ABSTRACT

An abstract of the thesis of Jeremy Greenwald for the Master of Science in Computer Science presented February 15, 2007

Title: A Novel Metaheuristic Search Technique: Iterative Treatment Learning

Increasingly modeling is being used in all the software life cycle stages studied in the field of software engineering. By using models from two such stages, the requirements stage and the testing stage, we show that our machine learning based metaheuristic search technique can be used to optimize these models. This technique is called Iterative Treatment Learning, ITL. We apply ITL to three NASA cost/benefit models used during the requirements stage and two biomathemitical and three NASA flight models used during the testing stage. These last five models are investigated using an integrated development and testing framework, SPY, that has built-in ITL search capabilities.

These studies improve upon previous work done with ITL by Menzies et. al. by 1) discussing ITL's characteristics in metaheuristic search terminology 2) increasing the number and complexity of the models studied 3) exploring the option space of ITL, including a new discretizer and 4) investigating the stability and variance of ITL's solutions.

With the cost/benefit models we show how our new method for discretizing the model's dependent variables 1) outperforms the previous discretizing method, both on solution quality and convergence speed and 2) outperforms a simulated annealer, a commonly used metaheuristic search technique, on convergence speed while finding the same quality solutions. With the biomathematical models we show that SPY can find potentially useful range restrictions to model inputs that restrict the model's output to specified behavior modes. Finally, with the NASA flight models we show that SPY can verify temporal properties in models with real valued inputs, comparing SPY's performance to a commercially available tool, Reactis, that uses a random and heuristic search strategy.

A NOVEL METAHEURISTIC SEARCH TECHNIQUE: ITERATIVE TREATMENT LEARNING

by

JEREMY GREENWALD

A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
COMPUTER SCIENCE

Portland State University
2007

I dedicate this thesis to my mother and father, and all my teachers who took the time to explain ideas.

ACKNOWLEDGMENTS

I would like to acknowledge the two people who most contributed to my development as a scientist. Dr. Steven Dytman from the University of Pittsburgh, my adviser for four years and Dr. Tim Menzies without whom this thesis would never exist.

Of course there are numerous other people who, through their friendship and understanding, helped make this work possible, but there isn't space to enumerate them all. They know who they are and I thank them.

Contents

LIST OF TABLES

List of Figures

CHAPTER 1

INTRODUCTION

Two methodologies that are becoming increasingly popular in software engineering are model based development and search based software engineering. The work presented in this thesis attempts to apply search based techniques to model based development. We apply a new metaheuristic search technique, Iterative Treatment Learning, ITL, to NASA requirements models and NASA flight models. These two types of models represent early life cycle stages and late stages, respectively.

1.1 Search Based Software Engineering

Software engineering is concerned with the process of software construction. The software construction process is commonly decomposed into the following parts: requirements, design, implementation, testing, and maintenance. Software engineering researchers have produced work on all of the above life stages. The overall goal of this body of work is to improve the quality of software. This might mean improving cost estimation for software projects, developing tools that make implementation easier, or automatically generating a suite of test cases that verify requirements.

Compared to the amount of research done in applying metaheuristic search techniques to combinatorial problems and to other engineering fields, the amount of work done with metaheuristic search in software engineering is quite small. Beginning in the mid-1990's some researchers in the software engineering field have argued for the application of common metaheuristic search techniques to classical software engineering problems [32,57,58]. This idea is accepted by this thesis as an important breakthrough in problem formulation for the field of software engineering. It is also beginning to be accepted by the software engineering field. For example, in a review completed in 2004 Rela [114] found at least 123 publications in the field of search-based software engineering that used evolutionary algorithms (evolutionary algorithms will be discussed in §2.2). Of the 123 publications 44% (54) were related to testing, while only 7% (8) were related to early project planning. In §2.3 we discuss these two topics, because much of the work presented in this thesis deals with early project planning, such as requirements engineering (chapter 4), and testing using model based verification (chapter 5).

The first step towards the application of metaheuristic search techniques to software engineering problems is the realization that many of these problems are already or could easily become a type of optimization problem. All that is needed is to reformulate the goal of the research as an *objective function*[1]. The objective function is a function that evaluates the desirability of a given candidate solution. This function can evaluate any property of the solution that the practitioner wishes to use, as long as the output is numeric. For example, the objective function could be path coverage or estimated project cost. Besides development of an objective function, a manipulatable language, as well as the operators that manipulate that language, need to be developed. A more complete discussion of these steps can be found in §2.3.1.

Once an objective function is formulated, a software engineering researcher does not have to develop their own optimization technique. Rather, researchers can borrow techniques developed for other problems that can be easily adapted to their own problem domain. While current research on metaheuristic search techniques may focus on complex issues such as robustness to noise, local optimum escaping, or hybrid approaches, software engineering researchers can use simple versions of many metaheuristic search techniques that have be shown to be effective, despite their simplicity. In fact, for some languages and platforms, all the researcher has to do is develop their objective function in a particular language and use a framework already developed[2].

[1]Sometimes called a *cost*, *fitness*, or *worth* function in the literature

[2]For example, see `http://www.cs.uwyo.edu/~wspears/freeware.html` for William Spears's C implementation of a genetic algorithm

This change in approach will facilitate the the adding of metaheuristic search techniques to the toolbox of all software engineering researchers.

1.2 Model Based Development

Software engineers build models at every phase of the life cycle. Some are paper-based and some are executable but all these artifacts are *models*, which are defined in [88] as

> "elements describing something (for example, a system, bank, phone, or train) built for some purpose that is amenable to a particular form of analysis, such as communication of ideas between people and machines; completeness checking; test case generation; etc."

The benefits of modeling include increasing productivity and reduced time-to-market for software products [119] and the imposition of "structure and common vocabularies so that artifacts are useful for their main purpose in their particular stage in the life cycle" [53]. The current ubiquity of the term *model-driven software engineering* is both a recognition of these benefits and an appreciation of the reality that we are not using these models as effectively as we may hope.

Model based software engineering has been put into practice by many groups. The Object Management Group [108], Microsoft [52], and Lockhead Martin [130], have all adopted model based development as part of their software engineering process. And just as their are search based tools available to researchers, tools to support model based development have been developed. These include tools for distributed agent-based simulations [28], discrete-event simulation [59, 75, 80], continuous simulation (also called system dynamics) [1, 123], state-based simulation [5, 56, 83], and rule-based simulations [98]. One can find models used in the requirements phase (see §4.1 for a description of such a tool, DDP), refactoring of designs using patterns [47], software integration [35], model-based security [71], and performance assessment [9].

Recently, AI has been successful applied to model based software engineering. For example, Whittle uses deductive learners to generate lower-level UML designs (state charts) from higher-level constructs (use case diagrams) [138]. More generally, the field of search based software engineering augments model based development with metaheuristic techniques, like those discussed in §1.3 and §2.2, to explore a model. Such heuristic methods are hardly complete but as remarked in [31]: "...software engineers face problems which consist, not in finding *the* solution, but rather, in engineering an *acceptable* or *near optimal solution* from a large number of alternatives."

1.3 Metaheuristics

Metaheuristic search is the catch-all phrase for a set of high-level heuristics that are abstract enough to be applied to many different fields. A central idea to all metaheuristic search techniques is that most interesting real-world problems are far too difficult to solve exactly. Therefore heuristic techniques must be used to find near-optimal solutions. Since heuristics techniques have no performance or convergence guarantees several different techniques have been developed, so that if a problem defeats one technique another may be used.

The most commonly used metaheuristic search techniques are simulated annealing, evolutionary algorithms, and tabu search. These techniques will be discussed in §2.2. This section will also discuss important ways to characterize metaheuristic techniques, such as mechanisms for escaping local optima and candidate solution encoding. These characteristics will be revisited in §3.1.3 to see how they can be applied to our new metaheuristic search technique, Iterative Treatment Learning. Occasionally we will review techniques from statistics and linear algebra that are used on some problems, which will be discussed in §2.3 and as they relate to search based software engineering.

1.4 Machine Learning as a Search Heuristic

It will be shown in this thesis that a machine learning technique has been successfully used as search heuristic. The machine learning technique used is treatment learning [91] and the search technique we call *Iterative Treatment Learning* (ITL), since the treatment learner is called in an iterative fashion. ITL has some characteristics that differentiate it from other mainstream metaheuristic search techniques. Most metaheuristics' base heuristic have a statistical or geometric interpretation. The base heuristic for ITL is treatment learning, which is considered a machine learning algorithm. Another important difference is that ITL's near-optimal solution is a *partial description* rather than a *complete description*. The difference between these two types of solutions is that partial descriptions do not comment on all the available attributes and these comments are not *exact* value assignments, but are value *range* assignments (see §3.1.2 for a more complete discussion of partial descriptions).

Menzies *et. al.* introduced ITL [89] as an alternative search heuristic in requirements engineering. While their work was interesting and novel, the methodology used to demonstrate the performance of ITL could use much improvement. A single model was used in their study, they did not explore design options within ITL, and ITL was not placed in the context of metaheuristic search.

Another version of ITL called SPY has also been introduced by Clarke [29]. SPY is an integrated model development and search environment. The search algorithm used by SPY is an ITL method with a slightly different search strategy and a more efficient treatment learner. SPY finds range restrictions to the model's inputs to confine model output behavior to specified modes. The original use case for SPY was that these behavior modes would be temporal properties in models with real valued inputs. In [29] SPY was only applied to a small number of simple models. In addition, there was no comparative analysis of SPY to benchmark it against other tools or analysis methods And like the original work on ITL, SPY was not presented in a search context. SPY will be described in more detail in §5.1.

1.5 Machine Learning

Machine learning is any algorithmic process that generalizes from a specific data set to a theory that can be applied to new unseen data. One type of machine learner is the classifier; §2.1 will describe several different classifiers, including treatment learning. Classifiers build theories that can be used to assign one of the target labels to unseen examples. Contrast this with the typical use of metaheuristic search techniques. Search techniques produce near-optimal solutions without attempting to generalize anything learned during their search. Even when viewing ITL as a search technique, which seeks a near-optimal solution, its solution is in a form that captures generalizations that can be used after the search.

1.6 Problem Statement

Can ITL be used as a search technique for models from different stages of the software life cycle?

To address that question the following topics will be discussed in the rest of this thesis.

1.6.1 Rethinking ITL as metaheuristic search

ITL has previously been presented as a machine learning technique. In this thesis we argue that ITL should be considered a metaheuristic search technique that has novel characteristics not common in other metaheuristic search techniques.

1.6.2 Improved methodology demonstrating ITL's ability to optimize early stage models in requirements engineering

The use of ITL to investigate early project requirements engineering has been limited to a single model. Other uses of ITL have focused on models with a small number of independent variables (fewer than 40). This thesis uses three models for early project requirements engineering that have 99, 83, and 58 independent variables, respectively. The new methodology also includes the development of extreme sampling, a new discretization method for ITL.

1.6.3 Can SPY be used in late stage life cycles?

Previous work with the SPY framework only used toy models which didn't have any previous analysis. This thesis uses the SPY framework on models that have been used elsewhere in the literature, hence a comparison of previous results with results generated by SPY is possible. We will investigate the use of SPY on real world models from the aerospace industry and compare the performance of SPY with a commercial tool. Lastly, we will investigate biomathematical models, using SPY to restrict the output behavior modes of these models.

1.7 Previous Work on Problem Statement

Treatment learning has been developed and tested extensively by Menzies, Hu, Clarke, et. al. [62, 92, 93, 124]. The machine learning properties of treatment learning will no be explored further in this thesis. ITL has been previously developed by Menzies et. al. [41, 94]. That work focused on either models not presented in this thesis or just one of the requirements models, *aero*, used in chapter 4. In addition, the SPY framework has developed by Clarke and Menzies [29]. That work focused on the validation of the SPY framework's methodology using small toy problems developed for testing purposes.

1.8 Contribution

This thesis makes the following novel contributions towards addressing the problem statement

- Presents ITL in a metaheuristic search context.
- Introduces the concept of an extreme sampling discretizer, and develops three variants of a discretizer.
- Three cost/benefit models, including two not previously analyzed, are used to
  - Conduct a study of the effect of parameter settings to extreme sampling's performance according to numerous criteria
  - Compare performance of ITL with extreme sampling discretizer vs. diagonal striping discretizer
  - Compare performance of ITL vs. simulated annealer
- A total of five models, from varied domains and with a higher complexity than models previously studied, are analyzed by SPY to find range restrictions that restrict the models' output behavior to specified modes.

The rest of this thesis will be structured around these contributions in the following manner: ITL will be placed in the context of metaheuristic search in §2.2 and chapter 3, extreme sampling and its three variants will be discussed in chapter 3, the three cost/benefit models will be studied, along with extreme sampling, in chapter 4, and the five models analyzed by SPY will be spread over chapter 5 and chapter 6.

CHAPTER 2

LITERATURE REVIEW

This chapter describes past and related work that has been done to date on topics discussed later in the thesis. The review will focus on three related topics

- General Machine Learning Techniques (§2.1)
- General Metaheuristic Search (§2.2)
- Search Based Software Engineering (§2.3)

The first 2 sections of this review are meant to give the reader an introduction to common techniques in machine learning and metaheuristic search. This introduction will allow the reader to appreciate how treatment learning differs from other machine learning techniques and how ITL differs from other metaheuristic techniques. These sections also serve to demonstrate some of the differences and similarities between machine learning and metaheuristic search. The last section is a more in-depth look at how metaheuristic search is used in software engineering.

2.1 Machine Learning Techniques

Since ITL utilizes a machine learning technique, a treatment learner, this section will introduce the field of machine learning. First some basic terminology used in the field will be defined, as well as some common characteristics used to describe different learners (§2.1.1). Then two popular techniques will be discussed (§2.1.2 and §2.1.3). These two techniques are described to highlight the differences between the form and usability of their theories and the form and usability of treatment learners. Before treatment learners (§2.1.5) are discussed, we discuss contrast set learners (§2.1.4) since treatment learners are a type of contrast set learner.

2.1.1 Introduction and Terminology

Machine learning deals with algorithmic processes that attempt to generalize knowledge learned form past examples to future, unseen, examples. The past examples are often called instances, records, or input data. Each record has a number of *attributes*. These attributes entirely describe the record, i. e. all information presented to the learner is in the form of attributes. Attributes can be *continuous*, e. g. the real numbers 0 to 10, or *discrete*. Discrete attributes can be *ordinal*, e. g. the integers 1-10 or *categorical*, e. g. the primary colors. For many problems there is a special attribute which labels its type. This attribute is called the *class* or the *target class*. Not all problems have a target class. The target class can also be real, ordinal or categorical. Learners take the input records provided and build a *theory* that describes what they learned. The learner is said to *train* on the input data to produce a theory. When trying to predict continuous classes the theory may be an equation, or a decision tree or probability table when trying to predict discrete classes. Other types of theories are possible.

Learners are often classified by what type of inputs they can use and what type of outputs their theories can predict for. Some learners, e. g. a naïve Bayes learner, (see §2.1.3) only work with discrete input data and can only predict for a discrete target class. Any learner can be adapted to handle continuous data by discretizing the data in a pre-processing step. The performance characteristics of different discretization methods are an active area of research [38, 97, 141] but a detailed description of these methods is beyond the scope of this review. Discretization is most commonly used on the independent attributes, but can be applied to the dependent attributes as well. Our extreme sampling discretizer, described in §3.1.4, is one that works on the dependent attributes.

Another way to describe machine learners is whether they are performance or explanatory systems. Performance learners only attempt to maximize their predictive performance according to some criterion, such as accuracy or precision. Explanatory learners also attempt to maximize their predictive performance, but their theories are in a form that can offer insight to human users. Naïve Bayes (§2.1.3) are an example of performance learners, while decision trees (§2.1.2) are an example of explanatory learners. Treatment learners (§2.1.5) are also explanatory learners.

2.1.2 Decision Tree Learning

Decision tree learning uses the common divide-and-conquer approach. The ID3 decision tree learner [111] is the most common, although other tree learners include M5 [113] and CART [21]. The modern incarnations of ID3 are C4.5 [112] and J4.8[1]. A decision tree's leaves are classification labels and its internal nodes are attribute-value tests. The pseudocode in Figure 2.1 describes how to classified a new instance using a decision already generated.

```
start at root node
repeat
   follow branch that matches the instance being classified
until current node is a classification node
return classification in leaf node
```

Figure 2.1: Pseudocode for classifying new instances with a decision tree

[1]Part of the weka project, downloadable at `http://www.cs.waikato.ac.nz/ml/weka/`

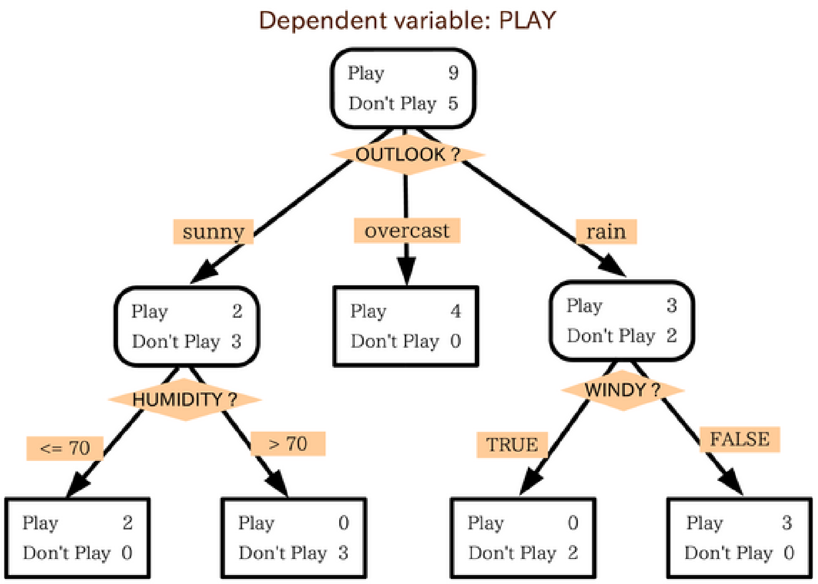


Figure 2.2: Example decision tree

See Figure 2.2 for an example of what a decision tree might look like after training on the ubiquitous golf data set. Note that the classification at the root nodes does not misclassify any of the training examples. This, of course, is not the typical performance of machine learning algorithms.

To train a decision tree an ordering heuristic is needed to decide which attribute to use when constructing an internal node. ID3 uses information gain [112], which is an entropy based metric. Starting at the root node and continuing until either there are no attributes remaining or the instances at the current location in the tree all belong to the same class, the attribute that would most reduce the information in the resulting tree is selected.

```
stat <= 11.66
| rm <= 6.54
| | lstat <= 7.56 THEN medhigh
| | lstat > 7.56
| | | dis <= 3.9454
| | | | ptratio <= 17.6 THEN medhigh
| | | | ptratio > 17.6
| | | | | age <= 67.6 THEN medhigh
| | | | | age > 67.6 THEN medlow
| | | dis > 3.9454 THEN medlow
| rm > 6.54
| | rm <= 7.061
| | | lstat <= 5.39 THEN high
| | | lstat > 5.39
| | | | nox <= 0.435 THEN medhigh
| | | | nox > 0.435
| | | | | ptratio <= 18.4 THEN high
| | | | | ptratio > 18.4 THEN medhigh
| | rm > 7.061 THEN high
lstat > 11.66
| lstat <= 16.21
| | b <= 378.95
| | | lstat <= 14.27 THEN medlow
| | | lstat > 14.27 THEN low
| | b > 378.95 THEN medlow
| lstat > 16.21
| | nox <= 0.585
| | | ptratio <= 20.9
| | | | b <= 392.92 THEN low
| | | | b > 392.92 THEN medlow
| | | ptratio > 20.9 THEN low
| | nox > 0.585 THEN low
```

Figure 2.3: Decision tree generated by J4.8 training on the Boston housing data

The Boston housing data [17] is a frequently used data set that describes the median value of homes in 506 suburbs of Boston. There are 12 continuous attributes, 1 boolean attribute, and the target class. Figure 2.3 shows a typical decision tree learned on this data by J4.8, where {`stat, rm, lstat, dis, ptratio, nox, b`} are independent variables[2] and {`low, medlow, medhigh, high`} are the target labels. Compare the complexity of this tree to the simple treatment shown in Figure 2.7. Also note that Figure 2.3 is a classifier, which can be used to predict the class of any new instance, whereas Figure 2.7 is a contrast rule that describes the biggest difference between the higher value target classes and the lower value target classes.

2.1.3 Naïve Bayes

Bayes theorem [11] is used to calculate the probability that a hypothesis is correct, given a set of evidence. It can be written as

$$P(H|E) = \frac{P(E|H) * P(H)}{P(E)}$$

where $P(H|E)$ is the probability of the hypothesis given the evidence, $P(E|H)$ is the probability of the evidence given the hypothesis, $P(H)$ is the probability of the

[2]Where `rm` is the average number of rooms, `ptratio` is the pupil-teacher ratio, and `nox` is the nitrogen oxide concentration, to define a few.

hypothesis, and $P(E)$ is the probability of the evidence.

The difficulty in using Bayes theorem is calculating the $P(E|H)$ term. The theorem suggests we most observe every possible combination of the different pieces of evidence.

Because of this difficulty, in machine learning a weaker form of Bayes theorem is often used called *Naïve Bayes*. Naïve Bayes assumes that the probabilities of the different attributes are independent of one another. This allows the $P(E|H)$ term to be calculated multiplicatively as

$$P(E|H) = \prod^{n} P(E_i|H)$$

where the $P(E_i|H)$ terms are the probabilities of a single piece of evidence given H and n is the number of attributes available.

To train a naïve Bayes learner, frequency tables are used to count the number of instances with each attribute value in each of the target classes. The attributes and the target class must be discrete to keep the number of tables and the number of rows in the tables finite. For example, if a naïve Bayes learner was trained on the data in Figure 2.4, the frequency tables might look like Figure 2.5.

To classify a new example, naïve Bayes calculates the likelihood of the example belonging to each target class, $L(H_i)$. The likelihood is the similar to the probability,

attributes						class
Make	coupe : SUV	Size	coupe : SUV	Hifi	coupe : SUV	coupe : SUV
Mitsubishi	1 : 1	Small	6 : 0*	yes	6 : 0*	8 : 4
Toyota	2 : 1	Medium	1 : 1	no	2 : 4	
Benz	1 : 1	Large	1 : 3			
BMW	2 : 0*					
Ford	1 : 1					
Honda	1 : 0*					

Figure 2.4: Training data. Cells indicated with a $*$ have a zero cell. In the frequency table all cells will have one added to their counts to avoid calculating any zero likelihoods.

attributes						class	
Make	coupe : SUV	Size	coupe : SUV	Hifi	coupe : SUV	coupe	SUV
Mitsubishi	$\frac{2}{9} : \frac{2}{5}$	Small	$\frac{7}{9} : \frac{1}{5}$	yes	$\frac{7}{9} : \frac{1}{5}$	$\frac{9}{14}$	$\frac{5}{14}$
Toyota	$\frac{3}{9} : \frac{2}{5}$	Medium	$\frac{2}{9} : \frac{2}{5}$	no	$\frac{3}{9} : \frac{5}{5}$		
Benz	$\frac{2}{9} : \frac{1}{5}$	Large	$\frac{2}{9} : \frac{4}{5}$				
BMW	$\frac{3}{9} : \frac{1}{5}$						
Ford	$\frac{2}{9} : \frac{2}{5}$						
Honda	$\frac{2}{9} : \frac{1}{5}$						

Figure 2.5: Frequency table, $P(E_i|H_j)$

but is not normalized by the $P(E)$ term. The likelihood can be used because naïve Bayes compares two probabilities, making the $P(E)$ terms cancel out. The likelihood is calculated by

$$L(H_i) = P(E|H_i) * P(H_i) \tag{2.1}$$

The example is classified as whichever class has the highest likelihood. For example, if a new instance was a medium sized Ford, the likelihood of it being an SUV would be

$$\begin{aligned} L(SUV) &= P(size = medium|SUV) * P(Make = Ford|SUV) * P(SUV) \\ &= \frac{2}{5} * \frac{2}{5} * \frac{5}{14} \\ &= 0.057 \end{aligned}$$

while the likelihood of it being a coupe would be

$$\begin{aligned} L(coupe) &= P(size = medium|coupe) * P(Make = Ford|coupe) * P(coupe) \\ &= \frac{2}{9} * \frac{2}{9} * \frac{9}{14} \\ &= 0.032 \end{aligned}$$

For this instance, since $L(SUV) > L(coupe)$, the naïve Bayes classifier would assign this instance to SUV target class.

2.1.4 Contrast Set Learning

Contrast set learners are a type of association rule learners that find the greatest difference between the target classes. Since treatment learning is an example of a contrast set learner, we will discuss some research done with contrast set learners other than treatment learners.

STUCCO STUCCO (Search and Testing for Understandable Consistent Contrasts) [10] uses the algorithm in Figure 2.6 to learn contrasting sets.

This algorithm builds trees (line 14) using a canonical ordering to avoid visiting the same node twice. The tree nodes consist of different possible attribute-value pairs, so that a path from the root to a leaf consists of several non-exclusive attribute-value pairs. The `prune` function (line 11) has three criteria, any one of which causes the function to return true:

1. minimum deviation size
2. expected cell frequencies
3. χ^2 bounds

These rules together make it possible to do a complete search of the remaining branches. The functions `significant` and `large` ensure that the proposed contrast set is both

```
    initialize set of candidates, $C$, to the empty set
    initialize set of deviations, $D$, to the empty set
    initialize set of pruned candidates, $P$, to the empty set
    let prune(c) return true if c should be pruned
 5: repeat
      scan data and count support $\forall c \in C$
      for all $c \in C$ do
        if significant(c) $\wedge$ large(c) then
          $D \leftarrow D \cup c$
10:     end if
        if prune(c) is true then
          $P \leftarrow P \cup c$
        else
          $C_{new} \leftarrow C_{new} \cup$ GenChildren(c, P)
15:     end if
      end for
      $C \leftarrow C_{new}$
    until $C$ is empty
    return Surprising(D)
```

Figure 2.6: STUCCO algorithm

	% holding	
contrast set	Ph. D	Bachelor
workclass = State-gov	21.0	5.4
occupation = sales	2.7	15.8

Table 2.1: Example rules learner by STUCCO from Adult Census data

statistically significant, based on the χ^2 test, and large, above a user specified threshold. Finally, the `Surprising` function ensures that the proposed contrast set is not just the result of multiplicative probabilities. For example if $P(color = red \mid shape = square) = .76$ and $P(hollow = yes \mid shape = square) = .45$, it would not be interesting if $P(hollow = yes \wedge color = red \mid shape = square) = .32$ since that probability is so close to probability we would expect if the attributes were independent, i. e. $.76 * .45 = .34$. This is only the basic idea behind the `Surprising` function; a more robust method based on *iterative proportional fitting* [40] was actually used in STUCCO.

Using the STUCCO learner to investigated Adult Census data [17] to answer the question "What are the differences between people with Ph. D and Bachelor degrees?" Bay and Pazzani [10] found 164 rules, while Apriori returned over 75,000 rules. For example the two rules in Table 2.1 were returned by STUCCO.

MINWAL The MINWAL learners [23], MINWAL(O) and MINWAL(W), extend the Apriori Gen Algorithm [3]. They are called weighted class learners, because they allow the instances to have an extra attribute, weight, that records how significant the instance is. The example used in [23] has to do with analysis of supermarket checkout baskets. If a rule learner like Apriori Gen finds the following two rules

- Buys(baby food) $\Rightarrow$ Buys(diapers)
- Buys(baby food) $\Rightarrow$ Buys(beer)

but the profit margin on beer is much higher than on diapers, the second rule is more useful to a business user.

Webb et. al. [132] did a study comparing STUCCO with two other rule learners, Magnum Opus, a commercially available tool and C4.5rules [112], which generates rules from a decision tree. Although these other learners were not designed to be contrast set learners, the authors were able to restrict the form of their theories to allow an almost direct comparison with STUCCO. The learners were applied to aggregated data from a large Australian discount department store taken on two different days. To their surprise they found a correspondence between the core contrast-set-discovery task in the Magnum and STUCCO learners. They found that the pruning rules of the learners are responsible for the difference in the theories returned by the two learners. These different pruning rules lead the STUCCO learner to return fewer rules than Magnum. A manual inspection of these rules found both interesting and spurious rules in the list generated by Magnum, suggesting that the pruning rules of STUCCO are possibly too strict and that the rules of Magnum are possibly too lenient. They did find that C4.5rules missed many of the key contrasts that were found by STUCCO and Magnum.

2.1.5 Treatment Learners

The tar family of learners have been developed by Menzies et. al. The original Tarzan [95] prototype was actually a post-processor that worked with classification tree generators (specifically C4.5, discussed in §2.1.2). Tarzan constructed a set of decision trees using different training data sets that had been generated by applying different restrictions to the software effort estimation model [2] being studied. Tarzan then used a set of pruning rules that applied to the individual trees, as well as rules that were applied to the ensemble of decision trees. These rules vastly reduced the complexity of the original decision trees. This prototype was quickly discarded as it was computationally inefficient. Tar2 was developed [63] to generate treatments quicker by training directly on the instances instead of constructing decision trees. Tar3, used in the studies described in chapter 4, was developed to increase the efficiency of tar2 by using stochastic sampling. Tar4 [29], used by SPY in the case studies described in chapter 5 and chapter 6, is the next generation treatment learner that further increases the runtime efficiency of the learner by using a Bayes style frequency table to eliminate the need to run over the data set for each proposed treatment or even store the training instances in memory.

The form of the treatments returned by the tar family of learners is conjunctions of attribute-value pairs. Each attribute-value pair consists of one attribute and either a range restriction, such as $6.7 \leq rooms \leq 9.8$ or a value assignment, such as $outlook = overcast$. Hence a treatment has the form

$$pair_1 \wedge pair_2 \wedge \ldots \wedge pair_n \Rightarrow P(c_1), P(c_2), \ldots, P(c_k)$$

where, typically, $1 \leq n \leq 5$ and $P(c_k)$ is the predicted probability of class c_k among instances that pass the treatment.

The tar learners are a type of contrast set learner, as discussed in §2.1.4. This means they attempt to find treatments that select for the desired class, but do not comment on instances that don't pass the treatment. In receiver operator characteristics, we would say these types of learners attempt to maximize the true positive rate of their theories, without penalizing theories that have a high false negative rate. A treatment should be interpreted as a region where desirable classes are more prevalent, while the region outside the treatment may not have significantly more instances of the undesirable classes than the baseline. In other words, if an unseen instance passes a treatment returned by a treatment learner there is a high likelihood that the new instance belongs to the desired class. But if the new instance does not pass the treatment, the treatment

```
IF: rm >= 6.6 AND ptratio <= 15.9
THEN: 97% of the found houses will be high quality
      3% of the found houses will be medhigh quality
BASELINE: 29% high, 29% medhigh, 21% medlow, 21% low
```

Figure 2.7: Treatment generated by tar3 on the Boston housing data

learner does not offer any guidance on what class that instance belongs to.

Figure 2.7 shows a typical treatment for the Boston housing market data used to build the decision tree in Figure 2.3, where {`rm, ptratio`} are independent variables, `high` is the most desirable target class, and `medhigh` is the second most desirable target class. Compare the succinctness of this theory to the complexity of the decision tree in Figure 2.3. This is the advantage to using a treatment learner. Treatments are a compact way to convey the differences between the higher valued target classes and the lower valued ones.

We will discuss the details of tar3 and tar4, since they are the two treatment learners utilized by the studies presented in chapters 4 - 6.

Tar3 The first step in tar3 is to calculate the *baseline*. The different possible target classes are assigned *weights*. Experience has shown that these weights should be exponential. For example, if there are four classes, they might be assigned the weights 2, 4, 8, 16. Of course, this is all done internally in the treatment learner. All the user has to do is list the possible target classes in the preferred order. The baseline is a normalized weighted sum of the proportions of the target classes in the training data. This calculation has the following steps

1. multiply the proportion of each class in the training set by its weight
2. sum the products from step 1)
3. normalize the sum from step 2) by the sum of the weights

Hence, the baseline can be calculated as

$$baseline = \frac{\sum_{i=0}^{n}(weight_i * prop_i)}{\sum_{i=0}^{n} weight_i}$$

where n is the number of target classes, $weight_i$ is the weight assigned to the ith class, and $prop_i$ is the proportion of the ith class in the training data.

The next step is to calculate the *lift*$_1$s. The *lift* for any treatment is the ratio of the normalized weighted sum of the instances covered by that treatment to the baseline, calculated as

$$\begin{aligned} lift &= \frac{treated\ average}{baseline} \\ &= \frac{\sum_{i=0}^{n}(weight_i * treated_i)}{\sum_{i=0}^{n}(weight_i * prop_i)} \end{aligned}$$

where $treated_i$ is the proportion of the ith class among instances that pass the treatment. The *lift*$_1$s are the *lift*s of all possible treatments with only one attribute-value pair. The number of attribute-value pairs in a treatment is often referred to as the treatment's *size*.

Next *lift₁*s are combined to form treatments of sizes 1 to max_size,which is specified by the user, but usually 5 works well. First the size of the treatment is picked randomly, then different attribute-value pairs are chosen randomly, biased according to each attribute-value pair's *lift₁*. The lift for each proposed treatment is calculated and after tar3 has run several trials, it returns the treatments it found ranked by their lift. This metric might tend to produce very overfitted theories, as treatments are learned that match only a few instances in the most desired class. A minimum best support term is used to counter this potential cause of over-fitting.

