KING'S College LONDON

University of London

6CCS3PRJ Final Year
Search-Based Software Engineering - Tree Search

Final Project Report

Author: Zhenjie Jiang

Supervisor: Steffen Zschaler

Student ID: 1701278

April 14, 2020

Abstract

Search-based software engineering (SBSE) redefines Software Engineering problems as search problems and optimization methods, and Model-Driven Engineering (MDE) is a method which aims to provide an abstract representation of the structure and constraint of a particular domain. MDE Optimizer is a tool that combines both techniques for solving optimization problems and aims to make the process of solving such optimization problems much easier, for users such as domain experts. Current version of the tool uses evolutionary search to solve optimization problems specified by the user through domain-specific language (DSL).

The goal of this project is to implement tree-search in MDEOptimiser as an alternative option to evolutionary search which diversifies the options for the user. Tree search has the ability to be competitive and even outperform evolutionary search on certain optimization problem. Domain experts will be able to choose the more efficient method for the problem.

Originality Avowal

Acknowledgements

I express my special thanks to my supervisor, Steffen Zscaler, who has assisted me during my individual, providing many helpful advice and recommendations, which helped me greatly during this project. I would like to also thank Alex Burdusel, who provided a lot of assistance and resources for this project and taking out times answering questions regarding the project.

Contents

Chapter 1

Introduction

1.1 Project Motivation

In Software Engineering (SE), problems often involve a large number of constraints to be satisfied while finding a balance between potentially competing objectives. Because there are a large number of options need to be considered, finding and deciding on a single solution is often very difficult. For instance, "What would be the best balance between development cost and customer satisfaction? " is a sample SE problem with the conflicting goals of minimizing development cost and maximise customer satisfaction, where lower development cost can often result in lower customer satisfaction. However, this task can also be seen as trying to find an optimal solution by optimizing the objectives, which makes SE problems essentially optimisation problems. Therefore optimisation algorithms and techniques can be used for solving them. Search-based Software engineering (SBSE) is the methodology of reframing SE problems as search/optimisation problems and applying metaheuristic search-based optimisation techniques on them, to find optimal/sub-optimal solutions [7]. Search-based techniques allow developers to explore solutions much faster in comparison to manually, which lead to quicker discovery of better solutions.

Model-Driven Engineering (MDE) provides the ability to represent the complex software problem as meta-models, with their structural constraints well-kept [15].

MDE Optimiser is a tool that combines MDE and SBSE, which enable the ability to perform searches directly on the models, with the goal of making the process of solving and defining SE problems much easier. This Project aims to build on to the existing MDE Optimiser tool and bring in tree search algorithms as an alternative option for the existing genetic algorithm. This will enable users to choose between the different types of algorithms and use the more suited one for the problem.

1.2 Project Aim

In this project, the main objective is to implement and experiment with Monte-Carlos Tree Search algorithm in MDEO, with an optional task of implementing Hill Climbing algorithm.

1.3 Report Structure

Firstly, the report begins with an in-depth look through the background and explain the related topics, which provides some prerequisite for the project.

Secondly, the project specification and requirement is listed, which provides a clear overview of what is going to be conducted during the project.

Then the report goes through the initial designs and implementations of the algorithms. The possible variants in the implementations and why certain choices are made are discussed.

Next, the different possible configurations of the algorithms are tested, with their differences in behaviour analysed. New potential optimizations on the algorithms are experimented in order to find better implementations, with a final configuration of the algorithms decided.

Afterwards, We also discuss some legal and ethical concerns and talk about how this project complies with them.

During the final evaluation, the original NSAGII algorithm and the final implementation of the new algorithms are compared and evaluated. The strengths and weaknesses of each algorithm are analysed.

Finally, the report concludes with a summary of what is learned and discovered during the project, and discuss the possible future works in MDEO.

Chapter 2

Background

2.1 MDEOptimiser (MDEO)

SBSE is about search-based optimization to Software Engineering Problems, which essentially involves two parts [7] :

1. A representation that captures the problem.
2. Fitness functions which evaluate the solutions for the problem.

MDEO is a tool originally created by Alex Burdusel and Dr. Steffen Zschaler [2], which uses the concept of MDE to represent the problems as metamodels. A model provides a simplified representation of the original problem, and a metamodel is a further abstraction of the model, which represents the constraints and associations of the problem similar to a class diagram. The user specifies the problems by using a simple domain-specific language (DSL), where the user is required to provide a problem specification, which is easily definable using the implemented DSL [2].

MDEO also provides necessary interfaces for implementing fitness functions for the problems, which simplifies the process for the user to specify the details of the fitnesses functions, and makes the fitnesses the same format regardless of the problem. Each fitness function returns a number with the "double" typing as the calculated fitness, which is always implemented in a minimising format, i.e. the smaller the fitness, the better the solution. The user implements the specifics in how fitness is calculated for the problem. A multi-objective problem can be defined with multiple fitness functions, with multiple fitnesses to minimise.

Each problem would also typically have some constraints to be satisfied, which is defined similarly to the fitness functions. Each constraint returns a number which represents how far away the solution is from satisfying the constraints, with the specifics defined by the user. The smaller the number, the better, with 0 meaning the constraint is satisfied.

Furthermore, the user can specify the algorithm that is going to be used to solve the problem in the problem specification, as well as details such as termination condition. With MDEO acts as the translator between implemented algorithms and the problems, by adapting models and problems for the algorithm, implemented algorithms can run on all the correctly specified problems.

MDEO uses the MOEA Framework as the main framework for implementing algorithms and related functionalities, which provides interfaces and abstract classes for implementing optimising algorithms, especially genetic algorithms.

The tool can be run using eclipse as a launching platform or directly from the command line.

2.2 Genetic algorithm

Genetic algorithm is the type of optimization algorithm which is being currently used in MDEO, which mainly involves four processes[13]:

1. **Initialization:**

   Create an initial collection of solutions for the problem known as a population, which is usually generated randomly

2. **Selection:**

   A portion of the existing population is selected, which is usually according to the fitness of the solution (number representing how good the solution is).

3. **Reproduction:**

   The selected population from the previous processes are then used to breed/produce a new population, through methods such as crossover (combine two solutions from the selected population) and mutation (randomly altered the solution slightly)

4. **Termination:**

   Repeat process 2 and 3, until set termination condition (specified by the case by case) is met.

