University of Sheffield

Evaluating the Performance of Hyper-Heuristics

Ruitao Feng

Supervisor: Pietro Oliveto

A report submitted in partial fulfilment of the requirements
for the degree of MSc. Computer Science with Speech and Language Processing

in the

Department of Computer Science

September 10, 2019

Declaration

All sentences or passages quoted in this document from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations that are not the work of the author of this report have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure.

Name: Ruitao Feng

Signature: Ruitao Feng

Date: September 10, 2019

Abstract

Many successful applications of randomised search heuristics (such as evolutionary algorithms) to real-world optimisation problems have been reported. Despite these successes, it is still difficult to decide which particular search heuristic is a good choice for the problem at hand, and what parameter settings should be used.

The high-level idea behind the field of hyper-heuristics is to overcome this difficulty by evolving the search heuristic for the problem rather than choosing one in advance. The overall goal is to automate the design and the tuning of the algorithm for the optimisation problem and hence achieve a more generally applicable system.

Many hyper-heuristics decide which search operators to use according to how well they perform during the optimisation process. Hence, learning which operators are most effective for the problem at hand is considered crucial for the success of hyper-heuristics.

This project aims to analyse the performance of selection hyper-heuristics on natural combinatorial optimisation problem: **SAT** problem, Besides, we extended the project to finish a selection hyper-heuristics framework that could test with different combinations of acceptance operators on all kinds of benchmark functions as problem fields to evaluate their performance.

Acknowledgement

During the whole process of my dissertation project, I have received a lot of support as well as assistance. I would like to express my deepest appreciation to my project tutor, Pietro Oliveto, who has guided me from the very beginning of this project and gives advises to both my development and presentation.

I would like to thank Chenxi Liu for her excellent cooking skills. We worked and cooked together during the whole dissertation period. I am very glad of your company and honored to stay by your side.

I would also like to thank Sheng Liang, Hao Xu and Zhen Yuan for their kindness of sharing house after my rent was due. I will never forget the days we were roommates. And a special thanks Hao Xu's pet for her cuteness eased my nerves every time.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Every corner in engineering exists combinatorial optimisation problem, scheduling, logistics, supply chain designing and so on. It also has applications in several fields, including auction theory, machine learning, and artificial intelligence. The goal behinds those application is to find the optima from a finite set of objects. However, many combinatorial optimisation problems are NP-hard and cannot be solved by exhaustive search. That is, the problem cannot be solved by a polynomial algorithm without giving polynomial algorithms. Although it has not been proven that no polynomial-time algorithm could solve NP-hard problems, they are considered unsolvable in practice. In this case, the approaches to get the best result that considered close enough with the global optima in polynomial time were investigated, heuristics, in other words.

Although heuristics algorithms have achieved much successful application in solving real-world computational search problem, there are still some difficulties to apply them to a newly encountered problem:

1. Select which particular search heuristic should be used.
2. What parameter setting should be applied.

As a result, hyper-heuristics found a place to overcome those difficulties. Seen as a high-level methodology, the hyper-heuristics automatically provides an adequate combination of the provided components to solve the given problem filed efficiently.

1.1 Aims and Objectives

The objective of this project is to analyse the performance of selection hyper-heuristics on a natural combinatorial optimisation problem. we also extended the project and present a selection hyper-heuristics framework that could test with different combinations of acceptance operators on all kinds of benchmark functions as problem fields to evaluate their performance.

Therefore, we have two major objectives in the project:

1. To verify the performance of hyper-heuristic for a natural combinatorial optimisation problem.
2. To verify the correctness and robustness of the hyper-heuristics framework.

1.2 Overview of the Report

The rest of the paper is organized as follows. As the readers may not be experts in combinatorial optimization and hyper-heuristics, we will start with some basic notions. In the next chapter, we will formally present a related technology survey, the problem in combinatorial optimization, heuristics, and hyper-heuristics. In chapter 3, we will analyze the project requirement and presents a simple selection hyper-heuristics program and its architecture to achieve all the objectives. Chapters 4 and 5 will introduce the design, implementation and test cases of the program. The experiment results are displayed and discussed in chapter 6. Finally, we will present a summarization of this project.

Chapter 2

Literature Survey

In this chapter, we will explain the concepts and review the related works from previous literature.

2.1 Combinatorial optimization

Combinatorial optimization is a topic that consists of finding an optimal object from a finite set of objects [1], optimization problems in which the feasible solutions are discrete and can be expressed using concepts from combinatorics (such as sets, subsets, combinations or permutations) and/or graph theory (such as vertices, edges, cliques, paths, cycles or cuts) [2]. It is a subset of mathematical optimization and a multidisciplinary research area that integrates three major scientific domains: mathematics, theoretical computer science, and management. As a result, the concepts of combinatorial optimization could, therefore, be explained from three aspect [3]:

- On the complexity of combinatorial optimization problems, presenting basics about worst-case and randomized complexity;
- Classical solution methods, presenting the two most-known methods for solving hard combinatorial optimization problems, that are Branch-and-Bound and Dynamic Programming;
- Elements from mathematical programming, presenting fundamentals from mathematical

The application scenarios can be found everywhere in engineering. Unfortunately, many of those problems are proven to be an NP-optimization problem which is considered impossible to be solved within polynomial time in practical. Some simple and classic problems are shown as follows:

- Job-shop Scheduling (JSP): given n jobs $J_1, J_2, \cdots, J_n$ of varying processing times, we need to be scheduled on m machines with varying processing power, while trying to minimize the makespan.

- Knapsack problem: Given a set of items, each with a weight and a value, determine the number of each item included in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.
- The Traveling Salesman Problem: given the (x, y) positions of N different cities, find the shortest possible path that visits each city exactly once.

2.2 Heuristics & Meta-Heuristics

The exact algorithms, which guarantee to find the global optima, must explicitly examine every combination in the search space unless they could guarantee no need to be examined. As the computational complexity theory had shown that many NP problems were likely to be intractable by exact algorithms. The search space would be extremely large at each iteration in the case of NP optimization problems. That makes exact algorithms impossible to found any solution within polynomial time. However, we often don't need an exact global optima of such problems in practice. The near-optimal solutions are also acceptable if they could be provided within a reasonable time. Hence the heuristic algorithm could be defined from 2 perspectives:

- An algorithm that could provide a feasible solution of each instance of a given problem field within a reasonable time. The derivation between the feasible solution and global optima cannot be predicted generally.
- A technology that searches for the best solution within the acceptable cost, but it can not guarantee to get the optimal solution, or even worse in most cases, can not explain the degree of approximate degree between the result and the optimal solution.

Although heuristics were believed to be the only hope to solve the optimization problem, the acceptance of heuristic algorithms was slow before it and achieved much success in solving the real-world computational search problem. A quote by Fred Glover by late seventies showed the common attitude to find an efficient heuristic:

"Algorithms are conceived in analytic purity in the high citadels of academic research, heuristics are midwifed by expediency in the dark corners of the practitioner's lair. . . and are accorded lower status. [4]"

The early research on heuristics was always built on intuitive and experiences. Simon and Newell [5] proposed a theory that the design of heuristics algorithms should focus on intuitive, insight and learning. In particular, a new type of heuristics called meta-heuristics attracted considerable research effort. In contrast to heuristics, the meta-heuristics build on simple problem-specific (local) search algorithms and aim at overcoming local optimality through some general-purpose mechanism. Examples include Evolutionary Algorithms (EAs), Ant Colony Optimisation (ACO) and Particle Swarm Optimization, which are all based on principles observed in nature but applied to optimization. A detailed classification of meta-heuristics are shown at 2.2.

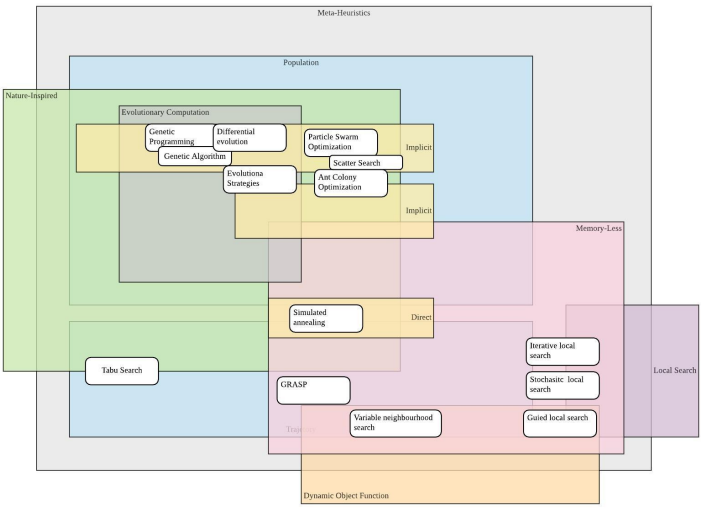


Figure 2.1: *Classification of meta-heuristics*

2.3 Hyper-Heuristics

Both heuristic and meta-heuristic methodology require problem-specific domain knowledge to design and determine the algorithm that could be applied to the new problems, or even a new instance of similar problems. It is still difficult to decide which of a set of heuristics to apply and what parameter setting would be a good choice to use. The theoretical understanding of which heuristic algorithm work efficient on a particular problem and why also does not provide any guidance. Therefore, designing an efficient heuristic algorithm for the given problem field is difficult and expensive. This fact proposed a new requirement, that is to automate the design and the tuning of the heuristic method to solve hard computational search problem [6, 7, 8]. It differentiates from the heuristic algorithms by operating on the search space of heuristics rather than searching directly for the solution to solve specific problem fields or their instance as 2.1.

	Heuristics	Meta-Heuristics	Hyper-Heuristics
Search Space	solutions to the underlying problem	solutions to a problem instance	heuristic algorithms
domain knowledge	Required	Required	Few or not required
Typical algorithms	Tabu search Hill climbing Greedy Algorithm	Random based hyper-heuristics Genetic algorithms Simulated annealing	Greedy based hyper-heuristics Hybrid hyper-heuristics

Table 2.1: *Difference between Heuristics, Meta-Heuristics and Hyper-Heuristics*

The term hyper-heuristics is first used by [9] in a peer-reviewed conference paper in 2001. The professor Cowling further developed the idea and applied it to scheduling problems in [10, 11, 12]. The hyper-heuristics were considered to be a high-level approach that, select and apply a low-level heuristic from a fixed set of low-level heuristics at each decision point to solve a given problem. However, the idea behind the hyper-heuristics is not new and could be traced back to the early 60s in many research fields such as Computer Science, Operational Research and Artificial Intelligence. They all investigated and developed approaches to solve

a different problem and could be classified into four types:

- Automated heuristic sequencing
- Automated planning systems
- Automated parameter control in evolutionary algorithms
- Automated parameter control in evolutionary algorithms

And with all the success achieved by applying hyper-heuristics, more and more research interest is attracted to this area. As a result, several tutorials and introduction of hyper-heuristics are published over the last few years, including [13, 14, 15, 16, 17, 18]. In particular, the classification of hyper-heuristics is proposed in [17] that provides a framework for investigating hyper-heuristic approaches. The classification considered two dimensions to classify a hyper-heuristic algorithm according to [17]:

- the nature of the search space of the heuristic algorithm.
- the different sources of feedback information.