Tar4 Tar3 still has to run through the data to evaluate each potential treatment. This also requires that the instances be stored in memory. Tar4 was developed to generate treatments while requiring only one pass through the data, which also means that the training data need not be stored in memory. During the training stage, tar4 stores the instances in frequency tables similar to a naïve Bayes classifier. The major difference between tar4's frequency tables and those found in a naïve Bayes classifier is that tar4 uses a two-class system, regardless of how many target classes exist in the training set. For example, if the target class is an integer from the range $[1, 5]$ with 5 being the best class, an instance with a class of 4 would count as $4/5$ *good* and $1/5$ *bad*. The *lift₁*s are then calculated from these tables. The final lift of a treatment is calculated by

$P(good) * support(good)$, where $P(good)$ is calculated in the same fashion as it would in a naïve Bayes classifier and $support(good)$ is the unnormalized likelihood from the frequency tables (see Equation 2.1).

For more complete details on tar4 see [29].

2.2 Metaheuristic Search

The most widely used metaheuristic techniques are

- gradient based techniques (§2.2.2)
- evolutionary algorithms (§2.2.3)
- tabu search (§2.2.4)

These three techniques will be discussed in this section. First some basic terminology used to describe the characteristics of different techniques will be discussed. Then the particulars of the techniques will be discussed.

2.2.1 Introduction and Terminology

Metaheuristic techniques are a set of high-level search techniques that are used on problems that have input spaces too large for a complete search. The underlying assumption of these techniques is that although there is no guarantee they will find the

```
1: generate initial candidate solution(s)
2: repeat
3:     evaluate candidate solution(s)
4:     generate new candidate solution(s)
5:     move to new solution(s) or keep old one(s)
6: until stopping condition is reached
7: report best candidate solution found
```

Figure 2.8: Metaheuristic pseudocode

optimal solution, finding a near-optimal solution is better than never finding the optimal solution. Again, it is assumed that near-optimal solutions exist.

The two key components of a metaheuristic technique are

1. objective function evaluation
2. new solution(s) generation

These two steps can be thought of as 1) where the search is and 2) where the search should go. The objective function is used to measure the value of the candidate solution(s). This function has to decode the candidate solution(s) and evaluate its desirability. Using that information the technique then forms its next generation candidate solution. This next generation may or may not be an improvement in terms of the objective function. These two steps are alternated until some stopping condition is reached. The pseudocode in Figure 2.8 describes, at a very abstract level, how metaheuristic search techniques work.

There are several ways to describe the characteristics of these techniques. Some of the more important characteristics are

- is the search local or global?
- how is the neighborhood determined?
- what type of solution encoding is needed?
- how are exploration and exploitation balanced?
- how robust is the technique is the presence of numerous local optima?

A search technique can be local or global, depending on whether only solutions close to the current solution are considered when new candidate solutions are generated. The neighborhood is the set of solutions that are considered before a move is made. How this set is determined may not be obvious. Higher dimensionality functions will have larger neighborhoods, since the current solution can move in many directions. This may require restricting in which directions the current solution can be modified when constructing its neighborhood. Mixed typed or mixed unit inputs may require domain knowledge to decide how great a change in one direction is equal to a change in another direction. The candidate solutions need to be encoded in a way that the neighborhood operators can work on and the objective function can efficiently evaluate. Every search has to balance exploration and exploitation. A search *explores* when it evaluates candidate solutions from portions of input space it hasn't visited before. But the search must also investigate regions of the input space where it has found promising solutions. This is called *exploiting*. If a technique sends too much time exploiting previously visited regions of the input space, it may get stuck in a local optimum. But if a search never exploits known locations of promising candidate solutions, solution quality may suffer.

A metaheuristic search technique has to balance when it explores new regions of the input space and when it exploits knowledge it has already learned. Regardless of how exploration and exploitation are balanced, since all metaheuristic search techniques are incomplete, they may still get stuck in local optima. Because of this each technique should have some method for escaping from local optima.

2.2.2 Gradient Based Techniques

In mathematics the gradient operator (∇) can be used to find the direction of greatest change for a multivariate function whose partial derivatives exist. When optimizing analytic functions the gradient can often be calculated analytically. For problem domains where the gradient can not be calculated analytically, it must be done numerically. By evaluating the objective function at points in the neighborhood of the current candidate solution, an estimation of the gradient can be calculated. As discussed in §2.2.1, how to generate this neighborhood is not always obvious and often requires some domain knowledge.

Hill climbing The simplest gradient based technique is hill climbing. A hill climber investigates the area around its current position and moves the candidate solution to a neighbor with a better score. There are several variants of the hill climber. The climber can move to the first candidate it finds that is better than the current solution. This is called *first-ascent* hill climbing. The climber could alternatively investigate its entire neighborhood and move the solution to the best neighbor it finds. This is called *steepest-ascent* hill climbing. Hill climbers terminate when they can not find a neighbor that has a better score than the current candidate solution. This means that the current solution is a local optimum.

Since gradient based techniques are not guaranteed to find global optima, methods that use hill climbers often do multiple restarts. After a local optimum is found, the hill climber is restarted from a new random starting point. Using restarts usually leads to the finding of a different optimum, increasing the likelihood that a high quality near-optimal solution is found. Of course this still does not guarantee finding the global optimum.

Simulated annealing Simulated annealing (SA) [24, 76] is a local search algorithm very similar to hill climbing, with one important difference. SAs will probabilistically make a move that does not improve the objective function according to the Boltzmann factor $e^{-(\nabla E/T)}$, where T is the *temperature* and ∇E is the change in the objective function. Note that $e^{-(\nabla E/T)} \rightarrow 0$ as $T \rightarrow 0$, i. e. SAs converge to hill climbers in the limit $T \rightarrow 0$. Also note that $e^{-(\nabla E/T)} \rightarrow 0$ as $\nabla E \rightarrow \infty$, i. e. the worse the move, the lower the probability it will be taken. Figure 2.9 shows a how a search might proceed

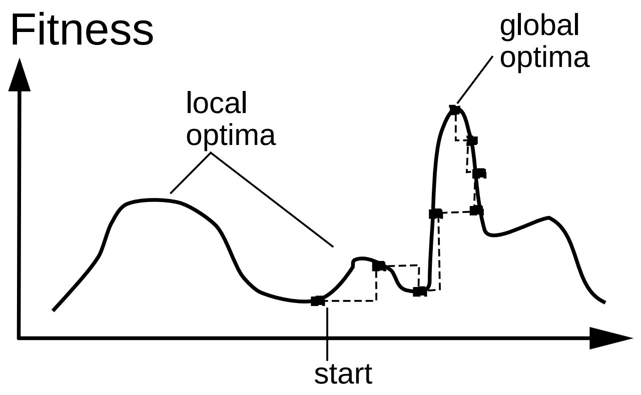


Figure 2.9: Typical simulated annealer

in a 2-D example, if a simulated annealer was used.

The cooling schedule (how the temperature changes during the search) is the way that SAs control the trade-off between exploration and exploitation. At the start of an SA run, when the temperature is high, the technique makes many bad moves and explores the search space. But as the temperature is lowered, the technique makes fewer and fewer bad moves and exploits what was learned (the possible location of a promising local optimum) during the exploring phase. Hence the benefit of allowing the SA to make bad moves is that these bad moves increase the amount of exploring done and decrease the likelihood of the search getting stuck in low quality local optima. As with restarting hill climbers, this technique can not guarantee that the global optimum will be found.

2.2.3 Evolutionary Algorithms

Evolutionary algorithms (EA) are inspired by Darwinian evolution developed in biology. The key parts of evolutionary algorithms are

- the population of candidate solutions
- the candidate solutions are scored, based on how well they solve the problem
- the candidate solutions are changed some to explore new possible solutions

Since more than one candidate solution is being considered, evolutionary algorithms are non-local techniques. Candidate solutions in EAs are encoded as *chromosomes*. These chromosomes are usually bit vectors, where each bit represents a single attribute of the solution. Typically the chromosomes are fixed length, but Whitley et. al. [137] has explored encoding master genes. These master genes can turn off other genes, so that while the chromosome stays a constant length during the search, the information expressed changes depending on how these master genes are mutated during the search. Exploitation and exploration are balanced by the values of the mutation operators. If mutation events are very common, the search performs more exploration, since the mutations push the search into new regions of the search space. If mutation events are not common, the search performs more exploitation, as the search stays in same region of the search space. An EA may have static mutation rates that don't change during the search, although it is common to find experimentation with adaptive mutation rates [25, 143]. The two most common EAs are genetic algorithms (GAs) and genetic programming (GPs). They will be discussed in the following section. In addition Evolutionary Programming [46] has been developed, but it will not be discussed here.

Genetic Algorithms and Genetic Programming

The following steps compose one *generation* in a genetic algorithm [51, 61]

1. evaluate the individuals to determine their fitness score
2. construct a new intermediate population based on those fitness scores
3. construct the next generation by using mutator operators on the intermediate population

The evaluation of the individuals is done by the objective function (usually called the fitness function by GA practitioners). This function must be called once for every individual in the population, every generation, hence it must be only modestly computational intensive.

Their are several ways that the intermediate population can be constructed, including proportional selection (also known as roulette wheel selection), stochastic universal sampling, and tournament selection. Each of these selection methods has been studied to determine some their characteristics. For example tournament selection was studied by Miller and Goldberg [101], who investigated the relationship between selection pressure, tournament size, and noisy objective functions, while stochastic universal sampling was studied by Baker [8], who demonstrated that this method was unbiased.

The two mutation operators used are the point mutation operator and the crossover operator. The point mutation operator works by probabilistically flipping individual bits among the chromosome population. An individual may have 0 or more bits flipped by the point mutation operator. A pictorial representation of the point mutation operator is shown in Figure 2.10. The crossover operator works by probabilistically exchanging information between two individuals in the population. There are several versions of the crossover operator, including the one-point, the two-point, and the uniform. The one-point operator selects a single location along the chromosome and exchanges the portion of the chromosome after that point from one individual with the corresponding portion from another individual. The two-point operator selects two locations along the chromosome and swaps the information between those two points between two individuals. The uniform operator probabilistically swaps each gene along the chromosome. A pictorial representation of a one-point crossover operator is shown in Figure 2.11.

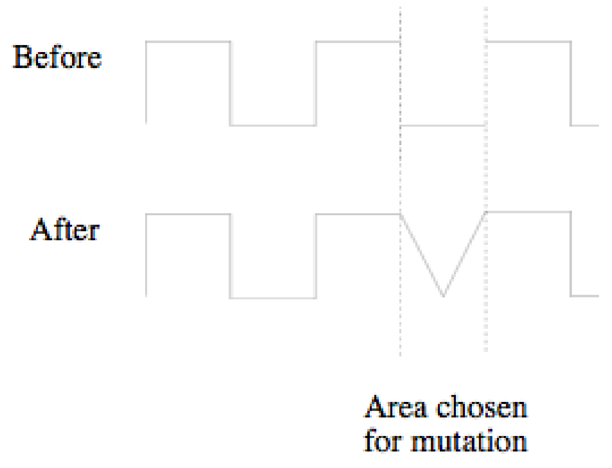


Figure 2.10: Example of a point mutation [115]

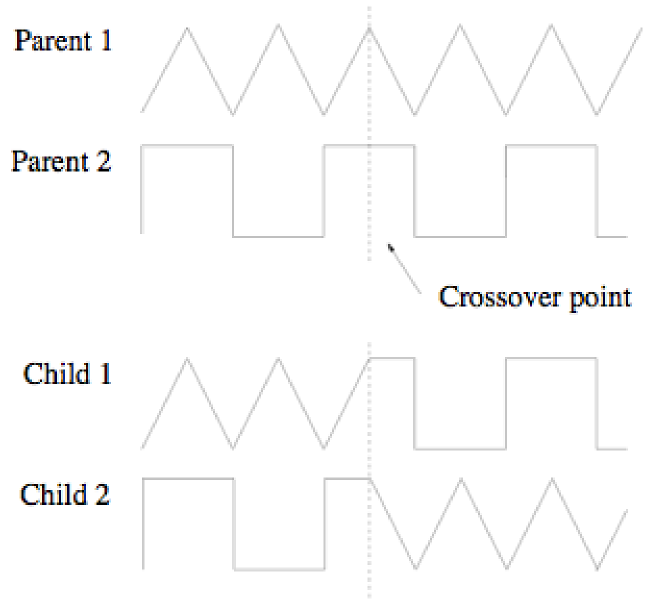


Figure 2.11: Example of a one-point crossover event [115]

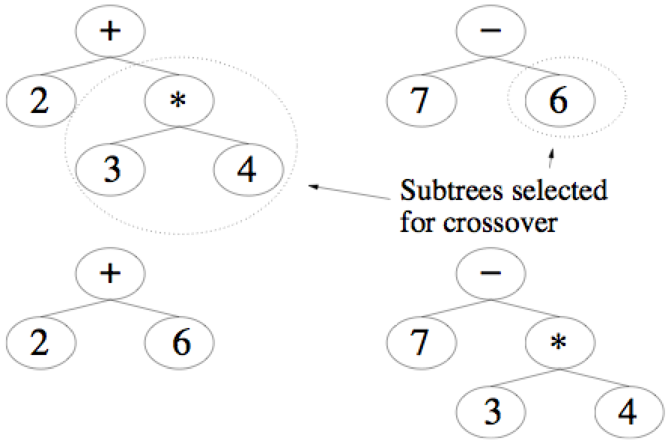


Figure 2.12: Example of crossover event in genetic programming [115]

There is also research on the topic of encoding schemes. Natural encoding has been advocated [99] since that eliminates the need to encode and decode the individuals before scoring them, but new mutation and cross-over operators need to be developed for each new problem. Runtime speed ups have also been been reported with natural representation [70]. The topic of encoding schemes will be revisited in §2.3.1.

Genetic programming [78, 106] is similar to genetic algorithms, except that the representation being "evolved" is a computer program. To ensure that all mutations are still valid programs, the representation chosen is often an abstract syntax tree. Since this representation is so different from the typical representation used in GA, specialized mutation operators have to be used. For example, a crossover event in a GP search might look like Figure 2.12. Besides specialized mutation operators, GPs use the same three basic steps discussed at the start of this section.

2.2.4 Tabu Search

Tabu search [49, 50] (a type of adaptive memory programming, or AMP) attempts to use memory in a flexible fashion, without recording the exact history of the search. Like SA, tabu search is a local algorithm. However, its neighborhood depends on previous moves. After each move is made a *tabu list* is updated based on the move. The list may contain the exact move, or it may contain certain attributes of the move. Members of this list have some form of tenure, so that they are eventually evicted from the list. This makes tabu search more scalable to large problems, since it does not attempt to store a complete history of its progress. In addition, multiple tabu lists might be maintained. For example, when solving the *minimum k-tree* problem [82] it is useful to maintain a separate list for edges that are dropped from the current solution and for edges that are added to the current solution. When constructing future neighborhoods, moves that match a move on the list (or share an attribute that is on the list) are not considered. That is, tabu tries to explore regions of the space that it has not visited previously. It is left to the practitioner to decide if a move should be considered tabu if it has any attributes that are in the tabu list or whether it must have several (or all) attributes on the tabu list to be considered tabu. Similarly, when maintaining multiple tabu lists, a move may be considered tabu if it has attributes on any of the lists, or if a move can be considered tabu only if its attributes are on several (or all) of the tabu lists.

The time each move (or attribute) remains on a tabu list is called its *tabu tenure*. This value of this tenure can be very flexible. If multiple lists are being maintained, each list can have a different tenure. The tenure for the different lists can be static or dynamic. The value of the tenure is the main way tabu search balances exploration with exploitation. Longer tenures forces tabu search to explore more the of the search space, as large parts will be considered tabu. Shorter tenures increases exploitation, allowing the technique to stay in a smaller part of the search space (without necessarily requiring it). For this reason, the tabu tenure may be a monotonically decreasing function, making it similar to the cooling schedule in SAs. This results in more exploration at the beginning of the search, with more exploitation as the tabu tenure decreases.

2.3 Search Based Software Engineering

Starting in the mid-1990's various researchers working in the field of software engineering started looking at how the field of optimization might be applied to their field. Many other engineering disciplines incorporated optimization techniques into their standard toolboxes long ago. These include mechanical engineering [79, 118], chemical engineering [16, 74], medical and biomedical engineering [110, 117, 135], civil engineering [7, 14, 44, 67], and electronic engineering [13, 33, 102]. In contrast to the multitude of areas outside software engineering, researchers in the field of software engineering are just starting to realize the applicability of search based optimization to their problem domain. This section will first describe how to view software engineering problems as search problems (§2.3.1). The key insight in search based software engineering is that many standard software engineering problems are not amenable to analytic solutions since the mathematics of the problem does not permit symbolic analysis nor complete methods since the size of the search space is too large to make such a search feasible. The rest of the section will review some of the research done in software engineering that has applied optimization techniques (§2.3.2 - §2.3.4).

2.3.1 Rethinking Software Engineering as Numeric Optimization

As described in [32,58] there are three key elements to rethinking a software engineering problem as a search problem.

1. The problem has to be described in a manipulatable language. The metaheuristic technique used must be able to generate new proposed solutions automatically using this language.
2. An objective function must be developed. This function needs to map proposed solutions to a single numeric output. This output then is either minimized or maximized (i.e. optimized) by the metaheuristic technique being employed.
3. Transformation operators that work on proposed solutions also need to be developed. This might mean defining a neighborhood function that generates solutions "close" to another solution based on some criteria. Or it might mean taking a single proposed solution and changing some aspects to generate a new

proposed solution.

If these conditions are not met, using a metaheuristic search technique may not be possible or may not be necessary.

Representation Solutions to the problem need to be encoded in such a way that the various operators of the search technique being used can operate on them. Therefore, the first issue that must be addressed when applying a search technique is to find a suitable representation for candidate solutions.

The problem of representation can be more complex than might be first assumed, because the most natural encoding may not be the best scheme for the search technique being used. For instance, consider the bit encoding of an integer, as might be required when using a genetic algorithm. It is possible to represent integers as "pure" binary numbers. One would be represented as 0001, two as 0010, three as 0011, and so on. According to the rules of arithmetic, seven can be transformed into eight through only one application of the successor function. But in pure binary seven, 0111, must have four bits flipped to be transformed into eight, 1000. Since the mutation operator in a genetic algorithm works on the bit representation, seven is no longer only one operation away from eight. For this reason *gray* encoding is preferred. With gray encoding seven would be 0100 and eight would be 1100[3]. Now seven and eight are

[3]Other gray encodings are possible, but this is the must popular.

only one mutation apart. It has been reported that gray encoding outperforms binary encoding [68, 136].

[125, 133] have also pointed out that both binary and gray encoding of real or integer numbers that have a restricted range can lead to the mutation and crossover operators that create out of bounds values. This means that after each mutation or crossover event the chromosome has to be inspected and, if necessary, repaired. Or if this possibility is ignored the semantics of a language with restricted range types (e. g. Ada) can be violated. They suggest using real-valued encoding to avoid this problem. [39] also argues for using real-valued encodings with evolutionary techniques.

Picking an encoding scheme must clearly include not only an understanding of the search technique but domain knowledge as well.

Objective function The next problem is developing an objective function. Software engineers are used to collecting software metrics, so those metrics are a ready supply of objective functions. For example, path or branch coverage for test case generation can be used as objective functions. There are a few new issues that should be addressed before these metrics, which were developed for other purposes, can be used as objective functions. [57] lays out four requirements for a metric function to be used as an objective function, each of which will be discussed in more detail below.

1. the input space of the metric function shouldn't be small enough to allow an exhaustive search
2. the function should have no known optimal solutions
3. the function must be computationally efficient
4. the function should be approximately continuous

Input space size and known optimal solutions Items 1 and 2 have bearing on the question of whether a search would beneficial. If the input space of the objective function is small enough to permit exhaustive search, then there is nothing to gain by using a incomplete search algorithm. Exactly how big is too big will depend on the hardware available and the computational costs of the objective function. For example, mission critical tasks might have the budget to allow the use of long periods of time on large amounts of hardware to tackle very large search spaces. Many of the case studies presented in chapter 4 have 2^{55} (10^{17}) or more possible input vectors, so it is unlikely that any amount of hardware or budget would allow a reasonable chance to do a complete search. The objective function must also have no known optimal solution for there to be any gain to using a metaheuristic search. For example, if a 3-SAT proposition has a single unsatisfiable clause (which can be found in linear time [48]), there would be no point in conducting an expensive search for that particular proposition. Any analytic solution should be preferred to a solution found through search, and any near-optimal solution found analytically would also be preferred. For example, an analytically found near-optimal solution might have a upper bound on its distance from the unknown optimal solution. This type of information would not be found by a metaheuristic search technique.

Objective function efficiency and continuity Items 3 and 4 have bearing on whether an effective search would be practical. While the exact number of times the objective function is evaluated depends on the technique being used, it will most likely be at least 10^4 and possibly more than 10^7. One of the main driving forces behind the development of ITL is reducing the needed number of evaluations of the objective function. As a search technique moves around the input space it must have some hint that it is moving in an undesirable or desirable direction. For that reason, the more continuous the objective function, the more efficiently we can expect the search to progress. If there exist many near-optimal solutions at points of high discontinuity, the probability that any search technique will find them approaches the probability that a random search would find them, as the discontinuity of the objective function increases. We encountered this problem in the studies presented in chapter 5 and chapter 6.

The next three sections will discuss the application of search-based software engineering to the fields of testing (§2.3.2), cost estimation (§2.3.3), and requirements analysis (§2.3.4). These fields are reviewed because cost estimation and requirements analysis are early life cycle activities, similar to the study presented in chapter 4, and testing is a late life cycle activity, similar to the studies presented in chapter 5).

2.3.2 Testing

One of the first areas in software engineering to use optimization techniques was test case generation. Test case generation has a few properties that make it uniquely suited for search-based optimization. First, there are many metrics that have already been developed by the testing community that serve as ready-made candidates for objective functions. There are static measures like the McCabe complexity values [84], and dynamic measures like path or branch coverage. Second, the input space to even the most trivial programs is far too large for any exhaustive search. Third, static program analysis is difficult for most real world programs. It is a well known result that it is not computationally possible to decide even the simplest program properties[4] for any Turing-complete language. Fourth, manual construction of test data is quite expensive. Testing often accounts for up to 50% of the cost for typical software projects [12]. For these reasons there has been much work on applying search techniques to the problem of automatic test case generation.

An early work using search based techniques on automatic test case generation

[4]For example, the halting property

is [140]. Since then there have been improvements to the original work [68, 77, 86] by removing the need for a human tester to select a path and by changing the way the objective function is calculated for conditional branches. For example, control dependence graph analysis has been included in the objective function [109]. If the search is trying to execute branch *c*, which is dominated by branch *b*, which in turn is dominated by branch *a*, the objective function reflects how many of the branches *a, b,* or *c* are executed. However, this type of function suffers because the objective function has a large plateau in the areas of the input space that don't execute branch *a*. The objective function doesn't provide any distance guidance as to how to execute any particular branch. Even with this disadvantage [109] still reported significant speed ups when compared to a purely random search. Later other researchers, [125, 129], added a distance metric to the objective function to remove some of the discontinuity of the function.

Early work using EA techniques to generate test data using a *Z* specification (see [122] for a full discussion of the *Z* specification) can be found in [69]. They used the canonical triangle classification problem and developed a *Z* specification for that program. The specification consisted of predicate statement in disjunctive normal form. Each disjunct was considered a path through the program. The objective function measured how close an input set came to satisfying one of the disjuncts. Success was reported for covering all nominal cases. More recent work on using specification driven test data generation has been done in [126, 129].

[134] investigated the objective landscape for timing behavior. They found that GAs outperformed SAs and they explained this difference by looking at the topology of the objective landscape. Recall that GAs perform global searches, since each individual in the population is initialized to a different location and cross-over events can generate individuals in radically different locations than their parents. The objective landscape had discontinuities wherever different paths through the control flow graph were taken. If the different paths had significantly different execution times, this creates a discontinuity at that location in the objective landscape. The landscape also had many plateaus. If different input values led to the same exact path being executed, their execution times were usually very close. These discontinuities and plateaus were less likely to deteriorate the performance of a global search algorithm like a GA.

Simulated Annealing (SA) is also a popular metaheuristic technique in automatic test data generation. For instance, [127, 128] investigate using SAs to find the worst case execution time and maximizing structural coverage. [126] uses pre/post conditions written in disjunctive normal form. The objective function was a distance metric that measures how close a given input comes during execution to violating the pre/post conditions.

Note the importance the objective function plays in all the work described in this section. The contributions of these publications are not new search techniques or even modifications to existing techniques, but rather new ways to formulate the objective function to allow the search to proceed more efficiently. Also note that symbolic analysis of the source code is necessary for the distance measures of branch points. The case studies presented by this thesis (chapter 5 and chapter 6) highlight the importance of a objective function that follows the advice laid out in §2.3.1.

For a broader survey of search based automatic test case generation see [87].

2.3.3 Cost Estimation

While a high quality estimate of a project's cost would be very useful to software planners, software cost estimations can be notoriously inaccurate. For example, [19] reports that 60% of large projects significantly overrun their estimates and 15% of software projects are never completed due to gross misestimation of development effort. A very popular technique for software cost estimation is the COCOMO [18] linear regression method[5]. The rest of this section will describe some efforts to improve software cost estimation by applying search-based techniques.

[5]Citeseer reports over 2000 citations to this work.

In [4] a sequential covering evolutionary algorithm was used to generate management rules to reduce the likelihood of effort and time overruns. This technique is a "divide-and-conquer" technique as rules are adjusted according to data points not already covered. Once a data point is covered by the rule it is removed from the pool of remaining points. The data points used in this study came from a dynamic model that had 12 numeric and ordinal inputs with restricted ranges. The two outputs of this model were development effort (developer-days) and development time (days to complete). The objective function combined the error for a rule, the support for the rule[6], and the coverage of the input space by the rule.

Although the technique can be generalized to any number of target classes, in this study the data points were split into two classes, GOOD and BAD. Those points that had both a development effort and time below a nominal value were labeled GOOD, all other points were labeled BAD. The nominal values for development effort and time were generated in the following fashion. An initial estimate of development effort and time was generated assigning values to the various inputs based on discussions with the project manager. The outputs of this initial run of the dynamic model were used as the nominal values. After finding the nominal values for time and effort, the simulator was then run randomly choosing inputs. The results of these runs were entered into a database. The database then became the source of the data points used by their

[6]The absolute number of correct classifications

sequential covering algorithm. They present several rules generated by their algorithm that the dynamic model predicts would have led to a 6% reduction in effort and a 9% reduction in time.

Similar to this thesis, [6] investigated using data mining techniques for software effort estimation. The work presented differs in a significant way from the current work. In [6] a dynamic model is used to generate data, but there is no feedback from the learner to the model. All records generated from the model are stored to a database and then the learner works on those records, after the simulation has ended. They do acknowledge that smaller rules should be preferred, but leave that advice to the user. In other words, their learner does not systematically search for smaller solutions. They also have the problem of needing to discretize the output parameters of the model for their learner. Their solution is for the project manager to define a "cutting-section". The cutting-section defines the maximum allowed values for each of the three outputs of the model (cost, development time, and quality). If any of the those three limits is exceeded the record is labeled "bad", otherwise the record is labeled "good". Their objective function (adapted from [4]) is slightly complex, taking into account the uniformity of the region, the coverage of the region, and the total hyper-volume[7] of the region. No attempt is made to demonstrate the superiority of the induced rules, rather just a validation of the data mining technique. Other modeling work with software cost

[7]Called amplitude in the original paper

estimation includes [37, 45, 120, 131].

Extreme sampling (discussed in §3.1.4) was developed independently of [4, 6] and follows a similar logic, but with a different motivation. The discretization methods in [4, 6] are used to guide the search towards acceptable solutions, because business restrictions made some solutions unacceptable. The extreme sampling discretization method was developed to complement the search heuristic used by ITL. §3.1.4 contains a fuller description of extreme sampling.

Other techniques not discussed in §2.1 and §2.2 are used in cost estimation. For instance in [36] linear regression, neural nets[8], and genetic programming were used to build cost estimators. They studied 46 student projects developed by a total of 148 students. All projects were accounting information systems for hypothetical firms. Different groups had different hypothetical firms with different requirements. In addition all the projects studied were developed in the same language. They found that neural nets slightly outperformed GP, and linear regression performed slightly worse.

[8]Neural nets are a another machine learning technique not discussed in this thesis, see [60] for a complete description.

2.3.4 Requirements Engineering

Since all software projects have a limited set of resources (money, staff, hardware, etc.), project managers must balance the expenditures of these resources to meet the specific goals of their projects. Additionally, it is an accepted fact that the earlier a fault is discovered, the less costly it will be to correct it [18, 22, 85, 105]. For these two reasons (and others) organizations have been spending more attention on the field of requirements engineering (RE). Search techniques have not been applied to requirements engineering as extensively as they have been in other fields. For instance, [107] does not even mention search-based techniques in its requirements engineering road map. To pick an example from that road map, [142] discusses model building and their argument for building models early in the requirements phase[9]. There is no discussion of, or even citation to, the application of search techniques to requirements engineering.

This lack of a large body of work in applying search to RE does not reflect any inherent inapplicability of search techniques to the problem of requirements engineering. Researchers in requirements engineering have already developed numeric metrics that can be easily used as objective functions. Most of these are some combination of projected cost of the project, risk inherent in the project, the value of the achieved

[9]As opposed to late in or after the end of the requirements phase

requirements, defect injection rate, and expected development time. As in the field of cost estimation, dynamic models can be used to model the interactions between the elements a requirements engineer is trying to study. For example, DDP (see §4.1) is a tool that facilities the development and analysis of risk-cost models.

The input space of a typical requirements problem is very large. Even if the input vector is boolean, a few dozen requirements can not reasonably be searched exhaustively. Some techniques have been explored, e. g. the Analytic Hierarchy Process [116] and Multi-criteria Decision Making, that are solvable by analytic techniques. But these techniques have some restrictions that limit their applicability. They have trouble dealing with dependency and ordering properties of requirements and costs. The use of dynamic modeling eliminates the possibility of using one of these analytic techniques. But dynamic models can be used to capture complex features, such as interdependent requirements or mitigations. The DDP tool already mentioned is capable of handling such dependencies.

Finally it should be noted that model building can be quite expensive (see discussion of DDP in §4.1), so developing high-quality solutions will help maximize the return-on-investment.

Analytic Hierarchy Process The Analytic Hierarchy Process (AHP) [116] is a method that uses pairwise comparisons to calculate the relative importance of any criterion. If

n alternatives are possible, a square matrix of size n is built. Each element in the matrix is the comparative importance of the alternative in the row to the alternative in the column. If the alternative in the row is more important an integer [1,9] is used, the reciprocals being used if the column is more important than the row. The matrix should have 1's along the diagonal. Various matrix operations are then performed which lead to a relative ranking of the alternatives and a consistency ratio. Advocates of the AHP claim that pairwise comparisons reduce the error common to this type of human judgment, when compared to absolute scores [81].

Since there has been little work applying search, we will discuss some related work on requirements engineering that doesn't utilize search based techniques. [73] developed a cost-value technique that uses the AHP to inform managers of the most cost effective requirements. They present two case studies from Ericsson Radio Systems, a telecommunications manufacturer. They had a group of project members develop the high level requirements for two projects (one currently in development, the other a mature product that had already had three major public releases). The group then filled in the pairwise matrix, first for the the value of each requirement, and then for the cost of implementing those requirements. The different requirements were then plotted, on a cost vs value graph. It is then very easy to see which requirements have the lowest cost-to-value ratio and which have the highest. Dropping the three requirements (out of 14) with the highest ratios still captured 94% of the value for only 78% of the cost in the first project; while dropping the three requirements (out of 11) with the highest ratios in the second project led to 95% of the value for only 75% of the cost. This is a archetypal example of requirements analysis balancing two or more competing goals. A simple technique like AHP can be used with these examples in large part because the models studied do not allow interdependencies. It is unlikely that it is possible for a group of experts to develop a completely orthogonal set of requirements. Developing an orthogonal set of requirements will get more difficult as the set of requirements grows larger[10]. In a later paper [72] they compare six different methods for prioritizing. None of the six techniques deal with the possibility of dependencies between the requirements.

[10]Due to the exponential growth of the number of possible edges as the number of nodes increases.

CHAPTER 3

ELEMENTS OF ITERATIVE TREATMENT LEARNING

This chapter describes Iterative Treatment Learning (ITL) in detail. First we will discuss topics related to ITL's use as a metaheuristic search technique (§3.1). Next we will discuss topics related to ITL's use as a model-based development tool (§3.2).

3.1 ITL as Search Technique

Iterative Treatment Learning has been previously introduced by Menzies et. al. [89,90]. But this previous work has not placed ITL in the context of metaheuristic search. This section will introduce the reader to several important features of ITL as they relate to ITL's use as a metaheuristic search technique.

3.1.1 Search Components

ITL has the following three key steps

1. objective function evaluation
2. discretization

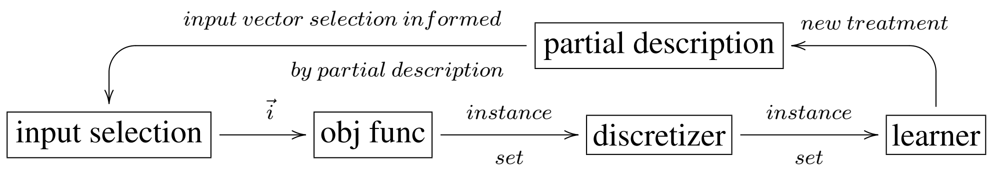


Figure 3.1: Diagrammatic view of ITL

3. treatment learning

The three steps together make an *iteration*. Figure 3.1 shows these steps, as well as the types of inputs to each step.

