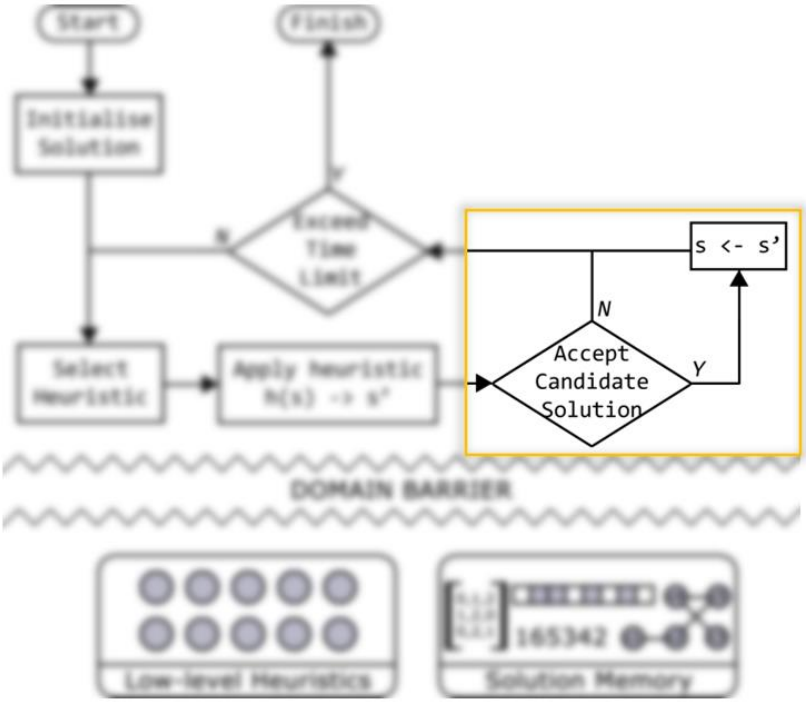
```
int n = problem.getNumberOfHeuristics();
int[] hs = new int[]{ rng.nextInt(n), rng.nextInt(n) };

problem.applyHeuristic(hs[0], 0, 1);
for(int i = 1; i < hs.length; i++) {

    problem.applyHeuristic(hs[i], 1, 1);
}
```

2.2.2 Move Acceptance

Move acceptance: Given the current search state, decide whether to accept or reject the candidate solution s_i' as the current solution in the following iteration. That is if we accept, then $s_{i+1} \leftarrow s_i'$, else if we reject, then $s_{i+1} \leftarrow s_i$.



In HyFlex, we perform move acceptance by copying solutions between memory indices (in the same way as the COMP2001 Framework) using the `copySolution(int sourceIndex, int destinationIndex)` method on the ProblemDomain Object.

Example:

Where the indices of the current and candidate solutions are 0 and 1.

Given the IE move acceptance method, $f(s_i) = 15.0$, and $f(s_i') = 13.2$:
$s\_(i+1) \leftarrow s_i'$
// accept – copy the "backup" as the current solution
`problem.copySolution(1, 0);`

Given the IE move acceptance method, $f(s_i) = 15.0$, and $f(s_i') = 18.7$:

$s_{i+1} \leftarrow s_i$

// reject – overwrite the "backup" with the current solution

```
problem.copySolution(0, 1);
```

2.2.3 Heuristic Selection – Simple Random

For practice in this worksheet, you can implement a simple (uniform) random (SR) heuristic selection. Given a set of low-level heuristics, or in the case of HyFlex a set of heuristic IDs, SR will select a single heuristic at random. Below is the pseudocode for SR.

```
 1 | INPUT: heuristic_set
 2 | h <- random ∈ heuristic_set;
13 | return h;
```

Code 2 - Pseudocode for simple random heuristic selection.

2.2.4 Basic Example of a Hyper-heuristic in HyFlex

All hyper-heuristics in HyFlex have the following common anatomy:

```
public void solve(ProblemDomain problem) {

    // define indices for current and candidate solutions
    int iCurrentIndex = 0, iCandidateIndex = 1;

    // set the memory size to two for single point-based search
    problem.setMemorySize(2);

    // generate an initial solution and store in the current index
    problem.initialiseSolution(iCurrentIndex);

    // keep a record of the current and candidate solution costs
    double dCurrentSolutionCost, dCandidateSolutionCost;

    // update the current solution cost to be equal to the objective
    // value of the initial solution
    dCurrentSolutionCost = problem.getFunctionValue(iCurrentIndex);

    // define the main loop of the hyperheuristic
    while(!hasTimeExpired()) {

        // select and apply heuristic(s)    ...

        // move acceptance                  ...
    }
}
```

We implement the main iterative part of the algorithm within the while loop which handles the termination criterion. Below is an example of a "hyper-heuristic" applying random heuristic selection and accepting all non-worsening moves, and 50% of worse moves (the naïve acceptance strategy).

```java
public void solve(ProblemDomain problem) {

    int iCurrentIndex = 0, iCandidateIndex = 1;
    int[] aiMutationHeuristicIds = HyFlexUtilities.getHeuristicSetOfTypes(problem, HeuristicType.MUTATION);

    problem.setMemorySize(2);
    problem.initialiseSolution(iCurrentIndex);

    double dCurrentSolutionCost, dCandidateSolutionCost;
    dCurrentSolutionCost = problem.getFunctionValue(iCurrentIndex);

    while(!hasTimeExpired()) {

        // simple random heuristic selection from mutation operators
        int h = aiMutationHeuristicIds[rng.nextInt(aiMutationHeuristicIds.length)];

        // apply heuristic 'h' to solution in iCurrentIndex and save in iCandidateIndex
        dCandidateSolutionCost = problem.applyHeuristic(h, iCurrentIndex, iCandidateIndex);

        // naive acceptance
        if(dCandidateSolutionCost <= dCurrentSolutionCost || rng.nextDouble() < 0.5) {

            // accept
            problem.copySolution(iCandidateIndex, iCurrentIndex);
            dCurrentSolutionCost = dCandidateSolutionCost;

        } else {

            // reject
            problem.copySolution(iCurrentIndex, iCandidateIndex);
            // 'dCurrentSolutionCost' stays the same!
        }
    }
}
```