2.2.1 NSGA-II

NSGA-II is the implemented genetic algorithm, which is an algorithm designed for multiobjective problems. After the reproduction process, it adds the produced population to the old population and performs Non-dominated sorting (put more dominating solutions in the front of the population), and truncate the solutions at the bottom. The solutions are then picked to add to the new population for the next iteration of the algorithm based on a crowding factor, the solution that's the furthest away from other solutions are more likely to get picked. [5]

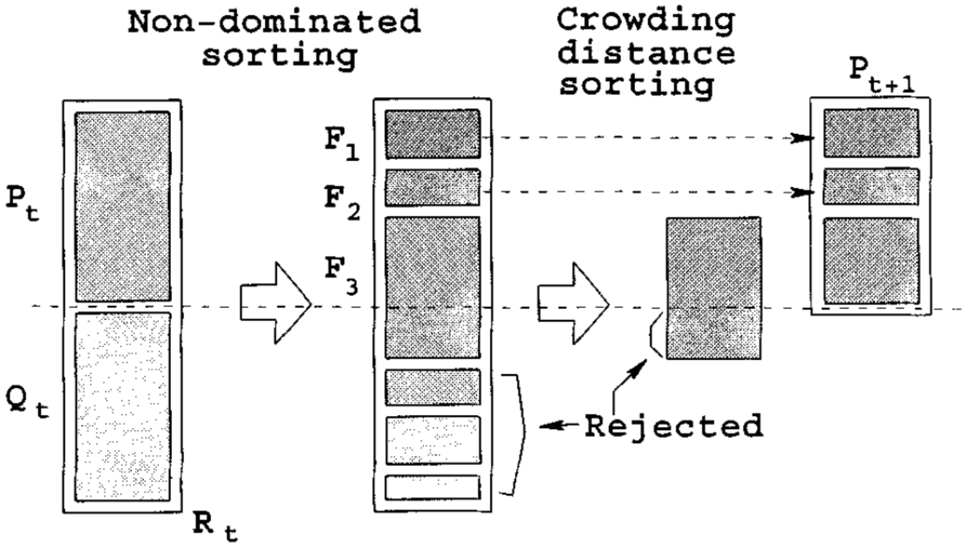


Figure 2.1: NSGAII process[5]

2.3 Tree-Search Algorithm

Tree-Search algorithms rely on the concept of a tree, which is a special type of graph, that contains a collection of linked nodes, and has no cycle. Tree-Search algorithms are algorithms designed to traverse and build these trees to reach certain objectives/nodes [6].

For MDEO, the nodes can represent the solutions of the problem, and the links between nodes can represent the relationship between the solutions. The tree-search algorithms traverse this tree to select a node, and creating new solutions based on the selected solution, the newly created nodes are linked to the selected node, with the new nodes being the children nodes, selected node as the parent node and the link representing this relationship. Due to the tree-search algorithm constantly maintains and updates this tree, which is effectively a history of all the solutions, tree-search algorithms have the ability to evaluate long term benefit versus short term benefit, as it can switch branches (path of creating new solution) to cover much larger search space, and roll back to older solutions. Compared to genetic algorithms, which only maintain solutions that do not keep a record of the discarded solutions, tree-algorithms are much less likely to stuck in local optimal. This makes tree-search algorithms an interesting alternative to NSGA-II.

2.3.1 Monte-Carlos Tree Search (MCTS)

MCTS is a stochastic heuristic tree-search algorithm, which has become extremly popular in the gaming field. It has shown to perform very well in both fully observable games like chess and go, and partially observable games like poker [12]. MCTS also does not have any constraints on the input, which should allow it to perform optimisation on a large variety of models. These characteristics should make MCTS an interesting addition to MDEO.

MCTS mainly includes 4 part[14]:

1. **Selection:**

   Start from the root (node of the initial solution), traverse through the tree and select a leaf node (a node with no expansion performed) according to a specific selection strategy.

   The selection strategy depends on the use cases and is usually based on the heuristic of the solution. The heuristic can often be equal to the fitness of the solution, in terms of a game, it would be the winning probability of that solution.

2. **Expansion:**

   Generate one or more new nodes from the selected node, add the new nodes on to the tree.

3. **Simulation:**

   Run the solution through the problem, to evaluate how good is the solution. In a game, it would be randomly playing different moves after the new solution/moves to evaluate its winning probability. For optimisation problems, it would evaluate the fitness of the solution, and check if the solution satisfies the constraints.

4. **Back-propagation:**

Traverse back to the root, and update the heuristics of the nodes along the way, according to the simulation [8]. For games, the winning probability for prior solutions/moves often changes when more solutions/moves are explored, and more simulations are run, hence the need for updating the tree. For optimisation problems, back-propagation can be updating the parent nodes' heuristic in conjunction with the new nodes, so the heuristics show how much solutions are improving, hence shows if the path worths to explore.

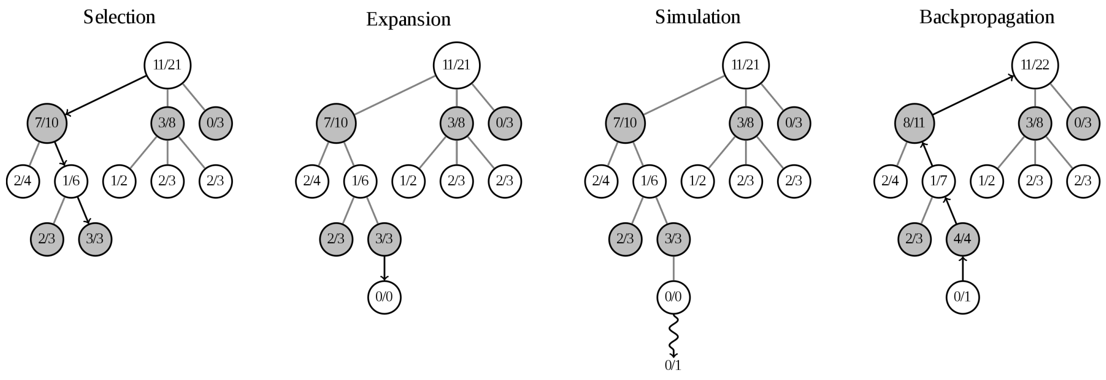


Figure 2.2: MCTS process[14]

MCTS requires functions for evaluating and calculating the fitness of the solutions, which is already implemented in NSAGII for comparing dominance. MCTS also keeps the record of all the explored nodes, and constantly re-evaluate them during backpropagation, which makes an interesting alternative to NSGAII.

2.3.2 Other Tree-Search Algorithms

The traditional algorithms, such as breadth-first search (BFS) and depth-first search (DFS) search the entire search space to find optimal solutions. These algorithms only explore the tree and lack the strategy for creating new solutions. Also, an optimization problem often has a vast number of possible solutions, the performance of these algorithms are likely very poor, considering they need to search through all the possible solutions. Especially in the case of DFS, where the algorithm can easily be stuck in an infinite loop; as DFS explores the branch until it reaches the end or a desired solution, and two solutions can have the nature of transforming into each other [6]. These features make them not suited to MDEO.

2.4 Hill Climbing

Hill Climbing is a local search algorithm that makes changes to a selected solution until a better solution is found, then moves on to that solution and re-apply the procedures [11]. Hill Climbing focuses on a local search space, instead of trying to cover the entire search space. T he small amount of explored solutions allow Hill Climbing to quickly discover better solutions, with the risk of potentially even better solutions being ignored.

This algorithm is added as an extra comparison point, as it possesses unique advantages. As an optimal solution for optimising problems is usually difficult or impossible to find due to the often massive amount of possible solutions [1], an algorithm that simply makes quick iterative changes to one solution might result in better potential result in realistic times.

Chapter 3

Requirement & Specification

3.1 Main Objective

- Implement Monte-Carlos Tree Search in MDEO
- (Optional) Implement Hill Climbing in MDEO

3.2 Secondary Objective

- Evaluate the performance and characteristics of the newly implemented algorithms

3.3 Specification

- Algorithms are implemented as part of MDEO's source code, directly integrated into the system.
- Evaluation is done by running algorithms through some case studies and evaluated with regard to how they perform in the given case study.

3.4 Requirement

- Implementations and modifications to MDEO should not hinder or limit MDEO's original functionalities.

  Part of the motivation of the project is to add more features to MDEO; hence MDEO's original functionalities should be kept.

- Implemented algorithms should be able to run on all problems originally runnable in MDEO.