The objective function decodes candidate solutions and scores each instance according to some criteria the user has decided on. As with the other search techniques described, the only restriction on the form of the objective function is that it outputs a single numeric value. Since treatment learning only targets ordinal target classes, ITL must discretize the output of the objective function before calling the treatment learner. The discretizer used by ITL will be discussed in §3.1.4. The treatment learner is responsible for generating the small theories that ITL accumulates as the last step of each iteration. It is the conjunction of these theories that constitutes ITL's solution when it finishes searching.

task number	schedule position
$task_1$	6
$task_2$	7
$task_3$	32
$task_4$	2
...	
$task_n$	41

Table 3.1: Complete description for a hypothetical scheduling problem

3.1.2 Solution Form

Recall from §2.1.5 that the theories returned by treatment learners comment on only a few of the available attributes. This means that after several iterations the candidate solution found by ITL does not comment on all the available attributes. We call this a *partial description*. Contrast this with the *complete descriptions* returned by the search techniques discussed in §2.2. These complete descriptions comment on all the available attributes. For example, when solving a scheduling problem they would return an exact schedule, like the one shown in Table 3.1. A partial description only assigns values or ranges to some of the attributes. In the scheduling example a partial description would resemble the one shown in Table 3.2.

We assert that these partial descriptions have value, particularly for management decisions. Because partial descriptions don't comment on all available attributes and don't necessarily assign a specific value to the attributes it does comment on, the descriptions are intrinsically more flexible. If new design constraints arise at a later date,

task number	schedule position
$task_1$	$6 \leq p_1 \leq 10$
$task_2$	unassigned
$task_3$	unassigned
$task_4$	$p_4 > 36$
	...
$task_n$	$40 \leq p_n \leq 41$

Table 3.2: Partial description for a hypothetical scheduling problem

they will be less likely to contradict a partial description. Put another way, if a single attribute of a complete description cannot be assigned the way it was assigned in the solution, there is no way to know its effect on solution quality without re-evaluating the new solution and possibly conducting a new search. Since a change to the allowed attribute values may not invalidate the entire region describe by the partial description, solutions in the set of solutions allowed by the partial description can still be used. The difficulty of conducting a new search should not be understated. Expertise in metaheuristic search is not likely to ever become a common skill among business managers. If the expert who conducted the original search is not available to the business user, a complete description that is no longer possible becomes worthless.

A possible drawback of partial descriptions is that, since they don't describe a single solution, but rather a set of solutions, the variance in that set could be so high that the partial description is not usable. Without going into the particulars of feature subset selection [55, 96, 139], the success of techniques that throw away attributes before training on the remaining attributes is indisputable. Building theories without commenting on all the available attributes is exactly what ITL does, so we have good reason to conjecture that it is possible that the variance of partial descriptions will be low. We investigate this conjecture in §4.4.

This concern may seem at odds with Miller's work [100], which suggests that removing attributes *reduces* the variance when using machine learners. But Miller is concerned with the learned theories from machine learners and whether the exact form of theories learned from different slices of the same data set will be similar to each other. The variance in the theories is reduced because by removing attributes available to the learner, the learned theory is less over-fit to the particularities of the data set. §4.4 investigates the variance of the *performance* of partial descriptions, not the variance of the different partial descriptions themselves.

3.1.3 ITL's Metaheuristic Search Characteristics

Since ITL is being presented as a metaheuristic search technique, ITL should also be described in the same terms as the metaheuristic techniques presented in §2.2. Some of the deficiencies mentioned in this section will be discussed further in chapter 8. These characteristics are

1. global search strategy

2. neighborhood defined by partial description
3. solution is encoded as a conjunction of attribute-value pairs
4. no turning knob for exploration vs. exploitation
5. no explicit mechanism to escape from local optima

Since ITL initially samples from the entire search space and only modestly reduces the size of the portion of that space each iteration, it should be considered a global search strategy. Although the neighborhood of the current solution is never enumerated during ITL's search, it can be thought of as all possible solutions that agree with the current partial description. This means that only points that agree with the current partial description are explored in the next iteration. As discussed earlier in this chapter, the candidate solution for ITL is encoded as a conjunction of attribute-value pairs. The most undesirable characteristic of ITL at the current time is the absence of some way to tune the exploration vs. exploitation trade-off to a specific domain. Nor is there any way for the technique to dynamically adjust the trade-off during its search. Instead, ITL becomes monotonically more exploitative as treatments are added to the candidate solution and the search space is reduced in size each iteration.

Once a treatment is added to the partial solution there is no operation that might remove that treatment, so there is no mechanism for ITL to escape from local optima. This is not as big a problem for ITL as it for some other techniques, such as hill climbing. Recall that hill climbing is a local search, therefore if a local optimum has a

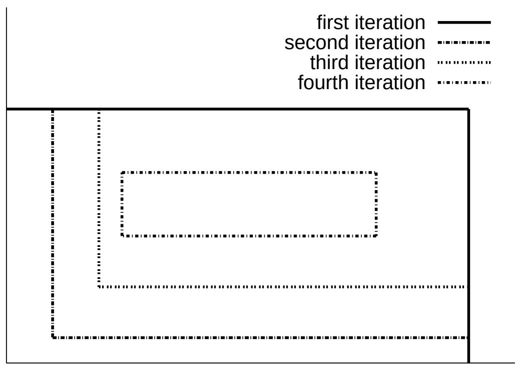


Figure 3.2: Search space sampling

large basin of attraction any initial solution starting in that basin will get stuck in that local optimum. ITL is a global search and hence is less suspectible to the presence of local optimum. Furthermore, since each iteration only reduces the size of the search space by a small amount (recall from §2.1.5 the small size of typical treatments), it is unlikely that many near-optimal solutions will be excluded from the search. See Figure 3.2 for a simple 2-D example of how the search space might be reduced during a typical search by ITL.

3.1.4 Extreme Sampling

Treatment learners (§2.1.5) produces theories that predict for ordinal classes (discrete classes that have an ordering), but since most useful objective functions produce real valued outputs, ITL must have some type of discretization policy. Note that while most discretization policies are applied to the input attributes, ITL needs a policy that discretizes the target class.

The original ITL work on cost/benefit models used a *striping* discretization policy. Four zones were demarcated by parallel lines drawn at $45°$ from the cost-axis. The zones were given exponentially increasing values, going from high cost/low benefit solutions to low cost/high benefit solutions. Figure 3.3 shows this policy graphically. Note that the diagonal lines are lines of constant benefit-to-cost ratios.

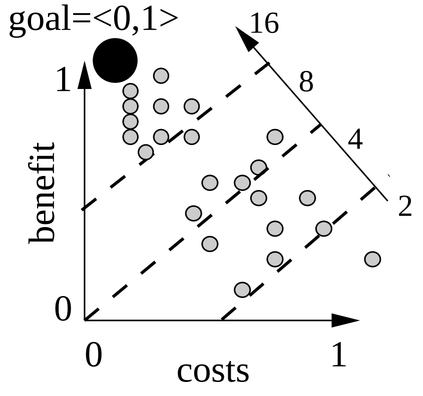


Figure 3.3: Diagonal striping

In this thesis we introduce a new discretization policy called *extreme sampling*. The intuition behind this idea is that, if treatment learning is going to be used as a search heuristic, having multiple target classes is unnecessary. There are only two classes, those that are close to the current best solution, *good*, and those that are not, *bad*. We developed a two-step process that takes a set of output vectors, $\{\vec{o}\}$, and labels some or all of them good or bad, shown in Figure 3.4. The first step is to take the n-dimensional $\vec{o}$ and map it to a single real number. We choose the euclidean distance function, which is simple and efficient and can work with any number of dimensions. The value for each coordinate is normalized by the maximum value for that coordinate to eliminate unit effects in the distance calculation. The distance is calculated from the theoretical best solution, which in our cost-benefit model is the point $(0, 1)$ in the cost-benefit plane (remember our coordinates have been normalized at this point). Now the distance values have to be mapped to our two classes, good and bad.

$$\vec{o} \xrightarrow{objective} [0-1] \xrightarrow{selector} good, bad$$

Figure 3.4: Extreme sampling

We developed three extreme sampling variants, which we call *selectors*, to map the distance values to our two classes shown in Figure 3.5. Each of our selectors has two control parameters, *M* and *N*. M is the batch size, i. e. the number of instances generated during each iteration. The first selector we developed, Best Or REst, *bore*, takes the N instances with the smallest distance and labels them good (shown in Figure 3.5a). The rest of the instances are labeled bad. After some initial experiments we developed two additional selectors. *bore′* (Figure 3.5b) puts N instances in the good class and a random sample of N instances from the rest and puts them in the bad class. It was thought this would allow M to be scaled up without increasing the run-time needed by the underlying learner (since the learner would only see $2N$ instances, instead of M instances). *wob* (Figure 3.5c), Worst Or Best, labels the N instances with the smallest distance measure as good and the N instances with the largest distance as bad. This selector was motived by the idea that the worst and the best instances might have the greatest contrast. And similar to *bore*′, *wob* only passes $2N$ instances to the underlying learner.

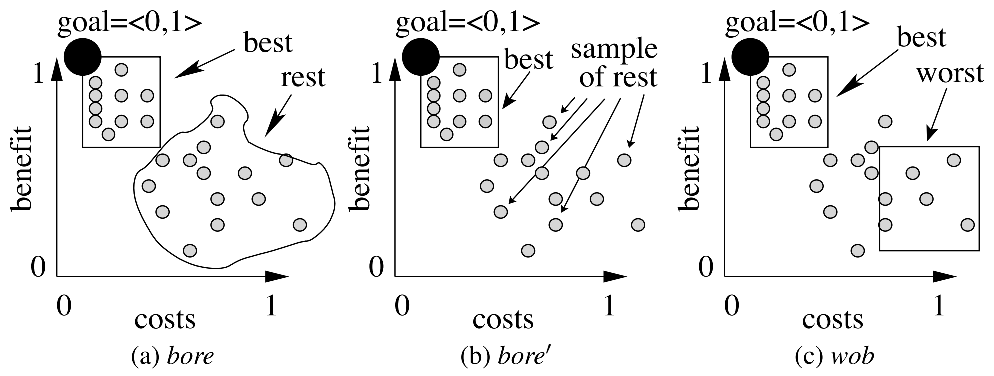


Figure 3.5: Three selectors developed

3.1.5 Search Strategy

In the original work on ITL it was imagined that a human expert would want to validate each treatment. In keeping with recasting ITL as a search heuristic, no human expert was used in the current implementation of ITL (see figure 3.1). Instead a simple greedy strategy was used to decide what treatment to use at the end of each iteration. The treatment learner suggests several treatments each time it is run, but ranks them according to their lift (with ties being broken by support). The best treatment returned by the treatment learner each iteration was used to constrain the next iteration. This strategy is also a strictly forward search; there is no back-tracking currently in ITL. Once a treatment is added to the partial description it cannot be removed by the search strategy.

Investigation into alternative search strategies

Initially we were concerned that search strategies that explore more might perform better than our simple greedy search. Therefore, we conducted a short study into using two different strategies, benchmarking them against our greedy search. The first alternative strategy completely ignored the current candidate solution when generating half the instances during each iteration. For example, if the current batch size was 500, 250 instances would be generated according to the current partial description, while another 250 would be drawn from the entire search space. The second alternative strategy was designed to allow ITL to search through multiple hyper-rectangles in the search space. At the end of each iteration, when the treatment learner returned several treatments, instead of just using the top treatment, all treatments that had a lift (see

$$\begin{tikzcd}
iteration_1 & iteration_2 & iteration_3 & iteration_4 \\
treatment_{1a} \arrow[dr] & treatment_{2a} \arrow[r] & treatment_{3a} \arrow[dr] & treatment_{4a} \\
treatment_{1b} & treatment_{2b} \arrow[dr] & treatment_{3b} & treatment_{4b} \\
treatment_{1c} \arrow[uur] & treatment_{2c} & treatment_{3c} \arrow[uur] & \\
treatment_{1d} & & treatment_{3d} &
\end{tikzcd}$$

Figure 3.6: Maintaining multiple lists of treatments. Notice that not all iterations find the same number of treatments that pass the restriction for inclusion into the table.

§2.1.5 for a definition of lift) within 20% of the lift of the best treatment were written to a list. Each iteration created a new list of treatments, and each list was remembered for the entire search. Only points that passed at least one treatment from each of the previous iterations were searched during the current iteration. Figure 3.6 shows these lists with two example paths through the lists. Each path through the lists was randomly generated, biased according the individual treatment's lift.

This study utilized the circuit model, which has a cursory description below. For a complete description of the model see [89].

Circuit model A qualitative model, previously developed in [89], was used to see if either of the new strategies developed could outperform the original greedy search. The model is built from *switches, bulbs, openers* and *closers*. A basic element was built using three bulbs and three switches. Elements were then connected using the openers and closers. The openers and closers connected a bulb from one element to

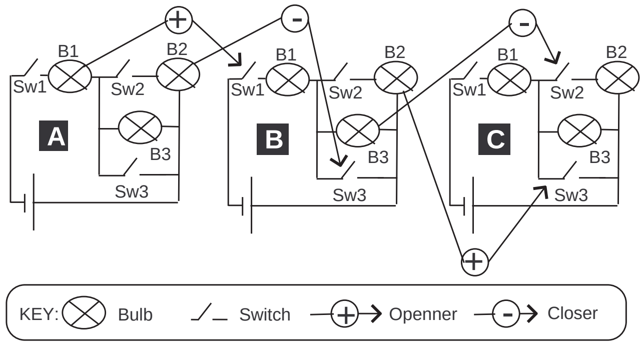
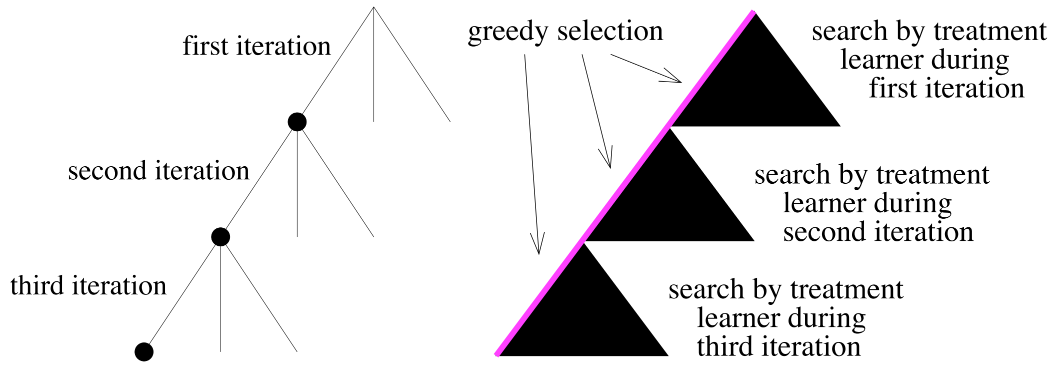


Figure 3.7: A qualitative network: Sw_i, B_i denote switches and bulbs. The network repeats the structure three times with labels $\{A, B, C\}$. Between each repetition are connectors that *open* or *close* switches depending on whether or not some bulb is lit or dark.

a switch from another and forced the switch open or closed depending on whether the bulb was lit or dark. Figure 3.7 shows a three element circuit. In this study we used two different sized models, one with three elements and one with six elements. In addition each model could be optimized in one dimension, maximizing the total number of shining bulbs, or in two dimensions, maximizing the ratio of shining bulbs to closed switches.

When compared to the original greedy search neither alternative strategy did well. The greedy search converged faster in all four versions of the circuit model, and found a higher quality solution in three of them.

Possible explanation for the performance of the greedy search Why does our greedy search seem to work so well? Figure 3.8a is a depiction of our greedy search through the treatments returned by our learner. But this is not the only searching being done by ITL. Recall from §2.1.5 that our treatment learner searches through dozens or hundreds of potential treatments according to its own ordering heuristic, *lift*$_1$. Figure 3.8b is a depiction of ITL's search including the heuristic search done by the treatment learner in between each iteration. Because the *lift*$_1$ heuristic works so well and the treatment learner does its own searching according to this heuristic, the search through the treatments can perform well without any additional exploring.


first iteration
second iteration
third iteration
greedy selection
search by treatment learner during first iteration
search by treatment learner during second iteration
search by treatment learner during third iteration

(a) greedy search between iterations (b) search done by ITL including learner search

Figure 3.8: Different views of ITL's search

3.2 ITL in Model Based Development

After putting ITL in a search context, we want to show how ITL can be used with model based development, to search through the models generated according to this development methodology. This section describes some of additional issues relating to ITL that come with using it in a model based environment.

3.2.1 Searching Through Models

With model based development, models are used by software engineering researchers to investigate different problems in different stages of the software life cycle. Earlier life stages might be modeled with a cost/benefit model, like DDP (described in §4.1), or a quality model like COQUALMO [27]. Later life stages like testing might be modeled with a digital logic circuit design (like these in §5.2). The purpose of using a model is to find which input settings of the model lead to certain desired behaviors. These inputs need to be controllable or observable quantities so that information gained from exploring the model is actionable outside the model. For example, a researcher might use a quality model to inform a business manager of the minimum level of expertise needed by a development team to reach an acceptable level of quality. Or a researcher might use a cost/benefit model to develop a plan that balances the projected cost of a project with its projected value.

Using ITL for early stage life cycle models can also reduces one of the potential drawbacks of ITL's partial descriptions. The earlier a model is constructed the more likely there will be changes before project completion. But since the earlier potential problems are found the greater the saving, postponing analysis would be a self-defeating solution. So the uncertainty inherent in ITL's solution due to the variance of a partial solution will be partially washed out by the uncertainty in the initial estimates made during model construction.

But what if the model has many uncertain inputs? This is when reformulating the original software engineering problem as a search problem is most useful. The model now becomes something that is executed thousands (or hundreds of thousands) of times, rather than something used to evaluate a single set of input values.

3.2.2 ITL as Model Controller

If we want to use ITL as a model controller to use with model based development, some changes have to be made in how ITL is implemented. Using ITL to control model execution breaks the objective function into two parts (see Figure 3.9). The objective function doesn't decode the candidate solutions, instead the model takes the candidate solution as an input vector and returns an output vector. The objective function takes that output vector and returns a single numeric value. The reason we do this has to do with the assumptions we make about the model (which will be discussed in §3.2.3). Now the theories returned by the treatment learner are used to restrict which input

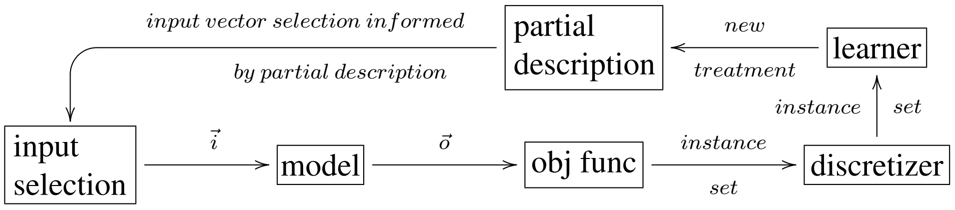


Figure 3.9: Diagrammatic view of ITL as a model controller

vectors are given to the model. Again, when ITL finishes searching, the conjunction of these theories constitutes ITL's solution.

3.2.3 Models

There are a few important assumptions we made about the models that would be used with ITL, which are

1. they have a well-defined input vector, $\vec{i}$, that can be supplied by ITL
2. they are black boxes, i. e. their internals can not be modified
3. they have a well-defined output vector, $\vec{o}$, that can be accessed by ITL

ITL must be able to control the region sampled by the model, so there must be some mechanism for ITL to direct the selection of input vectors. For example, the models used in chapter 4 can be given a file listing the points in the input space that should be visited, so ITL only has to generate that file in the correct format. Since ITL must have a low overhead for use, it cannot need access to any model's internals. This way the model can simply be a pre-compiled application. This greatly lowers the cost of using ITL with a model developed without any consideration given to the possible use of ITL. Finally the model must have an output that can be accessed by ITL, because the learner and discretizer must be able to read all the points generated by the model. The model could output something to the file system in a format that can be decoded by ITL, for instance.

CHAPTER 4

REQUIREMENTS ENGINEERING

The main case study presented in this thesis uses ITL to optimize cost/benefit models. In this chapter we describe our models and their execution framework, DDP, (§4.1), investigate the effect of different parameter settings on the performance of extreme sampling (§4.2), investigate the stability of the partial descriptions (§4.3), investigate the variance of the partial descriptions (§4.4), compare the performance of the original discretization policy with extreme sampling (§4.5), compare the performance of extreme sampling with the original simulated annealer in DDP (§4.6), and summarize the lessons learned from this study (§4.7).

4.1 Defect Detection and Prevention

The case studies in this chapter use models developed with the Defect Detection and Prevention (DDP) [34, 42] application. To familiarize the reader with this application, this section will describe DDP.

DDP is used at NASA's Jet Propulsion Laboratory to record a group's qualitative knowledge about design options of future deep-space satellite missions, their associated risks, and the costs of mitigations that can reduce those risks.

In DDP, a “design” is a decision about which set of mitigations to apply. One such “design” is better than another when it costs less, reduces the risks more, or achieves more requirements than some alternative design.

The design of DDP reflects the reality of group decision-making at JPL. Six to twenty experts are gathered together for short, intensive knowledge acquisition sessions, typically three or four half-day sessions. These sessions must be short since it is hard to gather together these busy experts for more than a very short period of time.

In those sessions, the DDP tool supports a graphical interface for the rapid entry of the assertions. Such rapid entry is essential, lest using the tool slow up the debate. Assertions from the experts are expressed by using an ultra-lightweight decision ontology. The ontology must be ultra-lightweight since:

- Only brief assertions can be collected in short knowledge acquisition sessions.
- If the assertions get more elaborate, then experts may be unable to understand technical arguments from outside their own field of expertise.
- Design rationale research cautions against elaborate notation languages since they can confuse the users [103]. Successful rational languages (e.g. QOC [121], DDP, etc.) all use very simple notations.

Hence, DDP assertions are either:

- *Requirements* (free text) describing the objectives & constraints of the mission and its development process;
- *Weights* (numbers) of requirements, reflecting their importance;
- *Risks* (free text), i.e. events that damage requirements;
- *Mitigations* (free text) describing actions that can reduce risks;
- *Costs* (numbers) of mitigations, i. e. the repair costs for correcting risks with these mitigations;
- *Mappings* that connect requirements, mitigations, and risks; or
- *Part-of relations,* which structure the collections of requirements, risks and mitigations.

This ontology is deliberately quite restrictive but even in this limited form, it has been useful for structuring and simplifying debates between NASA experts. For example, DDP has been applied to over a dozen applications to study advanced technologies such as (1) a computer memory device; (2) gyroscope design; (3) software code generation; (4) a low temperature experiment's apparatus; (5) an imaging device; (6) circuit board like fabrication; (7) micro electromechanical devices; (8) a sun sensor; (9) a motor controller; (10) photonics; and (11) interferometry. The DDP sessions have found cost savings exceeding $1 million in at least two of these studies, and lesser amounts (exceeding $100,000) in the other studies. These meetings have also generated numerous design improvements such as a savings of power or mass and a shifting of risks from uncertain architecture to better understood design. Also, at these meetings, some non-obvious significant risks have been identified and mitigated.

Specifically in this thesis, we will study three DDP models: *aero* , *holo* , and *cob*.

Full details of these models cannot be disclosed (since they are proprietary) but some brief notes follow.

Aero is a "portfolio" level-model, where each of the mitigations is a research program rather than an activity. The overall *aero* program was a complex system involving hardware, software and operators. Further, the risks (challenges) described in *aero* span technical and organizational concerns.

Cob and *holo* are two non-portfolio models. Both are examples of JPL Technology Infusion Maturity Assessment (TIMA) studies. Such TIMA sessions involve over a dozen stakeholders whose total experience spans the domains of systems engineering, space experiments, avionics, materials, packaging, manufacturing, testing, experimental design, failure analysis, quality assurance, mission technologies, MEMS research, and program management. A typical TIMA session might call together a set of stakeholders to discuss whether or not the MEMS technology was suitably mature and appropriate for the intended applications, and to construct a cost-effective development and testing plan. Specifically, *holo* was a study to identify risks that would arise in maturing a particular piece of technology to flight readiness.

After a model is developed, the major design decision is what set of mitigations to apply. Recall that mitigations cost money and affect the total benefit of a project[1]. The

[1]Some mitigations decrease the expected benefit in some way, e. g. increased vibration testing might damage circuit boards

goal then is to balance cost and benefit. This will be our main performance metric. The *holo*, *aero*, and *cob* models have 99, 83, and 58 mitigations respectively. This leads to design spaces with a cardinalities of 10^{29}, 10^{24}, and 10^{17} respectively. Clearly then, the set of possible designs can not be exhaustively searched, or even enumerated.

Note that DDP can model interdependency between the objectives and mitigations. Recall from §2.3.4 that some earlier work on prioritizing requirements (which is just a different terminology for objectives) [73] used a simpler matrix based approach that could not capture interdependencies.

4.2 Investigating Different Extreme Sampling Policies

Since this is the first use of extreme sampling we wanted to investigate the effects of different parameter settings on solution quality. This study made use of the three DDP models discussed in §4.1. Recall from §3.1 that our discretization method uses two parameters; M, which controls the length of the iteration, and N, which controls how the instances are labeled $good$ and bad. Also recall from §3.1 that we formulated three different selectors. We investigated different values of these control parameters to see what effect they had on the performance of ITL. Previous informal work had suggested M could be in the hundreds, so we used M equal to 100, 300, and 500. For the value of N we used 25, 50, and 75, although the case where M = 100, N = 75 was omitted because for the *wob* selector it was unclear how to handle cases when $N > \frac{M}{2}$. We used all three selectors in this study to see if they effected the performance of ITL. This gives us 24 $((3 * 3 - 1) * 3)$ different parameter combinations. Each parameter combination was used to optimize each model, giving us 72 parameter-model combinations. Finally we ran ten trials for each of the 72 different parameter-model settings, to give a total of 720 data sets. Each trial in this study was run for 10 iterations and then stopped; we did not investigate formulating an automatic stopping condition. The total time needed to collect these data sets was about 5 weeks, running on two single processor desktop Windows © machines.

4.2.1 Comparison Methodology

When attempting to compare two techniques it can be difficult to define what metric should be used for comparison. In this section we use *delta comparisons*. A delta is the simple difference between any two trials that are different according to the effect being isolated, but have the same value for all other parameters. The metric used for this difference is the normalized euclidean distance (from §3.1.4). Using this metric allows deltas from different models to be compared, since our euclidean distance function normalizes each coordinate. Also, since ITL attempts to minimize this distance function, it would not be useful to then study ITL's performance according to another metric. When we present the deltas that highlight the effects of M, for example, three lists of deltas are made, one each for the three different settings of M studied. The deltas for $M = 100$ are calculated by comparing each trial where $M = 100$ with every other trial where $M \neq 100$, when the other three important parameters are the same (N, selector, model). This isolates the effect of M from the effect of the other parameters. All deltas for $M = 100$ are added to the same list so that we can see the effect of M across all the other parameter settings. The lists of deltas are then sorted, so that we can identify the quartiles. Positive deltas indicate that the setting being isolated outperforms the other possible settings, whereas a negative delta indicates that the setting is outperformed by the other possible settings. In this section we present only the median results for space reasons. The 1$^{\text{st}}$, 2$^{\text{nd}}$, and 3$^{\text{rd}}$quartile plots can be found in appendix A.1.

We prefer this method to admittedly simpler statistical methods, because this method is *non-parametric*, i. e. it makes no assumptions about the form of the underlying data. For example, the commonly used t-tests are a parametric method that assume that the underling population distribution is Gaussian. Recent results suggest that there are many statistical issues left to explore regarding how to best to apply those t-tests for summarizing cross-validation studies [20].

4.2.2 Experimental Effects of M

In this section we present the results of our trials with the effects of M isolated. First, we will discuss the comparative performance of the three M values studied by plotting their performance after certain numbers of iterations had been completed by the search, i. e. after certain search depths had been reached by ITL. This comparison is shown in Figure 4.1. Second, we will discuss the comparative performance of the three M values studied by plotting their performance after certain numbers of points had been generated by the dynamic model, i. e. after certain numbers of evaluations of the objective function. This comparison is shown in Figure 4.2.

Figure 4.1 shows the median deltas after the trials had completed 3, 5, 8, and 10 iterations. It is clear that the trials with larger M values perform better than those with smaller M values. $M = 500$ outperforms $M = 300$, which outperforms $M = 100$, and the magnitude of this difference increases through all ten iterations. Examining Figure A.1 demonstrates that the difference between the 1st and 3rd quartiles is even larger for small numbers of iterations completed, but the differences between the M values decreases after 5 to 8 iterations. This suggests that searches with small M values are not getting stuck in local optima, but require more iterations to construct high quality-partial descriptions.

Of course it should not be surprising that a search with a much larger batch size

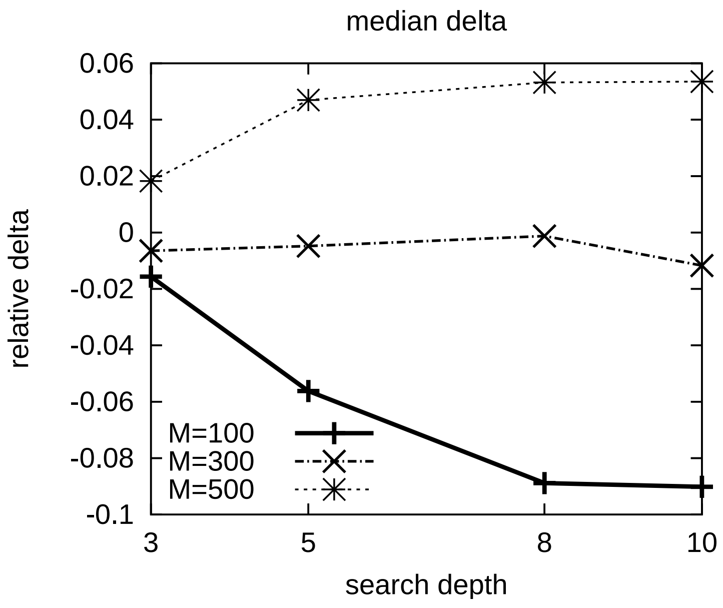


Figure 4.1: Effects of batch size, M, versus search depth

finds higher quality solutions. But given a fixed number of objective function evaluations, does ITL perform better by learning over many small batches or a few large batches? Figure 4.2 shows the median deltas after 300, 600, 1000, 1200, 1600, 1900, 2400, 2900, and 3500 evaluations of the objective function. These values correspond to iteration (learning) boundaries for M values of 300 and 500. Notice that the $M = 100$ and $M = 300$ lines do not extend to the right side of the graph. Recall that each trial was run for 10 iterations, so trials with $M = 100$ only evaluated the objective function 1000 times. Hence the $M = 100$ line stops at 1000 objective function evaluations and the $M = 300$ line stops at 3000 objective function evaluations. It is clear from this figure that for a fixed number of objective function evaluations, ITL performs much

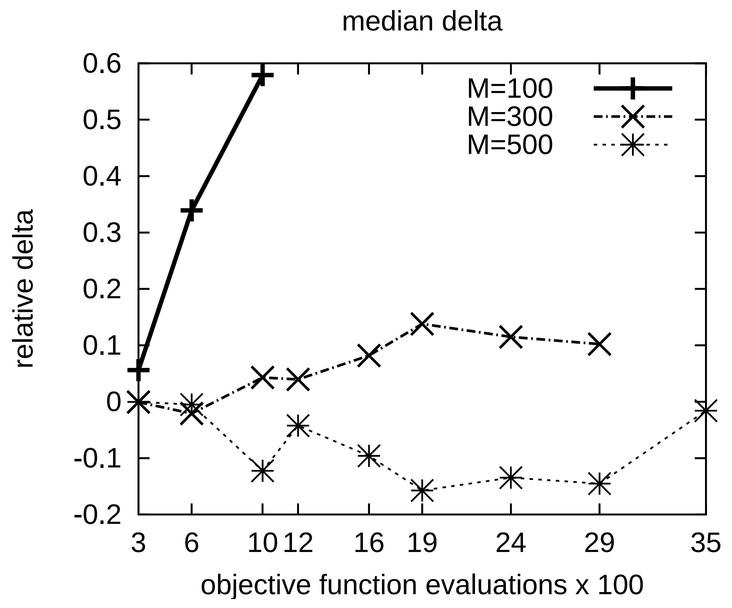


Figure 4.2: Effects of batch size, M, versus objective function evaluations

better by learning over a larger number of smaller batches. The $M = 100$ trials significantly outperform the two other M settings, and the $M = 300$ trials outperform $M = 500$, although to a lesser degree.