From the perspective of the first dimension, we have (i) selection hyper-heuristic: iteratively choose from a set of heuristics to form a new heuristic algorithm. (ii) generation hyper-heuristic: Generate a new heuristic algorithm from the component of the existing one. The framework to construct a new solution can be further classified into two types: (i) constructive heuristic: consider a partial solution and improve the performance by adding new components. (ii) perturbation heuristic: consider a complete solution and improve the performance by modifying some of its components.

2.4 Selection Hyper-Heuristics

A selection Hyper-Heuristics is a high-level approach that chooses the best heuristic rule from a set of pre-defined heuristics at a different stage of optimisation process until global optima are found or termination criteria are satisfied. Most of the meta-heuristic algorithms could be generated by the selection hyper-heuristic approach such as evolutionary algorithm and simulated annealing algorithm. Besides, many successful applications [18] of selection hyper-heuristics have already been found and utilized in many natural optimisation problems such as traveling salesman problems, educational timetabling problems [19], personnel scheduling [20] and packing problem [21]. The selection hyper-heuristics described as follow always contains two modules:(i) Benchmark function and its mutation operators (ii) and acceptance operators. A detailed introduction to the two modules could be found in the following sections.

2.5 Benchmark Function and mutation operators

The combinatorial optimisation problems we used to test our selection Hyper-Heuristics are the benchmark functions. All combinatorial optimisation problems could be tested as

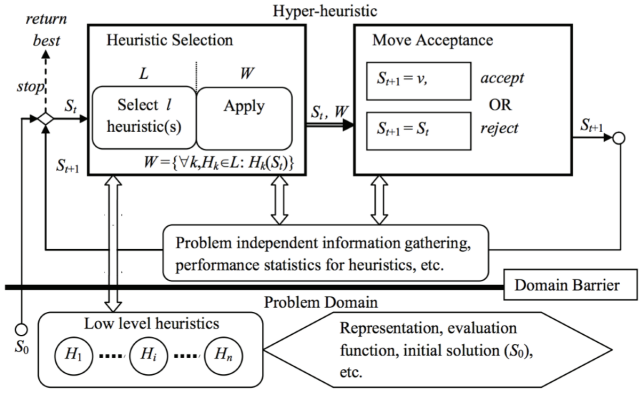


Figure 2.2: *Overview of Selection Hyper-Heuristics*

the benchmark functions theoretically. However, current research mainly focuses on toy benchmarks. Researchers could gain a deeper understanding of how the components of selection hyper-heuristics perform on such simple problems and analyzing its runtime rigorously. Two benchmark functions are commonly evaluated: (i) **OneMax** benchmark function, (ii) **LeadingOnes** benchmark function.. They are all 0-1 integer programming benchmark functions that operate on bit-string. The global optima could be achieved only when the bit-string includes no non-**1** bit.

The mutation operators depend on the search space of benchmark functions. They aim to find the neighbour solutions of current solution in the search space of the benchmark function. Since both the **OneMax** benchmark function and the **LeadingOnes** benchmark function operates on the bit-string search space, they could share the same set of mutation operators such as **Flip-1-bit** and **Flip-2-bits** operators. Researches especially theoretical runtime analysis on the two benchmark function have already proven the efficiency of selection hyper-heuristics and found the best parameter for different problem instances.

2.6 Acceptance Operators

The acceptance operators are the low-level heuristic rules that determine how the mutation operators are evaluated and accepted. Selection Hyper-Heuristics will search and select from the pre-defined set of acceptance operators performs randomly based on the specified probability distribution and use it to select the mutation operator used at each stage of optimisation process. Lissovoi et al. introduced several classical acceptance operators: (i) **Simple Random** which picks a mutation operator randomly at each step based on the given probability distribution, (ii) **Random Descent** which applies improving mutation operator until it finds the global optima or stuck at the local optima. (iii) **Permutation** that generates a random sequence of mutation operators and apply them in its order. (iv) **Greedy**: evaluate all the available mutation operators simultaneously and applies one that finds the best solution.. Generalized method is also investigated in the [22] that run the classical acceptance operators for a fixed period of time instead of just one step.

Chapter 3

Requirements and Analysis

3.1 Notation

In this paper, we use uppercase letters to denote vectors and lowercase to represent its element. The empty set is written with a empty set symbol $\emptyset$ while other set is denoted by listing its elements inside curly braces: $X = \{v_1, v_1, v_3\}$. Besides, we defined the set of integers $[n] \coloneqq \{1, 2, \cdots, n\}$ and $[0 \cdots n] \coloneqq \{0\} \cup [n]$. Given a vector $v \in \mathbb{R}$ and an integer $i \in [n]$, then v_i denotes the i-th element of v. We use $\sim$ to denote "distributed as" e.g. if X is distributed as D we write: $X \sim D$.

3.2 Project Requirement

Selection Hyper-Heuristics select from a set of heuristics at each stage of optimisation problem has been proven to escape local optima efficiently for toy benchmark problems such as $\mathbf{Cliff}_d$ and $\mathbf{Jump}_m$ with specific parameter setting [23, 24]. This project requires to verify the performance of Hyper-Heuristics on optimisation problem, especially natural combinatorial optimisation problems. So we will need to implement a flexible selection Hyper-Heuristics framework so that all kinds of benchmark functions could be evaluated with varying parameter settings (Both different sets of heuristics and different probability distribution). Besides, benchmark functions should be implemented as an individual module that including toy benchmark problems as well as natural optimisation problems. Numbers of the experiment should be designed and conducted with all those benchmark functions and different parameter settings. The result of experiments will then be collected and analyzed to verify our assumption and to provide the optimal parameters for different problem space.

3.2.1 Selection Hyper-Heuristics

The simple selection hyper-heuristic algorithm described in [24] is the foundation of our implementation. It focuses on the bit-string optimization problem and will flip one bit randomly every time and accept it by **AllMove**(AM) operator with probability p and by **OnlyImprovement**(OI) operator with probability 1-p. In our implementation, we extended the benchmark function classes to all optimization problems and allows to have multiple mutation operators instead of only one. Moreover, the benchmark function classes module and

acceptance operator module are designed to be hot-plugging so experiments could be designed and conducted with all kinds of benchmark functions and varying parameter settings. The pseudo-code 1 presents the basic idea of our framework and will be the prototype of further implementation.

Algorithm 1 Simple Selection Heuristics Framework

procedure HYPER-HEURISTIC
  $x \sim H$ ▷ H denotes the search space of solutions
  while Termination criteria not satisfied **do**
    Acc $\sim D_p(AO_1, AO_2, \cdots, AO_n)$ ▷ Select Acceptance operators
    Var $\sim D_p(NO_1, NO_2, \cdots, NO_n)$ ▷ Select Mutation operators
    $x' \sim$ **Var**(x)
    if ACC(x, x') **then** $x \leftarrow x'$
  end while
  return b

In detail, the algorithm will first generate an initial solution of x for the given benchmark function class randomly in its search space. Afterward, it will apply the mutation operators and acceptance operators interactively as follow:

- Step 1: Pick a acceptance operator ACC_i with probability given by parameter p_i.
- Step 2: Pick one or several mutation operators $Var_{j1}, Var_{j2}, \cdots$ included in the benchmark function according to the selected acceptance operator ACC_i.
- Step 3: Evaluating the mutation operators by applying them Var_j1, Var_j2 on the copy of current solution to get candidate solutions x'_{j1}, x'_{j2}.
- Step 4: Assign the new solution x'_j to current solution x if any of mutation is accepted by acceptance operator Acc_i.
- Step 5: If the termination criteria are still not satisfied, go back to step 1.

In each iteration of the algorithm, we will count the number of mutations ever tested and increment the step by 1. The runtime and maximum goal ever reached will also be updated and recorded for later analysis. The termination criteria will be either one of the following two cases or both: (i) The current runtime has not exceeded the maximum runtime limit, (ii) The current fitness has reached global optima.

As we mentioned before, the two modules in the framework, (i) Benchmark function classes and (ii) Acceptance Operators are designed as hot plugging modules that could be loaded individually. The former defines a fitness function that indicates the progress of the current solution from the initial state and determines the mutation operators that could be applied to the benchmark function class. For bit-string optimization problems like **OneMax**, flip-1-bit and flip-2-bits function could be picked as mutation operators. For traveling salesman problem, exchange-2-edge would be a reliable mutation operator. The latter determines the number of mutation operators is evaluated and whether the mutation caused by selected mutation operators would be accepted or not.

Each component has multiple implementations and will be introduced in the following subsections.

3.2.2 Benchmark Function Classes

There are three kinds of benchmark function classes we will introduce in the project:(i) **OneMax** classes, (ii) **LeadingOnes** classes and (iii) Boolean satisfiability problem(abbreviated as SATISFIABILITY or SAT) classes. All of them are 0-1 integer programming problems in which the variables in the optimization problem are restricted to be either 0 or 1. We pick them for both convenience and robustness. They all use similar mutation operators such as flip 1 bit and flip 2 bits randomly since they all operate on a bit-string search space to find the best solution. Besides, the SAT problem is the first problem that has been proven to be NP-complete[25], which means all optimization problem could be reduced to the SAT problem. Although we only implement the former problem classes due to time constraints, other kinds of combinatorial optimization problem classes such as traveling salesman problem and knapsack problem could be added to the framework with their as the benchmark.