  Algorithms do not need to perform well on all problems. However, users should have the choice to use any algorithms they desire for their problem.

- New algorithms should be consistent with the already implemented NSGAII in term of how to select and use them.

  MDEO aims to simplify the process of solving and defining optimization problems; therefore, dconsistency is essential not to make the tool too convoluted to use.

- MDEO should still be able to pass the original test cases after the new changes.

  The original test cases test if the components inside of MDEO function correctly, therefore they are essential to pass to ensure the tool works as intended

Chapter 4

Design & Implementation

MDEO is written in java and xtend (a dialect of java), it also uses the interfaces provided by the MOEA Framework as the baseline for the code implementations.

The algorithms implemented in this project extend the “AbstractEvolutionaryAlgorithm” interface and implement the “EpsilonBoxEvolutionaryAlgorithm” abstract class provided by the MOEA framework as the original NSGA-II algorithm despite not being evolutionary algorithms. MDEO has factories and adapters written for accommodating the interface and the abstract class to use the NSGA-II implementation. By also using them in the new algorithms, it allows them to seamlessly integrate into the system by removing the need to develop new adapters which saves more time for experimentation, despite the fact that neither extensions provide meaningful help for the new implementations

For the two Implemented algorithms, they both use java as the main language, and both store the best solution discovered and return that as the final output. Java is chosen as the primary language for implementation, as it is also used in the NSGA-II implementation, and has overall more robust support compared to xtend.

Functions for creating the new algorithms are added inside the “MoeaOptimisationAlgorithmProvider” class, which is a class in the original source code of MDEO responsible for adapting NSGAII to MDEO. The modification allows the tool to choose and execute the new algorithms. This class is implemented using xtend.

The implementation of MCTS is mainly separated into two part:

- Tree: The data structure that allows MTCS to perform search on.
- MCTS: The actual logic of the algorithm, which directs how the tree is searched and

constructed.

4.1 Tree

The tree contains the node class, which allows the tree-search algorithms to construct a tree and perform a search on. This part of the code can also be reused to facilitate future additional tree search algorithm implementations.

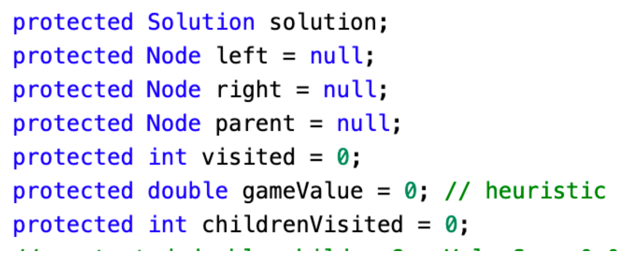
4.1.1 Node

Structure

Each node stores a solution, and the pointers to its parent node and children nodes. This structure makes the tree easily traversable, both backwards and forwards. In the initial implementation, each node only allows the construction of a binary tree with one left node and right node. Binary trees allow quicker exploration of the tree compared to trees that allow any number of children nodes, and allow search logistics to be implemented quickly. Many MCTS implementations have some limit on the amount of nodes to maintain a reasonable size tree, however an implementation that allows any number of nodes is later attempted in the next chapter.

Further information

To facilitate MTCS, each node also stores a number of different values/heuristics, such as the number of times the node is visited, the solution's heuristic and the number of times the children nodes are visited.



```
protected Solution solution;
protected Node left = null;
protected Node right = null;
protected Node parent = null;
protected int visited = 0;
protected double gameValue = 0; // heuristic
protected int childrenVisited = 0;
```

Figure 4.1: Node Structure

4.2 MCTS

MCTS is implemented under the "MCTS" class.

4.2.1 Heuristic

The initial heuristic for a solution is calculated by the function below:

$$-\sum f - 0.5 * \sum c$$

The first term is the sum of all the fitnesses multiplied by -1. Optimisation problems can often be multi-objective, which means each problem can have multiple fitnesses to be considered. However, the importance of each fitness is currently not specified in the problems for MDEO, which makes weighing the fitnesses difficult. As we want MCTS to support all problems for MDEO, where the fitnesses' weights are potentially all different, an unweighted sum of fitnesses is used, which essentially makes all problems essentially single-objective. As all fitnesses in MDEO are implemented in a minimising style, fitnesses can easily be summed up and used as one. However, because in MCTS, the larger the heuristic, the better, it is multiplied by -1. In the future, machine learning techniques can potentially be deployed to add weights to each fitness for multi-objective problems.

The second term is the sum of unsatisfied constraints in the number form. This term encourages MCTS to prioritize satisfying constraints and leads to a valid solution. Constraints are defined by the individual problems.

4.2.2 Initialization

During the first iteration of MCTS, the initial model for the problem is used as the first solution to the problem, which sets as the root of the tree with its heuristics calculated.

The NSGAII implementation uses a population of multiple solutions as the initial solutions, where each solution is achieved by performing one change (mutation) to the initial model. As a result, some of the original MDEO classes have to be altered to achieve the tasks mentioned above. The "MoeaOptimisationSolution" is a class responsible for transforming models to solutions, which originally performs mutation on the initial model upon creation to create an initial population for NSGAII. A new constructor is added inside the class to allow the ability to transform the initial model to a solution without mutation applied, which allows the initial mode to be used as a solution. A new creation function for "MoeaOptimisationSolution" class is also added in the “MoeaOptimisationProblem” class, which is responsible for creating "MoeaOptimisationSolution" instances.

4.2.3 Selection

During the selection stage, the tree is traversed based on a selection strategy to a leaf node (node with no child node), with that leaf node selected for expansion. Two different selection strategies have been considered for the MCTS implementation:

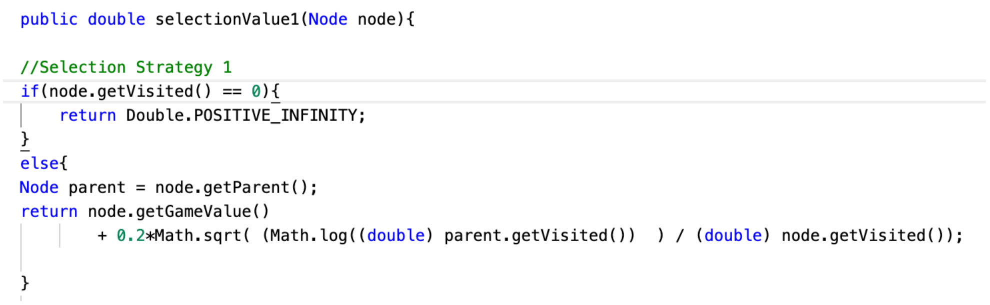
$$w_i + c\sqrt{\frac{\ln(N_i)}{n_i}}$$

The first strategy (S1) is the UCT (Upper Confidence bounds applied for Trees) algorithm, which is the current standard selection strategy for MCTS proposed by Kocsis and Szepesvari[9] and consists of two terms. The first term w_i is the heuristic of the current node, which encourages exploitation. The second term is made of the number of times the current node has been visited n_i and the number of times the parent node has been visited N_i, which is there to encourage exploration. If n_i is equal to 0, this term would return infinity. C is a constant for controlling exploration vs exploitation. The goal is to maximize this function, pick the nodes that give the largest value according to this function.

$$w_i + c\sqrt{\frac{\ln(n_i)}{\sum n_j}}$$

The second strategy (S2) is an alteration based on the original UCT algorithm introduced by myself, where the second term is changed to use the number of times the current node has been visited n_i and the sum of the number of times all of its children nodes been visited $\sum n_j$. The function return infinity if n_i or $\sum n_j$ equates to 0. The goal of the changes is to discourage deep exploration into bad branches, which are branches less likely to result in better solutions, while still encouraging branches to be explored enough to determine if the branch is bad. This goal is achieved by the second term decreasing with more children explored, which makes the heuristic the deciding factor in selection, where the heuristics are updated through back-propagation detailed later to represent the quality of the branches better.