4.2.3 Experimental Effects of N

In this section we present the results of our trials with the effects of N isolated, shown in Figure 4.3. The graph showing performance versus search depth is the only graph shown because the value of N does not effect the number of times the objective function is evaluated. For this parameter the effect is not so pronounced. While $N = 25$ is

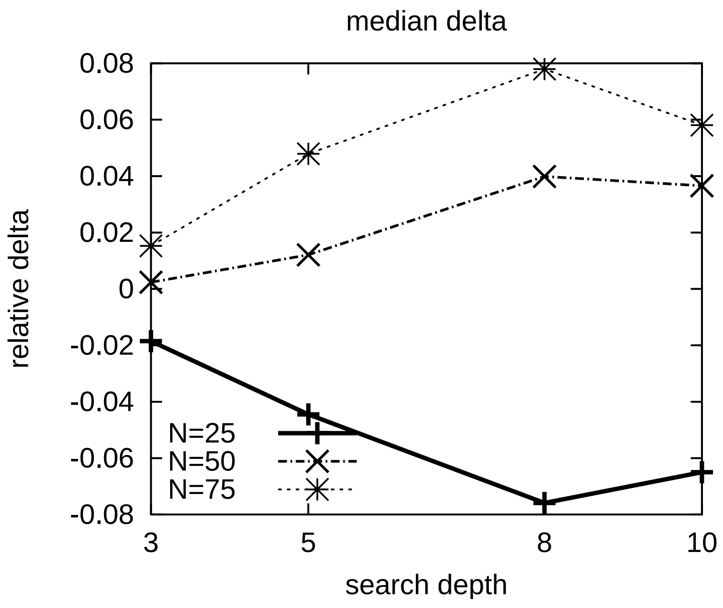


Figure 4.3: Effects of good/bad ratio, N

clearly an underperformer, the magnitude difference between $N = 50$ and $N = 75$ is not so large, with $N = 75$ being slightly higher performing. This difference in performance shows that it is important for there to be enough instances labeled $good$ for the treatment learner to find useful contrasts between the $good$ and bad instances.

4.2.4 Experimental Effects of M/N Combinations

In this section we present the results of our trials with respect to M/N combinations. As in §4.2.2, we first show the deltas after certain numbers of iterations (Figure 4.4), and after a certain number of objective function evaluations, (Figure 4.5). Note that

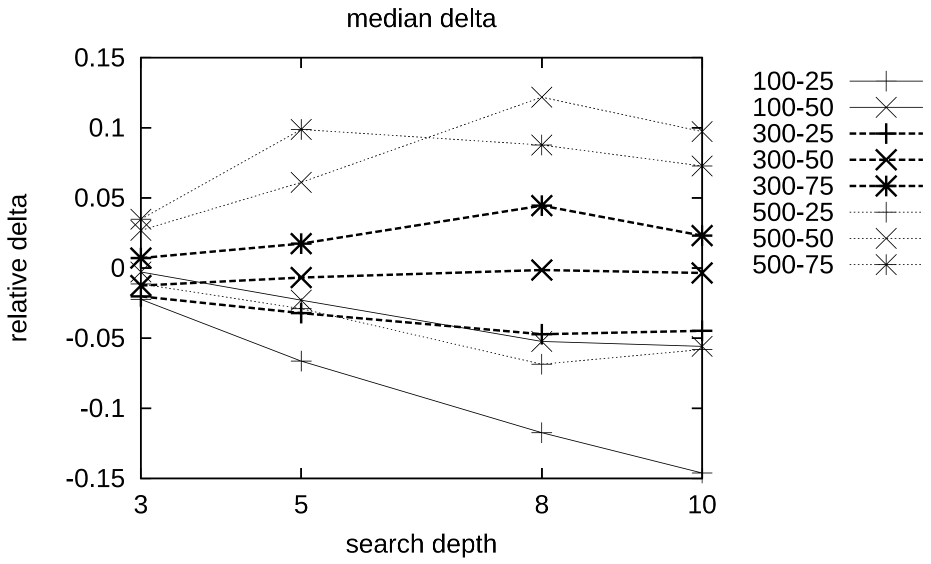


Figure 4.4: Effects of M/N combinations, versus search depth

lines with the same M value have the same line style (light dashed, dark dashed, light solid), while lines with the same N value have the marker (asterisk, X, or cross).

The most interesting effect in Figure 4.4 is that while 500/75 and 500/50 (light dashed with an asterisk or X) clearly outperform the other combinations, 300/75 and 300/50 (dark dashed with an asterisk or X) out perform 500/25 (light dashed with a cross). In fact, 500/25, 300/25, and 100/50 all perform at about the same level, with 100/25 significantly underperforming compared to all other M/N values. This shows that having a large batch size is important to performance, but almost as important is to have a large enough fraction of the instances labeled *good*.

Figure 4.5 repeats the result from Figure 4.2; using smaller batch sizes for a fixed

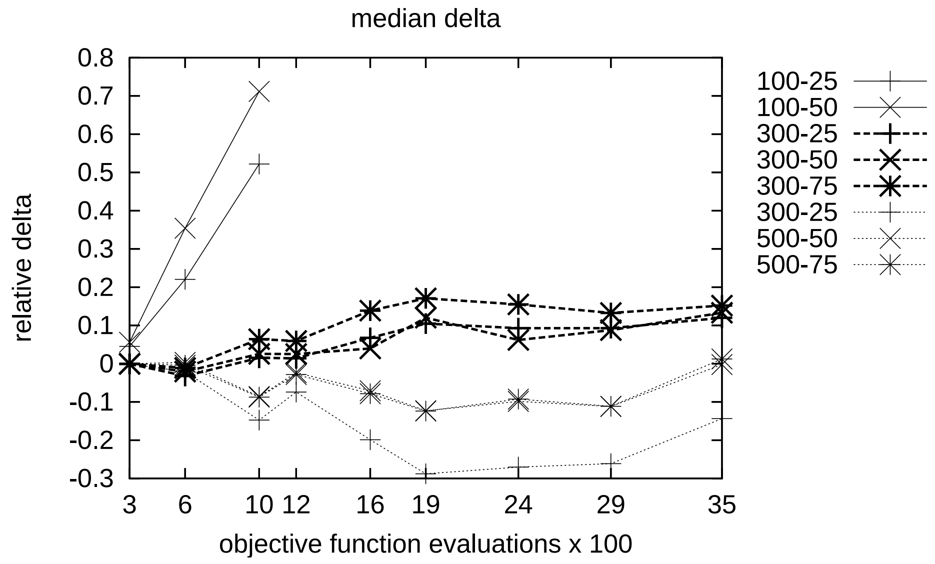


Figure 4.5: Effects of M/N combinations, versus objective function evaluations

number of objective functions evaluations leads to a higher quality solution. Figure 4.5 also shows that this effect is more important than the value of N, since the lines with the same M value (same color) cluster together.

4.2.5 Experimental Effects of Selector

In this section we present the results of our trials isolating the effects of the selector used, shown in Figure 4.6. Recall that *wob* was an attempt to maximize the contrast between the $good$ and bad instances, while both *wob* and *bore*′ pass only $2N$ instances to the learner rather than M instances (in our study it is always the case that $M > 2N$). Figure 4.6 clearly shows that *wob* is an underperformer in our study. The difference

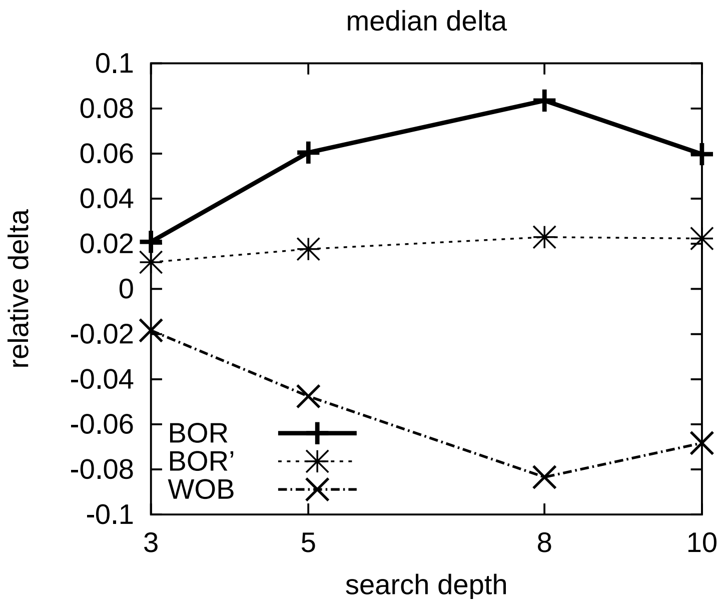


Figure 4.6: Effects of selector

between *bore* and *bore*$'$ is smaller, but *bore* performs slightly better than *bore*$'$. *wob*'s inferior performance can likely be attributed to the same cause as the inferior performance of the smaller values of N. There must be a certain amount of contrast between the $good$ and bad classes for ITL to build useful partial descriptions. Small values of N reduce the amount of useful contrast by reducing the number of $good$ of instances. Since the *wob* selector lowers the performance of ITL, we conjecture that there is useful information being hidden by only passing on the worst instances to the learning stage of ITL. In other words, we conjecture that the bad instances used by *wob* are more similar to each than the bad instances used by *bore*$'$. It is this similarity that reduces the ability of the treatment learner to find highly valuable treatments. Since the *bore*$'$ selector also performs worse than the *bore* selector, we can conjecture that the smaller number of instances utilized by the *wob* selector is also partially responsible for its degraded performance, but since the *wob* selector performs significantly worse than the *bore*$'$ selector, the primary reason is likely the exact nature of the instances utilized. The difference between the N instances with lowest distance score and the N instances with the highest distance score seems to be smaller than the difference between the best N instances and the N instances spread out among the $M - N$ instances picked by *bore*$'$.

4.3 Stability of Repeated Trials

The last section showed the performance of our different trials aggregated across all ten trials. But are we sure that ITL produces stable results? If, in a time sensitive setting, we run only one trial, can we be sure that the solution found by this trial won't be significantly outperformed by another run. In this section we present the results previously discussed by displaying the *search trajectories* for all ten trials. A search trajectory is the path the search takes through the output space. The outputs from DDP are cost and benefit, so the search trajectory can be plotted as 2-D line. The x-axis is the normalized mean cost and the y-axis is the normalized mean benefit. These means were calculated by averaging the costs and benefits of the instances that were generated by DDP each iteration. Hence each point represents one iteration. The lines are the successive iterations from a single trial.

Since we can not aggregate this data across models or selectors, as we did in the last section, this section will not present the results of all parameter-model combinations. Instead we will focus on the trials with M/N values of 500/75 and the *bore* selector, since these were the best combinations found in §4.2. When there is something interesting to comment on, other combinations or selectors will be discussed. See §A.2 for a complete listing of the results from this section.

The rest of this section will break down the results by model, first *aero*, then *holo* and finally *cob*.

4.3.1 *aero*

First we present some results from the *aero* model. We display the 500/75 and 500/25 trials using the *bore* selector in Figure 4.7. First note that both combinations shown find the same approximate benefit ceiling, slightly above .55 units. But the superior performance of 500/75 is evident. More trials find a lower cost, between .42 and .44 units, and if you look closely at the point (.45,.20) on the 500/25 graph there is an outlier which did not find the benefit ceiling (the trial represented by the plus sign).

Examining the results from all combinations with the *bore* selector, we see the

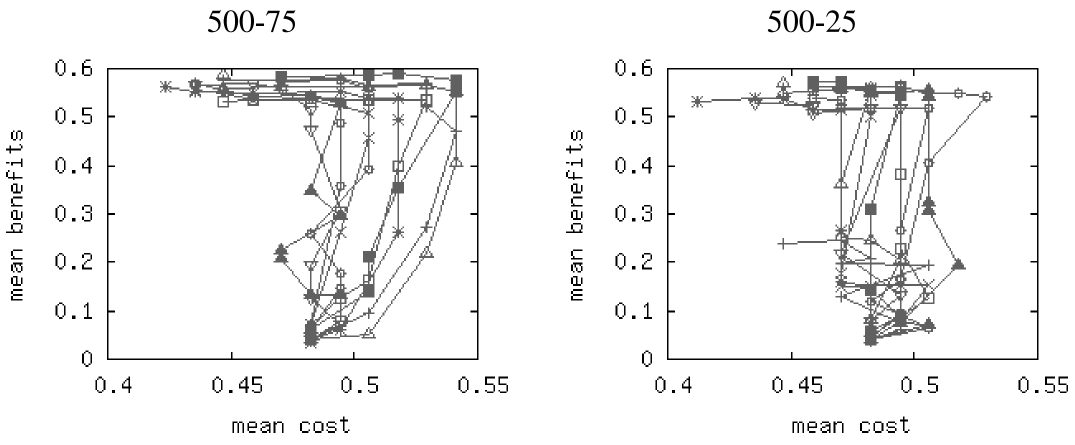


Figure 4.7: Stability of cost and benefit in the *aero* model with the *bore* selector

same pattern as the performance based comparison demonstrated in §4.2. Larger values of M show less variability run-to-run than do smaller values. But a large value of M coupled with a small value of N (like the 500/25 graph in Figure 4.7) does poorly when compared to medium values of M coupled with large values of N (specifically, the 300/75 trials show less variability than the 500/25 trials, see Figure A.7). We can also see a model specific pattern in these comparisons. The final average benefit of the partial descriptions is very similar for almost all M/N combinations; the performance difference in the trials comes almost entirely from their differing costs.

Figure 4.8 displays the same M/N settings as Figure 4.7, but from trials with the *wob* selector. Figure 4.8 clearly shows the inferior performance of the *wob* selector that we discussed in §4.2. These graphs also clearly shows that in addition to inferior performance, the *wob* selector also has a lower stability in solution quality. The end of

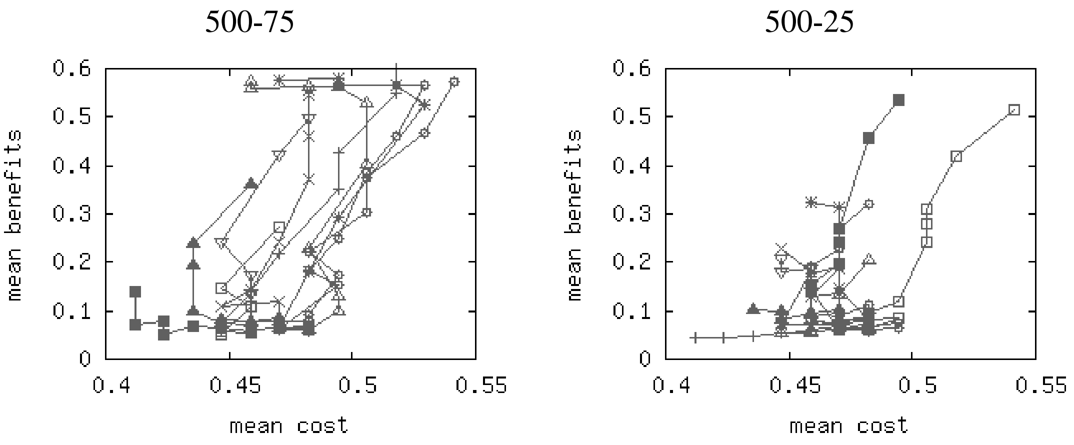


Figure 4.8: Stability of cost and benefit in the *aero* model with the *wob* selector

points of the trajectories have significant outliers for all parameter settings shown, even for the parameter combination that showed the most stability with the *bore* selector (500/75). The difference between 500/75 and 500/25 is also greater with the *wob* selector, suggesting that *wob* is more sensitive to changes in the M/N values.

4.3.2 *holo*

There was much less variation between the three different selectors in the *holo* model. Therefore, we only present the results of using the *bore* selector. Figure 4.9 shows the trials that used M/N combinations of 500/75 and 500/25 with the *bore* selector. The endpoints of the 500/75 trials are much closer together and this is a pattern repeated for all values of M in the *holo* model. Notice that trajectories follow different paths

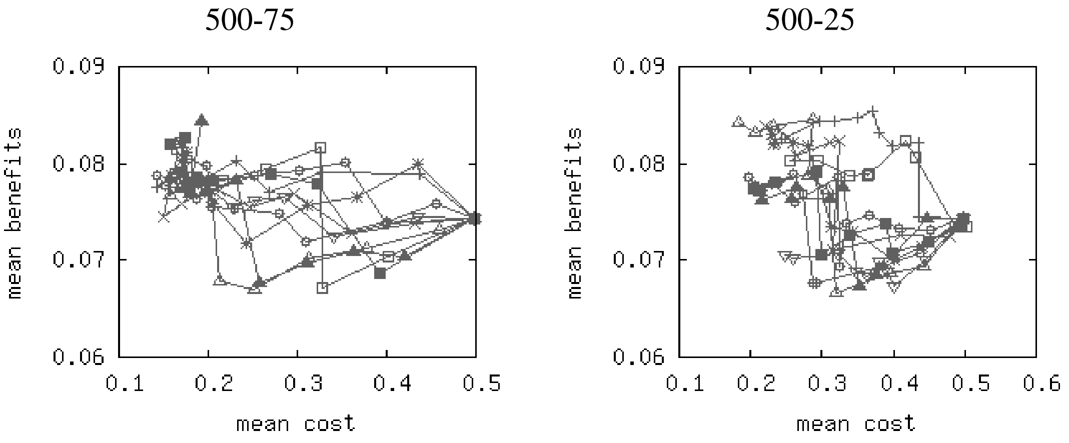


Figure 4.9: Stability of cost and benefit in the *holo* model with the *bore* selector

through the cost/benefit plane, unlike the trajectories in Figure 4.7 which all followed the same path. All trajectories lowered the average cost during each iteration, but some initially raised the average benefit, while some lowered the average benefit.

4.3.3 *cob*

Finally we present some results from the *cob* model. The *cob* model shows even less variability between the trials than the *holo* model, for different M/N combinations and for the different selectors. To illustrate the similar variance, Figure 4.10 shows the M/N values 500/75, 500/25, 100/50, and 100/25, all from the *bore* selector. There is little to no difference between the 500/75 and 500/25 trials and while the 100/50 trials cluster at a slightly higher cost, the trajectories do not show any more variability than the trajectories for the 500/75 combination. The 100/25 combination, which has been the worst performer in all comparisons shown as far, shows a wider spread of trajectories, but still clusters tightly after ten iterations. In particular notice how tightly clustered all the trials are for 500/75.

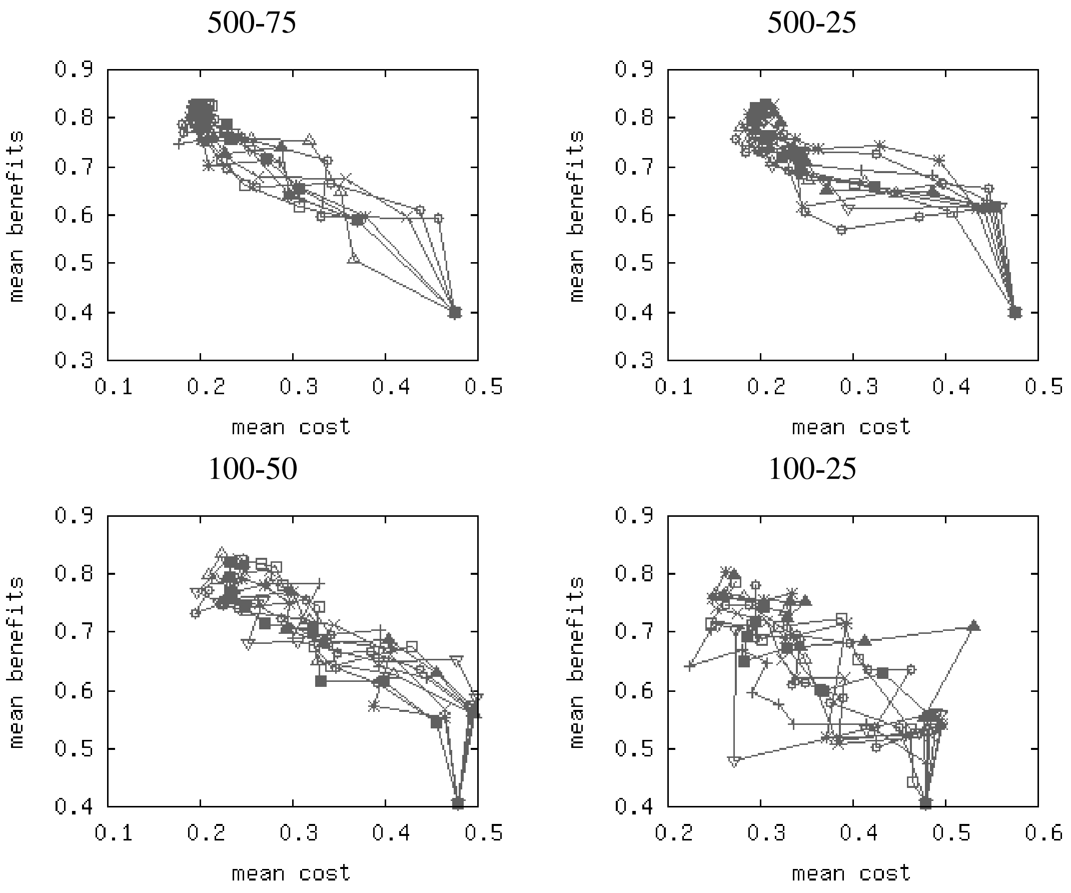


Figure 4.10: Stability of cost and benefit in the *cob* model with the *bore* selector

4.4 Variance of Partial Descriptions

In §4.3 we looked at the variability between each trial; in this section we will look at the variability within the trials. Recall from §3.1.2 that a partial description actually forms a set of solutions, comprising all solutions that pass the treatments in the partial description. These solutions will naturally be scored differently according to the objective function. A key question then is: what amount of variability can we expect from a partial description returned by ITL?

To address this question we present some results that show the estimated variance based on the standard deviation of the points generated during each iteration. The figures in this section show the average cost and benefit side by side. The averages were calculated in the same way as in §4.3, with a small amount of horizontal jitter added to separate the ten different trials. The standard deviation of the points in each iteration are shown as error bars. Also as in §4.3, we will focus on the results from the 500/75 combination and the *bore* selector. §A.3 has a complete set of results.

4.4.1 *aero*

Figure 4.11 displays the per-iteration average of the 500/75 combination for the *bore* and *bore*′ selectors. The *wob* selector (shown in §A.3), which has been shown to find

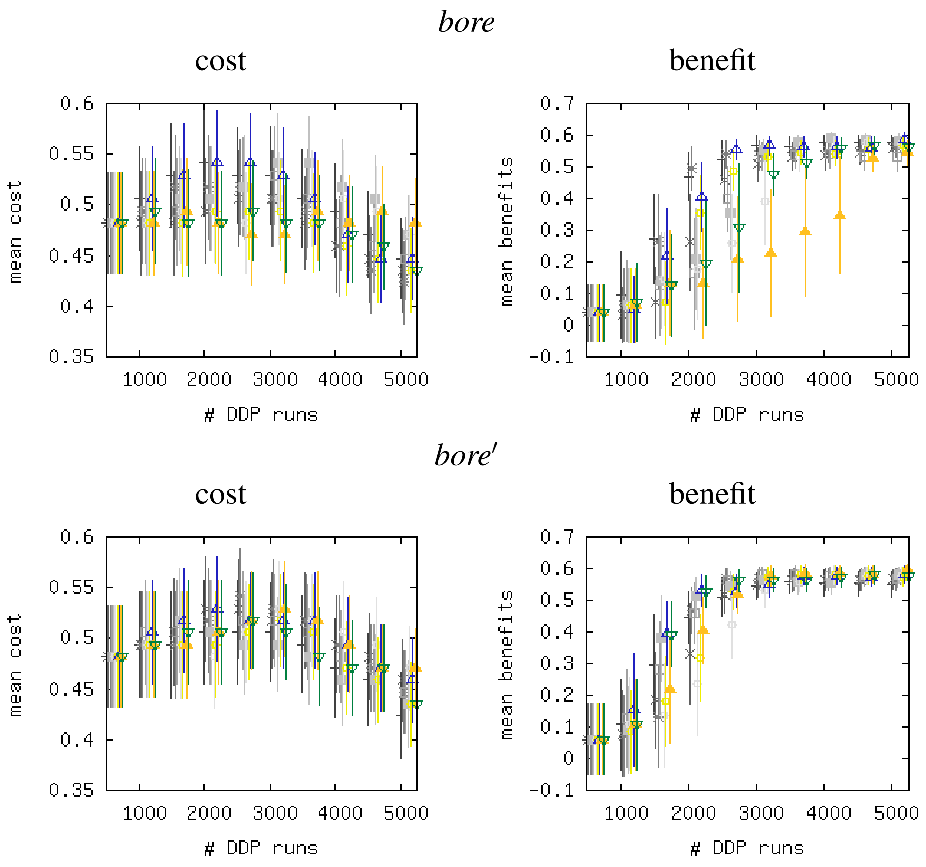


Figure 4.11: Cost and Benefit in the *aero* model with M/N values set to 500/75. Error bars represent the standard deviation of the points in each iteration.

lower quality solutions, also has a much higher variance due to the partial description. This is in line with the results from §4.2 and §4.3 which also showed similar poor performance (according to the criteria used in those sections) of the *wob* selector.

Looking at the *bore* and *bore′* results we see some of the patterns seen in §4.2 and §4.3 repeated and some contradicted. Using the *bore* selector it appears the least variable trials are from the combinations where N = 75. Both the M = 500 and M =

300 cases (the $M = 300$ cases can be found in §A.3) have only one outlier that takes 3-5 more iterations to converge to the average benefit seen in the other nine trials. But after these "late" trials do converge they have small standard deviations similar to the other trials. This is similar to the pattern we saw in §4.2 and §4.3 where N = 25 combinations are underperformers compared to N =75 combinations.

Something that we have not seen in previous sections is the apparently superior performance of the *bore*$'$ selector for M = 500 combinations. Figure 4.11 shows faster convergence in the average benefit, as well as lower variance. It is not clear why this is the case for this particular model, because this pattern is not seen in the other models (discussed below).

4.4.2 *holo*

For the *holo* model all three selectors show a similar pattern of decreasing variance, therefore we will only display the *bore* selector. Figure 4.12 shows the 500/75 and 500/25 combinations. This figure shows an extremely low variance in the cost, with a decreasing, but higher variance in the benefit, for the 500/75 combination. The 500/25 has a higher variance in both the average cost and average benefit. The $M = 300$ cases are similar and the $M = 100$ cases show a significantly higher variance, particularly in the average benefit.

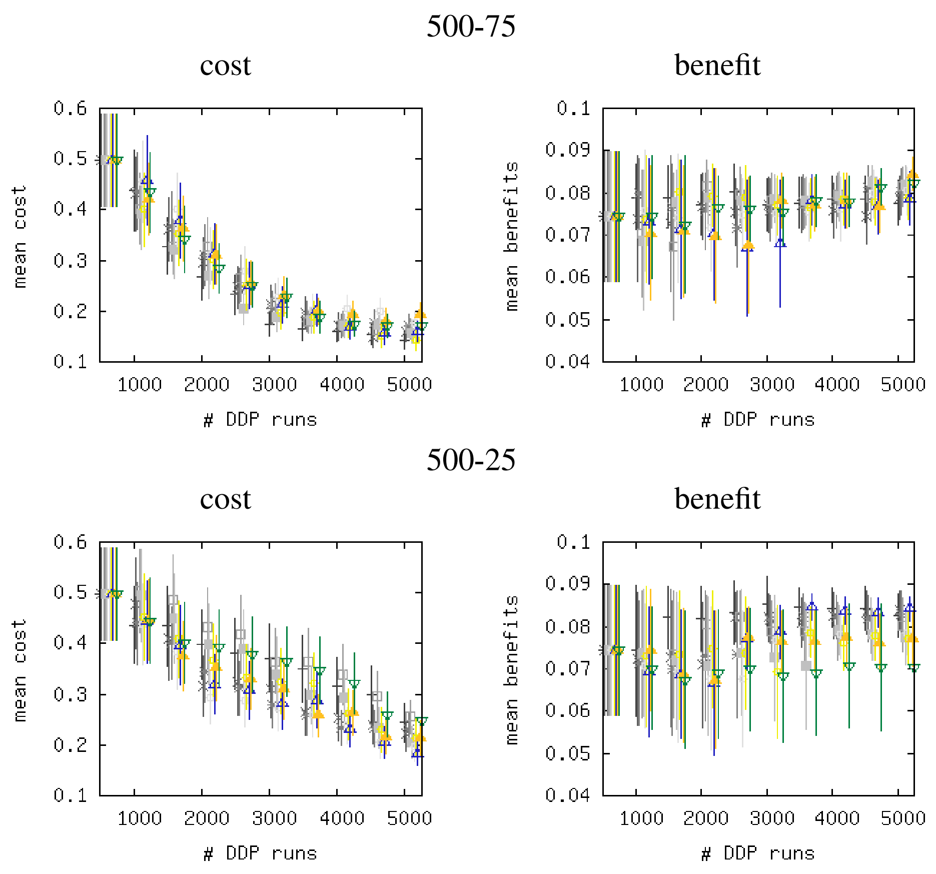


Figure 4.12: Cost and Benefit in the *holo* model with the *bore* selector. Error bars represent the standard deviation of the points in each iteration.

4.4.3 *cob*

The cob model seems to have a particularly low variance inherent in its structure. This is similar to the high stability we saw in the *cob* model in §4.3. For all selectors and all combinations the variance in both the average cost and benefit quickly diminishes and by the fourth to sixth iteration is quite low. The *bore* and *wob* selector are shown in Figure 4.13 to demonstrate how the usually underperforming *wob* selector has the

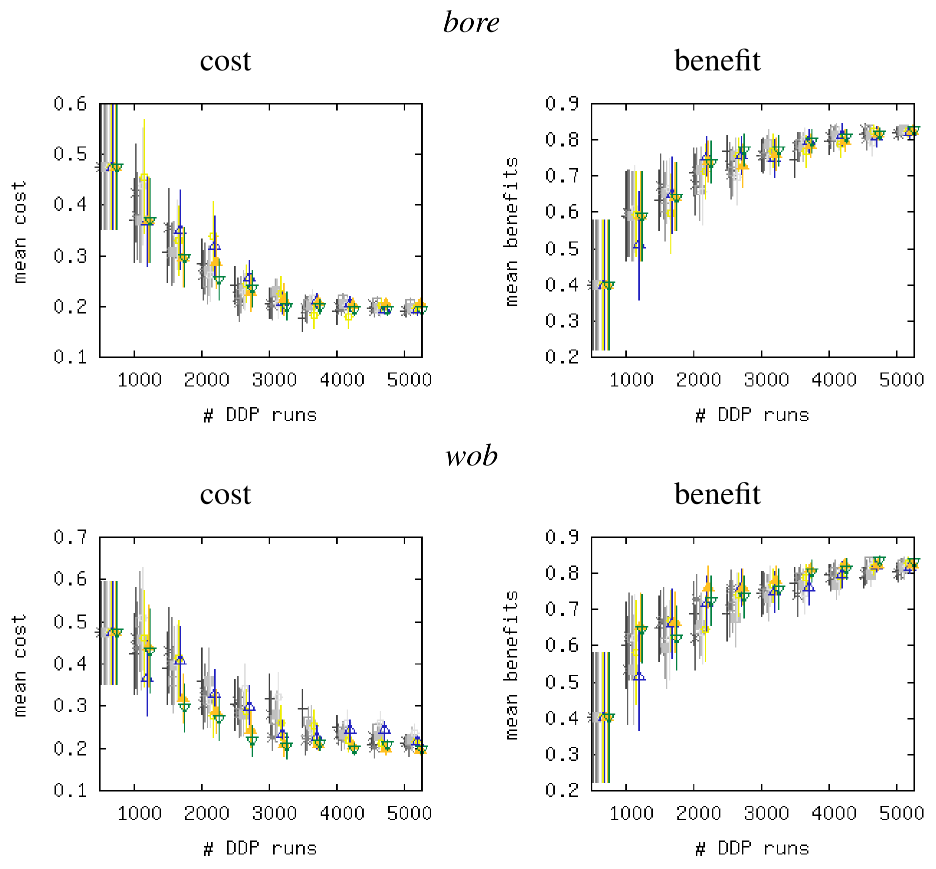


Figure 4.13: Cost and Benefit in the *cob* model with M/N values set to 500/75. Error bars represent the standard deviation of the points in each iteration.

same level of variance as the *bore* selector. Both selectors show a steadily reducing variance as the search proceeds, with the *cob* selector taking just 1-3 more iterations for the variance in the average cost to reach the level seen in the *bore* selector.

The familiar patterns, with regard to the M/N values, from §4.2 and §4.3 are seen in the *cob* results, although with reduced significance. That is, we see higher variance in lower values of M and in the lower values of N, but the 300/75 combination performs slightly better than the 500/25 combination.

4.5 Extreme Sampling Performance Compared to Diagonal Striping

Having investigated the performance characteristics of extreme sampling in §4.2 - §4.4, this section compares the performance of extreme sampling with the previous discretization method. As mentioned in §3.1, the original discretization method used with ITL [26,43,89] was a diagonal striping discretization. We will compare this older method with our best-performing extreme sampling method, i. e. the *bore* selector with M/N set to 500/75. Figure 4.14 compares diagonal discretization with the *bore* method as a reminder from §3.1.