OneMax Problem Classes

The **OneMax** problem is one of the simplest optimization problem and the perfect one to start with. Many meta-heuristic algorithms like evolutionary algorithms use it to explain their basic idea and potential to solve a difficult problem. This kind of problem uses a sequence of integers consists of 0 and 1 as their initial solution and will find the global optima until the current solution is filled with only 1 and no 0 anymore. The classic **OneMax** problem calculates its fitness by counting the number of bits that valued as '1' in a given bit-string. The fitness function can be described as:

$$\textbf{OneMax} \coloneqq \sum_{i=0}^{n} X_i$$

Obviously the search space of solutions of this problem can be expressed as $\textbf{UNIF}\{0,1\}^n$. The mutation operators for **OneMax** problem classes could be (i) **F**lip**O**ne**B**it function: randomly flip 1 bit; (ii) and **F**lip**T**wo**B**its function: randomly flip two bits in the bit-string. Actually, this two mutation operators will be used in all the benchmark functions we implemented since all of them operates on the search space of integer vectors contains either 0 or 1.

Besides, this kind of problem could find the global optima only if the solution contains only '1' bits. We could use the number of '1' bits $|X|_1$ as an indicator of their fitness function. The fitness of classic **OneMax** function will monotony increases with the growth of number of '1' bit like figure 3.1. The fitness function of other variants of standard **OneMax** like $\textbf{Cliff}_d$ tend to raise with the increase of $|X|_1$ but have some gaps and traps in the middle of the fitness function. But as we mentioned before, they will reach the global optima if and only if the bit-string contains only 1 bit.

LeadingOnes Problem Classes

The **LeadingOnes** problem classes, on the other hand, works differently to find the global optima. As the name implies, this class of optimization problems have an increasing tendency with the number of consecutive 1-bits in the bit-string before the first 0-bit: **Ridge(x)**. The global optima is still the same as the **OneMax** problem classes that could be reached only if

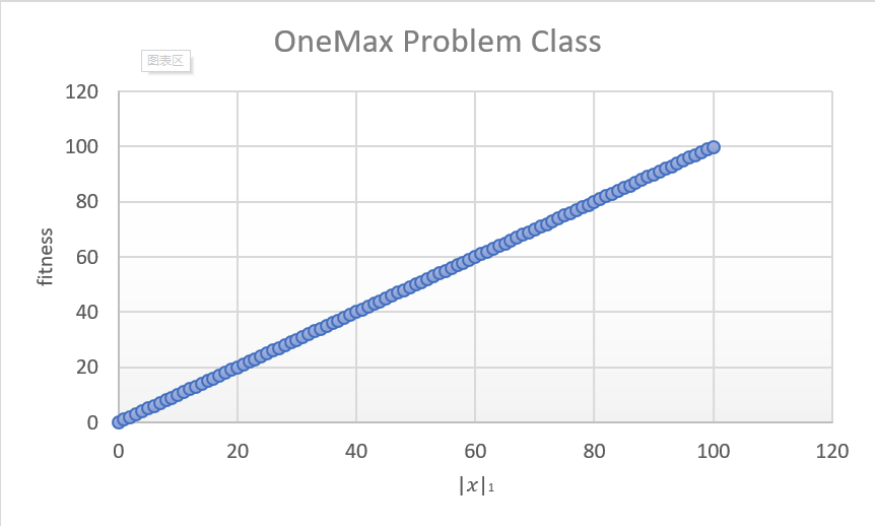


Figure 3.1: *Fitness function of **OneMax** problem classes*

the bit-string includes only 1 bits. The classic **LeadingOnes** algorithm could be described as follow:

$$\mathbf{LO}(x) \coloneqq \mathbf{Ridge}(x) = \sum_{i=1}^{n} \prod_{j=1}^{i} \mathbf{x}_j$$

The fitness function of standard **LeadingOnes** with **Ridge**(x) represented in the x-axis is presented at figure 3.2. And the search space and mutation operator of **OneMax** problem could also work for **LeadingOnes** problem. Instances of this problem classes such as **GapPath** and **Generalised GapPath** problem contains gaps where the fitness of solution is dropped dramatically and bounced back after few steps with the increase of **Ridge**(x). We will introduce them later in the design section 4.

SAT Problem Classes

The boolean satisfiability problem, abbreviated SATisfiability or SAT, is a classic natural combinatorial optimisation problem that determines whether the given boolean expression consists of n clauses is satisfiable or not as well as find the assignment for each variable in the boolean expression that makes the formula to be true. In particular, we will focus on **Max-SAT** problem, a generalised version of the SAT problem, that try maximizing the number of successful clauses and find the best assignment for each variable to find the maximum value. The reason to use **c** problem includes (i) SAT problem is the first optimisation problem that was proven to be NP-complete, which means all the NP problems could be reduced to the SAT problem. (ii) **Max-SAT** problem could provide a consistent fitness function by counting the number of satisfied clauses for analysis.

We will use an example to better illustrate the nature of **Max-SAT** problem. Let a set of boolean variables be denoted by $x_1, x_2, \cdots, x_i$ that could only be assigned with **true** or **false** and $C_1, C_2, \cdots, C_m$ represent the clauses. Any **Max-SAT** problem instance could

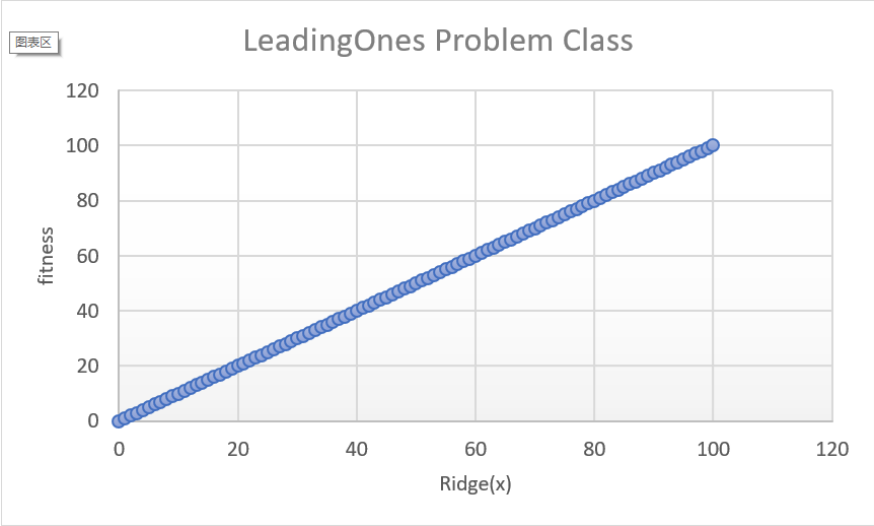


Figure 3.2: *Fitness function of* ***LeadingOnes*** *problem classes*

be describe as a conjunctive normal formula consists of several clauses connected by logic and operator with each other. Each clause is a disjunction of a boolean variables or their negation. Therefore, a **Max-SAT** problem instance with 4 variables and 5 clauses would be:

$$(\neg x_1 \vee x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4) \wedge (x_1) \wedge (x_3)$$

With $x_1 \coloneqq True, x_2 \coloneqq False, x_3 \coloneqq True, x_4 \coloneqq False$, the fitness for this **Max-SAT** problem instance would be 4 since 4 clauses evaluate to True. The global optima of this problem instance would be 5 and could be reached if $x_1 \coloneqq True, x_2 \coloneqq True, x_3 \coloneqq True, x_4 \coloneqq True$.

However, the fitness function of **Max-SAT** problem cannot be described as mathematical formula like **OneMax** problem class or **LeadingOne** problem class. In other words, we could not find a clear mapping relationship between any set related to the problem instance like the most natural combinatorial optimisation problem. If we let x-axis denotes the number of satisfied clauses and the y-axis denotes the fitness value, the fitness function could be only one global optima just like the standard **OneMax** function or a random hill-climbing curve that has multiple local optima that could only be escaped by non-elitism heuristic rules.

3.2.3 Acceptance operator

The acceptance operators are designed to decide the way to pick mutation operators of the specific benchmark function and whether to accept the current solution or not. In order to perfect our hyper-heuristic framework so that the framework could generate as many types of the meta-heuristic algorithm as possible, we divided the acceptance operators space to 4 quadrants by whether it is generalised and whether it accepts equal and inferior solution than the previous one:

1. **All Moves** (AM) and **Only Improvement** (OI): selects a mutation operator randomly and accepts it immediately (AM) / only if any improvement is found (OI) .

Figure 3.3: *Acceptance Operator Matrix*

2. **Greedy**: evaluates all available mutation operators and applies the one with the best improvement.

3. **Generalised Random/Gradient**: select a mutation operator randomly and applies the mutation for τ steps immediately (Random) / only if any improvement is found (Gradient) .

4. **Generalised Greedy**: evaluates all available mutation operators and accepts the one returns best solution for τ steps.

.

3.3 Analysis

3.3.1 Analysis of the Objective

Based on the project requirement, the project should develop the flexible Hyper-Heuristic framework including a benchmark function module and acceptance operator module. The benchmark function module should contains implementation classes of all kinds of benchmark problems including **OneMax** class, **GapPath** class and **Sat** problem. The acceptance operator module could implement a set of heuristics rules including: (i) **AM** and **OI** operators, (ii) **Greedy** operator, (iii) **Generalised Random/Gradient** operator and (iv) **Generalised Greedy** operator.. Besides, a random operator selector component should be de-

signed to select different heuristic rule based on given probability distribution. The structure of this project would be like 3.4.

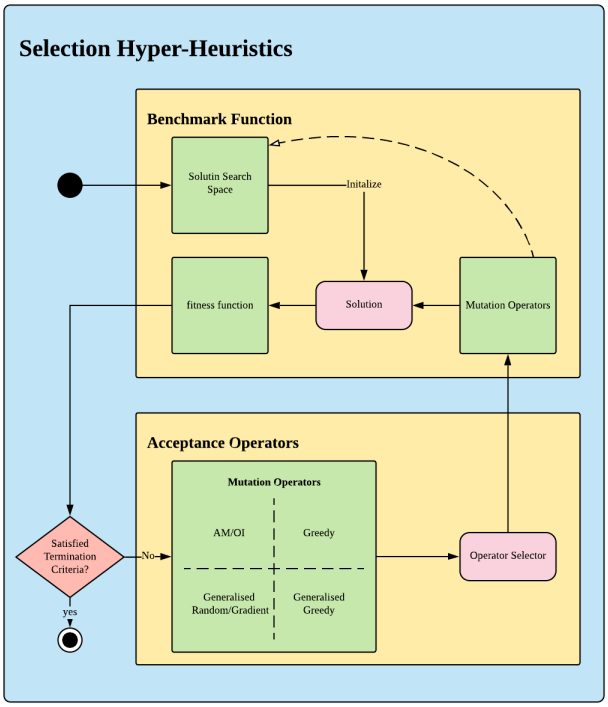


Figure 3.4: *Hyper-Heuristics Framework*

In addition, the experiments with this framework should be designed properly by setting a reasonable value to the problem size of the benchmark function and the probability distribution of the operator selector. According to the paper [24], the hyper-heuristics which switches between elitism (OI operator) and non-elitism (AM operator) for toy benchmark including **OneMax** problem and $\mathbf{Cliff}_d$ problem will find the global optima at its most efficient setting when $p = \mathbf{O}(1/n)$. The experiments [26] conducted on **Cliff** problem with only elitism and non-elitism acceptance operator shows that the optimal parameter for $\mathbf{Cliff}_d$ would be $p = 2/n$ while the approximate runtime of it would between $2n \ln n and 2.5n \ln n$. So our experiments on three problem classes will focus on elitism and non-elitism operators with $p = 1/n$ and $p = 2/n$. Experiments that only use elitism heuristic ($p = 0$) or non-elitism ($p = 1$) heuristic will also be tested as the control group.