There is also a selection strategy for single-player MCTS (SP-MCTS) introduced by Schadd et al [10], where a third term is introduced beyond the original UCT for better efficiency in single-player games such as puzzles. However, during limited testing, this strategy has shown significantly worse result and higher memory cost, it is not being considered.

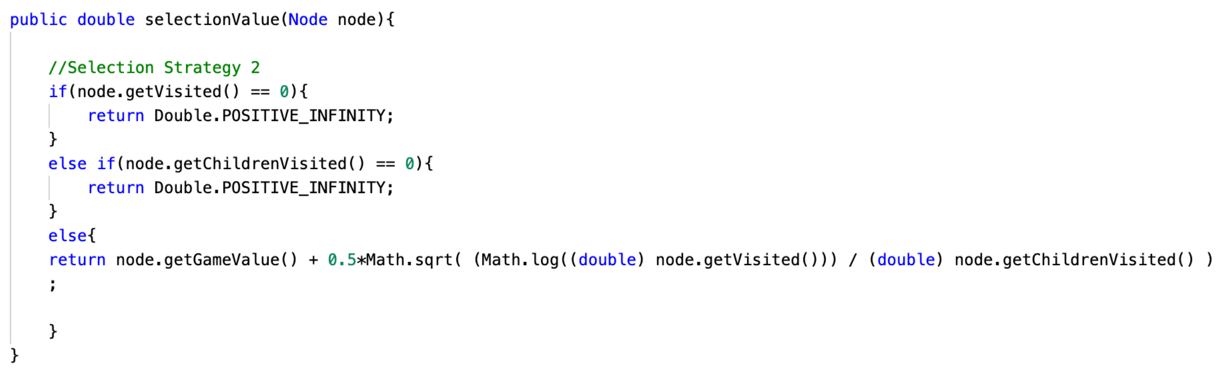


```
public double selectionValue1(Node node){

//Selection Strategy 1
if(node.getVisited() == 0){
    return Double.POSITIVE_INFINITY;
}
else{
Node parent = node.getParent();
return node.getGameValue()
        + 0.2*Math.sqrt( (Math.log((double) parent.getVisited())  ) / (double) node.getVisited());

}
```

Figure 4.2: S1 implementation



```
public double selectionValue(Node node){

    //Selection Strategy 2
    if(node.getVisited() == 0){
        return Double.POSITIVE_INFINITY;
    }
    else if(node.getChildrenVisited() == 0){
        return Double.POSITIVE_INFINITY;
    }
    else{
    return node.getGameValue() + 0.5*Math.sqrt( (Math.log((double) node.getVisited())) / (double) node.getChildrenVisited() )
    ;

    }
}
```

Figure 4.3: S2 implementation

4.2.4 Expansion

The first expansion strategy (E1) creates new nodes by applying one random transformation (mutation) on the selected solution. This is done by using the "RandomOperatorMutation-Strategy" class from the original MDEO source code, which performs this task.

The second considered expansion strategy (E2) performs the mutation mentioned before for a maximum amount of times and tries to create a new solution with better overall fitness than the parent solution. This choice is to reduce the number of bad solutions within the tree. The maximum upper bound is applied to prevent infinite loop in case no better solution exists, and the solution generated from the latest will be used. The number 50 is chosen for the initial implementation because it provides a sufficient amount of attempts.

4.2.5 Simulation

For MDEO, the solutions are essentially always "simulated", because fitnesses and unsatisfied constraints are already calculated and defined through MDEO using the fitness functions specified by the problem when the solutions are created, and they are always fixed. In games, every simulation can result in different winning probability due to the different moves that could be chosen by the opponent. However, for optimisation problems, this is unnecessary, as a better solution will always be a better solution, hence specific simulation strategy is not implemented.

4.2.6 Backpropagation

During backpropagation, we travel back from the selected node to the root. For this implementation, we also update the heuristics as:

$$w_i = \frac{\sum w_j}{k_i}$$

w_i is the current node's heuristic, $\sum w_j$ is the sum of the current node's child nodes' heuristics and k_i is the number of child nodes of the current node. Since the leaf nodes are the nodes that are going to be expanded, the purpose of other nodes' is only to guide the algorithm to the leaf node. The updated heuristics changes gradually upon the tree based on the leaf nodes. Therefore it allows the heuristic to be representative of how good the given branch is and encourages the selection strategy to select the node that is most likely to lead a better solution. As such, the heuristics also gradually decrease, if the leaf node's solution gets worse, thus discourages the exploration of branches that already potentially reached local optimal.

The sum of children visited is also updated to facilitate the second selection strategy.

4.3 Hill Climbing

Hill Climbing initially chooses the best solution out of an initial population as the first selected solution. The population is generated using the same method as NSGA-II by applying one random mutation to the initial model several times to create several initial solutions as an initial population. Next, Hill Climbing makes several random one step transformations (mutation same as the one used in MTCS' expansion) to the selected solution until a better solution is found and moves on to that solution and repeats the process of performing mutation. The mutation is performed for a maximum of 50 times each iteration to avoid an infinite loop; however, the amount of 50 is not important here, as its purpose is to start a new iteration. If a better solution cannot be discovered during this iteration, a second attempt will be made in the next iteration. The solutions are compared by using the dominance of the solution.

4.4 Automated Unit Testing

MDEO has around 231 automated unit tests, which are run during compilation, to check if each component works correctly. Extra 6 tests are implemented under the ”AlgorithmTests” class, which runs some sample problems using Hill Climbing and MCTS, to confirm they do not have bugs. All tests are passed. The specific behaviour of the algorithms are tested manually, by checking the solutions produced when running through the problems.

Chapter 5

Experimentation & Optimisation

5.1 Experimentation method

The experimentations are conducted by comparing the changes in performances between algorithms in the two selected case studies.

Results are gained by running the algorithm for 1 minute for 30 times, as results from individual runs can vary. The values are rounded to 3 decimal places.

5.1.1 Class responsibility Assignment Problem (CRA)

CRA is the first case study selected which is a problem introduced at the 2016 Transformation Tool Contest. The CRA problem involves transforming a procedural software application to an object-oriented architecture while maintaining the same features and aims to achieve a minimum coupling and good cohesion. A CRA value is used to measure the quality of the solution and acts as a single fitness for the solutions. Because MDEO is using a minimise framework (aims to reduce the fitness), the smaller the CRA value, the better. This makes CRA a single objective problem that aims to minimise the CRA value with a constraint of maintaining all the features [3].

A medium-sized software application model is used in the experiment with 20 attributes, 15 methods, 50 data dependencies and 50 functional dependencies.