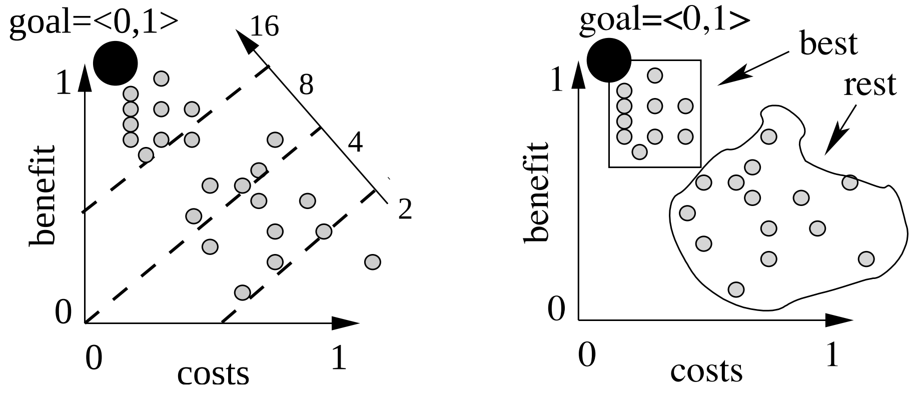


Figure 4.14: Diagonal striping and *bore*

Figure 4.15 shows the search trajectories for diagonal striping and *bore* discretization in the *aero* and *holo* models. For the *bore* trajectory this figure uses the best of

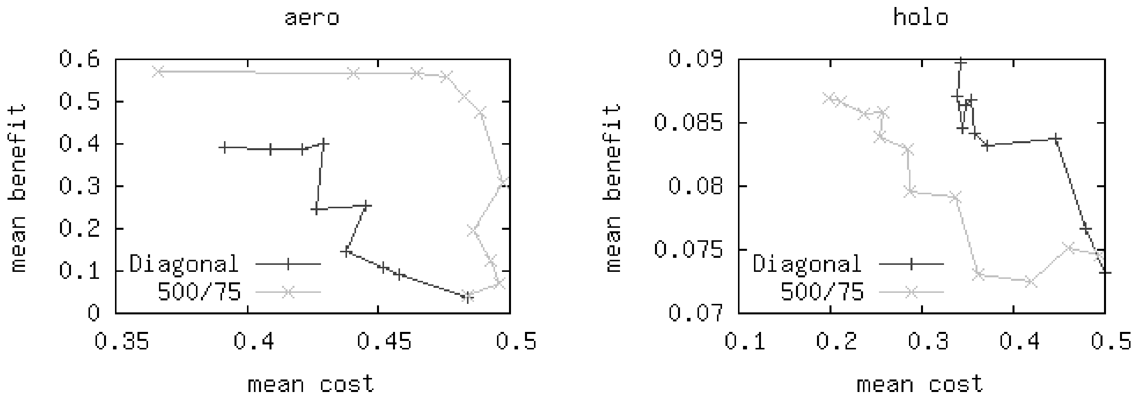


Figure 4.15: Comparison of *bore* to diagonal striping

the ten trials from each of the models, based on benefit/cost ratio. For the diagonal trajectory this figure uses the best of the 30 trials from each of the models, based on benefit/cost ratio. In addition, the original diagonal experiments required batch sizes (i. e. the value of M) of 2000. Even with this larger batch size and more trials to choose from, Figure 4.15 makes it clear that *bore* outperforms diagonal striping significantly in both models shown.

For the *aero* model, *bore* found a solution with a much higher expected benefit and a slightly lower cost. In the *holo* model, *bore* found a solution with a much lower cost and only a slightly lower expected benefit. Table 4.1 gives the exact values of the endpoints for the cost, benefit, and normalized benefit/cost ratio.

	bore			diagonal		
model	cost	benefit	normalized ratio	cost	benefit	normalized ratio
aero	.366	.569	1.55	.392	.392	1.00
holo	.198	.0869	.439	.343	.0898	.262

Table 4.1: Normalized benefit/cost ratios in *bore* and diagonal striping

4.6 Extreme Sampling Performance Compared to Simulated Annealing

So far in this chapter we have investigated the performance characteristics of extreme sampling using different parameters and compared extreme sampling to our previous discretization method, diagonal striping. In this section we will compare ITL to the previously used search strategy, simulated annealing (see §2.2.2 for a description of simulated annealing).

Original Search Technique Used by DDP As part of the DDP environment a simulated annealer can be used to maximize the benefit/cost ratio. In DDP's simulated annealer, the current best solution is mutated as follows. The candidate solution is a boolean vector representing whether to apply each mitigation. Each boolean has a 10% chance of flipping to the other boolean value. The user specifies the number of objective function evaluations desired so DDP can automatically set the cooling schedule.

Comparison of ITL to Simulated Annealing Figure 4.16 shows the trajectories of DDP's simulated annealer using 30,000 objective function evaluations, and the trajectories of the best trial using the *bore* selector with M/N values of 500/75. (Hence 5000 objective function evaluations were done during the *bore* search.) Table 4.2 lists the normalized endpoints of the six different searches in Figure 4.16. (Remember that the normalized benefit/cost ratio is our objective function.) We can see that extreme sampling outperforms SA in the *cob* model, finds a solution with almost the same value for the objective function for the *aero* model, and underperforms on the *holo* model.

Notice also that ITL seems to systematically find solutions with higher benefit and cost (with the exception of the cost of the solution found for the *cob* model). In personal communication with an experienced DDP user it was suggested that this was due to the nature of a typical DDP model. A typical DDP model has some high-cost mitigations with large benefits and many more low-cost mitigations that have small benefits. The simulated annealer can find a subset of low-cost mitigations that together have a high benefit. But this subset will likely be much larger than a few treatments (as previously discussed, ITL uses treatments with a maximum size of five attribute-value pairs). Since ITL only searches one treatment deep per iteration, it cannot find this type of subset. Instead the treatment learner finds the few mitigations that most greatly effect the normalized ratio. But obviously this search bias does not stop ITL

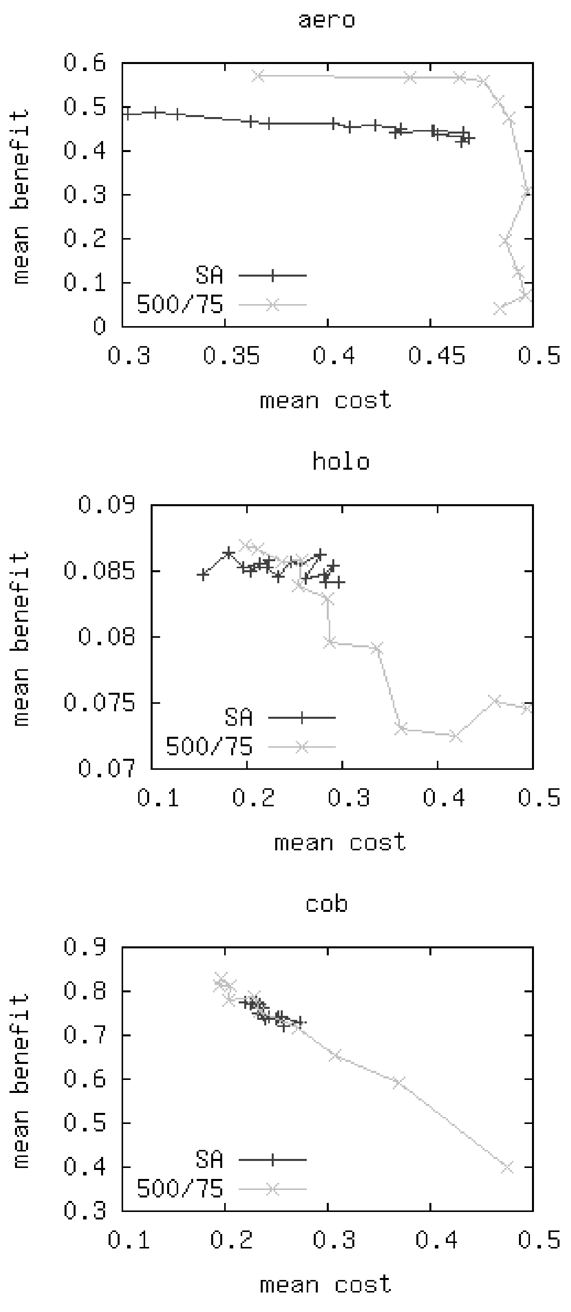


Figure 4.16: Comparison of *bore* to SA

	bore			SA		
model	cost	benefit	normalized ratio	cost	benefit	normalized ratio
aero	.366	.569	1.55	.302	.482	1.60
holo	.198	.0869	.439	.153	.0845	.552
cob	.197	.829	4.21	.220	.776	3.53

Table 4.2: Normalized benefit/cost ratios in *bore* and SA

from finding high quality solutions, as discussed in the previous paragraph.

4.7 Conclusions from the DDP Studies

This chapter has used requirements engineering to investigate several important characteristics of ITL. First we investigated different possible policy settings to extreme sampling, a key part of ITL. Then we compared ITL's performance with extreme sampling to its performance with diagonal striping. Lastly, we compared the performance of ITL with extreme sampling to simulated annealing, a well studied and frequently used metaheuristic search technique.

There are several conclusions we can draw from all these experiments. Extreme sampling works best with the *bore* selector. While we hoped that *bore*′ or *wob* could be used to reduce the amount time needed by the learner, by reducing the number of instances trained on, both *bore*′ and *wob* caused a degradation in the performance of ITL when compared to the *bore* selector. Our intuition that the *wob* selector might perform better by highlighting the difference between the instances with a high objective score and those with a low objective score also turned out to be incorrect.

Learning on large batch sizes positively effects solution quality when the search is iteration limited, but if the search is limited by the number of times the objective function can be evaluated, smaller batches positively effects solution quality. The experiments in this chapter did not use a wide enough range of M values to hint at the value of M at which ITL's performance stops increasing with increasing values of M.

We found that reducing the number of instances labeled good negatively affects solution quality. Extreme sampling needs a certain percentage of instances in each iteration to be labeled good. The ratio of N to M appears to be at least 15% to 25% (75/500 to 75/300). The experiments in this chapter did not use a wide enough range of N values to hint at the upper limit to this ratio.

Sections 4.3 and 4.4 demonstrated that ITL with extreme sampling has both stable performance (restarting a search does not significantly change the performance of the method) and that the variance inherent in partial descriptions is low after several iterations.

In §4.5 we demonstrated the clear superiority of the *bore* version of extreme sampling to our previous discretization method, diagonal striping. Averaged over the two models, *bore* found a solution with an objective score 161% of the average found by diagonal striping.

Finally §4.6 showed that extreme sampling can find higher, equal, or lower quality solutions than simulated annealing, but in many fewer objective function evaluations. Averaged over the three models, ITL found a solution with an objective score 98.6% of the average found by the simulated annealer.

The next set of case studies have to do with the SPY framework. Like ITL, SPY finds range restrictions to model input variables by using a treatment learner. However, SPY takes a different approach to integrating the search algorithm with the model. The models used by SPY are written in the SPY language, which was specifically designed to make the model-learner interface as smooth as possible. These studies highlight a problem we discussed earlier in §2.3.1; using a nearly continuous objective function is critical for a metaheuristic search to be successful. Chapter 5 will use SPY to validate temporal properties in NASA flight models and chapter 6 will investigate restricting the behavior modes of biomathematical models.

CHAPTER 5

MODEL PROPERTY CHECKING

In the last chapter we saw how ITL with a new discretizer was able to perform at the same level as a well respected metaheuristic technique. An important feature of that work was the continuous nature of the objective function used by ITL (the normalized benefit/cost ratio). In this chapter and the next we will present results from experiments using the SPY framework (described in §5.1), which will demonstrate how important this feature is.

The first case study with SPY uses NASA flight models. We attempt to verify temporal model properties in production flight models used by NASA contractors. Property validation was the original motivation for developing the SPY framework. Although there have been several breakthroughs in static verification and validation (V & V) techniques such as model checking, the usefulness in verifying properties of software systems has been limited because important classes of software systems involve large input domains (e. g. unbounded integer variables and real valued variables) as well as interrelated numeric constraints over the variables in the input domain.

These characteristics severely limit the usefulness of verification techniques like model checking. There are several modifications to model checking that can be used to allow model checkers to work with models that have unbounded numeric inputs. Bounded model checking [15, 30] can be used to check a model with discrete inputs, by exhaustively checking the model using a narrow range of values for the model inputs. Models can also be abstracted to remove or isolate the effect of numeric inputs. The drawback of this technique is that the model checker correctness and completeness depend on the abstraction technique. It may not be possible to develop an abstraction that preserves the essential semantics of the model being checked. The SPY framework was designed to give analysts a tool that didn't suffer that limitation. The price an analyst has to pay to get around these limitations is the incomplete nature of SPY's search. Many model checkers have a completeness guarantee, so if the checker reports no violations, the model is guaranteed to never violate the properties checked. With SPY if a property violation is not found, the random nature of its search means that a violation may still occur, if different inputs are feed to the model.

Throughout §5.2 we will mention a commercial tool, Reactis, that was used by our collaborators at the University of Minnesota (UMN) as a baseline in a comparison with SPY. Reactis was used because because it is a common commercial tool that was available in-house to our collaborators at UMN. The advantage of Reactis is that it performs a random and heuristic search through a model, without any restriction on the type of the model inputs. This means that this technique can be used to check models with real valued inputs without any preprocessing of the model, just like SPY. The random nature of Reactis's search means that it does not have a completeness guarantee, as discussed in the previous paragraph.

5.1 SPY

The essential question the SPY framework tries to answer is: what input range restrictions are most likely to constrain the state of the model to states that are considered more desirable? SPY does not use formal methods to investigate the models under question. Instead the SPY framework includes an execution engine for driving the models under examination. SPY executes the model a prescribed number of times generating a set of input-output pairs. The desirability of these pairs is evaluated by an objective function. SPY then finds correlations between input-output pairs and their desirability by using tar4 (see §2.1.5 for a description of tar4). This cycle of executing the model, objective function evaluation, and data mining is called an iteration. The SPY framework runs for several iterations, with the learner finding treatments in between each iteration. Notice this is the same work flow as ITL's work flow, discussed in chapter 3, in particular in Figure 3.9.

During each iteration the SPY framework randomly chooses inputs for the models according to range restrictions on the input variables. These restrictions are initially described by the analyst, but after the first iteration are modified by SPY's learning process. The restrictions take the form of upper and lower bounds for each input value. SPY reduces the space it searches by increasing the lower bound and/or decreasing the upper bound according to the treatments returned at the end of each iteration. Hence the points that are randomly sampled during one iteration are always from a space smaller than the space sampled from during the previous iteration. However, rather than just using the treatments themselves as the upper and lower bounds (as the ITL method used in chapter 4 did), the bounds are adjusted in the *direction* suggested by the treatments, but not necessarily in the *amount* suggested by the treatments.

The selection of input points from the search space is completely random; SPY attempts no symbolic analysis of the model's source code. This is different from other random tools, such as Reactis, which use heuristic analysis of the model's source code (such as path coverage) when picking input points. Refer to [29] for a complete description of the SPY framework.

In addition to introducing SPY, [29] also applies the framework to a "magic" bus that tries to transport passengers the farthest it can before running out of fuel. While this artificial model was useful in demonstrating the capability of SPY to learn model input constraints, the next section returns to the original motivation for developing SPY: verifying temporal properties in models with real valued inputs. To do so we will investigate three NASA flight models and ten properties the the original model developers describe in the requirements documentation available to us.

5.2 Using SPY on NASA Flight Models

The original purpose for developing the SPY framework was to give NASA's IV&V (independent verification and validation) facility a new method to check model properties. Since, as discussed in §5.1, SPY allows any data type in the models being analyzed, it can be used to verify properties in models that would defeat techniques with completeness guarantees, like model checking.

Recall from §5.1 that SPY does not perform any symbolic analysis of the model's source code. Some other tools, like our baseline tool Reactis, use heuristic analysis of the model's source code, such as trying to achieve full path coverage, when picking input points. SPY's selection of input points is completely random, but it uses partial descriptions to restrict the region of the search space that these points are chosen from.

This section describes the models analyzed (§5.2.1), the experimental goals of this analysis (§5.2.2), and the results of these experiments (§5.2.3).

5.2.1 Models Under Consideration

The models used in this section were originally developed in Simulink[1]. The models were translated to the SPY language by a LUSTRE-based translator [54] developed as part of the SPY framework development, at the University of Minnesota.

To gauge the effectiveness of SPY in verifying properties and uncovering defects, we analyzed three different models

- Sensor Voting
- Dual FGS
- Altitude Switch

This section will describe some of the particulars of the models.

Sensor Voting

The Sensor Voting model[2] is a generic triplex voter. The voter takes inputs from three redundant sensors and synthesizes a single reliable sensor output. Each of the redundant sensors produces both a measured data value and self-check bit (validity flag) indicating whether or not the sensor considers itself to be operational. The output of a sensor is amplitude limited in hardware by the A/D conversion.

The functionality of the triplex voter is as follows:

[1]Available from The MathWorks Inc, at `www.mathworks.com`.
[2]Developed at Honeywell Laboratories.

- Sample digitized signals of each sensor measurement at a fixed rate appropriate for the control loop, e. g. 20 Hz. A valid flag supplied by sensor hardware indicating its status is also sampled at the same rate.
- Use the valid flag and comparison of redundant sensor measurements to detect and isolate failed sensors.
- Output at a specified sample rate a signal value computed as a composite average of the signals of non-faulty sensors. Also output, at the same specified rate, the status of the composite output by setting an "outputValid" flag.
- Tolerate "false alarms" due to noise, transients, and small differences in sensor measurements. Sensors are not marked failed if they are operating within acceptable tolerances and noise levels.
- Maximize the availability of valid output by providing an output whenever possible, even with two failed sensors.
- The algorithm is not required to deal with simultaneous sensor failures since this is a very low probability event.

The operation of the sensor voter algorithm is as follows. All valid sensor signals are combined to produce the voter output. If three sensors are available, a weighted average is used in which the outlying sensor value is given less weight than those that are in closer agreement. If only two sensors are available a simple average is used. If only one sensor is available, it becomes the output. There are two mechanisms whereby a faulty sensor may be detected and no longer considered valid; either by comparison of the redundant sensor signals or by the validity flags produced by the sensors themselves.

Dual FGS

A Flight Guidance System is a component of the overall Flight Control System (FCS) in a commercial aircraft. The FGS compares the measured state of the aircraft (position, speed, and attitude) to the desired state and generates pitch and roll guidance commands to minimize the difference between the measured and desired state. The FGS subsystem accepts input about the aircraft's state from the Air Data System (ADS) and Flight Management System (FMS). Using this information, it computes pitch and roll guidance commands that are provided to the autopilot (AP). When engaged, the autopilot translates these commands into movement of the aircraft's control surfaces necessary to achieve the desired changes about the lateral and vertical axes.

The flight crew interacts with the FGS primarily through the Flight Control Panel (FCP). The FGS has two physical sides corresponding to the left and right sides of the aircraft. These provide redundant implementations that communicate with each other over a cross-channel bus. Normally, only one FGS (the pilot flying side) is active, with the other FGS operating as a silent, hot spare. In this dependent mode of operation, the active FGS provides guidance values to the AP and the Flight Director (FD). The pilot and copilot can switch which side is the pilot flying side by pressing the Transfer Switch on the FCP. This is frequently done when switching to a different navigation source. However, in some critical modes, such as Approach and Go Around, both sides are active and independently generate guidance values for their own FD. In this independent mode of operation, both sets of guidance values are provided to the AP, which first verifies that they agree within a predefined tolerance value. If in agreement, the values are averaged and executed. If not in agreement, the situation is annunciated to the pilot and the AP disconnects.

Altitude Switch

The Altitude Switch (ASW) is a re-usable component that turns power on to a Device Of Interest (DOI) when the aircraft descends below a threshold altitude above ground level. If the altitude cannot be determined for more than two seconds, the ASW indicates a fault. The detection of a fault turns on an indicator lamp within the cockpit. The DOI is turned back off again if the aircraft ascends above the threshold altitude plus some hysteresis value. The ASW receives a status indication from the DOI indicating whether the DOI is powered on. If the DOI does not indicate that it is powered on within two seconds after power is applied, a fault is indicated. The ASW does not apply power to the DOI if the DOI is already powered on. The ASW is not in complete control of the DOI, since the DOI may be turned on and off by other systems or the pilot. If the DOI is turned off after the aircraft descends below the threshold altitude, the ASW does not reapply power to the DOI unless the aircraft again descends below the threshold altitude. The ASW also accepts an inhibit signal that prevents it from turning on power to the DOI or indicating a fault. All other ASW functions are unaffected by the inhibit signal. The ASW also accepts a reset signal that returns it to its initial state.

5.2.2 Experimental Goals

This section will describe the properties we want to verify in each of the models described in §5.2.1.

Sensor Voting

The Sensor Voting model combines the output from three sensors to produce a high quality output. It operates in such a fashion as to tolerate transient errors or multiple failures in the sensors, to detect significant differences in the signals, and to isolate that difference (if possible) to a single faulty sensor. The full details of this model are given in §5.2.1.

For the Sensor Voting model we checked the following properties

1. If sensor valid flag goes bad 3 time steps in a row, then the sensor output shall be flagged as bad. This sensor will stayed flagged as bad even if the sensor valid flag becomes good at a later time.

2. If a persistent threshold violation is detected by the model, the sensor output shall be flagged as bad. This sensor will stayed flagged as bad even if the persistent threshold violation disappears at a later time.

Both these properties were applied separately for each of the three sensors.

The first property was implemented in SPY using a trio of simple accumulators that tracked the number of consecutive time steps that a sensor valid flag was bad. If any of the accumulators exceeded the 3 time steps allowed, that sensor was recorded as bad by SPY, and if the model ever reported that sensor as good a property violation was reported.

The second property, which measures disagreement between the values reported by the different sensors, was a bit more complicated to implement. Two sensors had to disagree by an amount above a certain threshold (Sensor Magnitude Threshold) for a time exceeding another threshold (Sensor Persistence Threshold). In addition, since with only two valid sensors it would be impossible to determine which sensor was faulty, all three sensors had to be considered valid by the model for it to isolate which sensor was producing the disagreeing value. If the objective function detected a persistent threshold violation that was not reported by the model, a property violation was reported by SPY.

Together these two properties were also checked in the other direction. That is, if the objective function decided a sensor was valid according to both criteria, but the model reported the sensor as invalid, a property violation was reported by SPY.

Dual FGS

The FGS compares the measured state of the aircraft (position, speed, and attitude) to the desired state and generates pitch and roll guidance commands to minimize the difference between the measured and desired state. §5.2.1 gives a more detailed description of the Dual FGS.

For the Dual FGS model, we checked the following properties

1. At least one FGS side shall always be active
2. Exactly one side shall be the pilot flying side. This property was stated in two forms
   (a) It is always the case that property 2 holds
   (b) It is always the case that when property 2 is false then in the next step property 2 is true
3. If the system is in independent mode (defined in §5.2.1), both sides shall be active. This property was stated in two forms:
   (a) It is always the case that property 3 holds
   (b) It is always the case that when property 3 is false then in the next step property 3 is true
4. If the system is in dependent mode, it shall not be the case that both sides are active. This property was stated in two forms:
   (a) It is always the case that property 4 holds
   (b) Property 4 cannot be false for more than two consecutive time steps
5. Pressing the transfer switch shall cause the system to change PF sides. This property is expressed in two parts:

(a) It is always the case that if the Left FGS is not the pilot flying side and the transfer switch is not pressed, then in the next step if the transfer switch is pressed, Left FGS shall become the pilot flying side.

(b) It is always the case that if the Right FGS is not the pilot flying side and the transfer switch is not pressed, then in the next step if the transfer switch is pressed, Right FGS shall become the pilot flying side.

Properties 2b, 3b, and 4b capture the properties the developers intended the models to have. We checked properties 2a, 3a, and 4a to ensure that SPY checked for some properties that did not hold in the model.

The first property was easy to implement by checking the two boolean variables used to record the status of each side of the FGS. One side had to be active at every time step.

The second property checked the two boolean variables that recorded which side of the FGS was giving input to the AP, i. e. the "flying" side. An accumulator was used to count the number of time steps that both sides of the FGS were flying. When checking property 2a, if this accumulator was ever greater than 0, a violation was reported. When checking property 2b, if this accumulator was ever greater than 1, a violation was reported.

The third property checked that if the system was in independent mode (defined in §5.2.1) both boolean variables that recorded if a FGS side is active were set to true. An accumulator was used to count the number of time steps this was false. When checking property 3a, if this accumulator was ever greater than 0, a violation was reported. When checking property 3b, if this accumulator was ever greater than 1, a violation was reported.

The fourth property was implemented in a way similar to property 3. The accumulator was incremented if the FGS was in dependent mode and both FGS sides were active. When checking property 4a, if this accumulator was ever greater than 0, a violation was reported. When checking property 4b, if this accumulator was ever greater than 2, a violation was reported.

The final property was checked for both sides of the FGS. If the side being checked was the flying side and the transfer switch wasn't pressed during the current time step, but was pressed during the previous step and the side was the flying side in the last step, a violation was reported.

Altitude Switch

The Altitude Switch(ASW) is a re-usable component that turns power on to a Device of Interest (DOI) when the aircraft descends below a threshold altitude above ground level. Refer to §5.2.1 for a more detailed description of the Altitude Switch.

For the Altitude Switch, we checked the following properties:

1. The ASW shall command the DOI to be turned on if and only if the following

conditions are satisfied
   (a) The aircraft descends below the threshold altitude (nominally 2000 ft)
   (b) The DOI is not already on
   (c) The ASW is not inhibited
   (d) The ASW is not reset

2. The ASW shall command the DOI to be turned off if and only if all of the following conditions are satisfied
   (a) The aircraft has attained an altitude greater than the threshold plus hysteresis. (Hysteresis is assumed to be 0.1 * Threshold)
   (b) The DOI is not already OFF
   (c) The ASW is not inhibited
   (d) The ASW is not reset

3. The ASW shall indicate a fault, if any of the following conditions are satisfied:
   (a) ASW is not able to determine the altitude for more than 2 seconds
   (b) DOI does not indicate that it is powered ON within 2 seconds after ASW applies power to it
   (c) Altimeter indicates that the Altitude Quality is bad
   (d) ASW is not inhibited

The first two properties were very similar and checked in the same way. They list four conditions that must exist for the DOI to be turned on or off. Both simply check that all four conditions hold when ever the DOI is turned on or off.

The third property checks that the ASW has not been inhibited or reset and then checks that if any of the three alarm modes (properties 3a - 3c) are true indicating that the alarm signal should be activated.

5.2.3 Experimental Results

In this section we present the results of the experiments in which we tried to verify the properties described in §5.2.2. Each of the models was run for 10 iterations, with each iteration having a 100 instances batch size, and each instance running the model for 40 discrete time steps.

Sensor Voting model

Both properties formulated for the Sensor Voting model were not expected to be violated, and SPY did not detect a violation of either property. Recall from §5.2.2 that property 1 said that the sensor valid flag had to be bad 3 time steps in a row for the model to invalidate that sensor. Internally in the SPY code this threshold was stored in `numBadFlags`. Additional experiments were run with the Sensor Voting model, with `numBadFlags` set to have different values. Whenever this number differed from the 3 (the formulation desired by the original model developers) SPY reported a property violation, showing that it could detect differences between the property formulations and the implementation of the models. These results are summarized in Table 5.1.

There was one subtle bug that SPY missed which was discovered by manual inspection . The Sensor Voting model detects a persistent threshold violation (property

Property	SPY Result	Expected Result
Property 1	Not Violated	Not Violated
Property 2	Not Violated	Not Violated
`numBadFlags` = 2	Violated	Violated
`numBadFlags` = 4	Violated	Violated

Table 5.1: Sensor Voting property check results

2 in §5.2.2) by accumulating the time that any sensor differs (above a set threshold) from any of the other sensors (provided that the sensors are still considered valid by the model). This accumulator value was stored in a floating-point variable. At each time step this accumulator is checked against the allowed time for a sensor to miscompare with the other sensors. However, when accumulating .05 seconds (since the model runs at 20 Hz) ten times, the sum was actually just below .5 seconds. But since both the model and the code used to check the model suffered from this form of numeric creep, SPY did not recorded a failure. Hence a property violation escaped SPY's notice. Our collaborators at the University of Minnesota (who developed the translation system discussed in §5.2.1) independently checked this violation using our commercial tool Reactis. That tool also failed to detected this numeric creep. Since both tools failed to detected this violation, this example demonstrates the danger in using floating-point variables for some applications. After manual inspection detected this error, the property checking code in SPY was re-written to use an integer accumulator to count the number of time steps that two sensors miscompared. Using this integer accumulator,

SPY detected the property violation, demonstrating that the original failure of SPY was due to the numeric creep issue. This means that SPY, when the objective function was properly implemented, found a subtle bug that apparently had not been previously discovered.

Dual FGS

Table 5.2 shows the different properties and their expected results for the Dual FGS model. The results we obtained from SPY differ from the expected results for only one of the nine properties checked, property 4.b. The FGS system was also independently check by our UMN collaborators. Reactis also reported that property 4.b was violated. Since this model has been in production use for some time, it is likely that the property is formulated incorrectly in the requirements documentation we had access to. Even if the property formulation is incorrect and there are no faults in the model, having consistent requirements is obviously valuable. Discovering inconsistent requirements, while not the stated goal of the SPY framework, is always possible when testing requirements against a development artifact.

Property	SPY Result	Expected Result
Property 1	Not Violated	Not Violated
Property 2.a	Violated	Violated
Property 2.b	Not Violated	Not Violated
Property 3.a	Violated	Violated
Property 3.b	Not Violated	Not Violated
Property 4.a	Violated	Violated
Property 4.b	Violated	Not Violated
Property 5.a	Not Violated	Not Violated
Property 5.b	Not Violated	Not Violated

Table 5.2: FGS property check results

Property	SPY Result	Expected Result
Property 1	Violated	Violated
Property 2	Violated	Violated
Property 3	Not Violated	Not Violated

Table 5.3: ASW property check results

Altitude switch model

Table 5.3 shows the results of our experiments with the altitude switch model. When these experiments were started it was thought that properties 1 and 2 should not be violated, but when Reactis was used as an independent check, it also reported properties 1 and 2 violated by the model. This led to a closer examination of the requirements documentation. It became apparent that the model's behavior was not fully encapsulated in the properties, as the properties were recorded. Since the original developers were not available to us, this was as far as our investigation could go.

5.3 Conclusions from NASA Flight Models

This chapter has shown, through comparative analysis, that our current methodology for verifying temporal properties in real valued models has much potential. We showed that

- the translation framework preserves the semantics of our models
- SPY agrees with Reactis on which properties are violated in our models
- SPY was able to find defects in the models that were either
  - already present in the models
  - injected into the formulation of the properties.

A difficulty we encountered in this chapter that was not seen in chapter 4 was the interface between the learner and the model input variables. In chapter 4 ITL could learn directly on the input variables because the requirement models were non-temporal. With the temporal models studied in this chapter, SPY could not learn directly on the model input variables, because they changed during the execution of the model. In two of our flight models, ASW and Dual FGS, the model inputs represented the states of control inputs that were expected to change during the course of a simulation. In our third flight model, Sensor Voting, we had some real value inputs that changed during the simulation, namely the values reported by the individual sensors, as well as discrete control inputs, namely the validity flags reported by each sensor. In fact, the properties explicitly detailed how the model was supposed to react when the control states changed. Presenting SPY with the exact value that each input took during each discrete time step was not desirable for two reasons. First, presenting the learner with that many input variables, the number of model inputs times the number of discrete time steps, would overwhelm the learner's ability to find correlations between the inputs and the objective function evaluation. Second, if the learner were presented with the value of each input at each time step, the learner would return treatments suggesting value assignments to particular inputs *at particular time steps*. For our purposes these would not be useful suggestions. For these reasons we parameterized the model's control inputs as probabilities. These probabilities were constant during a simulation and were the values that the SPY framework learned on. This meant when a defect was discovered by SPY, its advice would be to decrease or increase the probability of some of the control inputs being in one state or the other. In the models studied in this chapter, the learner would reduce some of these probabilities to zero, effectively saying not to use that control input. This advice might not be particularly useful to the model developers.

Since we would like to investigate SPY's ability to find useful range restrictions, we decided to investigate another class of temporal models whose inputs were all real valued, but whose values did not change during the execution of the model. The next chapter investigates SPY's ability to find useful range restrictions with biomathematical models. These models have the above-mentioned advantage of having all real valued inputs that do not change during the execution of the model. In addition, the models we chose have had previous analysis work, so we can compare the results of using SPY with previous results.

CHAPTER 6

VALIDATING SPY ON BIOMATHEMATICAL MODELS

The previous chapter introduced SPY and demonstrated its ability to check certain types of properties in NASA flight models. We discussed how the nature of the model inputs made validating a useful feature of SPY difficult. The nature of the range restrictions offered by SPY, e. g. the pilot should never press the transfer switch, were not practically useful. To demonstrate SPY's ability to find useful range restrictions we investigated a class of models that have been developed and reviewed in research fields outside of machine learning or metaheuristic search.

SPY will attempt to find input range restrictions to these new models, described in §6.1.1, that confine the behavior of the model outputs to specified modes.

6.1 Using ITL in Biomathematical Models

This section discusses two biomathematical models that have been analyzed using techniques outside of the machine learning field. The models will be introduced, along with a description of the experimental goals and results of the experiments using the

SPY framework.