3.3.2 Evaluation

To satisfy all the requirement and provide reliable conclusions, we will design the evaluation standard of the project as follow:

1. The Hyper-Heuristic framework should be designed and implemented correctly and running well. It should go through test-cases with different parameter settings and be able to reproduce the experiment result of previous research, especially [26].

2. The framework should provide functions to accept the parameter settings as its input and adjust the execution logic based on the given input.
3. The framework should restore the logging information as well as the output properly for debugging.
4. The results of experiments should be collected by the framework and analysed with statistic methods.
5. The framework should restore all the experiment results and generate analysis graphs.
6. Experiments should be tested with the framework on three benchmark problem class with different parameters

3.3.3 Risk Analysis

All the risks that might happen in this project are analyzed in the table 3.1. Each risk is measured with three parameters: (i) Likelihood: the probability of risk; (ii) Impact: the influence of risk on the project; (iii) Exposure: the overall risk score that takes both the likelihood and impact into account. Actions are also stated to prevent risks from happening and minimize their effects.

3.3.4 Ethical, Professional and Legal Issues associated with project

It is in everyone's interests to promote research ethics and support the integrity and reputation of research. We checked the objective and production carefully with the code of ethics (also with an exhaustive search) and find no obvious ethical, professional and legal issues associated with our project in all respects.

Rank	Risk	Likelihood	Impact	Exposure	Action
1	Inefficient Methodology	1	5	8	Investigate the related research and discuss the basic idea with tutor.
2	Underestimate the time cost to complete the framework	2	3	7	Follow the schedule defined before and adjust the plan by current progress.
3	Underestimate the time cost to conduct all experiments	4	2	6	Apply for high-performance computing account and run experiments simultaneous on multiple .
4	Overlooked the dependency between tasks	4	3	6	Finish the development of task before the milestone in the timetable and design test for each task.
5	Fail to meet all the requirement	2	5	5	Have a detailed analysis of realizability of each requirement and adjust the plan flexibly
6	Omission of a key task	1	8	4	Follow the timeplan and manage the task with a software engineering application like Trello
7	Fail to understand the requirements	1	9	3	Meet the project tutor regularly and discuss the detailed plan
8	Server Breakdown	1	6	3	Start the experiments early, divide them into independent groups and test them simultaneously
9	Code lost	1	5	3	Manage the project code with github and commit changes regularly

Table 3.1: *The Risk Analysis Table*

Chapter 4

Design

Design is the first and the most fundamental part of the whole process while the entire development of the project is based on it. We will introduce the structure of our framework as well as its input and output.

4.1 Selection Hyper-Heuristics Framework

As we have analysed in the 3.3.1, our implementation would contain a framework and two hot-plugging modules. This structure naturally involves the interaction between different components and would perfectly suit for object-oriented programming paradigm (abbreviated as OOP). The OOP programming paradigm packs related data and procedures into an object and uses its inner procedures to modify the data fields of the object. As a result, the paradigm would design a program by creating objects first and interact with them to finish the specific task. With OOP paradigm, our program is design as figure 4.1

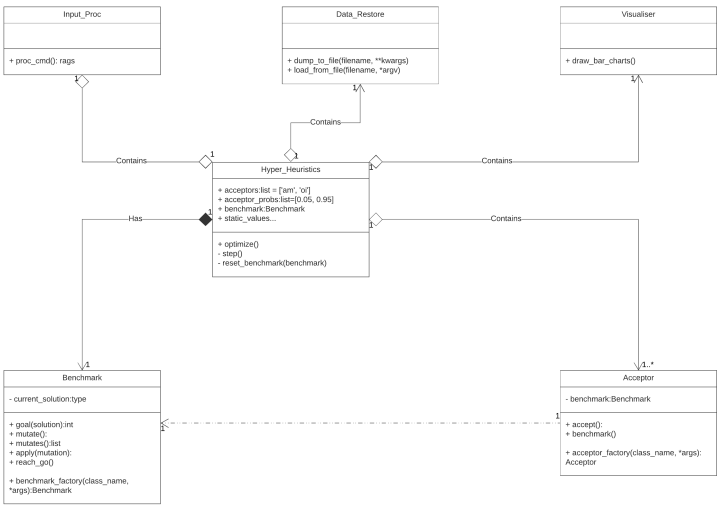


Figure 4.1: *Class Diagram for Hyper-Heuristic Framework*

As the figure is shown, the framework is consist of six modules:

1. Hyper-Heuristics control module
2. Benchmark module
3. Acceptance operator module
4. Input processing module
5. Data restore module
6. Visualiser module

We will go deep into each module in the later sections:

4.2 Hyper-Heuristics control module

The Hyper-Heuristics framework control module is implemented as `Hyper_Heuristics` class in the figure 4.1. As its name implies, it acts as the main control component to interact with all other modules in our program. The control module will first receive the parameters passed from the input processing module to initialise its data fields. In detail, the data field `benchmark` will be initialised with exactly one benchmark problem instance that inherits from Benchmark virtual interface. The acceptance operator list `acceptors` will be assigned with at least one acceptance operator that inherits the acceptance operator virtual interface while its probability will be initialised to the data field `acceptor_probabilities`. After initialisation, it will use `optimize()` function to start the procedure described in algorithm 1 and invoke `step()` function every time to iterate. The statistic data collected during execution would be saved by the data restore module to a specified file. Finally, the visualiser module would use the collected data to drawbar charts for visualisation.

4.3 Benchmark Function Module

In order to test all kinds of benchmark functions with our program, we will need to first generalise their common features. Combinatorial optimisation aims to find the global optima in the search space of a particular problem by definition. Therefore, all combinatorial optimisation problems would have a global optima and search space to search for feasible solutions. Besides, those problems should also have methods to mutate from a feasible solution to another one. In addition, some acceptance operators such as elitism and greedy operator would need fitness function to decide their choice. Therefore, we created a virtual benchmark function interface that has the following properties:

1. `current_solution`. This attribute denotes the current solution for this benchmark function. Our goal is to optimize the current solution to the optimal solution.
2. `goal(solution)`. This function gives the fitness value of solution.

3. `mutates()`. This function returns a list of mutation function that could find neighbouring solution of current solution.
4. `mutate(mutate_fun_name, *args)`. This function would invoke the a mutation operator function specified by `mutate_fun_name` with `*args` as its parameters.
5. `apply(mutation)`. This function apply the mutation caused by a mutation operator function to the current solution.
6. `reach_go()`. This function returns whether the current solution is the global optima or not. The control module described in 4.2 uses the result of this function as a termination condition.
7. `benchmark_factory`. This static method is designed as a factory method to generate a instance of all combinatorial optimisation problem.

All types of combinatorial optimisation problem could inherit from the interface and override the properties with their own implementation. Due to the time limit, we have only implemented bit-string benchmark problems for our experiment. A special function called `flip_n(n)` is created to return a function that flip n bit randomly in a bit-string instead of values. Three types of bit-string benchmark problems are developed in the program and they all use the function return by `flip_n(1)` and `flip_(2)` as their mutation operators. All problem classes are featured by their measurements but they all belong to 0-1 integer programming problems. We will introduce them respectively in the later sections.

4.3.1 OneMax problem classes

As we introduced before, the **OneMax** problem classes are measured by the number of `1` bits and reach the global optima when all the bit in the bit-string are converted to `1`. However, we set some traps and gaps to the fitness function to make it have more complex curve properties other than a straight line directly to the global optima. In other words, the optimisation algorithm will need to escape from several local optima to find the optimal value. The two variant benchmark function we used in the program is:

1. $\mathbf{Cliff}_d$ function
2. $\mathbf{Jump}_m$ function

The $\mathbf{Cliff}_d$ function is defined as follows:

$$\mathbf{Cliff}_d(x) = \begin{cases} \mathbf{OneMax}(x), & \text{if } |x|_1 \leq n - d, \\ \mathbf{OneMax}(x) - d + 1/2, & \text{Otherwise} \end{cases} \tag{4.1}$$

It is first proposed by [27] as an instance that the non-elitism evolutionary algorithm performs better than elitism. The fitness function that uses the number of `1` bit as an independent variable would rise until a local optima is reached. Then the fitness function will drop dramatically for one point and rise again monotonically to the global optima. The optimisation process could use elitism operators to rise with the fitness-increasing slope efficiently. However, the local optima could only be escaped by accept non-elitism operators

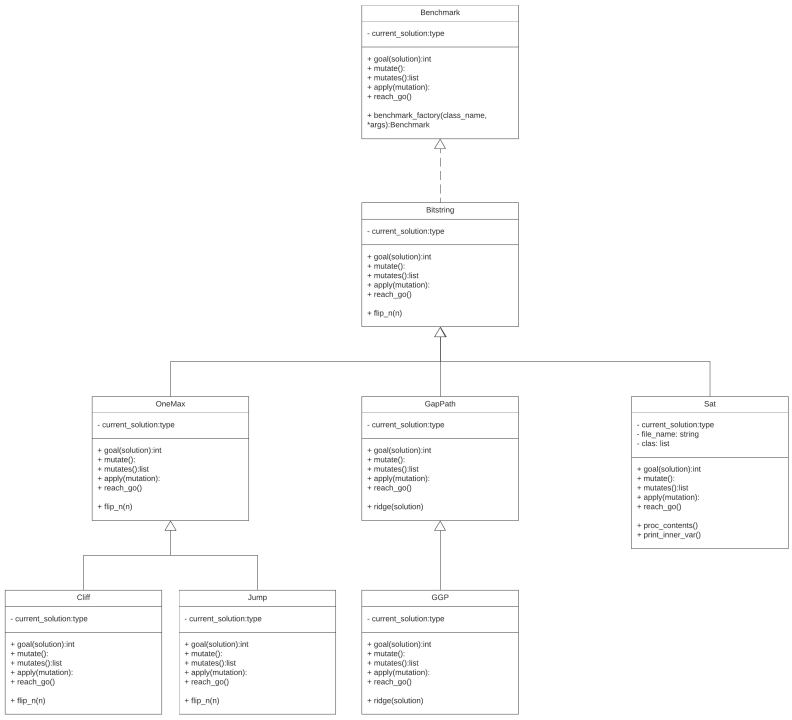


Figure 4.2: *Class Diagram for Benchmark Module*

at least once. The parameter `d` represents the point where the fitness function drop from the previous fitness-increasing slope to another. This toy benchmark function stands for the real-world problems that have narrow basins in the optimisation process before reaching the global optima.