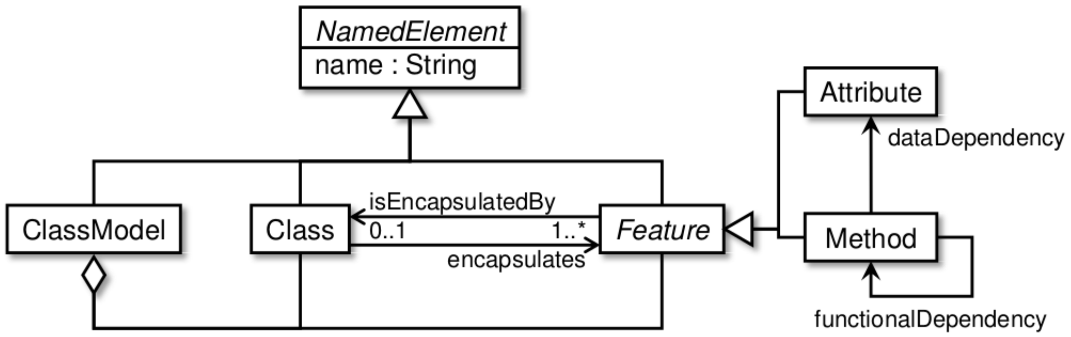


Figure 5.1: CRA metamodel[3]

5.1.2 Next Release Problem (NRP)

NRP is a case study about finding the best set of tasks to include in the next release for a software product. The goal of the solution is to minimise the cost and to maximise customer satisfaction [4]. Individual customers can have one or many desired software artefacts, and each software artefacts can have a dependency on other software artefacts. Cost and customer satisfaction each has a different value representations, customer satisfaction value is multiplied by -1, so the goal becomes minimising both values.

For NRP problem, the output of the algorithms is changed to include a list of best solutions compared by dominance instead of just one best solution. This change is made to allow a better comparison between the performance of the algorithms for multi-objective problems, as solutions often carry non-dominance relationship.

The results are compared using a scattered diagram, average cost and average satisfaction, with solutions from all 30 runs plotted.

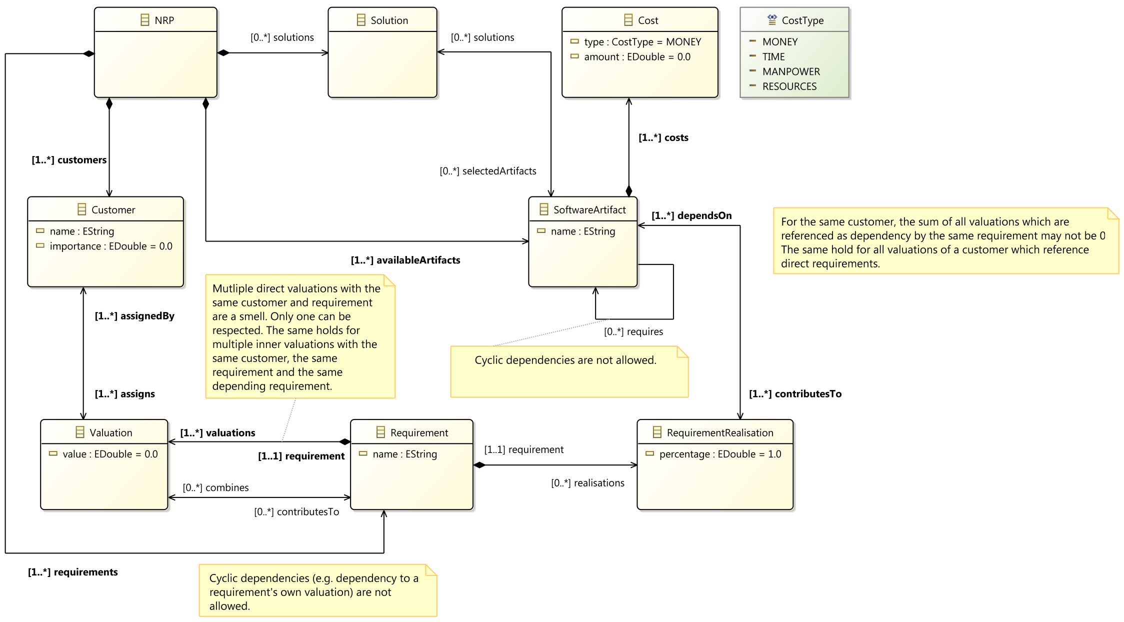

NRP
Solution
Cost
type : CostType = MONEY
amount : EDouble = 0.0
CostType
MONEY
TIME
MANPOWER
RESOURCES
[0..*] solutions
[0..*] solutions
[1..*] costs
[1..*] customers
[0..*] selectedArtifacts
Customer
name : EString
importance : EDouble = 0.0
SoftwareArtifact
name : EString
[1..*] dependsOn
For the same customer, the sum of all valuations which are referenced as dependency by the same requirement may not be 0 The same hold for all valuations of a customer which reference direct requirements.
[1..*] availableArtifacts
Mutliple direct valuations with the same customer and requirement are a smell. Only one can be respected. The same holds for multiple inner valuations with the same customer, the same requirement and the same depending requirement.
[1..*] assignedBy
[0..*] requires
Cyclic dependencies are not allowed.
[1..*] assigns
[1..*] contributesTo
Valuation
value : EDouble = 0.0
[1..*] valuations
[1..1] requirement
Requirement
name : EString
[1..1] requirement
RequirementRealisation
percentage : EDouble = 1.0
[0..*] combines
[0..*] realisations
[0..*] contributesTo
[1..*] requirements
Cyclic dependencies (e.g. dependency to a requirement's own valuation) are not allowed.

Figure 5.2: NRP metamodel[4]

5.2 MCTS Implementation Performance

Different combinations of the implementations mentioned in the last chapter are compared to check the difference in performances and behaviours.

5.2.1 Implementation Comparison

Constant c is set to 0.2 for S1, which is the value commonly used in go and many board games. For S2 it is set 0.5, to provide balanced exploitation and exploration.

CRA	Minimum	Maximum	Mean	Median
S1 & E1	-1.598	-0.628	-1.182	-1.191
S2 & E1	-1.677	-0.629	-1.282	-1.327
S1 & E2	-1.502	-0.577	-1.149	-1.178
S2 & E2	-1.877	-0.784	-1.270	-1.242

The CRA result shows that the altered version of UTC (S2) performed much better than the original, with lower CRA values across the board. This proves that it is better to not explore too deep into bad branches, as there is probably way too much possible solutions for an optimisation problem like CRA. Exploring branches that are very unlikely to produce a good result is not worth it. This result also indicates that the performance of S1 can potentially be increased by lower constant c, and might result in better results if each run lasts much longer, considering it will cover more search space.

E1's better performance over E2 is unexpected, especially considering the solutions generated by E2 should be better than E1, as E2 aims to create solutions with better heuristics. This is likely due to that better solution does not necessarily lead to another better solution. By restricting what solutions are created, solutions with a large amount of development potentials have a good chance to be ignored. Despite E2 managed to produce the best solution with S2, the overall results being worse is a good indication of this.