6.1.1 Models Under Consideration

Competitive Exclusion

[104] develops dozens of biomathematical models. Dynamic population models are one of the classic types of biomathematical models. Populations are modeled using coupled differential equations. The first derivatives (in time) are constructed from first principles. These derivatives usually contain several parameters. These parameters can be studied analytically to define different types of behaviors of the model. We will be studying a system of 2 different species that compete in some way in the same niche. We can think of this competition in terms of food supply, space, toxicity, or anything else that would lower the carrying capacity of the niche for both species. This system is called competitive exclusion because analytic analysis shows that under most conditions, one species will be driven to extinction. The general form of the population equations is

$$\frac{dN_1}{dt} = r_1 N_1 \left(1 - \frac{N_1}{K_1} - b_{12}\frac{N_2}{K_1}\right) \tag{6.1}$$

$$\frac{dN_2}{dt} = r_2 N_2 \left(1 - \frac{N_2}{K_2} - b_{21}\frac{N_1}{K_2}\right) \tag{6.2}$$

where N_1 and N_2 are the size of the two species, K_1 and K_2 are the carrying capacities for the two species in the absence of the other species, r_1 and r_2 are the growth rates of the species, and b_{12} and b_{21} measure the degree to which the two species effect each other. All these constants are positive and real-valued. Following the methodology of [104] we rewrite Equations 6.1 and 6.2 in dimensionless terms as

$$\frac{du_1}{d\tau} = u_1(1 - u_1 - a_{12}u_2) \tag{6.3}$$

$$\frac{du_2}{d\tau} = \rho u_2(1 - u_2 - a_{21}u_1) \tag{6.4}$$

where $u_1 = \frac{N_1}{K_1}$, $u_2 = \frac{N_2}{K_2}$, $a_{12} = b_{12}\frac{K_2}{K_1}$, $a_{21} = b_{21}\frac{K_1}{K_2}$, $\rho = \frac{r_2}{r_1}$, and $\tau = r_1 t$. We have eliminated one term (we scaled r_1 to one and redefined r_2 to a ratio) and normalized our population sizes so that the carrying capacity of each species is 1.

Animal Neurons

A simple model of animal neurons was developed in [65]. This model has the advantages of being computationally efficient, while also being able to simulate most of the behaviors of much more complex models. Its main drawback is that its parameters have no physical significance. The model is a pair of coupled ordinary differential equations. The entire model has the form

$$\frac{dv}{dt} = 0.04v^2 + 5v + 140 - u + I \tag{6.5}$$

$$\frac{du}{dt} = a(bv - u) \tag{6.6}$$

$$\text{if } v \geq 30mV, \text{then } v \leftarrow c \text{ and } u \leftarrow u + d$$

Where v and u are dimensionless variables, a, b, c, d are dimensionless parameters, and I is an input DC signal. The variable v represents the membrane potential of the neuron and is the experimentally observable quantity in this model. The variable u is a recovery variable, which models the rate at which K^+ and Na^+ channels can open and close in the neuron.

[65] and [66] state that almost all behaviors seen in real neurons can be modeled using this model, depending on the settings of the different parameters. There is matlab program available for download[1] that lists the different values of the parameters that lead to different classes of behavior.

[1] `http://vesicle.nsi.edu/users/izhikevich/publications/izhikevich.m`

6.1.2 Previous Analytic Work with Models

Competitive Exclusion

The dynamic population model we are using is simple enough to be analyzed using a number of different techniques. Here we simply summarize the linear algebra based technique used in [104]. First we construct the matrix A around a point where $\frac{du_1}{d\tau} = \frac{du_2}{d\tau} = 0$. Then the sign of the eigenvalues of the matrix A determine whether the equilibrium points are stable or unstable. After finding the eigenvalues at the four different equilibrium points, it turns out there is only a single set of conditions that will lead to a stable equilibrium point where $N_1, N_2 \neq 0$. If $0 \leq a_{12} \leq 1$ and $0 \leq a_{21} \leq 1$ then there will be a stable equilibrium point at (u_1^*, u_2^*), where $0 < u_1^* < 1$ and $0 < u_2^* < 1$ (recall equations 6.3 and 6.4 were nondimensionalized, so the carrying capacity for the two different species is 1).

Animal Neurons

Our neuron model is much harder to analyze symbolically, both because it is not first-order (recall the v^2 term in equation 6.5) and because of the reset condition. Izhikevich has spent much time describing some of the analytic properties of different models of animal neurons [64]. All of these discussion are beyond the scope of this thesis. It should be noted that this analysis takes dozens of pages and a high degree of mathematical training to discover. The various parameter settings that lead to the different behaviors are published, but no explanation is given as to how they were found (which suggests a simple trial-and-error search).

6.1.3 Experimental Goals

There are two slightly different goals we have in mind for our experiments. We would like our framework to find the relevant range restrictions and we would like it to NOT offer any range restrictions on parameters that are irrelevant.

Competitive Exclusion

For the dynamic population model we want to discover the conditions that create a stable equilibrium point where $N_1, N_2 \neq 0$, as discussed in 6.1.2. The range restrictions that lead to this behavior are $0 \leq a_{12} \leq 1$ and $0 \leq a_{21} \leq 1$. Since symbolic analysis shows that the value of ρ and the initial populations do not affect the location or existence of equilibrium points, we hope that SPY will not offer any range restrictions on those variables.

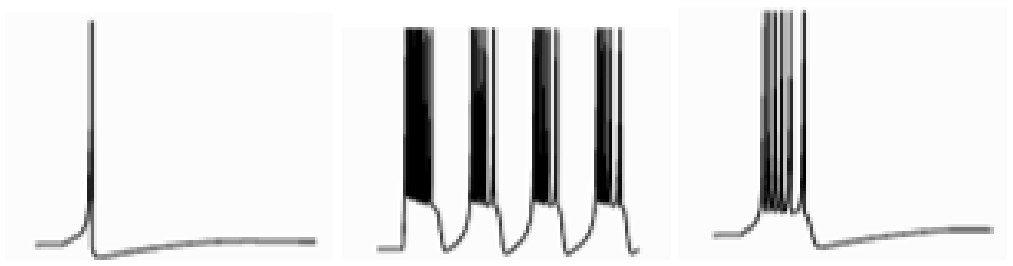

(a) phasic spiking (b) tonic bursting (c) phasic bursting

Figure 6.1: Behavior modes of animal neuron

Animal Neurons

Of all the different behaviors described in [66], three were chosen because it was easy to describe them in mathematical terms. First, *phasic spiking*, Figure 6.1a, which is a single spike and reset. Second, *tonic bursting*, Figure 6.1b, which is several spikes close together in time, followed by a quiescent state, repeated several times. Third, *phasic bursting*, Figure 6.1c, is several spikes close together in time, followed by a quiescent state without any repeats. We attempted to find the input value ranges that lead to these different behaviors.

6.1.4 Methodology

Competitive exclusion model in the SPY language

To highlight some of the interesting features of the SPY language we will discuss the particulars of the competitive exclusion model, as it was developed in SPY. The SPY code for the competitive exclusion model is listed in Figure 6.2 - Figure 6.4.

Figure 6.2 shows the use of the `Choice` function. This function is the main interface between the model and the learner. The `Choice` function maintains a cache of all the values it has returned during an iteration, and these values are what the learner takes as training data. At the start of each run through the model, the function picks a new value and always returns the same value until the start of another run. Currently SPY interacts with its learner best if the values remembered by the `Choice` function are all $[0-1]$, hence the five functions after the `Choice` functions in Figure 6.2 rescale the numbers $[0-1]$ to $[0-15]$. This range was chosen to include the values that would lead to our desired behavior, $[0-1]$, as well as many values that wouldn't. The initial populations are chosen from $[.04-2.04]$. Zero was disallowed because $(0,0)$ is a trivial stable state in the model.

Figure 6.3 shows the model appropriately formulated in the SPY language. The change in the current populations of the two species is calculated and returned to the

```
Choice(a1SpyVal,=,n){
  return linear(a1SpyVal_min,a1SpyVal_max,n); }
Choice(k1SpyVal,=,n) {
  return linear(k1SpyVal_min, k1SpyVal_max,n); }
Choice(k2SpyVal,=,n) {
  return linear(k2SpyVal_min, k2SpyVal_max,n); }
Choice(alphaSpyVal,=,n){
  return linear(alphaSpyVal_min,alphaSpyVal_max,n); }
Choice(betaSpyVal,=,n) {
  return linear(betaSpyVal_min,betaSpyVal_max,n); }

function rho(   tmp) {
  tmp = a1SpyVal();
  #split 0-1 in half and then rescale
  if(tmp>.5) return 28*tmp-13;
  else return tmp*2;
}
function a12(   tmp){
  tmp = alphaSpyVal();
  #split 0-1 in half and then rescale to 0-1 and 1-15
  if(tmp>.5) return 28*tmp-13;
  else return tmp*2;
}
function a21(   tmp) {
  tmp = betaSpyVal();
  #split 0-1 in half and then rescale to 0-1 and 1-15
  if(tmp>.5) return 28*tmp-13;
  else return tmp*2;
}
# don't allow initial populations of zero
function X0() {
  return (k1SpyVal()+.02)*2;
}
function Y0() {
  return (k2SpyVal()+.02)*2;
}
```

Figure 6.2: Picking model variables in the competitive exclusion model

```
# Model:
#    dx/dt = x * (1 - x - a12*y)
#    dy/dt = rho*y * (1 - y - a21*x)

function dx(x,y) {
  return x*(1-x-a12()*y);
}
function dy(x,y) {
  return rho()*y*(1-y-a21()*x);
}
```

Figure 6.3: Competitive exclusion model, adapted from [104]

```
function main(warmup,   n,delx,dely,xt,yt) {
  a1SpyVal(n);
  k1SpyVal(n);k2SpyVal(n);
  alphaSpyVal(n);betaSpyVal(n);

  xt=X0();yt=Y0();
  REDO=0
  if (!warmup) {print "# new experiment" >> "points.dat";
    print count++ " " a12() " " a21() >> "parameters.dat";}
  do {
    REDO++;
    if (!warmup) print xt " " yt " " sumworth >> "points.dat";

    # calculate change to populations
    delx = dx(xt,yt)/UNITSTEP;
    dely = dy(xt,yt)/UNITSTEP;
    xt += delx;xt=(xt<SMALL?0:xt);
    yt += dely;yt=(yt<SMALL?0:yt);
    totalChange = (abs(delx)<SMALL?0:delx) + (abs(dely)<SMALL?0:dely);
  } while (totalChange != 0 || REDO<10)
  return ((xt>.SMALL?1:0)+(yt>.SMALL?1:0));
}
```

Figure 6.4: Model execution code in the competitive exclusion model

main model executing code. Notice that the two functions call the functions described in Figure 6.2.

Figure 6.4 shows the main control loop for the competitive exclusion model. It first sets the values that will be used for the current simulation, lines 2-4. Then it initializes the populations, line 6. Lines 8, 9, and 12 write some useful information to disk. Lines 15 and 16 call the two functions that comprise this model, shown in Figure 6.3. Lines 17 and 18 adjust the current sizes of the two species. Lines 19 and 20 ensure that model runs for at least `REDO` time steps, but then stops when the populations approach a steady state. Finally line 21 is the worth function, which will be described below. This function is SPY's version of the objective function.

Worth Functions

One practical detail of using SPY that turned out to be more challenging than expected was how to code the worth function. The worth function is what SPY uses to decide whether an instance belongs to the preferred class or not, and hence plays the role of the objective function. Tar4 needs discrete classes, but since tar4 has a built-in discretizer, worth functions can generate numeric class labels. The worth function then must produce a single numeric value that describes how close to the preferred class any instance is. The difficulty in formulating these functions should be familiar to anyone who works at an interface between natural languages and mathematics. While it may be easy to describe a certain type of behavior in a natural language, developing a mathematical formulation can be difficult.

Competitive exclusion As mentioned above, in the dynamic population model we were looking for a single behavior, both species surviving at the equilibrium point. So our worth function was

$$worth = (\text{if } N_1 > 0 \text{ then } 1 \text{ else } 0) + (\text{if } N_2 > 0 \text{ then } 1 \text{ else } 0) \tag{6.7}$$

Note that the analytic analysis in [104] shows if $N_1^0 > 0$ and $N_2^0 > 0$, one species will always survive. Equation 6.7 therefore will only return 1 or 2.

Animal neurons The worth function used in the animal neuron model depended on which behavior we wanted SPY to look for.

Phasic spiking A neuron exhibits phasic spiking when its membrane potential (v from equation 6.5) spikes once and is then quiescent. The model increments the *reset* counter every time the reset condition is reached during each run. The worth function used was

$$worth = \text{if } reset = 1 \text{ then } 1 \text{ else } 0 \tag{6.8}$$

Tonic bursting A neuron exhibits tonic bursting when its membrane potential spikes in repeated groups, where each group has a very high frequency, but there is a long quiescent period between each burst. To classify this behavior the time between each reset was recorded. If the time since the last reset was below a threshold, *low*, the counter *fast* was incremented. If the time since the last reset was above a higher threshold, *high*, (some times were therefore ignored completely), the counter *slow* was incremented. Finally at the end of the run the following worth function was used

$$worth = \text{if } (fast > 10 \textbf{ and } slow > 2) \text{ then } 1 \text{ else } 0 \tag{6.9}$$

The small number of *slow* resets delineated the groups, while the large number of *fast* resets ensured that each group had several resets.

Phasic bursting Phasic bursting is similar to tonic bursting, except there is only a single group of high frequency spikes. Using the same counters as in the tonic bursting case, the worth function was

$$\begin{aligned} &worth = \\ &\text{if } (fast > 4 \textbf{ and } slow = 0 \textbf{ and } last\_spike > high) \text{ then } 1 \text{ else } 0 \end{aligned} \tag{6.10}$$

where *last_spike* was the time since the last reset when the simulation ended. This worth function was particularly hard to develop. Recall that the definition of *slow* was based on the time between two resets. With phasic bursting the quiescent period should last until the end of the simulation, so there would be no reset to cause the *slow* counter to be incremented. This last detail was difficult to formulate.

6.1.5 Experimental Results

Competitive Exclusion

We ran 10 repeats with each repeat lasting 10 iterations. Each iteration used only 100 instances. The results of these experiments are summarized in Table 6.1 and Table 6.2. Table 6.1 shows the bounds that SPY found on the parameters that our analytic analysis showed were relevant (a_{12} and a_{21}). Table 6.2 shows the bounds found on the irrelevant parameters (ρ, u_1^0, and u_2^0). If there is a number in parentheses after the bound, it indicates the number of times that particular bound was found. The final column shows the average number of iterations before SPY stopped adjusting the bounds on a particular parameter (lower or upper). The original bounds on a_{12}, a_{21} and ρ were $[0, 15]$, while the original bounds on u_1^0 and u_2^0 were $[0.04, 2.04]$ (0 could not be allowed because $(0, 0)$ is a trivial stable state).

name	lower bounds	upper bounds	speed
a_{12}	0.0(10)	0.2(6), 0.6(2), 0.8(2)	1.8
a_{21}	0.0(10)	0.2(3), 0.3, 0.4(4), 0.8(2)	1.2

Table 6.1: Relevant parameters in competitive exclusion model

name	lower bounds	upper bounds	speed
ρ	0.0(10)	15.0(10)	1.0
u_1^0	0.04(10)	2.04(10)	1.0
u_2^0	0.04(10)	2.04(10)	1.0

Table 6.2: Irrelevant parameters in competitive exclusion model

Table 6.1 shows that SPY was successful in finding ranges that restrict the model to our desired behavior. Every repeat found restrictions that would always lead to our preferred behavior (both species surviving in the stable state). The speed to convergence was also very fast. It took SPY only slightly more than 1 iteration to settle on the range restrictions for a_{21} and just under 2 iterations for a_{12}. Analytically there is no difference between the influence of a_{12} and a_{21}, so we suspect the difference in the speed of convergence for the two parameters is an artifact of the way SPY utilizes the treatments returned by tar4 or an artifact of the random search. We can't call this experiment a complete success though, because the range restrictions were, for the most part, too restrictive. That is, the ranges SPY suggested eliminated a large part of the parameter space that would lead to the preferred class.

Table 6.2 shows that SPY was completely successful in ignoring irrelevant parameters. Of the six irrelevant bounds SPY could have adjusted, not a single one in ten

parameter	lower bound	upper bound
a	0	.1
b	0	1
c	-80	-40
d	0	10

Table 6.3: Original bounds on parameters for animal neuron model

trials was moved by SPY.

Animal Neurons

In this section we will discuss the results of the experiments with the animal neuron model. All experiments were run for 10 iterations, with 10 repeats for each behavior. For this set of experiments a much larger batch size, 400, was needed to allow SPY to converge to reasonable answers.

There is a pair of figures for each behavior. The first figure shows the average worth at the end of each iteration for the ten iterations ran during the search for all repeats that had a final average worth above .3 (in practice all repeats either had a final average worth close to 1 or close to 0). (Recall a worth of 0 means that the behavior was not observed and a worth of 1 means that the behavior was observed.) The second figure shows the range restrictions that SPY found by the last iteration for all repeats that had a final average worth above .3. Going from left to right, the four columns represent the four independent variables in our model, a, b, c and d. Going from the

	parameter name			
behavior	a	b	c	d
phasic spiking	.02	.25	-65	6
tonic bursting	.02	.20	-50	2
phasic bursting	.02	.25	-55	.05

Table 6.4: Published parameter settings [66]

bottom to the top, the different rows represent the different successful repeats. The top row shows the original bounds on the independent variables. This makes it easy to see how much SPY restricted the range for each variable. The original bounds of the variables are given in Table 6.3. The same original bounds were used for all behaviors. These bounds were set after reviewing all the values used in [66] for the 20 different behaviors demonstrated. The parameter settings used in [66] and [65] for the different behaviors we are interested in are listed in Table 6.4.

Phasic spiking Only four of the ten repeats found range restrictions that exhibited phasic spiking. The speed of convergence, when the search was successful, was moderately fast. Figure 6.5 shows that for the four successful searches, convergence happened after 6-8 iterations. A few qualitative statements can be made about Figure 6.6, which shows the final range restrictions found by SPY. The value of the b parameter seems critical, because all successful repeats found the same narrow range. The values of the c and d parameters seem less critical, as a wide range (covering almost the entire range in the case of parameter d) of range restrictions was successful. It is harder to say something about parameter a. All successful repeats found very restrictive bounds for a, but the bounds found do not coincide. There may be a dependence between a, c, and d that SPY is finding, but this is only a conjecture.

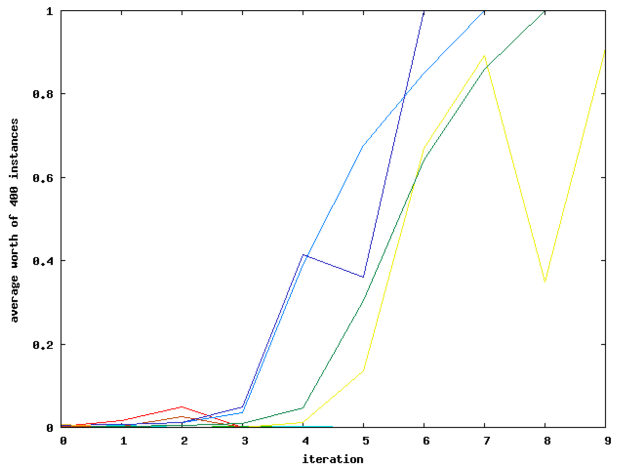


Figure 6.5: Average worth for 10 repeated trials, trying to find phasic spiking

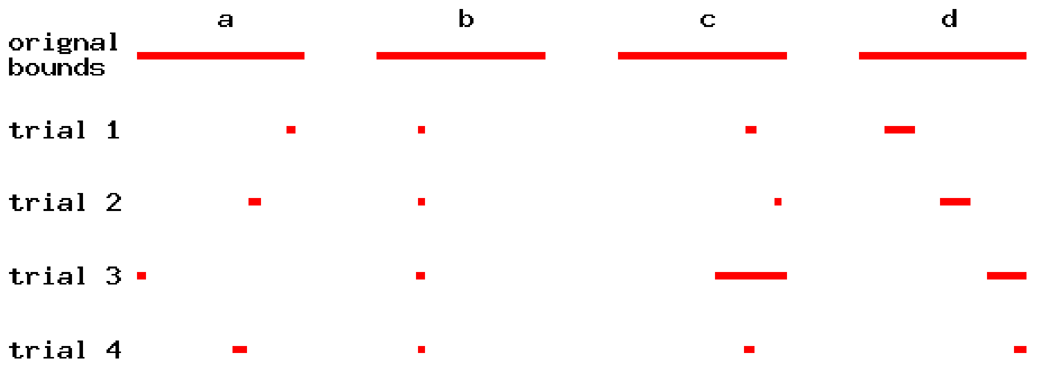


Figure 6.6: Final bounds for all successful repeats, trying to find phasic spiking

Tonic bursting SPY was more successful at finding range restrictions that exhibited tonic bursting. Eight of the ten repeats ended with an average worth very close to 1. The speed of convergence was also faster while trying to find tonic bursting. It ranged from only 2 iterations to 5 iterations, as seen in Figure 6.7. The larger number of successful repeats makes it easier to make qualitative statements about the range restrictions found, which are shown in Figure 6.8. First, we see something that was not observed with phasic spiking: some of the bounds were not changed at all by SPY. For instance, many of the successful repeats never lowered the upper bound on the d parameter. The same is true of the lower bound on the c parameter and the upper bound on the b parameter. As with phasic spiking, the d parameter has wide range restrictions, covering slightly more than half the original range. Tonic bursting is also less sensitive to the b parameter than phasic spiking. The noncoincidence of the found range restrictions on the a and b parameters again might be evidence of some dependence between the parameters.

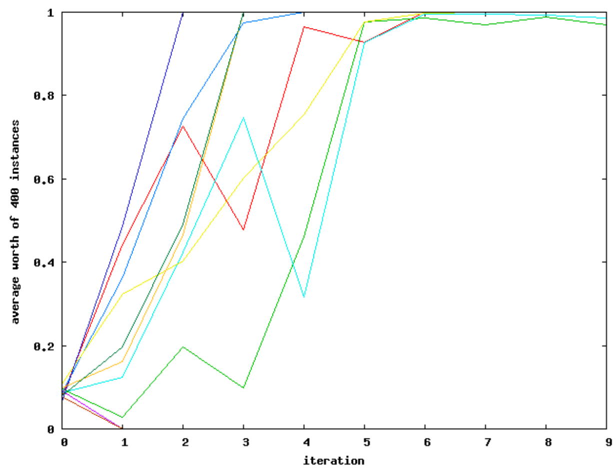


Figure 6.7: Average worth for 10 repeated trials, trying to find tonic bursting

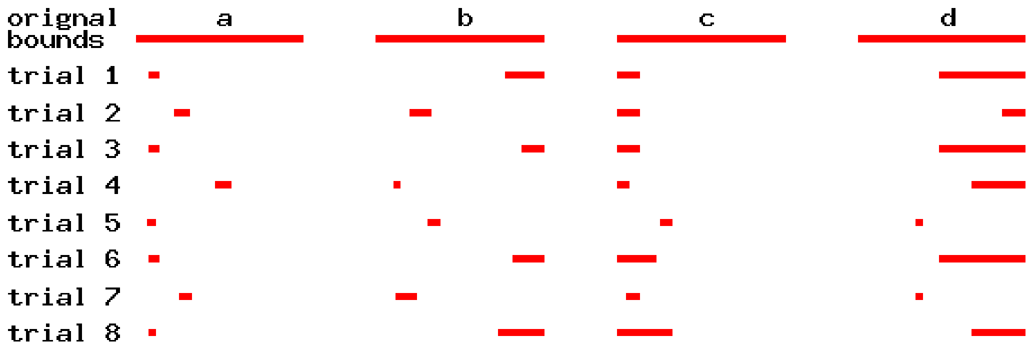


Figure 6.8: Final bounds for all successful repeats, trying to find tonic bursting

Phasic bursting SPY was more successful at finding good range restriction for phasic bursting than it was at finding phasic spiking, although not as successful as with tonic bursting. Six of the ten repeats ended with an average worth close to 1. The speed of convergence was slightly slower than the speed for tonic bursting, 3-4 iterations, with one outlier needing 8 iterations, as can be seen in Figure 6.9. Phasic bursting appears to be very sensitive to the a parameter, with all successful repeats finding the same narrow range, see Figure 6.10. The other three parameters show a wide range of bounds, although it is difficult to say anything specific, other than to note that the 2nd successful trial has the highest bounds on the parameters b, c, and d.

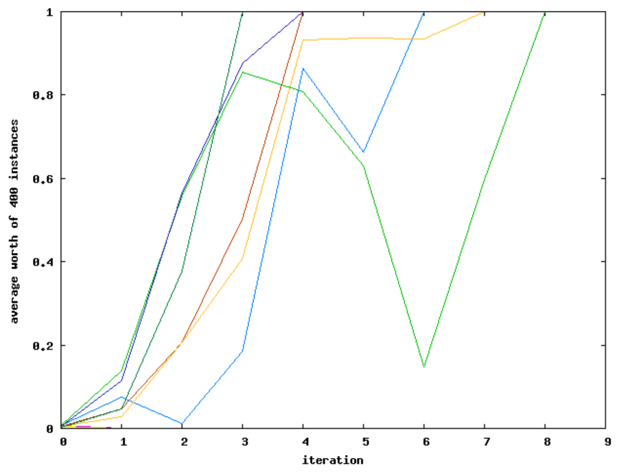


Figure 6.9: Average worth for 10 repeated trials, trying to find phasic bursting

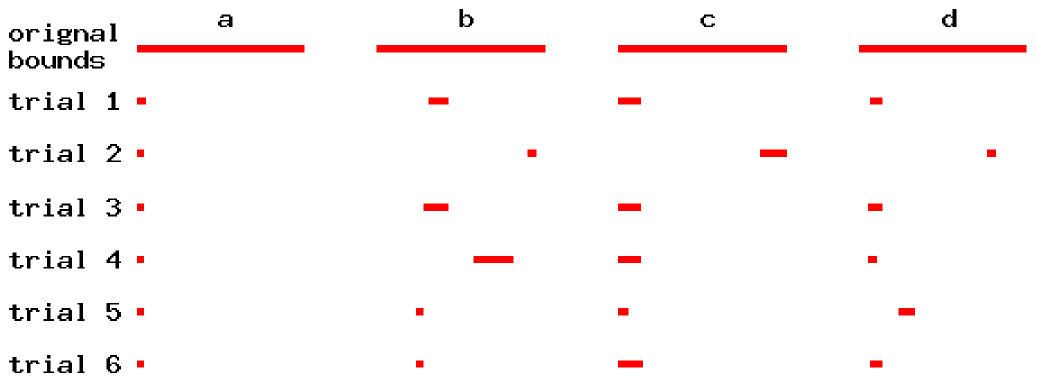


Figure 6.10: Final bounds for all successful repeats, trying to find phasic bursting

6.2 Conclusions from Biomathematical Models

Recall that in §2.3.1 we discussed the desirability of using a continuous objective function. The work presented in this section violates that advice. In other words, if an instance was close to, but not in, a portion of the parameter space that had a positive worth value, the objective function would return 0 for that instance. In the work presented in chapter 4 we had a continuous objective function, but this was our first experience trying to optimize what was essentially a boolean function. We believe this is a general problem when trying to identify types of behaviors when there is no concept of being "close to" the desired behavior.

This discontinuity of the objective function causes what should be a heuristically

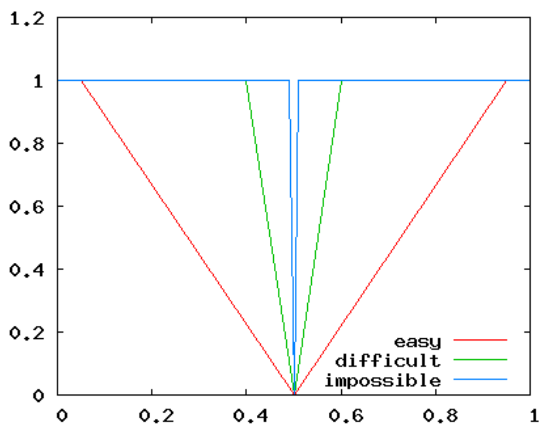


Figure 6.11: Example of discontinuous functions

driven search to degrade to a purely random search. If the objective function is discontinuous, then, during the first few iterations, SPY won't be able to find treatments that restrict future iterations to portions of the input space that have promising values. In addition, the more discontinuous the objective, i. e. the smaller the region with a high objective evaluations, the harder it will be for SPY (or any other stochastic search method) to find these promising regions. This can be seen in Figure 6.11. All three lines are discontinuous, but the likelihood of finding the optimal solution in the *impossible* line is so much smaller than the likelihood of finding the optimal solution in the *easy* line, that no heuristic could outperform a purely random search.

In some problem domains the objective function can be made more continuous through domain specific knowledge. In §2.3.2 we saw how some researchers trying to automatically generate test cases modified the objective function through symbolic analysis of the source code being tested. This particular technique might be adapted to the experiments presented in this chapter by including the value of some the intermediate variables used in the objective function (§6.1.4) as part of the value returned by the objective function. For example, the original objective function for tonic bursting was

$$worth = \text{if } (fast > 10 \textbf{ and } slow > 2) \text{ then } 1 \text{ else } 0$$

perhaps this could be modified to

```
if ($fast > 10$ and $slow > 2$) then
    return 1
else
    return -($|10 - fast|$ + $|2 - slow|$)
end if
```

This function would return 1 when tonic bursting was present. But when tonic bursting wasn't present it would return a negative number, whose value approaches zero as the intermediate values used by the objective function, $fast$ and $slow$, approach the values that indicate the presence of tonic bursting. This type of modification might not work because we can not be sure that small changes in the value of the independent variables will only cause small effects on the intermediate values used in the new proposed objective function, i. e. we can not be sure that the values of $fast$ and $slow$ are approximately continuous in the independent variables. Experimentation with objective functions of this type would reveal how useful this modification is. Of course it is possible that this will be a domain specific feature: some intermediate values may be approximately continuous and some not depending on the model being analyzed.

Regardless of the continuousness of the objective function, the search bias of SPY's underlying learner, tar4, can lead to over-restrictive bounds, of the type we saw in §6.1.5. To see why imagine that the true bound for some parameter q is (x, y). If the largest value for q sampled that exhibited the desired behavior was r ($r < y$) and the smallest value that didn't exhibit the desired behavior was s ($s > y$), no learner could reliably say anything about the range (r, s) (particularly if the objective function is undifferentiable). Because tar4 favors true-positives over true-negatives, it will always pick a range with an upper bound of r. Then in future iterations the range (r, y) will never be sampled from, and the upper bound will have no chance to increase to y. This problem is similar to, but distinct from, getting trapped in a local optimum. While both may be solved by randomly sampling outside the current range restrictions, reporting overly restrictive bounds to the analyst that still constrain the model to optimal behavior modes is not the same as reporting bounds to the analyst that constrain the behavior of the model to sub-optimal modes. This problem in particular should be the focus of our future work.

CHAPTER 7

CONCLUSIONS

This thesis addresses the question of whether ITL can be successfully used as a meta-heuristic search technique for model-based development. To address this question we investigated models from two stages of the software life cycle. Chapter 4 experimented with requirements models, often used in the earlier life stages of the software life cycle. Chapter 5 experimented with digital logic circuit models, often used during the testing stage, i. e. late stage, of the software life cycle. This chapter we will discuss conclusions from our work with requirements models, §7.1, and conclusions from our work with the SPY framework, §7.2.

7.1 Conclusion from Early Life Cycle Models

We did a comprehensive study of three requirements models to investigate some preference characteristics of ITL and our new discretization method, extreme sampling. We found

- repeated success with *holo* model and extended success to *aero* and *cob* models

- *bore* is currently the best version of extreme sampling
- larger batch sizes, up to 500, increase performance on requirements models
- the good/bad ratio should be in the range 15% to 25%, or possibly higher
- *bore* has high stability and low variance
- *bore* clearly outperforms diagonal striping on
  - solution quality
  - speed of convergence
- *bore* outperforms simulated annealing on speed of convergence, while finding similar quality solutions

7.2 Conclusions from Late Life Cycle Models

We used five models written in the SPY language, two by hand and three translated from Simulink, to investigate the capabilities of the SPY framework. We found that

- SPY could find property violations in temporal models with real-valued inputs
- SPY is able to find range restrictions that constrain model behavior
  - SPY found these restrictions quickly, often only 2-5 iterations
  - SPY could tell the difference between relevant and irrelevant parameters
- the search bias of the underlying learner can lead to over restrictive bounds
- discrete worth functions reduce learning efficiency. Worth may need to
  - be augmented with additional heuristics, like path coverage
  - include hints as to how close intermediate variables were to correct values
- translation framework preserves model semantics

The availability of analytic techniques for one of our models allows us to say conclusively that SPY may find overly restrictive bounds on input variables, but that the bounds were always correct. While this is a result of the search bias of the underlying learner (maximizing true positives without penalizing false negatives), an improved search strategy in SPY may correct this.

The discrete nature of behavior mode identification caused significant problems in one of our models. Optimizing low cardinality discrete functions, particularly boolean functions, will be a problem for any stochastic search. Addressing this problem will involve new methodologies for developing objective functions, rather than a change to the search strategy.

The only area where domain knowledge or apriori mathematical knowledge was needed was in picking the original bounds on the parameters.

CHAPTER 8

FUTURE WORK

In the future we would like to see more acceptance of ITL as a general metaheuristic search technique as well as see its use in model-based software development. For ITL to become a well respected techniques, such as those described in §2.2 and §2.3, the following research would have to be completed

1. ITL should be applied to instructive toy problems
2. ITL should be used to solve other software engineering problems
3. the conjecture that ITL's use of partial solutions prevents it from getting stuck in local optima should be verified
4. a more rigorous study of alternative search strategies for ITL should be conducted
5. a downloadable, ready-to-use implementation of ITL should be made available to the public
6. methodologies for dealing with low-cardinality discrete functions should be investigated
7. different automatic stopping conditions could be investigated

The list's length reflects the short time that has elapsed since ITL was developed and the even shorter time that ITL has been considered a metaheuristic search technique, rather than any intrinsic difficulty foreseen in completing the proposed work.