Another variant of standard **OneMax** function, the $\mathbf{Jump}_m$ benchmark function could be described as follows:

$$\mathbf{Jump}_m(x) = \begin{cases} n + m, & \text{if } |x|_1 = n, \\ m + \mathbf{OneMax}(x), & \text{if } |x|_1 \leq n - m \\ n - \mathbf{OneMax}(x), & \text{Otherwise} \end{cases} \tag{4.2}$$

This benchmark function also has a fitness-increasing slope at the beginning that leads to the local optima. However, the second slope after the drop is fitness-decreasing. After the second slope hits the horizontal axis, it bounces back to the global optima. The optimisation process must accept many non-elitism to jump from the local optima to the global optima instead of just once. Natural optimisation problems that have a basin followed by a rapid increasing ridge would benefit from the experiments designed on the $\mathbf{Jump}_m$.

Two variants of **OneMax** function differed in the process that leads to the global optima after the drop from local optima. The $\mathbf{Cliff}_d$ function guides the optimisation process by the second fitness-increasing slope to the global optima while the $\mathbf{Jump}_m$ function would make

Figure 4.3: *Fitness function for* $\boldsymbol{Cliff}_d$

the optimisation process stuck between the local optima and fitness-decreasing slope.

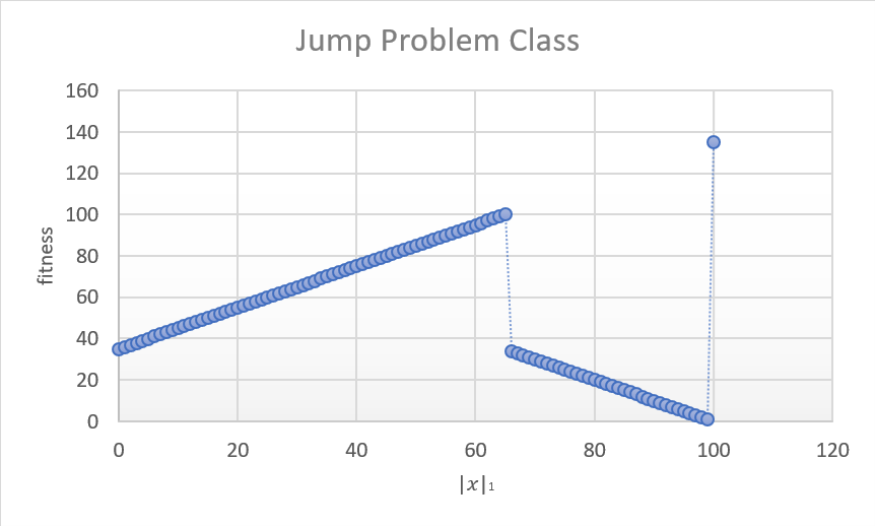


Figure 4.4: *Fitness function for* $\boldsymbol{Jump}_m$

4.3.2 LeadingOnes problem classes

Although the **LeadingOnes** problem class could also reach the global optima when the bit-string contains only `1` bits, they are differed from the **OneMax** problem classes in the order to flip `0` bits to `1` bits. The **LeadingOnes** problem classes could only increase their fitness by increasing the number of leading consecutive **1** bits iteratively and keeping the rest of bit-string as `0`. Here we introduce two variants of standard **LeadingOnes** function:

1. **GapPath** function

2. **GGP** function

The **GapPath** function proposed by [23] is defined as follow:

$$\textbf{GapPath}(x) = \begin{cases} \textbf{ZM(x)} & \text{if } \textbf{Ridge}(x) mod 3 = 1, \\ \textbf{ZM(x)} + 2n\textbf{Ridge}(x) & \text{Otherwise} \end{cases} \tag{4.3}$$

where $\textbf{ZM(x)} := \sum_{i=1}^{n}(1 - x_i)$, and $\textbf{Ridge}(x) = \begin{cases} i & \text{if } x = 1^i 0^{n-i}, \\ 0 & \text{Otherwise} \end{cases}$

The fitness function has different equations under different conditions. When the number of leading consecutive **1** bits modulo 3 equals to 1, the fitness is inferior to another condition. Therefore, the optimisation process would need to jump over the first condition to follow the gradient to find global optima. Optimisation that only applies flip-1-bit operator or flip-2-bit operator could not jump over the gaps with elitism operator and would require infinite time to escape all the local optima with non-elitism operators. The optimisation process would need to switch between flip-1-bit operator and flip-2-bits.

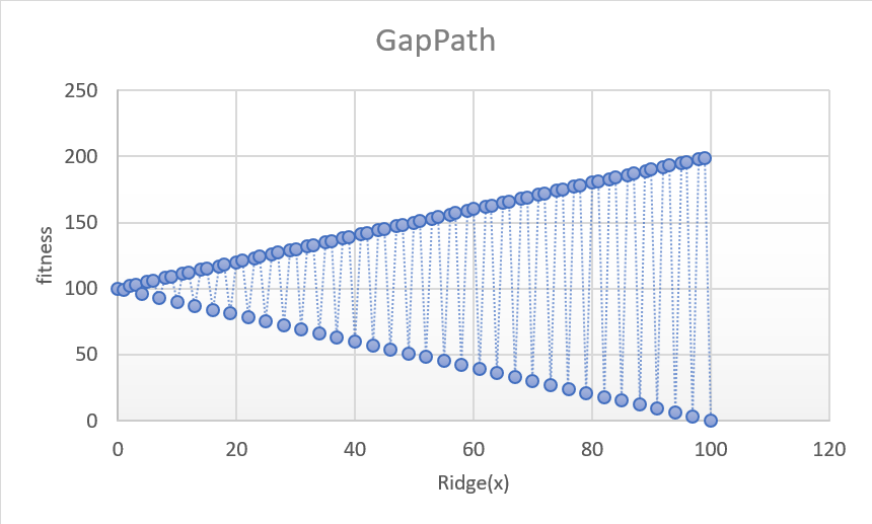


Figure 4.5: *GapPath function for* $\boldsymbol{Jump}_m$

On the other hand, we define the **GGP** function as follow:
Let **n** be of the form $n = d(2k + 1)$ for some $d, k \in \mathbf{N}$:

$$\textbf{GGP}(x) = \begin{cases} \textbf{ZM(x)} & \text{if } \textbf{Ridge}(x) \in S_{\mathbf{k}}, \\ \textbf{ZM(x)} + 2n\textbf{Ridge}(x) & \text{Otherwise} \end{cases} \tag{4.4}$$

Where $S_{\mathbf{k}} = c(2k + 1) + 1 - 2\beta \mid c, \beta \in \mathbb{N} : c \leq d, \beta \leq k$
This benchmark function is similar to the **GapPath** function but only differs in the condition to use the first equation. The benchmark function divided the optimisation process to **d** groups. Each group has one point which provides inferior fitness every other point except for the first and last points. As a result, the **GGP** benchmark function requires k consecutive flip-2-bits followed by a flip-1-bit at each search point of the form $1^{(c-1)(2k+1)}0^{n-((c-1)(2k+1))}$

to jump over gaps instead of switching between flip-1-bit operator and flip-2-bits operator. The optimisation process needs to learn to use flip-2-bits operator consecutively but still need to switch to flip-1-bit operator sometime.

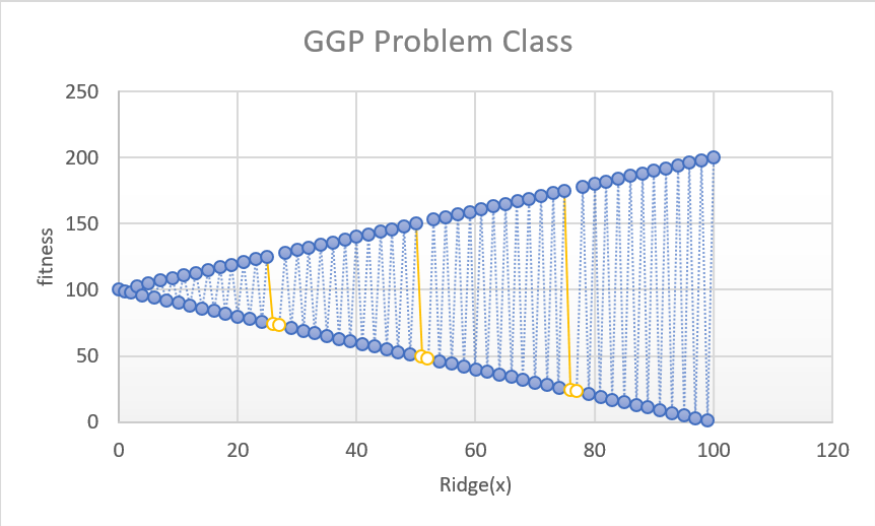


Figure 4.6: *GGP function for* $\boldsymbol{Jump}_m$

4.3.3 SAT Problem

We have also implemented the forementioned **Max-SAT** benchmark problem. Unlike other benchmark function we implemented, the **Max-SAT** benchmark problem is a natural optimisation problem and don't have a simple mathematical expression. All **Max-SAT** benchmark problem could be described in conjunctive normal form that connected a number of clauses with logic and operator $\wedge$ while the clauses are boolean variables connected with logic or operator $\vee$. A typical **Max-SAT** benchmark problem would be:

$$(\neg x_1 \vee x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4) \wedge (x_1) \wedge (x_3)$$

The instances of such a problem are stored in files of the **CNF** format to contain all the information needed to define a **Max-SAT** benchmark problem. **CNF** format is encoded by **ASCII** code and contains 2 major sections: the preamble and the clauses.