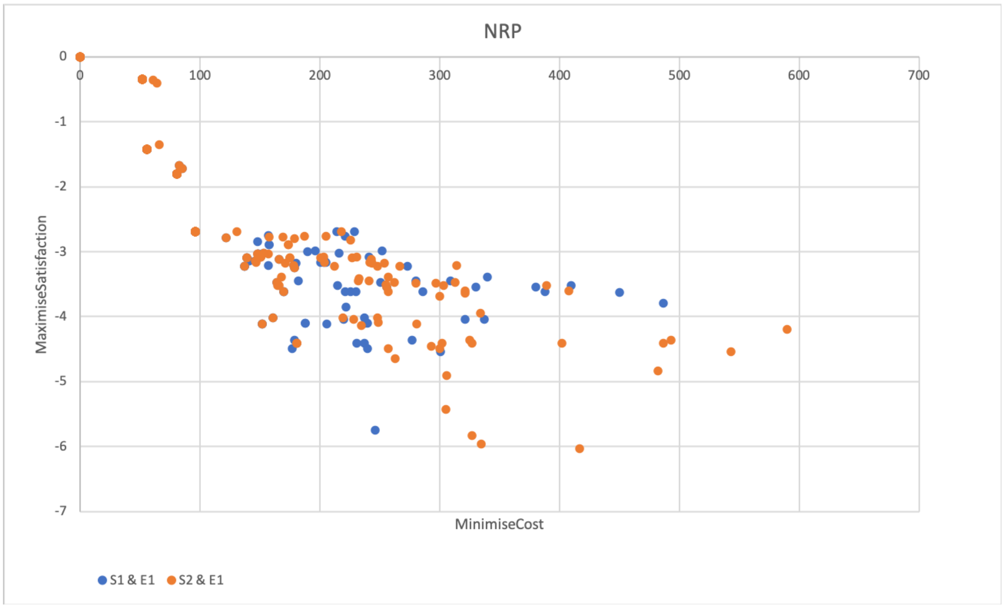


Figure 5.3: NRP result 1

NRP	Average MinimiseCost	Average MaximiseSatisfaction	Sum
S1 & E1	130.877	-2.383	128.494
S2 & E1	161.738	-2.728	159.010

The NRP result shows very little performance difference between S1 and S2. The sum of the average minimise cost, and average maximum satisfaction of S1 is smaller than S2, which indicates S1 might be better than S2. However as the average shows a non-dominance relationship between the two, the conclusion cannot be confidently drawn. Also, there are more dots for S2 than S1, which suggests S2 managed to find more solutions with a non-dominance relationship with each iteration, which is likely due to the lesser focus on worse branches.

5.2.2 Exploitation vs Exploration

With the hypothesis that a higher rate of exploitation might result in better performance for S1, both S1 is tested with C = 0.1, as well as C = 0.5 to confirm this hypothesis with E1.

CRA	Minimum	Maximum	Mean	Median
c = 0.1	-1.632	-0.751	-1.296	-1.370
c = 0.5	-1.649	-0.635	-1.196	-1.250

The result shows that while lower exploration rate does result in better averages and better consistency, it has a lower chance to discover a more optimal result. Interestingly both C values performed better than the original 0.2, which breaks the pattern; it is likely an anomaly or 0.2 is simply a bad ratio for the algorithm. To consolidate what has been discovered, S2 is also tested with C = 0.2 and E1.

CRA	Minimum	Maximum	Mean	Median
c = 0.2	-1.548	-0.932	-1.299	-1.348

The result follows the trend of higher exploitation leads to better consistency, and higher exploration can produce more optimal results, the result with S1 and E1 with c = 0.2 is likely an anomaly.

5.2.3 Heuristic Experimentation

The purpose of a heuristic is to help MCTS navigate the tree, in the case of the optimisation problems, it should be a path leading to an increasingly better solution. We hypothesize that a heuristic represents that more clearly might perform better.

5.2.4 Heuristic Function Experiement

A different heuristic which better represents the improvement between the solution and its parent solution is introduced here, which is calculated by using the difference of the original heuristics and updated using backpropagation similar as before.

The experiments are run using the original C values.

CRA	Minimum	Maximum	Mean	Median
S1 & E1	0.029	6.1875	2.180	2.138
S2 & E1	0.245	10.583	3.573	2.588

The result shows the solutions are inferior compare to the original heuristics, with S1 showing better performance over S2. This finding suggests S1 is perhaps more tolerant with worse heuristics compare to S2, which is likely due to S1 explores more, as heuristics mainly contributes to exploitation, which results in less impact on S1.

Some limited experimentation is done on increasing the weight of the new heuristics, as the bad results might be due to the difference between the solution's original heuristics being too small. However, no significant improvement has shown. This is likely caused by the significant variation in the new heuristics throughout the execution of the algorithm. A dynamic weighting function might be able to increase the performance of this heuristic, however, due to the time constraint, it has not been experimented.

5.2.5 Heuristic Update Experimentation

Continuing with the hypothesis, the original update function for heuristic during back-propagation changes according to child nodes' average heuristics, which shows how good the leaf nodes are for that branch. However, as the experiment with E2 shows, a good solution does not necessarily lead to a better solution, thus perhaps the heuristics should be updated according to the improvement in heuristics rather than the heuristics itself.

$$w_i = w_i + diff(w_i, \frac{w_l + w_r}{2})$$

w_i is the current node's heuristic, and $diff(w_i, \frac{w_l+w_r}{2})$ is the difference in heuristics between the current node and its right and left child's average. original back-propagation comes from the inclusion of w_i in the formula, instead of just using the average heuristic of child nodes, which reduces the changes in heuristic during each update. $diff(w_i, \frac{w_l+w_r}{2})$ is positive if the two child nodes' heuristic is better than the current, and it is negative otherwise. This property makes the heuristics of all the nodes in the same branch gradually increase if this branch continually produces better solutions and gradually decreases if otherwise, thus better representing the quality of the branches.

For the experiment, the two implementations that have shown the best mean results so far are chosen, which are S1 & E1 with C = 0.1 and S2 & E1 with C =0.2.

CRA	Minimum	Maximum	Mean	Median
S1 & E1 C = 0.1	-1.673	-0.840	-1.316	-1.344
S2 & E1 C = 0.2	-1.656	-0.923	-1.324	-1.341

The result shows an overall modest improvement over the original updating strategy for both S1 and S2.

5.2.6 Tree Structure Experimentation

The initial tree structure is limited to a binary tree. For games, MCTS often nodes depending on the possible moves available during each expansion. Each move can be seen as each different solution reachable from the parent solution through one step transformation. Although for some optimisation problems, the number of available solutions during each expansion can potentially be too large to feasibly create a node for each one of them, for CRA, it is typically limited to a maximum of 10. By making the tree structure unrestricted, and MCTS creating all the possible solutions during each expansion, it reduces the chance that a potentially great solution is skipped. In Order to achieve the unrestricted structure, the left and right child nodes are replaced with an array to store all the possible child nodes. Selection and back-propagation are changed to go through every child node instead of just the left and right nodes. A new mutation strategy is also implemented under a new class named "AllMatchingOneStepMutationStrategy", which returns all the reachable solutions from on step transformation from the given solution, which is used to allow the new expansion in MCTS.

However, after several 60 seconds runs through CRA with both S1 and S2 attempted, the changed structure was not able to produce a solution that satisfies the constraints with around 27 classless features, which means the solutions are invalid. he CRA values vary between -1 and -3, however as the solution is invalid, the values do not hold. The transformation chain length for the solutions are noticeably low, which is typically below 20 compared to 80 to 100 for the previous implementations. This length suggests that MCTS wasn't able to explore deep into the tree as before, which is likely due to the increased number of nodes inside each layer of the tree. After a 900 second run, the solution shows no improvement with the transformation chain length only increased to 23. Inorder to achieve a reasonable result with the unrestricted structure, a much longer run time would be necessary. The number of nodes inside of the tree drastically increases as time increases, the result suggests that MTCS only explored slightly deeper despite the large increase in run time.