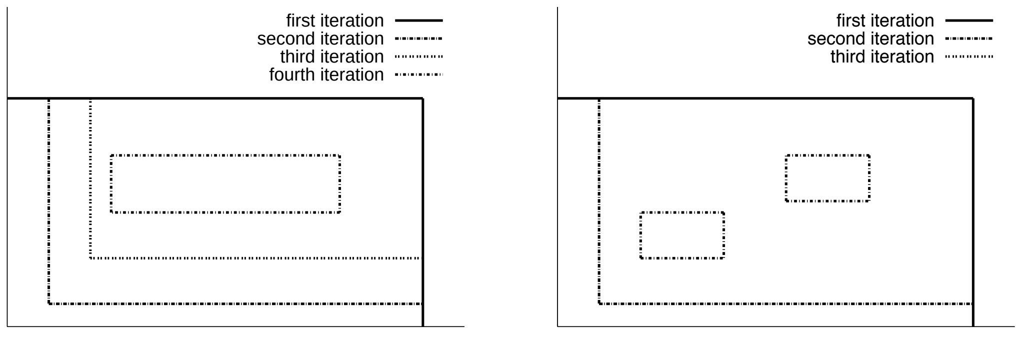
Toy problems New techniques are often applied to simple problems when first introduced. This thesis introduced ITL as a metaheuristic search technique, but skipped the step of proving ITL could be effective on simple problems. For example, ITL could be used to find satisfying assignments for propositional logic statements, e. g. in LSAT problems, to find maxima in piecewise-linear functions, or to find function approximators.

Other software engineering problems As shown in §2.3, metaheuristic search techniques have been applied to numerous problem domains in software engineering. It would bolster the case for ITL if it was used to solve problems in domains other than requirements analysis, chapter 4, and model property verification, chapter 5.

ITL robustness in the face of local optima It was conjectured in chapter 3 that the use of partial solutions makes ITL unlikely to get stuck in local optima. This claim should be investigated, most likely through the use of toy problems have have a variable amount of deceptiveness. This question has great bearing on the issue of search strategy. If ITL is currently, as conjectured, robust in the presence of numerous local optima, then improvements to the search strategy would not have to be tailored to this problem. If, however, this conjecture turns out to be incorrect, then new search strategies specifically designed to combat the presence of local optima need to be developed.

Improved search strategy As discussed in chapter 3, ITL currently uses a simple greedy forward select without any backtracking. Investigation into more sophisticated strategies should be done. There are at least two features that the search strategy could use. First, the search should be able to proceed backwards as well as forwards, i. e. ITL should be able to retract some of the previous treatments according to some rule. Second, the search should have some parameter that controls the amount of exploration and exploitation done. This could be set by the user at the start of the search or could be dynamically adjusted during the search.

Another feature that might prove useful is allowing ITL to maintain multiple partial descriptions so that multiple hyper-rectangles be searched each iteration. This is shown in Figure 8.1. While some attempt was made to conduct this type of search, as described in §3.1.5, that search strategy did not maintain multiple hyper-rectangles. While searching points that passed any treatment found during each iteration, ITL did not keep track of which treatments were actually responsible for increasing the objective score. Searching only one hyper-rectangle makes the assumption that there are not multiple locations in the input space that have equal quality near-optimal solutions. A traditional metaheuristic search technique would not necessarily try to capture multiple near-optimal solutions, even if they were of the same quality, since the user only wants a single high-quality solution. But ITL, particular when viewed as a model controller,



single hyper-rectangle multiple hyper-rectangles

Figure 8.1: Restricting search to a single vs. multiple hyper-rectangles

attempts to restrict output behavior, so if a model exhibits highly desirable behavior in more than one location in the input space, ITL should try to find all such locations to provide the user with a more flexible set of advice.

Public availability Our current implementation of ITL[1] grew out of a GUI developed for instructional purposes. While adherence to OO design principles has made extension of the code base possible, it is probably time for the prototype to be thrown away now that we have gained valuable insight into what works and what needs to be tried. This next version of the code should be as portable and ready to use as the GA package gac[2] or the machine learning weka package[3]. In particular, the code should be developed with the possibility of replacing certain parts of the algorithm easily; for

[1]Internally called surfer.
[2]Found at `http://www.cs.uwyo.edu/~wspears/freeware.html`.
[3]Found at `http://www.cs.waikato.ac.nz/ml/weka/`.

instance, we have already mentioned that the search strategy might need improvement. This code also needs to have a publicly accessible location so that other researchers can access it without needing to get in contact with its developers.

New methodologies for objective function development The difficulties we saw ITL have with the models in chapter 6 might be solved with improved methodologies for developing objective functions. While objective function development will always require a certain amount of domain-specific knowledge, it is possible that certain methodologies tailored for use with ITL can be found that offer general guidance to analysts faced with discrete objective functions.

Automatic stopping conditions All the studies in this thesis ran the ITL search for a fixed number of iterations, i. e. a fixed search depth. The size of partial descriptions that will be useful is a domain specific feature. Different models with input spaces with different cardinalities will require different sized partial descriptions to constrain output behavior. So that users of ITL do not have to experiment with different search depths to determine the size of useful partial descriptions, different automatic stopping conditions should be developed. These stopping conditions might be based on the average values of the partial descriptions or the variance of the partial descriptions.

REFERENCES

[1] T. Abdel-Hamid and S. Madnick. *Software Project Dynamics: An Integrated Approach*. Prentice-Hall Software Series, 1991.

[2] C. Abts, B. Clark, S. Devnani-Chulani, E. Horowitz, R. Madachy, D. Reifer, R. Selby, and B. Steece. COCOMO II model definition manual. Technical report, Center for Software Engineering, USC,, 1998. `http://sunset.usc.edu/COCOMOII/cocomox.html#downloads`.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th International Conference on Very Large Databases*, 1994. Available from `http://www.almaden.ibm.com/cs/people/ragrawal/papers/vldb94_rj.ps`.

[4] Jesus S. Aguilar-Ruiz, Isabel Ramos, Jose Riquelme, and Miguel Toro. An evolutionary approach to estimating software development projects. *Information and Software Technology*, 43(14):875–882, 2001.

[5] M. Akhavi and W. Wilson. Dynamic simulation of software process models. In *Proceedings of the 5th Software Engineering Process Group National Meeting (Held at Costa Mesa, California, April 26 - 29)*. Software engineering Institute, Carnegie Mellon University, 1993.

[6] J. L. Alvarez, J. Mata, Jose C. Riquelme, and I. Ramos. A data mining method to support decision making in software development projects. In *ICEIS'2003: Fifth International Conference on Enterprise Information Systems*, 2003.

[7] V. Babovic. Mining sediment transport data with genetic programming. *Proceedings of the First International 10 Conference on New Information Technologies for Decision Making in Civil Engineering*, pages 875–886, 1998.

[8] James E. Baker. Reducing bias and inefficiency in the selection algorithm. In *Proc. of the 2nd Intl Conf on GA*, pages 14–21. Lawrence Erlbaum Associates, Inc. Mahwah, NJ, USA, 1987.

[9] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30(5), May 2004.

[10] S.B. Bay and M.J. Pazzani. Detecting change in categorical data: Mining contrast sets. In *Proceedings of the Fifth International Conference on Knowledge*

Discovery and Data Mining, 1999. Available from `http://www.ics.uci.edu/~pazzani/Publications/stucco.pdf`.

[11] T Bayes. An essay toward solving a problem in the doctrine of chances. *Philosophical Transactions of the Royal Society of London*, 53:370–418, 1764.

[12] B. Beizer. *Software Testing Techniques*. Van Nostrand Rheinold, New York, 1990.

[13] Forrest H. Bennett III, Martin A. Keane, David Andre, and John R. Koza. Automatic synthesis of the topology and sizing for analog electrical circuits using genetic programming. In Kaisa Miettinen, Marko M. Makela, Pekka Neittaanmaki, and Jacques Periaux, editors, *Evolutionary Algorithms in Engineering and Computer Science*, pages 199–229, Jyvaskyla, Finland, 30 May - 3 June 1999. John Wiley & Sons.

[14] Peter J. Bentley and Jonathan P. Wakefield. Generic reporesentation of solid-object geometry for genetic search. *Microcomputers in civil engineering*, 11(3), 1996.

[15] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. *Lecture Notes in Computer Science*, 1579:193–207, 1999.

[16] Robert R. Birge. Protein-based optical computing and memories. *Computer*, 25(11):56–67, 1992.

[17] C.L. Blake and C.J. Merz. UCI repository of machine learning databases, 1998. URL: `http://www.ics.uci.edu/~mlearn/MLRepository.html`.

[18] B. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[19] M. Boraso, C. Montangero, and H. Sedehi. Software cost estimation: an experimental study of model performances. Technical Report TR-96-22, Departimento Di Informatatica, Universita Di Pisa, 6, 1996.

[20] Remco Bouckaert. Choosing between two learning algorithms based on calibrated tests. In *International Conference on Machine Learning*, 2003. Available from `http://www.cs.pdx.edu/~timm/dm/10x10way`.

[21] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. Classification and regression trees. Technical report, Wadsworth International, Monterey, CA, 1984.

[22] F. P. Brooks. *The Mythical Man-Month, Anniversary edition*. Addison-Wesley, 1995.

[23] C.H. Cai, A.W.C. Fu, C.H. Cheng, and W.W. Kwong. Mining association rules with weighted items. In *Proceedings of International Database Engineering and Applications Symposium (IDEAS 98)*, August 1998. Available from `http://www.cse.cuhk.edu.hk/~kdd/assoc_rule/paper.pdf`.

[24] V. Cerny. A thermodynamical approach to the travelling salesman problem: an efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45:41–51, 1985.

[25] M.-S. Chen and F. H. Liao. Crossover operators with adaptive probability. In *INTSYS '98: Proceedings of the IEEE International Joint Symposia on Intelligence and Systems*, page 10, Washington, DC, USA, 1998. IEEE Computer Society.

[26] E. Chiang and T. Menzies. Simulations for very early lifecycle quality evaluations. *Software Process: Improvement and Practice*, 7(3-4):141–159, 2003. Available from `http://menzies.us/pdf/03spip.pdf`.

[27] S. Chulani and B. Boehm. Modeling software defect introduction and removal: COQUALMO. Technical Report USC-CSE-99-510, University of Southern California, Center for Software Engineering, 1999.

[28] W. Clancey, P. Sachs, M. Sierhuis, and R. van Hoof. Brahms: Simulating practice for work systems design. In P. Compton, R. Mizoguchi, H. Motoda, and T. Menzies, editors, *Proceedings PKAW '96: Pacific Knowledge Acquisition Workshop*. Department of Artificial Intelligence, 1996.

[29] Ryan Clark. Faster treatment learning. Master's thesis, Portland State University, 2006.

[30] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[31] J. Clarke, J.J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. Reformulating software engineering as a search problem. *IEE Proceedings-Software*, 150(3):161–175, 2003.

[32] John Clarke, Jose J. Dolado, Mark Harman, R. M. Hierons, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, M. Shepperdand, and Bryan Jones. Reformulating software engineering as a search problem, 2003.

[33] Oscar Cordón, Francisco Herrera, and Luciano Sánchez. *Evolutionary learning processes for data analysis in electrical engineering applications*. John Wiley

and Sons, 1997.

[34] S.L. Cornford, M.S. Feather, and K.A. Hicks. DDP a tool for life-cycle risk management. In *IEEE Aerospace Conference, Big Sky, Montana*, pages 441–451, March 2001.

[35] P. Denno, M. P. Steves, D. Libes, and E. J. Barkmeyer. Model-drven integration using existing models. *IEEE Software*, 20(5):59–63, Sept.-Oct. 2003.

[36] Jose J. Dolado. A validation of the component-based method for software size estimation. *IEEE Transactions on Software Engineering*, 26(10):1006–1021, October 2000.

[37] Jose J. Dolado. On the problem of the software cost function. *Information and Software Technology*, 43(1):61–72, 1 January 2001.

[38] James Dougherty, Ron Kohavi, and Mehran Sahami. Supervised and unsupervised discretization of continuous features. In *International Conference on Machine Learning*, pages 194–202, 1995.

[39] L. J. Eshelman and J. D. Schaffer. Real-coded genetic algorithms and interval-schemata. In *Foundations of Genetic Algorithms-2*, pages 187–202, 1993.

[40] Brian Everitt. *The Analysis of Contingency Tables*. Chapman and Hall, London, 1977.

[41] M. Feather, H. In, J. Kiper, J. Kurtz, and T. Menzies. First contract: Better, earlier decisions for software projects. In *ECE UBC tech report*, 2001. Available from `http://menzies.us/pdf/01first.pdf`.

[42] Martin Feather and Steve Cornfordi. Quantitative risk-based requirements reasoning. *Requirements Engineering Journal*, 8(4):248–265, 2003.

[43] M.S. Feather and T. Menzies. Converging on the optimal attainment of requirements. In *IEEE Joint Conference On Requirements Engineering ICRE'02 and RE'02, 9-13th September, University of Essen, Germany*, 2002. Available from `http://menzies.us/pdf/02re02.pdf`.

[44] Chung-Wei Feng, Liang Liu, and Scott A. Burns. Using genetic algorithms to solve construction time-cost trade-off problems. *Journal of Computing in Civil Engineering*, 11(3):184–189, 1997.

[45] G. R. Finnie, G. E. Wittig, and J.-M. Desharnais. A comparison of software effort estimation techniques: using function points with neural networks, case-based reasoning and regression models. *Journal of System and Software*, 39(3):281–289, 1997.

[46] L. Fogel. *Artificial Intelligence through Simulated Evolution*. Wiley and Sons, 1966.

[47] R. France, S. Ghosh, E. Song, and D. Kim. A metamodeling approach to pattern-based moel refractoringt. *IEEE Software*, 20(5):52–58, Sept.-Oct. 2003.

[48] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[49] Fred Glover and M. Laguna. Tabu search. In C. Reeves, editor, *Modern Heuristic Techniques for Combinatorial Problems*, Oxford, England, 1993. Blackwell Scientific Publishing.

[50] Fred Glover and Claude McMillan. The general employee scheduling problem: an integration of MS and AI. *Comput. Oper. Res.*, 13(5):563–573, 1986.

[51] D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[52] Jack Greenfield and Keith Short. *Software factories : assembling applications with patterns, models, frameworks, and tools*. Wiley Publishing, Indianapolis, IN, 2004.

[53] B. Hailpern and P. Tarr. Model-driven develpment: the good, the bad, and the ugly. *IBM Systems Journal*, 45(3):451–461, 2006.

[54] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[55] M.A. Hall and G. Holmes. Benchmarking attribute selection techniques for discrete class data mining. *IEEE Transactions On Knowledge And Data Engineering*, 15(6):1437– 1447, 2003.

[56] D. Harel. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

[57] Mark Harman and John Clark. Metrics are fitness functions too. *10th International Software Metrics Symposium*, 2004.

[58] Mark Harman and Bryan F. Jones. Search-based software engineering. *Information & Software Technology*, 43(14):833–839, 2001.

[59] H. Harrell, L. Ghosh, and S. Bowden. *Simulation Using ProModel*. McGraw-Hill, 2000.

[60] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall, 1999.

[61] John Holland. *Adaption in natural and artificial systems*. MIT Press Cambridge, 1975.

[62] Y. Hu. Treatment learning, 2002. Masters thesis, Unviersity of British Columbia, Department of Electrical and Computer Engineering. In preperation.

[63] Y. Hu. Treatment learning: Implementation and application, 2003. Masters Thesis, Department of Electrical Engineering, University of British Columbia.

[64] Eugene M. Izhikevich. Neural excitability, spiking, and bursting. *International Journal of Bifurication and Chaos*, 10(6):1171–1266, 2000.

[65] Eugene M. Izhikevich. Simple model of spiking neurons. *IEEE Transactions on Neural Networks*, 14(6):1569–1572, November 2003.

[66] Eugene M. Izhikevich. Which model to use for cortical spiking neurons? *IEEE Transactions on Neural Networks*, 15(6):1063–1070, September 2004.

[67] W. M. Jenkins. The genetic algorithm-or can we improve design by breeding. *IEE Colloquium on Artificial Intelligence in Civil Engineering*, pages 1/1–1/4, 1992.

[68] B. Jones, H. Sthamer, and D. Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, 11(5):299–306, 1996.

[69] B. Jones, H. Sthamer, X. Yang, and D. Eyres. The automatic generation of software test data sets using adaptive search techniques. In *Proceedings of the 3rd International Conference on Software Quality Management*, pages 435–444, 1995.

[70] C. Z. Jonikow and Z. Michalewicz. An experimental comparison of binary and floating point representations in genetic algorithms. In *International Conference on Genetic Algorithms*, pages 31–38, 1991.

[71] Jan Jrjens and Jorge Fox. Tools for model-based security engineering. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 819–822, New York, NY, USA, 2006. ACM Press.

[72] J. Karlsson, C. Wohlin, and B. Regnell. An evaluation of methods for prioritizing software requirements. *Journal of Information and Software Technology*, 39(14-15):939–947, 1998.

[73] Joachim Karlsson and Kevin Ryan. A cost-value approach for prioritizing requirements. *IEEE Softw.*, 14(5):67–74, 1997.

[74] C. L. Karr, S. K. Sharma, W. J. Hatcher, and T. R. Harper. Fuzzy control of an exothermic chemical reaction using genetic algorithms. *Engineering Applications of Artifical Intelligence*, 6(6):575–582, 1993.

[75] D. Kelton, R. Sadowski, and D. Sadowski. *Simulation with Arena, second edition*. McGraw-Hill, 2002.

[76] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[77] Bogdan Korel. Automated test data generation for programs with procedures. In *ISSTA '96: Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, pages 209–215, New York, NY, USA, 1996. ACM Press.

[78] John Koza. *Genetic Programming: on the programming of computers by means of natural selection*. MIT Press Cambridge, 1992.

[79] J. E. Labossiere and N. Turrkan. On the optimization of the tensor polynomial failure theory with a genetic algorithm. *Transactions of the Canadian Society for Mechinical Engineering*, 16(3-4):251–265, 1992.

[80] A. Law and B. Kelton. *Simulation Modeling and Analysis*. McGraw Hill, 2000.

[81] Albert L. Lederer and Jayesh Prasad. Nine management guidelines for better cost estimating. *Commun. ACM*, 35(2):51–59, 1992.

[82] A. K. Lokketangen and S. Storoy. Tabu search within a pivot and complement framework. *International Transactions in Operations Research*, 1(3):305–316, 1994.

[83] R.H. Martin and D. M. Raffo. A model of the software development process using both continuous and discrete models. *International Journal of Software Process Improvement and Practice*, June/July 2000.

[84] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[85] Steve McConnell. *Code Complete, Second Edition*. Microsoft Press, Redmond, WA, USA, 2004.

[86] G. McGraw, C. Michael, and M. Schatz. Generating software test data by evolution. Technical report, Reliable Software Technologies, Sterling, VA, 1997. Submitted to IEEE Transactions on Software Engineering.

[87] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[88] S.J. Mellor, A.N. Clark, and T. Futagamii. Model-driven development - guest editor's introduction. *IEEE Software*, 20(5):14– 18, Sept.-Oct. 2003.

[89] T. Menzies, E. Chiang, M. Feather, Y. Hu, and J.D. Kiper. Condensing uncertainty via incremental treatment learning. In Taghi M. Khoshgoftaar, editor, *Software Engineering with Computational Intelligence*. Kluwer, 2003. Available from `http://menzies.us/pdf/02itar2.pdf`.

[90] T. Menzies and Y. Hu. Constraining discussions in requirements engineering. In *First International Workshop on Model-based Requirements Engineering*, 2001. Available from `http://menzies.us/pdf/01lesstalk.pdf`.

[91] T. Menzies and Y. Hu. Data mining for very busy people. In *IEEE Computer*, November 2003. Available from `http://menzies.us/pdf/03tar2.pdf`.

[92] T. Menzies and Y. Hu. Just enough learning (of association rules): The TAR2 treatment learner. In *Artificial Intelligence Review (to appear)*, 2004. Available from `http://menzies.us/pdf/02tar2.pdf`.

[93] T. Menzies and J.D. Kiper. How to argue less, 2001. Available from `http://menzies.us/pdf/01jane.pdf`.

[94] T. Menzies and J.D. Kiper. Machine learning for requirements engineering, 2001. Available from `http://menzies.us/pdf/01ml4re.pdf`.

[95] T. Menzies and E. Sinsel. Practical large scale what-if queries: Case studies with software risk assessment. In *Proceedings ASE 2000*, 2000. Available from `http://menzies.us/pdf/00ase.pdf`.

[96] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, January 2007. Available from `http://menzies.us/pdf/06learnPredict.pdf`.

[97] Tim Menzies and Andres Orrego. Incremental discreatization and bayes classifiers handles concept drift and scaled very well. In *IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING*, 2005. Submitted, IEEE TKDE, Available from `http://menzies.us/pdf/05sawtooth.pdf`.

[98] P. Mi and W. Scacchi. A knowledge-based environment for modeling and simulation software engineering processes. *IEEE Transactions on Knowledge and Data Engineering*, pages 283–294, September 1990.

[99] Zbigniew Michalewicz. *Genetic algorithms + data structures = evolution programs (3rd ed.)*. Springer-Verlag, London, UK, 1996.

[100] A. Miller. *Subset Selection in Regression (second edition)*. Chapman & Hall, 2002.

[101] Brad L. Miller and David E. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, 9:193–212, 1995.

[102] Yuichi Miyamoto, Tatsuya Miyatake, Soh Kurosaka, and Yoshinobu Mori. A parameter tuning for dynamic simulation of power plants using genetic algorithms. *Transactions of the Institute of Electrical Engineers of Japan C*, 113-D(12):1410–1415, 1993.

[103] T.P. Moran and J.M. Carroll. *Design Rationale: Concepts, Techniques, and Use*. Lawerence Erlbaum Associates, 1996.

[104] J. D. Murray. *Mathematical Biology*. Springer-Verlag, 1980.

[105] Takeshi Nakajo and Hitoshi Kume. A case history analysis of software error cause-effect relationships. *IEEE Trans. Softw. Eng.*, 17(8):830–838, 1991.

[106] M. Negnevitsky. *Artificial Intelligence, A Guide to Intelligent Systems*. Addison-Wesley Longman Publishing, 2002.

[107] Bashar Nuseibeh and Steve Easterbrook. Requirements engineering: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 35–46, New York, NY, USA, 2000. ACM Press.

[108] Object Management Group. *MDA Guide Version 1.0.1*, June 2003.

[109] Roy P. Pargas, Mary Jean Harrold, and Robert Peck. Test-data generation using genetic algorithms. *Software Testing, Verification & Reliability*, 9(4):263–282, 1999.

[110] R. Poli, S. Cagnoni, and G. Valli. Genetic design of optimum linear and nonlinear QRS detectors. *IEEE Transactions on Biomedical Engineering*, 42(11):1137–41, 1995.

[111] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[112] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufman, 1992. ISBN: 1558602380.

[113] J. R. Quinlan. Learning with Continuous Classes. In *5th Australian Joint Conference on Artificial Intelligence*, pages 343–348, 1992. Available from `http://citeseer.nj.nec.com/quinlan92learning.html`.

[114] L. Rela. Evolutionary computing in search-based software engineering. Master's thesis, Lappeenranta University of Technology, 2004.

[115] Conor Ryan. *Automatic re-engineering of software using genetic programming*. Kluwer Academic Publishers, 2000.

[116] T. L. Saaty. *The Analytic Hierarchy Process*. McGraw-Hill, Inc., 1980.

[117] E. Sanchez, H. Miyano, and J. P. Branchet. Optimization of fuzzy queries with genetic algorithms. *Proceding Sixth International Fuzzy Systems Association World Congress*, 2:293–296, 1995.

[118] M. Sebag, M. Schoenauer, and H. Maitournam. Parametric and non-parametric identification of macro-mechanical models. In D. Quagliarella, J. Périaux, C. Poloni, and G. Winter, editors, *Genetic Algorithms and Evolution Strategy in Engineering and Computer Science*, pages 327–340. John Wiley and Sons, Chichester, 1998.

[119] S. Sendall and W. Kozacaynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, Sept.-Oct. 2003.

[120] M. Shepperd and C. Schofield. Estimating software project effort using analogies. *IEEE Transactions on Software Engineering*, 23(12), November

1997. Available from `http://www.utdallas.edu/~rbanker/SE_XII.pdf`.

[121] S. Buckingham Shum and N. Hammond. Argumentation-based design rationale: What use at what cost? *International Journal of Human-Computer Studies*, 40(4):603–652, 1994.

[122] Mike Spivey. *The Z notation: a reference manual*. Prentice Hall, 2nd edition, 1992.

[123] H. Sterman. *Business Dynamics: Systems Thinking and Modeling for a Complex World*. Irwin McGraw-Hill, 2000.

[124] T.Menzies and Y. Hu. The TAR2 treatment learner, 2002. Available from `http://www.ece.ubc.ca/twiki/pub/Softeng/TreatmentLearner/intro.pdf`.

[125] Nigel Tracey. *A search-based automated test-data generation framework for safety-critical systems*. PhD thesis, University of York, 2000.

[126] Nigel Tracey, John Clark, and Keith Mander. Automated program flaw finding using simulated annealing. In *ISSTA '98: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, pages 73–81, New York, NY, USA, 1998. ACM Press.

[127] Nigel Tracey, John Clark, and Keith Mander. The Way Forward for Unifying Dynamic Test Case Generation: The Optimisation-based Approach. In *IFIP International Workshop on Dependable Computing and its Applications (DCIA 98)*, Johannesburg, January 1998.

[128] Nigel Tracey, John Clark, Keith Mander, and John A. McDermid. An automated framework for structural test-data generation. In *Automated Software Engineering*, pages 285–288, 1998.

[129] Nigel Tracey, John Clark, John McDermid, and Keith Mander. *A search-based automated test-data generation framework for safety-critical systems*. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[130] David Waddington and Patrick Lardieri. Model-centric software development. *IEEE Computer*, 39(2):28–29, February. 2006.

[131] Fiona Walkerden and Ross Jeffery. An empirical study of analogy-based software effort estimation. *Empirical Softw. Engg.*, 4(2):135–158, 1999.

[132] Geoffrey I. Webb, Shane Butler, and Douglas Newlands. On detecting differences between groups. In *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 256–265, New York, NY, USA, 2003. ACM Press.

[133] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.

[134] Joachim Wegener, Hartmut Pohlheim, and Harmen Sthamer. Testing the temporal behavior of real-time tasks using extended evolutionary algorithms. In *RTSS '99: Proceedings of the 20th IEEE Real-Time Systems Symposium*, page 270, Washington, DC, USA, 1999. IEEE Computer Society.

[135] Ron Weiss and Thomas F. Knight, Jr. Engineered communications for microbial robotics. *Lecture Notes in Computer Science*, 2054:1–15, 2001.

[136] D. Whitley. A free lunch proof for gray versus binary encodings. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, pages 726–733, Orlando, Florida, USA, 13-17 1999. Morgan Kaufmann.

[137] D. Whitley, J. R. Beveridge, C. Guerra-Salcedo, and C. Graves. Messy genetic algorithms for subset feature selection. In Thomas Bäck, editor, *Proceedings of the Seventh International Conference on Genetic Algorithms (ICGA97)*, San Francisco, CA, 1997. Morgan Kaufmann.

[138] J. Whittle and P. Jayaraman. Generating hierarchical state machines from use case charts. In *IEEE International Conference on Requirements Engineering (RE2006)*, 2006.

[139] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, 1999.

[140] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, and K. Karapoulios. Application of genetic algorithms to software testing. *5th International Conference on Software Engineering and its Applications*, pages 625–636, 1992.

[141] Ying Yang and Geoffrey I. Webb. A comparative study of discretization methods for naive-bayes classifiers. In *Proceedings of PKAW 2002: The 2002 Pacific Rim Knowledge Acquisition Workshop*, pages 159–173, 2002.

[142] Eric S. K. Yu. Towards modeling and reasoning support for early-phase requirements engineering. In *RE '97: Proceedings of the 3rd IEEE International Symposium on Requirements Engineering (RE'97)*, pages 226–235, Washington, DC, USA, 1997. IEEE Computer Society.

[143] Jun Zhang, Henry S. H. Chung, and Jinghui Zhong. Adaptive crossover and mutation in genetic algorithms based on clustering technique. In *GECCO '05:*

Proceedings of the 2005 conference on Genetic and evolutionary computation, pages 1577–1578, New York, NY, USA, 2005. ACM Press.

APPENDIX A

EXTREME SAMPLING RESULTS

In chapter 4 we presented the results of our experimentation with extreme sampling. For space reasons not all the results were presented. This appendix contains all the data analyzed from that experimentation.

A.1 Results of Solution Quality

§4.2 displayed only the median delta comparisons This section displays the 1st, 2nd, and 3rdquartiles for the delta comparisons.