The preamble section is consists of comments and problem parameters to describe the current problem instance. Both of them are contained in lines and appear at the beginning of the **CNF** file. The comment lines that start with a lowercase character **c** gives the human-readable description about the instance and are ignored during the processing. The problem line begins a lowercase character **p** give the parameters of current instance: number of boolean variables and number of clauses. The **CNF** file could only contain one problem line in avoid of ambiguity.

The clauses section indicates the specific definition of the conjunctive normal form. It will appear immediately after the problem line. Each clause in the instance is separated by a whitespace character such as a space, a tab, and a new line. The clauses contain a sequence

of numbers. Each positive numbers **i** represents a boolean variable $\mathbf{x_i}$ while the negative numbers denotes the negation of a boolean variable $\neg\mathbf{x_i}$. For example, a clause like:

$$(\neg x_1 \vee x_2 \vee x_4 \vee \neg x_2 0)$$

could be defined as

$$-1\ 2\ 4\ -20$$

In order to parse the **CNF** format, we added two new functions: `proc_content()` and `print_inner_var()`. The former function read the content of given **CNF** file and parse it to setup the instance while the latter one print the parsed information from problem line for debugging.

4.4 Acceptance Operator Module

The acceptance operator module also implements a virtual interface to enable the capability to use all types of acceptance operator with our program. The common behaviors of acceptance operators are extracted as the `accept()` function to determine the mutation operators to evaluate and accept. All acceptance operators we implemented could are classified by two dimension: (i) The decision criteria to accept a mutation. (ii) The number of steps the accepted mutation operator will apply for.. We will introduce them in the following sections.

4.4.1 AM, OI and IE Operators

The **AM**, **OI** and **IE** operators could be classified into one category. They all select one random mutation operator to evaluate and accept for one step in one iteration in the algorithm 1. The three operators differ in the decision criteria to accept the mutation operator. The **AM** operator accepts the randomly selected mutation operator regardless of its performance. The **OI** operator, on the other hand, only accepts the mutation operator that improves in the fitness. The **IE** operator accepts the mutation operator that either improves or keeps in the quality. The **OI** and **IE** operators are considered elitism operators, the **AM** operator that works like random search is considered as a non-elitism operator.

4.4.2 Greedy Operator

The **Greedy** operator will evaluate all the mutation operators provided by the benchmark function in parallel and accept the one that performs the best. In order to evaluate all the available mutation operators, we used the multiprocessing technique and created a pool of processes to evaluate all the mutation operators. In detail, the process pool is established with the number of available mutation operators in the benchmark function. Every time the **Greedy** operator is selected, it will submit a job to each process. Therefore, each process would evaluate the performance of a mutation operator.

4.4.3 Generalised Greedy Operator

The **Generalised Greedy** operator is the generalised version of the **Greedy** operator. Instead of evaluating all the available mutation operators and apply the one that finds the best solution once in each iteration, the **Generalised Greedy** operator will apply the best-found mutation operator for a period of fixed length τ.

4.4.4 Generalised Random/Gradient Operator

The **Generalised Random/Gradient** operator is the generalised version of the **AM** and **OI** operators. This operator will choose uniformly a mutation operator and evaluate its performance. The **Generalised Gradient** operator will run the mutation operator for τ times if the mutation operator obtains any improvement. In contrast, the **Generalised Random** operator will run the mutation operator for a fixed period regardless of its quality.

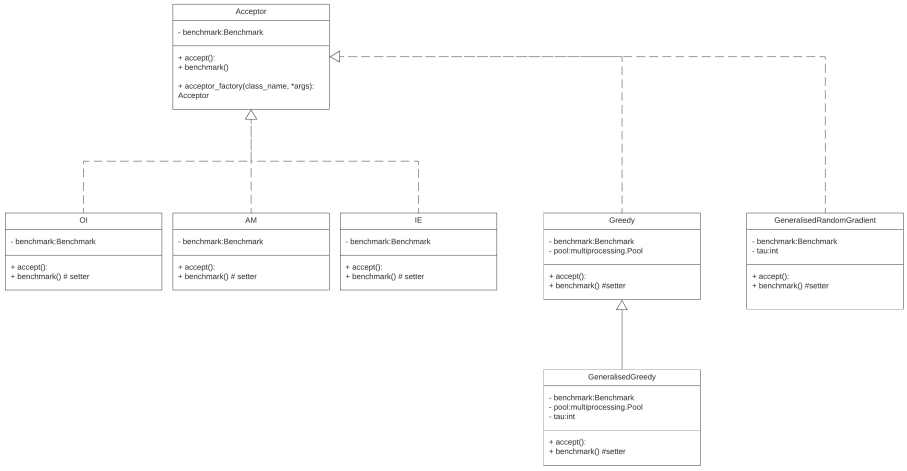


Figure 4.7: *Class Diagram for Acceptance Operator Module*

4.5 Input Processing Module

This module takes charge of reading input string from the command-line and parse the string into parameters to setup our framework and start our experiments. The acceptable parameters are listed as follows:

1. `acceptors` option.

   The option specifies the acceptance operators we will use in the program. The value of this option should be a list that consists of several names of acceptance operators. For example, the The default value for this option is `['AM', 'OI']`, which represents **AM** and **OI** acceptance operator.

2. `acceptor_probs` option.

   This option should provide the probabilities for each acceptance operator we specified in `acceptors` option. The value of this option should be a list of float point numbers represents the probability of acceptance operator specified in the `acceptors` option respectively. The default value would give equal probability to all the acceptance operator.

3. `benchmark` option.

   The option specifies the benchmark function classes on which we will test our framework. The parameter should be selected from [**OneMax**, **Cliff**, **Jump**, **GapPath**, **GGP**, **Sat**] and the default value would be **Sat**.

4. `benchmark_params` option

   This option makes sense only if the `benchmark` option specified is one of [**OneMax**, **Cliff**, **Jump**, **GapPath**, **GGP**]. It should provide the parameters for those benchmark functiosn. For example, the $\mathbf{Cliff}_d$ problem object is initialised by the construction function `Cliff(n, d)`. If the `benchmark` option specifies $\mathbf{Cliff}_d$ problem and `benchmark_params` option provides `100, 35` as its parameter, the $\mathbf{Cliff}_d$ problem instance will be created by `Cliff(100, 35)`.

5. `sat_files` option

   This option makes sense only if the `benchmark` option specified is `SAT`. In this case, the program will accept the value of `sat_files` option as the Unix style pathname pattern. The program will test on all the **CNF** files whose name matches the pattern. For instance, all the files such as [`/home/frt/sat/sat20/sat_01.cnf`, `/home/frt/sat/sat20/sat_02.cnf`, ..., `/home/frt/sat/sat20/sat_99.cnf`] will be tested for the given option `/home/frt/sat/sat20/sat_[0-9]2.cnf`.

6. `num_run` option

   This option specifies the number of times to run each problem instance. The default value is 20, which implies the program will run each problem instance 20 times.

7. `log_file` option

   This option specifies the file to restore logging information for debugging. The default value is 'hh.log'.

8. `max_mutation` option

   This option specifies the upper limit of number of mutations. It will act as one of the termination criteria in the Hyper-Heuristic framework described in 1. The default value for this option is 200000.

9. `dump_file` option

   This option specifies the file to restore the statistic data collected during the test. The data restore module is used to collect, analyse and save the data to the given file. The default value for this option 'hh_data.dump'

10. `show` option

    This option controls the visualisation switch. If the option presents in the input, the visualiser module will be used to drawbar charts from the statistic data stored in the log file.

4.6 Data Restore Module

This module saves the collected data to a file specified by `dump_file` option in the input processing modules 4.5. The process is divided into 3 steps:

1. Collect data

   In this step, the data restore module will collect several types of data for later analysis including maximum fitness ever reached in the single test, number of mutation steps to reach the maximum fitness and the runtime of a single test.

2. Analyse data

   After the data is collected, the data restore module will calculate the average and median value of the collected data.

3. Save data

   The analysed data will then be restored to a file specified by `dump_file` option.

4.7 Visualiser Module

This module utilised the python `matplotlib` module to visualise the experiment results. The module read data from the dump file and drawbar charts like figure 4.8.

The graphs show the result of an experiment that ran 20 instances 20 times on the uf75 dataset. The data are divided into 4 rows with each row contains the data of 5 instances. The blue bar at the first column shows the times for each instance that find the global optima within the maximum mutation step (specified in `max_mutation` option) while the orange bar shows the times didn't find the optimal value. The charts in the second column describe the maximum fitness they have ever reached for those who didn't find the global optima. The charts at the final column display the average mutation step takes to find the global optima for those who reach the global optima.

4.8 Overview

In summary, the whole picture of our program would look like figure 4.9. The workflow of our program is controlled by the center control module. It will first read input from the input processing module to set up the framework and create the needed instance from the benchmark problem module and acceptance operator module by their factory method (`benchmark_factory()` and `acceptor_factory`). The optimisation process is conducted by the control module with `optimize()` function and useful data such as maximum fitness are collected during the process. The data is then analysed and restored by the data restore

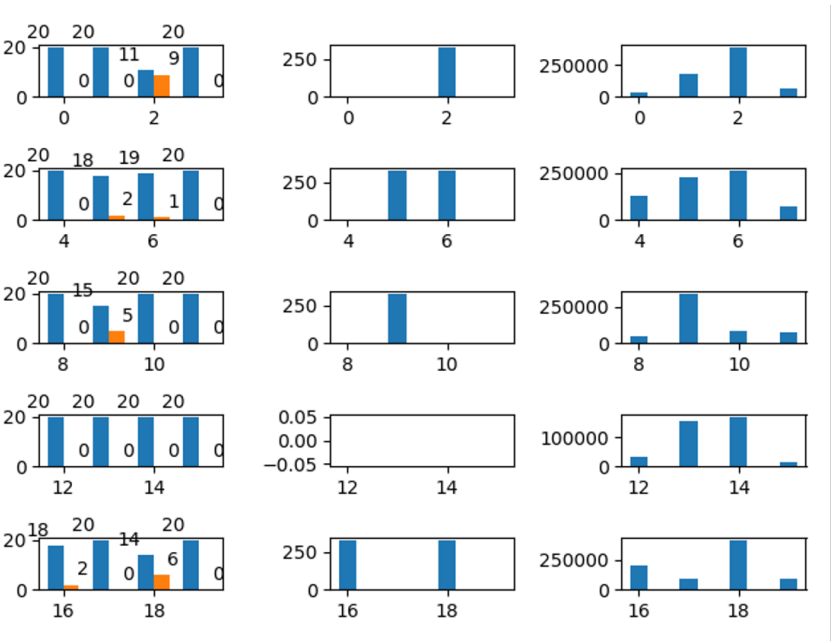


Figure 4.8: *An Example of Visualisation Graph*

module to dump files. The visualisation graphs will be generated if the `show` option is specified.