As the results are considerably worse, MCTS is going to stay with the original binary tree structure, the altered version of Node and MCTS is included under the "UnrestrictedNode" and the "UnrestrictedMCTS" class name inside the appendix.

5.3 Hill Climbing Performance

CRA	Minimum	Maximum	Mean	Median
Hill Climbing	-3.298	-2.252	-2.722	-2.680

The initial implementation of the Hill Climbing algorithm shows great performance. The result suggests focusing on solutions that are strictly better than the previous solution is an effective strategy in some optimisation problems.

5.3.1 Optimization

The first optimization attempted is to ensure that Hill Climbing does not move down the hill to a worse solution. The modification is made by remembering a solution that is at least as good as the solution (not dominance solution). If during the 50 attempts a better solution cannot be found, it moves to an equivalent solution. If only worse solutions are found, the algorithm stays on the same solution, and start another iteration.

```
Solution equal = null;
Solution[] solutions = new Solution[variation.getArity()];
for (int i = 0; i < 50; i++) {
    solutions = variation.evolve(parent);
    evaluateAll(solutions);
    if(compareDomin(solutions[0], parent[0]) == -1){
        return solutions[0];
    }
    else if(compareDomin(solutions[0], parent[0]) == 0){
        equal = solutions[0];
    }
}
return equal;
```

Figure 5.4: modified Hill Climbing function for selecting next solution

CRA	Minimum	Maximum	Mean	Median
Hill Climbing	-3.637	-1.674	-2.764	-2.713

The result shows moderate improvement. This is likely due to the original implementation enters the local optimal earlier and more often, as when the original implementation run without the 50 attempts limit, it usually ends up in an infinite loop, which indicates no better solution can be found from the selected solution. This modification helps to slightly alleviate this issue.

5.4 Further Experimentation

Due to the incredible performance from Hill Climbing, E2 is modified to bring in ideas from Hill Climbing. E2 now compares solutions uses dominance instead of heuristic, and in order to keep the exploration property of MCTS, the 50 attempts limitation is kept, with an implementation similar to Figure 4.1. However, when a better or equal solution cannot be found, MCTS still creates a new solution derived from the selected solution, as MCTS updates the heuristics of the solutions, which encourages MCTS to explore other branches.

As S1 with c = 0.1 and S2 with c = 0.2 combined with new heuristic update function in back-propagation had the best results previously, they are also used here for the experiment.

CRA	Minimum	Maximum	Mean	Median
S1 c = 0.1	-2.437	-1.429	-1.849	-1.876
S2 c = 0.2	-2.700	-1.503	-1.854	-1.872

The result shows significant improvement. One interesting discovery shows that the transform chain (number of mutation from root to the solution) for the best solution of the new implementations are significantly shorter than the original, which was around 80 to 100 transformations, which now reduced to 40 to 60. The phenomenon suggests that the solutions discovered are much shallower in the tree. Given that for the Hill-Climbing algorithm has a transformation chain around 100 to 140, with longer running time significantly better results can be expected.

5.5 Final Implementation

The final implementation uses the configurations provided the best average results. MCTS is going to use S2 and the new E2 and heuristic update modification. Hill Climbing will use the modification during optimisation.

Chapter 6

Legal, Social, Ethical and Professional Issues

This project follows the Code of Conduct and Practice provided by the British Computing Society (BCS). The project puts in all effort in following the guidelines. The guidelines prevent the use of third parties' intellectual properties without appropriate recognitions. All the third party resources that are used in implementations or draw inspirations from are referenced and credited. Codes that uses external libraries are all adequately mentioned and credited. Due to the nature of the project, very minimal ethical issues are involved; however, the project pays full regards where concerns arrive. The guidelines encourage professional behaviour. The project pays full respect for all viewpoints and criticisms.

Chapter 7

Results & Evaluation

Comparisons uses the same methodology as Experimentation & Optimisation by default, changes are mentioned where they are made.

7.1 Performance Comparison

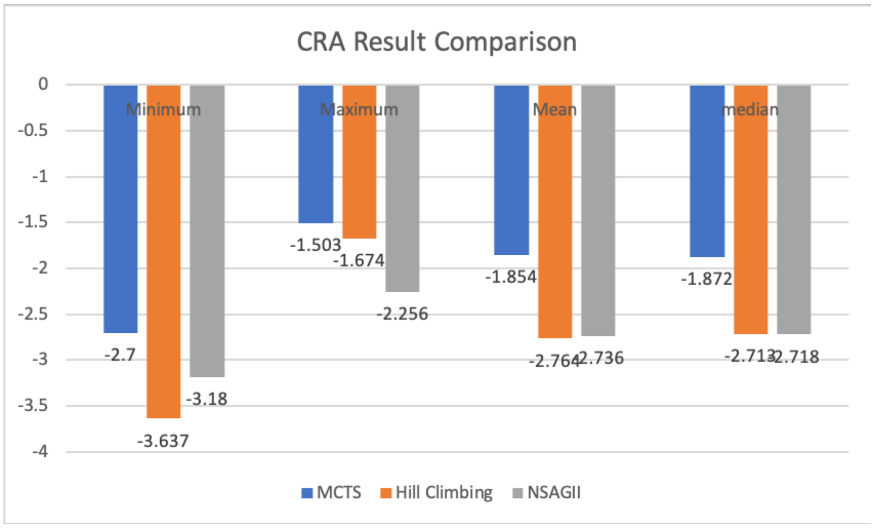


Figure 7.1: CRA Result Comparison

CRA	Standard Deviation
MCTS	0.193
Hill Climbing	0.367
NSGA-II	0.265

Hill Climbing and NSGA-II show similar performances overall in the CRA case study, with

Hill Climbing producing better best solution, and worse worst solution. NSGAII also has the longest transform chains around 90 to 125, which is around twice the length in comparison to MCTS.

The standard deviation represents how much each solution differ from each other (higher represents a higher difference). The result shows that the algorithms cover more search space produce more consistent solutions, with MCTS has the lowest standard deviation which covers the most search space and Hill Climbing with the highest standard deviation which only focuses on a single solution.

NRP	Average Cost	Average Statisfaction
MCTS	107.824	-1.896
Hill Climbing	104.394	-1.895
NSGAII	1604.121	-25.560

For NRP, MCTS and Hill Climbing demonstrates similar results, with hill climbing's results close to dominating MCTS's result with only a slightly worse average cost, which is likely due to the similar expansion techniques used. On contrast, NSGAII opts for a much worse cost in favour of better satisfaction, and NSGAII's output solutions' are much more varied in comparison to MCTS and hill Climbing, where the solutions are much closer to one another, although, with more run times, the variation might increase with MCTS and Hill Climbing. Below is a graph, which demonstrates this.