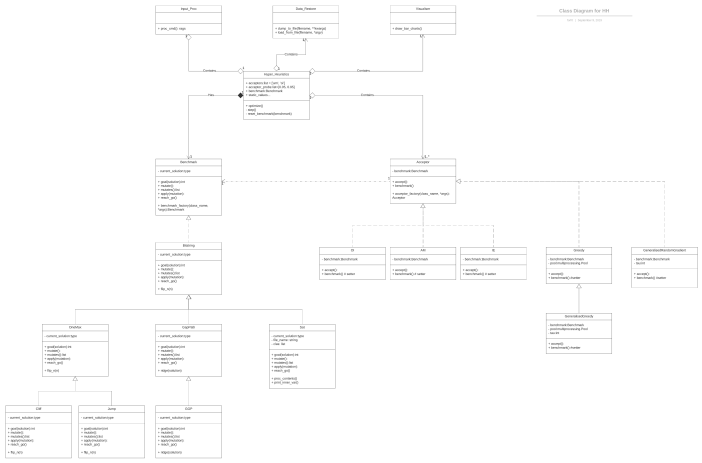


Figure 4.9: *Class Diagram for Hyper-Heuristic Framework*

4.9 Testing conditions

We have two major objectives in the project:

1. To verify the performance of hyper-heuristic for a natural combinatorial optimisation

problem.

2. To verify the correctness and robustness of the hyper-heuristics framework.

In order to achieve the first objective, we designed experiments with 3 datasets with different problem sizes (n=20, n=50, n=75), and executed 20 ran for 20 instances for each dataset. In each ran, we will collect the number of runs find the global optima, the average value of maximum goal ever achieved and the average mutation steps took to achieve the maximum goal of four different parameter setting:

1. $acceptance_operators \coloneqq [AM, OI], acceptor_probability \coloneqq [0, 1]$
2. $acceptance_operators \coloneqq [AM, OI], acceptor_probability \coloneqq [1, 0]$
3. $acceptance_operators \coloneqq [AM, OI], acceptor_probability \coloneqq [1/n, 1 - 1/n]$
4. $acceptance_operators \coloneqq [AM, OI], acceptor_probability \coloneqq [2/n, 1 - 2/n]$

The first setting that always accepts the elitism heuristic rule work like gradient search while the second set that only accepts the non-elitism heuristic rule work works like random search. The hyper-heuristic would be considered helpful if the last two settings perform better than the first two settings.

The second objective could be achieved by design experiments with **Generalised Greedy** acceptance operators.

Chapter 5

Implementation and Testing

5.1 Deployment

We use the Git to manage our project and github to store and mange our code. Anyone could download our project from this repository. Packages that are needed for the runtime environment are listed as follow and could be found in the requirement.txt:

1. numpy package which could convert a sequence of number to a number vector.
2. matplotlib package which provides the functions to draw visualisation graphs.
3. logging package which controls the minimum debugging level to restore log information as well as the output file.
4. statistics package which helps calcuating the average and median value of a list of numbers.
5. tictoc package which provides a easy-to-use runtime recorder.

The program could be launched by simply run the command `python main.py [options]`. The help message will be displayed if no option is given and the could be found in the A. The project will be carried on for quite a long time and any questions and suggestion are welcomed.

5.2 Testing

We designed the test cases for two major objective respectively. The validation test cases are designed to validate the correctness of our framework while the experiment test cases are designed to verify the performance of hyper-heuristics on the SAT problem. The validation test cases shown as table 5.1 mainly focus on the reproduction of previous research that already have clear conclusions. The experiment test cases shown in the table 5.2 contains the performance evaluation of 3 datasets in the Satlib with different problem sizes (n=20, 50, 75). The result of experiments will be analysed and discussed in the next chapter.

No.	Testing conditions	Expected Results	Actual Results
1	Test the [AM, OI] combination with [5%, 95%] on OneMax Benchmark function (n=100) for 10 times	Find the global optima very soon. OI operator accepts all the operators that convert 0 to 1	Meet the expectation
2	Test the [AM, OI] combination with [5%, 95%] on $\mathbf{Cliff}_d$ Benchmark function (n=100, d=35) for 10 times	Find the global optima very soon. The optimisation stuck until AM operator is selected to jumps over the gap of 1^{n-d}	Meet the expectation
3	Test the [AM, OI] combination with [5%, 95%] on $\mathbf{Jump}_m$ Benchmark function (n=100, m=35) for 10 times	Spend a long time to find the global optima . The number of 1 bits grows only when AM operator is selected after the point 1^{n-m}	Meet the expectation
4	Test the [AM, Generalised Greedy ($\tau = 2$)] combination with [5%, 95%] on **GapPath** Benchmark function (n=100, k=12, d=4) for 10 times	The Generalised Greedy jumps over most of gaps but get stuck at points $1^{x(2K+1)}0^{n-x(2K+1)} for x \in \mathbf{N} and x \leq k$ until the AM operator is selected	Meet the expectation

Table 5.1: *The Validation Tests Table*

No.	Testing conditions
1	Test the [AM, OI] combination with [1/n, 1-1/n] on **SAT** Benchmark function (n=20, 50, 75) for 20 instances 20 times
2	Test the [AM, OI] combination with [2/n, 1-2/n] on **SAT** Benchmark function (n=20, 50, 75) for 20 instances 20 times
3	Test the [AM, OI] combination with [0%, 100%] on **SAT** Benchmark function (n=20, 50, 75) for 20 instances 20 times
4	Test the [AM, OI] combination with [100%, 0%] on **SAT** Benchmark function (n=20, 50, 75) for 20 instances 20 times
5	Test the [AM, Generalised Greedy] combination with [5%, 95%] on **SAT** Benchmark function (n=20) for 20 instances 20 times
6	Test the [AM, Generalised Greedy] combination with [5%, 95%] on **SAT** Benchmark function (n=20) for 20 instances 20 times
7	Test the [AM, Generalised Greedy] combination with [5%, 95%] on **SAT** Benchmark function (n=20) for 20 instances 20 times
8	Test the [AM, Generalised Greedy] combination with [5%, 95%] on **SAT** Benchmark function (n=20) for 20 instances 20 times

Table 5.2: *The Experiments Table*

Chapter 6

Results and Discussion

6.1 Results of Experiments

We have collected the success rate and average runtime of all the experiments on three different datasets. The statistic data and visualisation graphs could be found at **??**. The success rate and average runtime are calculated and displayed as follows:

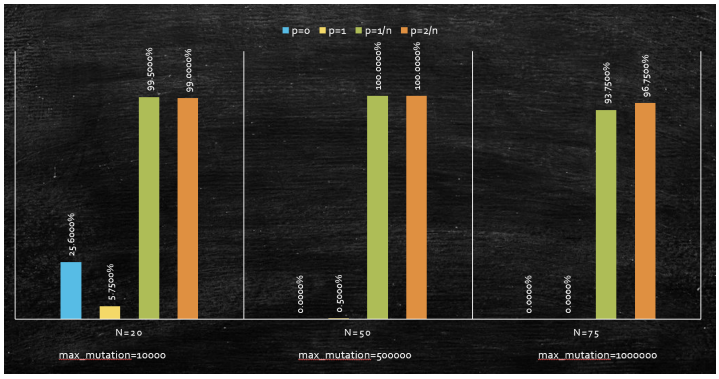
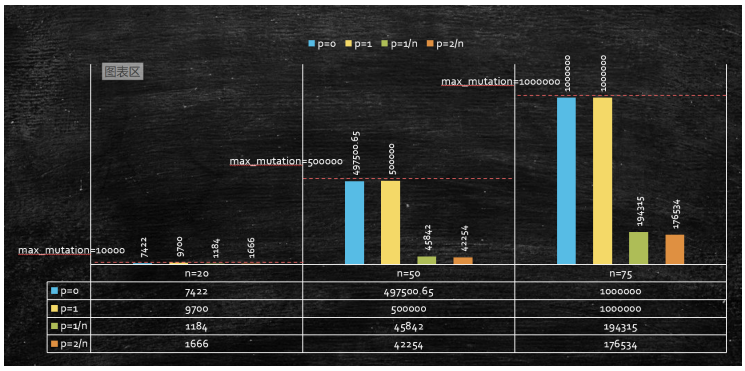


Figure 6.1: *Success Rate of finding global optima within max_mutation step*

The success rate of finding global optima within the maximum mutation step is shown as figure 6.1. As you can see, algorithms with $p = 1/n$ and $2/n$ have a much higher probability to reach the global optima. It is because the program will only accept the elitism strategy when $p = 0$. As a result, the program could find the global optima only if it's lucky and there are no local optima to escape from. For p=1, the algorithm did the random search, the program will accept all the mutation and bounces back and forth around the middle value of fitness. The program could find the global optima when the problem is small so the variance could reach the global optima within the limit of runtime. However, the average step for a variance to reach the edge will increase exponentially with the growth of problem size. As a result, the success rate of finding the global optima on the large problem would stick to 0.

Besides, we also collected the average runtime to find the best fitness. Since the experiments of mixed operators (p=1/n or p=2/n), always find the global optima while the single ones (p=0 or p=1) cannot. The best fitness for mixed operators is always larger than a single operator. However, Hyper-Heuristic with mixed operators could still find the best fitness

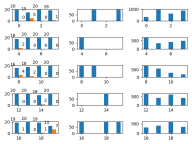


	n=20	n=50	n=75
p=0	7422	497500.65	1000000
p=1	9700	500000	1000000
p=1/n	1184	45842	194315
p=2/n	1666	42254	176534

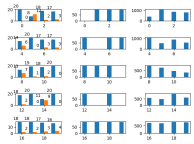
Figure 6.2: *Average Runtime to Find The Best Fitness*

much quicker than a single operator.