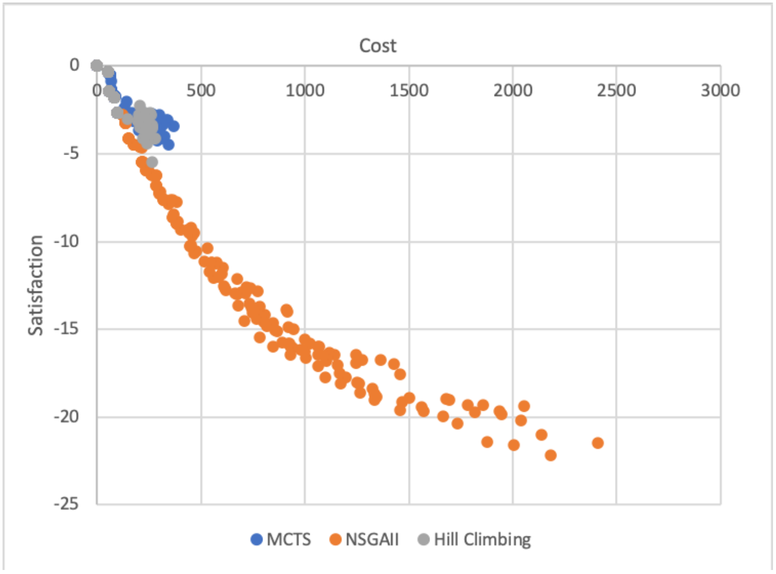


Figure 7.2: NRP Scattered Solutions Graph

7.1.1 Performance Over Time

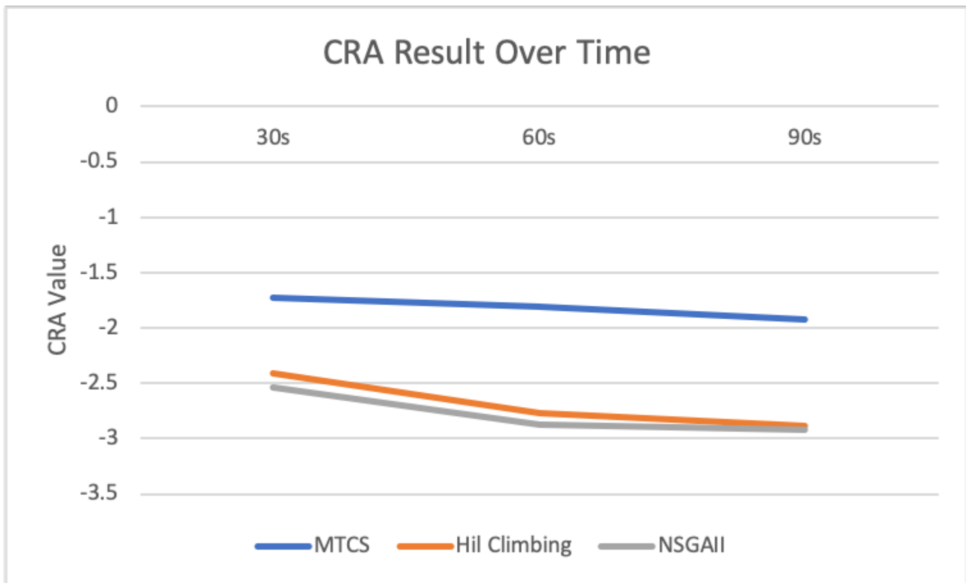


Figure 7.3: CRA Result Over Time

The result shows similar trajectory for Hill Climbing and NSAGII with performance increase drastically drop after 30 seconds, which can suggest both are reaching a local optimal at about 30 seconds. MCTS shows a slow but constant increase in performance over time, which indicates MCTS has the potential to surpass the other two algorithms. However, it can also be the case that the other algorithms are close to a global optimal, and the improvement for MCTS will also flat line around the same CRA value.

7.1.2 Performance Evaluation

NSGAII and Hill climbing demonstrates a clear advantage over MCTS, which likely due to the underlying similarly concept behind both algorithm. NSGAII discards the worst solutions, and make changes to the best solutions, as long as the best solution does not change; it is essentially the same as Hill Climbing. Tree search algorithms like MCTS attempts to cover an ample search space, which limits their ability to discover a good solution quickly, thus the worse results.

7.2 Memory Usage Evaluation

MCTS uses significantly more memory as the run-time increase in comparison to the other algorithms, with a max of 10 GB memory usage for a single 5 minutes run, which is an inherent characteristic of MTCS due to the continually growing tree. Both NSGAII and Hill Climbing has a constant memory usage, which never surpasses 1 GB due to the limited solution that is kept in each algorithm.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

Tree Search demonstrates limited advantage in MDEO. MDEO's model-based approach essentially allows any solution to transform into any other solution given enough transformation. Search algorithm such as local search and genetic which only focuses on a minimal amount of solutions and performs a significant amount transformation only on the limited solutions is far more effective than cover a large search space. Tree search algorithms such as MTCS can discover solutions which are typically ignored by the other, which can be interesting to some user, however overall it does perform much worse compare to local search and genetic search. Algorithm like MCTS, which scales poorly with memory usage also presents an inherent disadvantage, as optimisation problems can often have an effectively infinite number of possible solutions to be discovered, the memory costs are particularly high.

8.2 Future Work

For future work, DSL that is used to specify problems can also be extended to specify the specific configuration of the algorithms, which gives users more control when solving the problems. Also, Hill Climbing has demonstrated the significant potential of local search in MDEO, and more local search options can be explored. Furthermore, only the CRA case study works with the MDEO with or without the implementations made in this project; more sample case studies could be updated to allow more test cases. The compilation time for MDEO is relatively long, which can also be improved in the future.

References

[1] Lubna Abdulahad. Find optimal solution for eggcrate function based on objective function, 01 2015.

[2] Alexandru Burdusel, Steffen Zschaler, and Daniel Strüber. Mdeoptimiser: a search based model engineering tool. pages 12–16, 10 2018.

[3] Burdusel, Alex and Zschaler, Steffen. Class Responsibility Assignment — MDEoptimiser, 2020. [Online; accessed 2-April-2020].

[4] Burdusel, Alex and Zschaler, Steffen. Next Release Problem — MDEoptimiser, 2020. [Online; accessed 2-April-2020].

[5] Kalyan Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A fast and elitist multi-objective genetic algorithm: Nsga-ii. *Evolutionary Computation, IEEE Transactions on*, 6:182 – 197, 05 2002.

[6] Benyamin Ghojogh, Saeed Sharifian, and Hoda Mohammadzade. Tree-based optimization: A meta-algorithm for metaheuristic optimization, 09 2018.

[7] Mark Harman, Afshin Mansouri, and Yuanyuan Zhang. Search based software engineering: A comprehensive analysis and review of trends techniques and applications. 05 2009.

[8] Steven James, George Konidaris, and Benjamin Rosman. An analysis of monte carlo tree search. 02 2017.

[9] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. volume 2006, pages 282–293, 09 2006.

[10] Maarten Schadd, Mark Winands, H. Herik, Guillaume Chaslot, and Jos Uiterwijk. Single-player monte-carlo tree search. pages 1–12, 09 2008.

[11] Bart Selman and Carla Gomes. *Hill-climbing Search*. 01 2006.

[12] Otakar Trunda and Roman Barták. Using monte carlo tree search to solve planning problems in transportation domains. volume 8266, pages 435–449, 11 2013.

[13] Wikipedia contributors. Genetic algorithm — Wikipedia, the free encyclopedia, 2020. [Online; accessed 3-March-2020].

[14] Wikipedia contributors. Monte carlo tree search — Wikipedia, the free encyclopedia, 2020. [Online; accessed 3-March-2020].

[15] Steffen Zschaler and Lawrence Mandow. Towards model-based optimisation: Using domain knowledge explicitly. volume 9946, pages 317–329, 07 2016.