In addition, we have also tested different combinations of acceptance operators on **SAT** problem. The experiment results that use **Generalised Greedy** and **Generalised Random** are shown as figure **??**. From the visualisation graph and data from the dump file, we could find out that acceptance operators **Generalised Greedy** and **Generalised Random** works correctly just like **AM** and **OI** operators

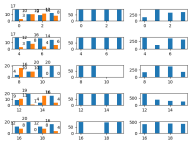


(a) 5% Generalised Greedy + 95% Generalised Random

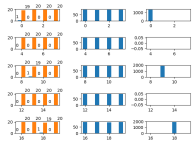


(b) 10% Generalised Greedy + 90% Generalised Random

Figure 6.3: *Mixed Operators on uf20*



(a) 0% Generalised Greedy + 100% Generalised Random



(b) 100% Generalised Greedy + 0% Generalised Random

Figure 6.4: *Single Operators on uf20*

6.2 Finds and Goals Achieved

From the experimental results shown above, we could generalise findings as:

1. The mutation step used to find the global optima increases significantly with the growth of problem size.

2. The hyper-heuristics that switch between elitism and non-elitism operator have a higher probability to find the global optima within a large enough runtime limit.

3. The success rate of a single operator drops toward 0 very quickly with the growth of problem size.

4. The mutation steps taken to find the best fitness for hyper-heuristics are much smaller than the gradient search or random search.

Combining the results of our experiments, we could give conclusions that:

1. The Hyper-Heuristics which switches between elitism and non-elitism performs much better than elitism or non-elitism operator alone for the NP-complete Max-SAT problem instances.

2. The Hyper-Heuristic framework works correctly with other acceptance operator combinations then **AM** and **OI** operators on **SAT** problem.

6.3 Future Work

Despite so many works have done in our project, the program still cannot generate all kinds of meta-heuristic algorithms for all types of problems such as evolutionary algorithm. It is because that the acceptance operators we implemented didn't cover all the low-level heuristic rules used in the meta-heuristic algorithms. For example, the evolutionary algorithm that employs the reproduction, mutation, recombination, and selection is still beyond the capability of this framework. The **Greedy** operator could generate a population-based algorithm but we still need recombination heuristic rule to generate evolutionary algorithm. What's more, adaptive mechanisms that consider more parameters related to the optimisation process such as momentum could also be further investigated and integrated into the framework.

Besides, all the benchmark function we tested belong to 0-1 integer programming problems. They all search in the bit-string search space and use **flip-1-bit** and **flip-2-bits** operators as their mutation operators. With the virtual benchmark function interface we provided in the framework, all combinatorial optimisation problems could be used as benchmark functions. Therefore, more benchmark problems, especially natural combinatorial optimisation problems could be implemented to test the completeness of our framework and gain a deep understanding of the nature of optimisation process.

In addition, experiments that use combinations with **Greedy**, **Generalised Greedy** and **Generalised Random/Gradient** operators should be conducted to find the optimal parameters for different problem instances. The performance on benchmark problems like $\mathbf{Cliff}_d$ and $\mathbf{Jump}_m$ could be improved by learning the patterns to switch between different mutation operators. A learning mechanism that learns from the feedback of optimisation process would provide a performance increase to many kinds of benchmark functions.

Last but not least, theoretical runtime analysis especially drift analysis could be introduced to estimate the time complexity of hyper-heuristics with a different combination of acceptance operators on different kinds of problems. By adding a drift analysis module with drift theorems, the framework could automatically estimate the expected runtime bound of optimisation process on particular optimisation problems.

Chapter 7

Conclusion

The two major objectives of this project are all satisfied with the result and analysis of our experiments. We proved that the selection Hyper-Heuristics which switches between elitism and non-elitism performs much better than elitism or non-elitism operator alone for the NP-complete Max-SAT problem instances. Besides, our selection framework could test different combinations of acceptance operators on different benchmark functions.

- Chapter 1(Introduction): This chapter introduces the related background information and the aims of our project.
- Chapter 2(Literature Survey): This part elucidates the selection hyper-heuristic and the Cliff function from the past literature, and highlights the algorithm and optimisation function that will be discussed in this project.
- Chapter 3(Analysis): Expanding the brief introduction of aims in Chapter 1. Stating what will be covered in this project and the evaluation standard.
- Chapter 4(Design and Implementation): The design of the implementation and the parameter value will be mentioned in this chapter.
- Chapter 5(Deployment and Testing): This chapter gives a detailed description of the methods in the program and the test case for the experiments.
- Chapter 6(Results and discussion): The experimental results are displayed in the figures in this chapter, also there is a discussion and comparison based on experimental results.
- Chapter 7(Conclusions): This chapter is a summary of the previous chapters.

Bibliography

[1] Alexander Schrijver. A first course in combinatorial optimization. *Choice Reviews Online*, 42(03):42–1619–42–1619, 2013.

[2] Adam N. Letchford. What is the meaning of combinatorial optimization?

[3] Vangelis Th. Paschos. *Applications of combinatorial optimization*.

[4] Fred Glover. Heuristics for integer programming using surrogate constraints. *Decision sciences*, 8(1):156–166, 1977.

[5] Herbert A Simon and Allen Newell. Heuristic problem solving: The next advance in operations research. *Operations research*, 6(1):1–10, 1958.

[6] Sam Allen, Edmund K Burke, Matthew Hyde, and Graham Kendall. Evolving reusable 3d packing heuristics with genetic programming. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 931–938. ACM, 2009.

[7] Belarmino Adenso-Diaz and Manuel Laguna. Fine-tuning of algorithms using fractional experimental designs and local search. *Operations research*, 54(1):99–114, 2006.

[8] Samad Ahmadi, Rossano Barone, Peter Cheng, Peter Cowling, and Barry McCollum. Perturbation based variable neighbourhood search in heuristic space for examination timetabling problem. *Proceedings of multidisciplinary international scheduling: theory and applications (MISTA 2003), Nottingham*, pages 155–171, 2003.

[9] Kendall G Cowling P and Soubeiga E. A hyperheuristic approach to scheduling a sales summit, 2000.

[10] Peter Cowling, Graham Kendall, and Eric Soubeiga. A parameter-free hyperheuristic for scheduling a sales summit. In *Proceedings of the 4th Metaheuristic International Conference, MIC*, volume 2001, pages 127–131. Citeseer, 2001.

[11] Peter Cowling, Graham Kendall, and Limin Han. An investigation of a hyperheuristic genetic algorithm applied to a trainer scheduling problem. In *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No. 02TH8600)*, volume 2, pages 1185–1190. IEEE, 2002.

[12] Peter Cowling, Graham Kendall, and Eric Soubeiga. Hyperheuristics: A tool for rapid prototyping in scheduling and optimisation. In *Workshops on Applications of Evolutionary Computation*, pages 1–10. Springer, 2002.

[13] Edmund Burke, Graham Kendall, Jim Newall, Emma Hart, Peter Ross, and Sonia Schulenburg. Hyper-heuristics: An emerging direction in modern search technology. In *Handbook of metaheuristics*, pages 457–474. Springer, 2003.

[14] Peter Ross. Hyper-heuristics. In *Search methodologies*, pages 529–556. Springer, 2005.

[15] Konstantin Chakhlevitch and Peter Cowling. Hyperheuristics: recent developments. In *Adaptive and multilevel metaheuristics*, pages 3–29. Springer, 2008.

[16] Edmund K Burke, Mathew R Hyde, Graham Kendall, Gabriela Ochoa, Ender Ozcan, and John R Woodward. Exploring hyper-heuristic methodologies with genetic programming. In *Computational intelligence*, pages 177–201. Springer, 2009.

[17] Edmund K Burke, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Ozcan, and J Woodward. Handbook of metaheuristics, volume 146 of international series in operations research & management science, chapter a classification of hyper-heuristic approaches, 2010.

[18] Edmund K Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and Rong Qu. Hyper-heuristics: a survey of the state of the art. *Journal of the Operational Research Society*, 64(12):1695–1724, Dec 2013.

[19] Edmund K Burke, Barry McCollum, Amnon Meisels, Sanja Petrovic, and Rong Qu. A graph-based hyper-heuristic for educational timetabling problems. *European Journal of Operational Research*, 176(1):177–192, 2007.

[20] Peter Cowling, Graham Kendall, and Eric Soubeiga. A hyperheuristic approach to scheduling a sales summit. In *International Conference on the Practice and Theory of Automated Timetabling*, pages 176–190. Springer, 2000.

[21] Peter Ross, Sonia Schulenburg, Javier G Marín-Bläzquez, and Emma Hart. Hyper-heuristics: learning to combine simple heuristics in bin-packing problems. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, pages 942–948. Morgan Kaufmann Publishers Inc., 2002.

[22] Andrei Lissovoi, Pietro S Oliveto, and John Alasdair Warwicker. On the runtime analysis of generalised selection hyper-heuristics for pseudo-boolean optimisation. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 849–856. ACM, 2017.

[23] Per Kristian Lehre and Ender Özcan. A runtime analysis of simple hyper-heuristics: to mix or not to mix operators. In *Proceedings of the twelfth workshop on Foundations of genetic algorithms XII*, pages 97–104. ACM, 2013.

[24] Andrei Lissovoi, Pietro S Oliveto, and John Alasdair Warwicker. On the time complexity of algorithm selection hyper-heuristics for multimodal optimisation. In *AAAI Conference on Artificial Intelligence*, 2019.

[25] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[26] Xiaoye Sun. Identification of the optimal parameter value for simple selection hyper-heuristic to efficiently optimise multimodal cliffd functions. https://www.dcs.shef.ac.uk/intranet/archive/campus/2017_2018/projects/msc/acp17xs.pdf. Accessed: 2019-08-30.

[27] Jens Jägersküpper and Tobias Storch. When the plus strategy outperforms the comma strategy and when not. *Proc. of FOCI*, 7:25–32, 2007.

Appendices

Appendix A

User manual

```
------------------------------------------------------------
USAGE: <PROGNAME> [-h] [-b BENCHMARK] [-s] [--sat_files SAT_FILES]
               [--acceptors ACCEPTORS [ACCEPTORS ...]]
               [--acceptor_probs ACCEPTOR_PROBS [ACCEPTOR_PROBS ...]]
               [--max_mutation MAX_MUTATION]
               [--num_run NUM_RUN] [--log_file LOG_FILE]
               [--dump_file DUMP_FILE]

OPTIONS:
    -s --show : show the statistic bar chart based on pervious data
    --acceptors : a list of acceptors chosen to be selected
    --acceptor_probs : probabilities of given acceptors
    -b --benchmark BENCHMARK : the benchmark to be tested on (Benchmark in {OneMax, Cliff, J
    --benchmark_params *Params : benchmark problem parameters if one of [OneMax, Cliff, Jump
    --sat_files : CNF files if SAT benchmark is selected
    --num_run : number of run for each problem instance
    --log_file : logging file
    --dump_file : file that store the json.dumps() result
    --max_mutation : maximum number of mutation
------------------------------------------------------------\
```