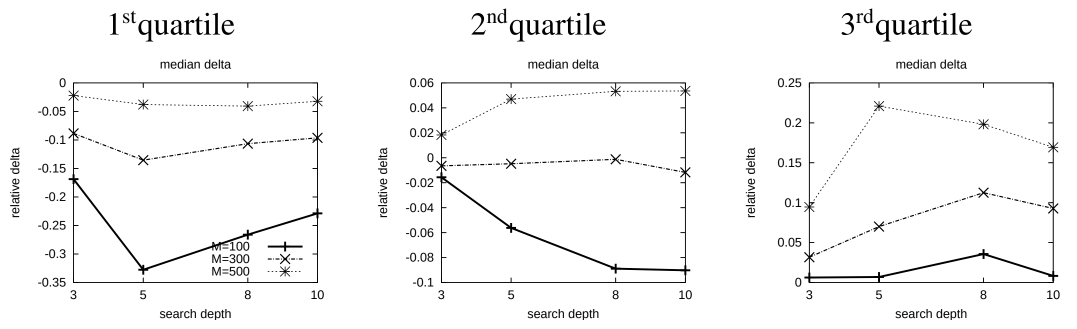


Figure A.1: The effects of batch size, M, versus search depth

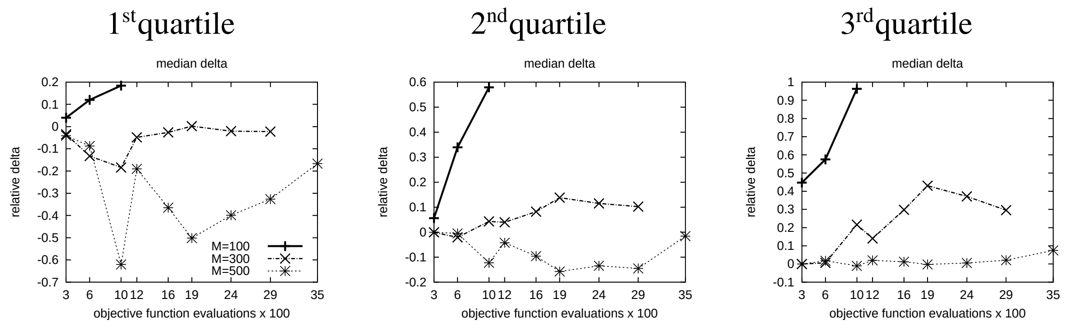


Figure A.2: The effects of batch size, M, versus objective function evaluations

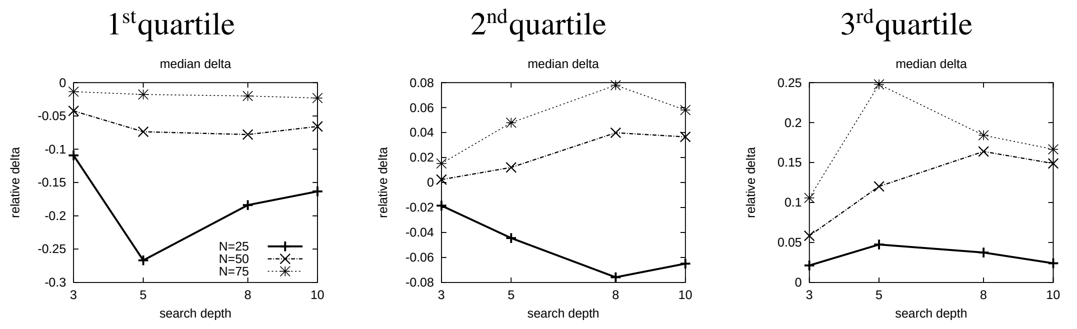


Figure A.3: The effects of the good/bad ratio, N

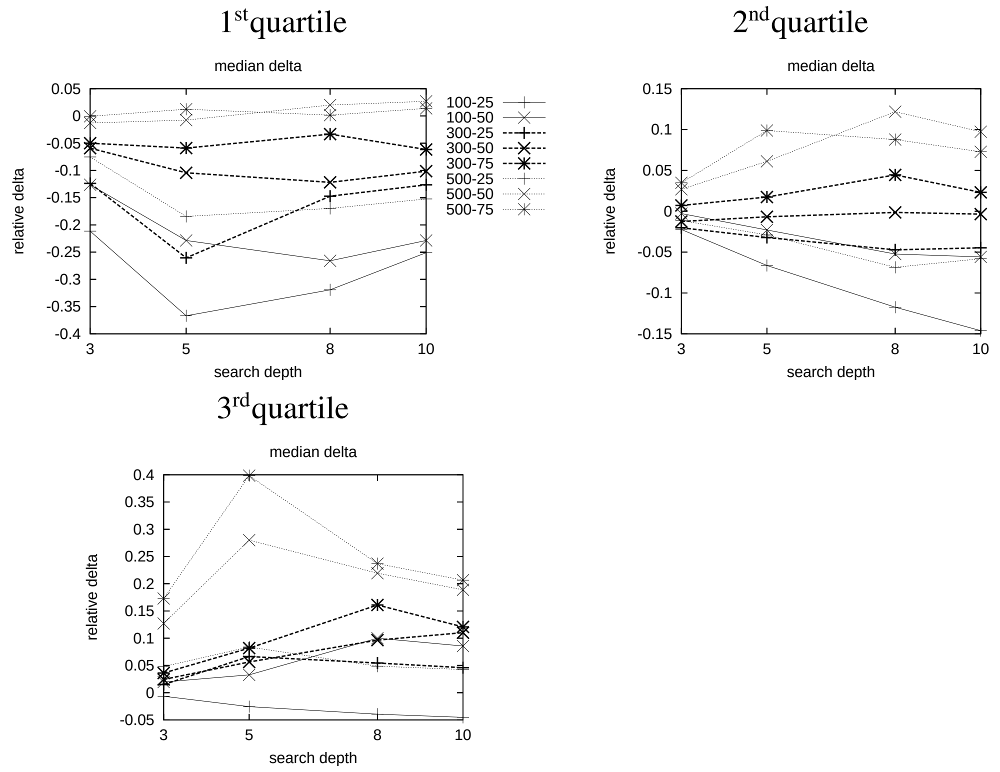


Figure A.4: The effects of the M/N combinations, versus search depth

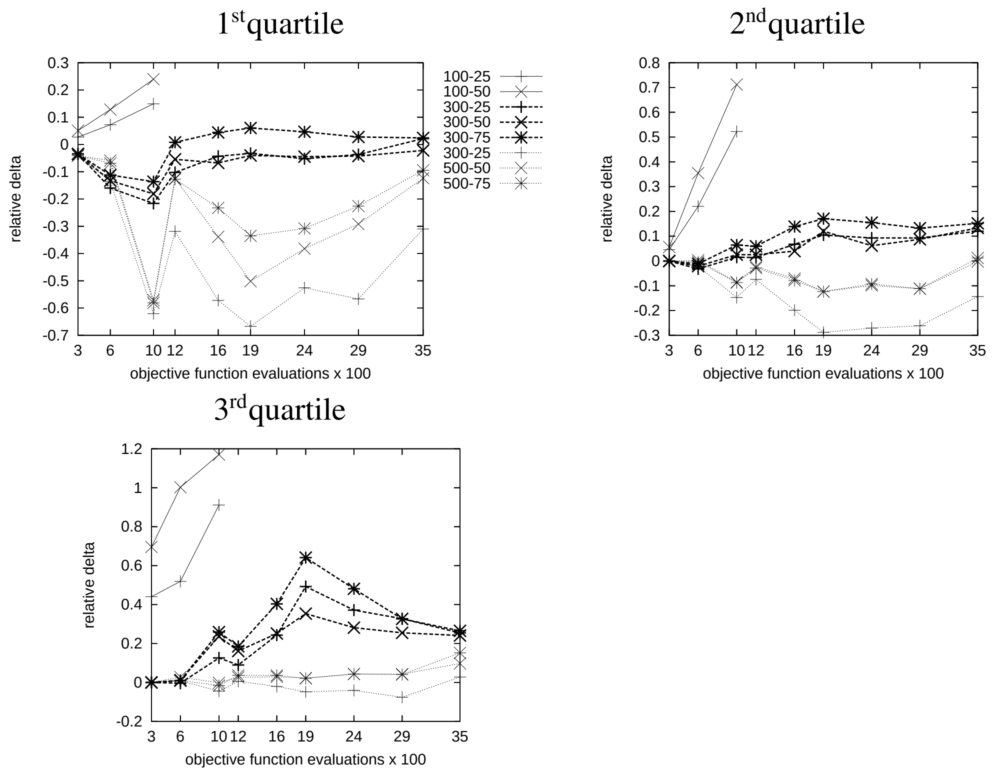


Figure A.5: The effects of the M/N combinations, versus objective function evaluations

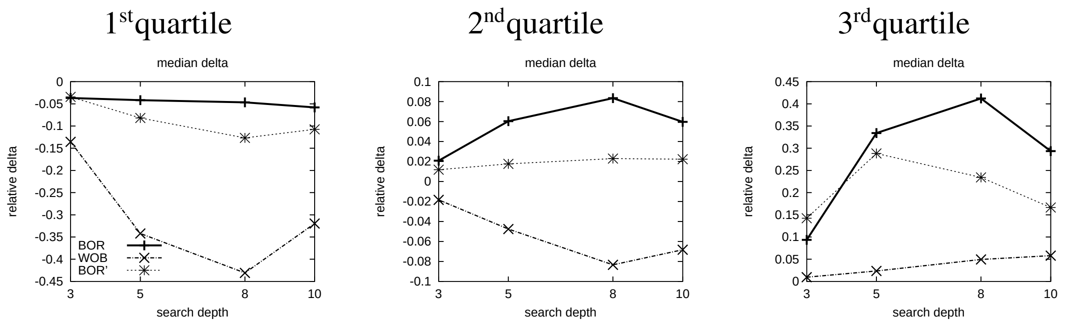


Figure A.6: The effects of the selector

A.2 Stability of Repeated Trials

§4.3 did not display the stability results from all selectors and all M/N combinations. This section displays the search trajectories for all 720 trials described in §4.2.

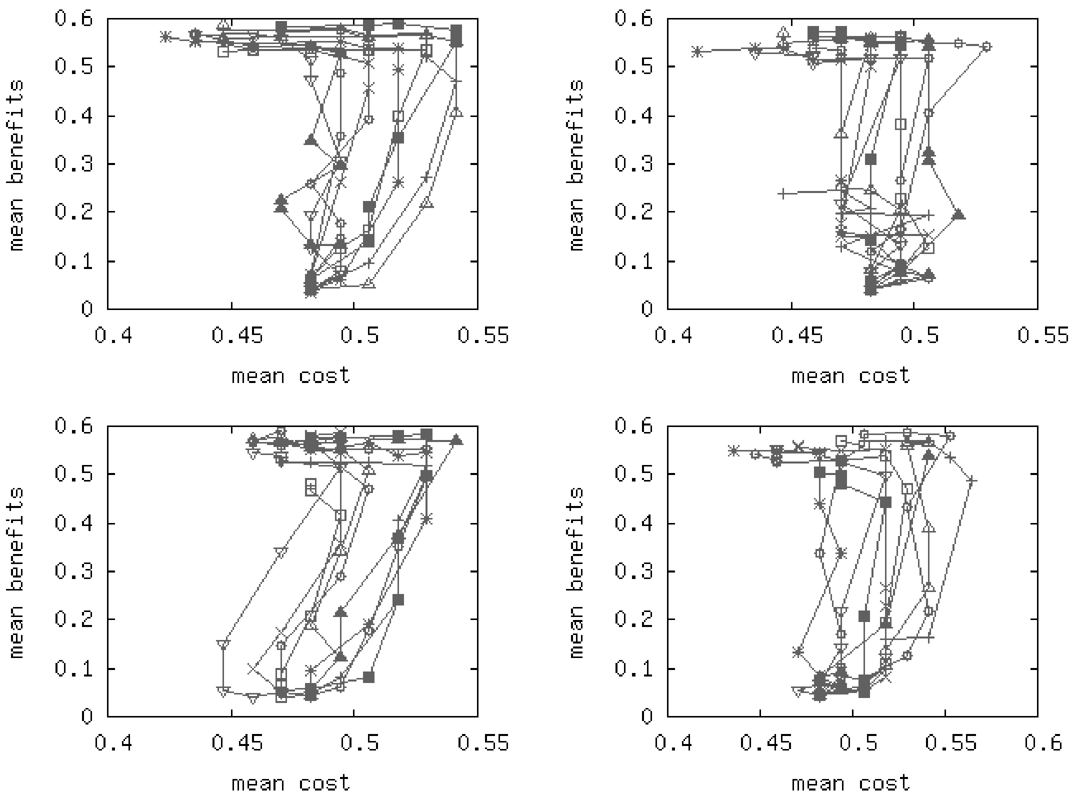


Figure A.7: Cost vs. Benefit in the *aero* model with the *bore* selector

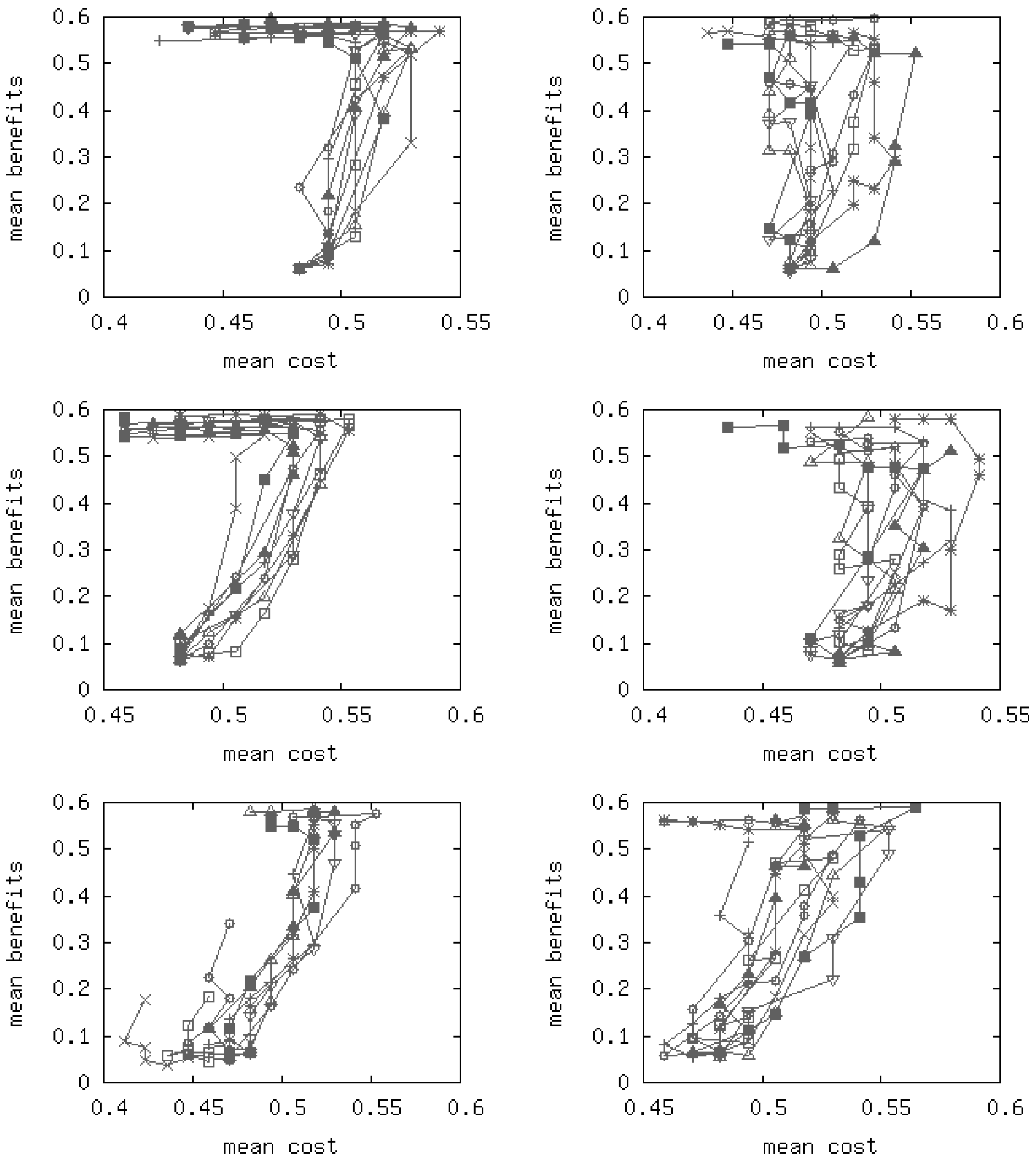


Figure A.8: Cost vs. Benefit in the *aero* model with the *bore′* selector

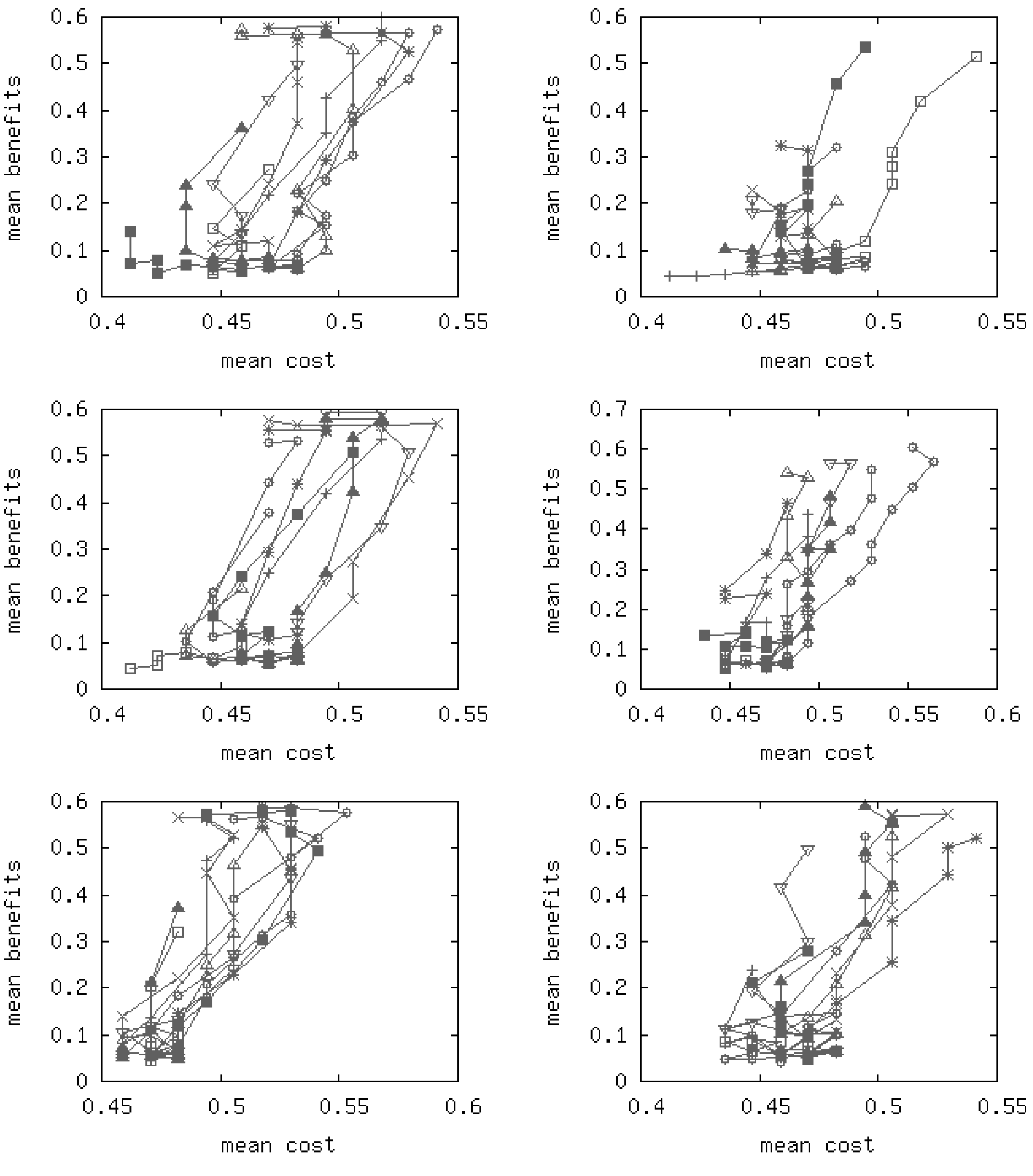


Figure A.9: Cost vs. Benefit in the *aero* model with the *wob* selector

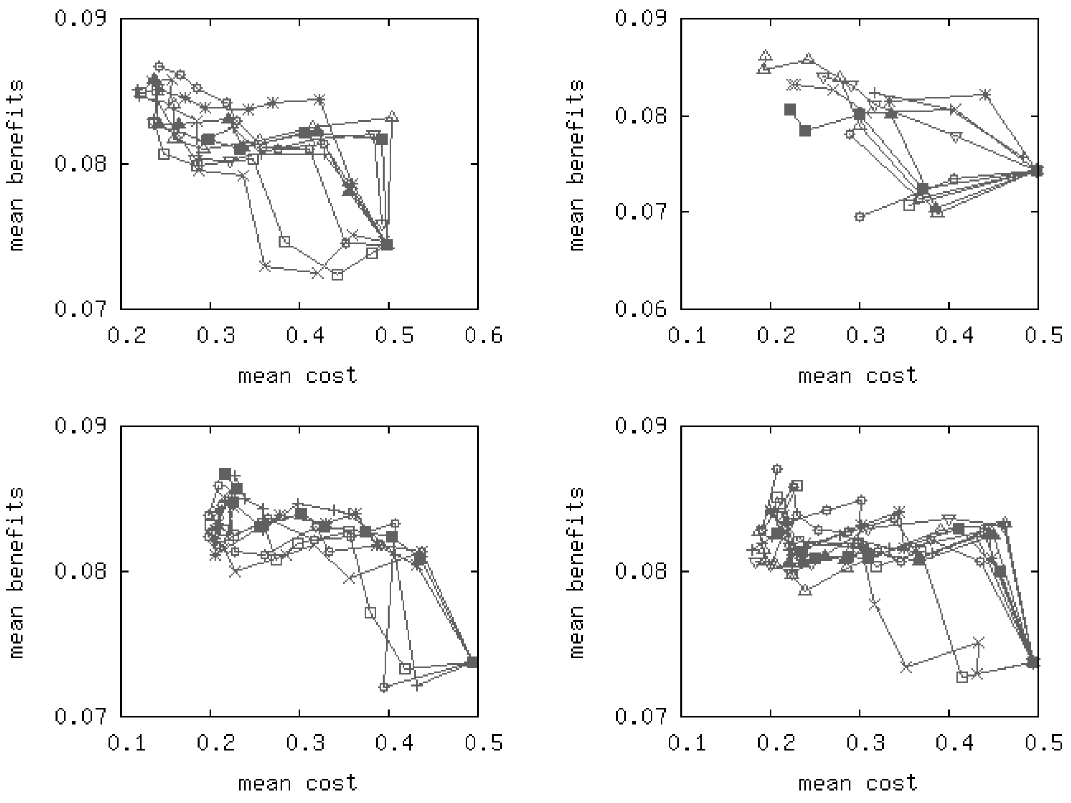


Figure A.10: Cost vs. Benefit in the *holo* model with the *bore* selector

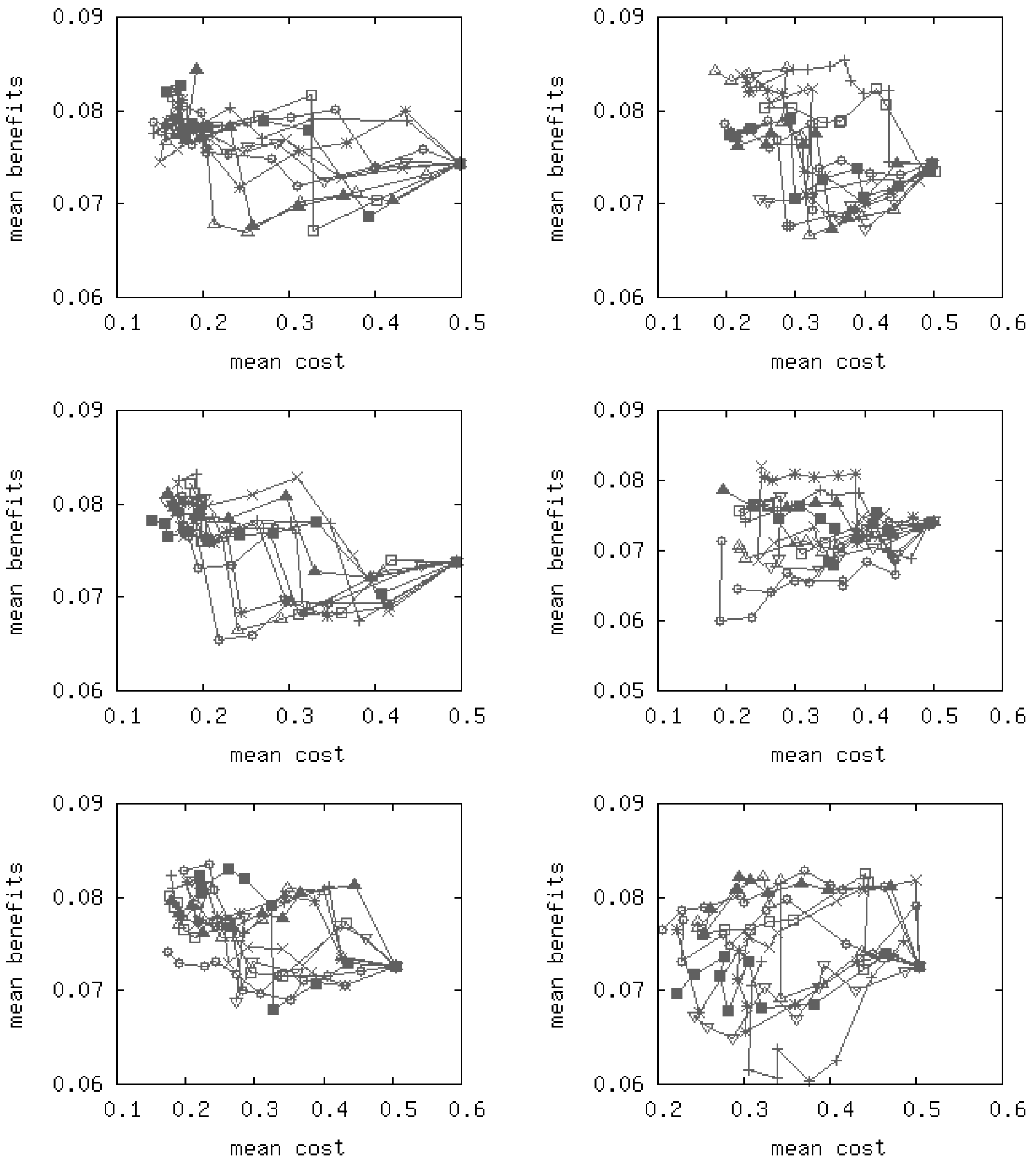


Figure A.11: Cost vs. Benefit in the *holo* model with the *bore′* selector

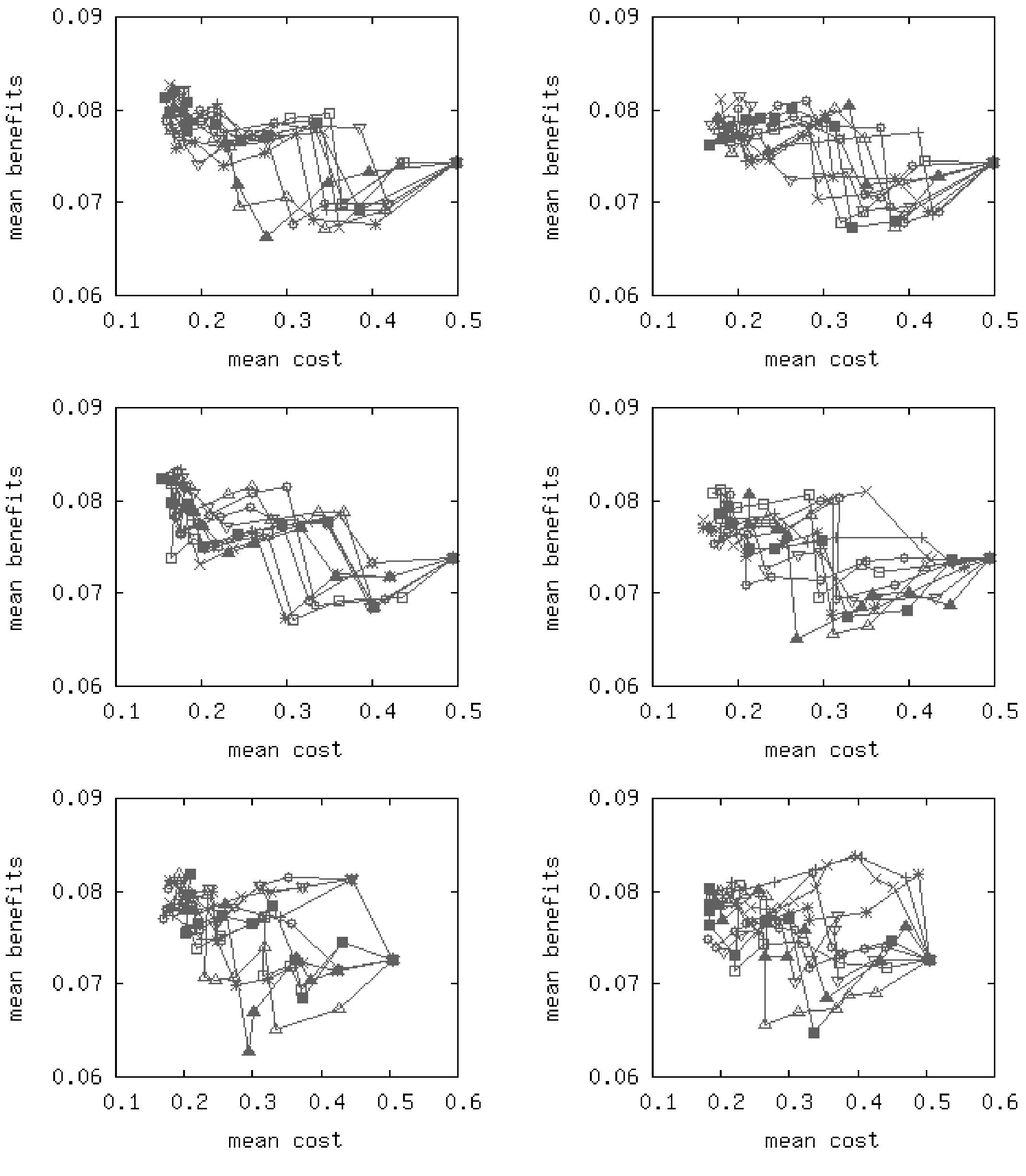


Figure A.12: Cost vs. Benefit in the *holo* model with the *wob* selector

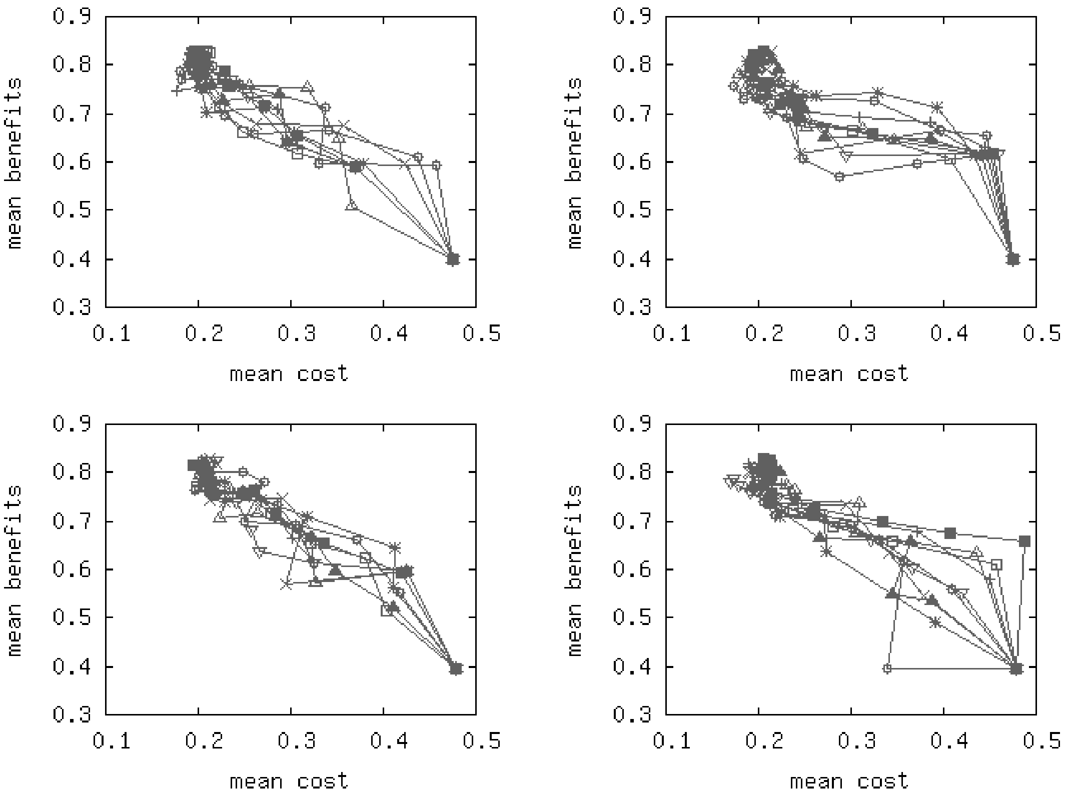


Figure A.13: Cost vs. Benefit in the *cob* model with the *bore* selector

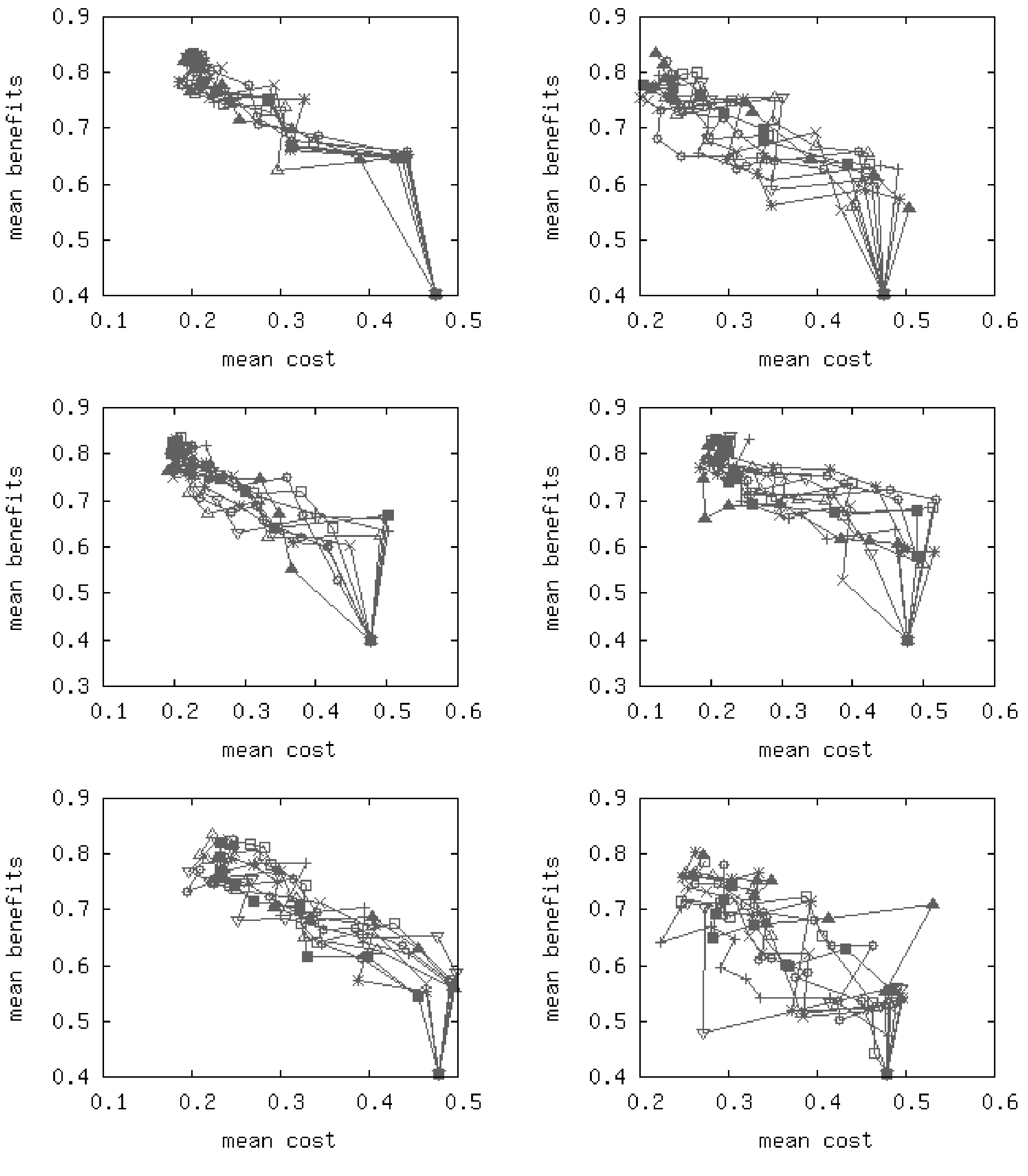


Figure A.14: Cost vs. Benefit in the *cob* model with the *bore*′ selector

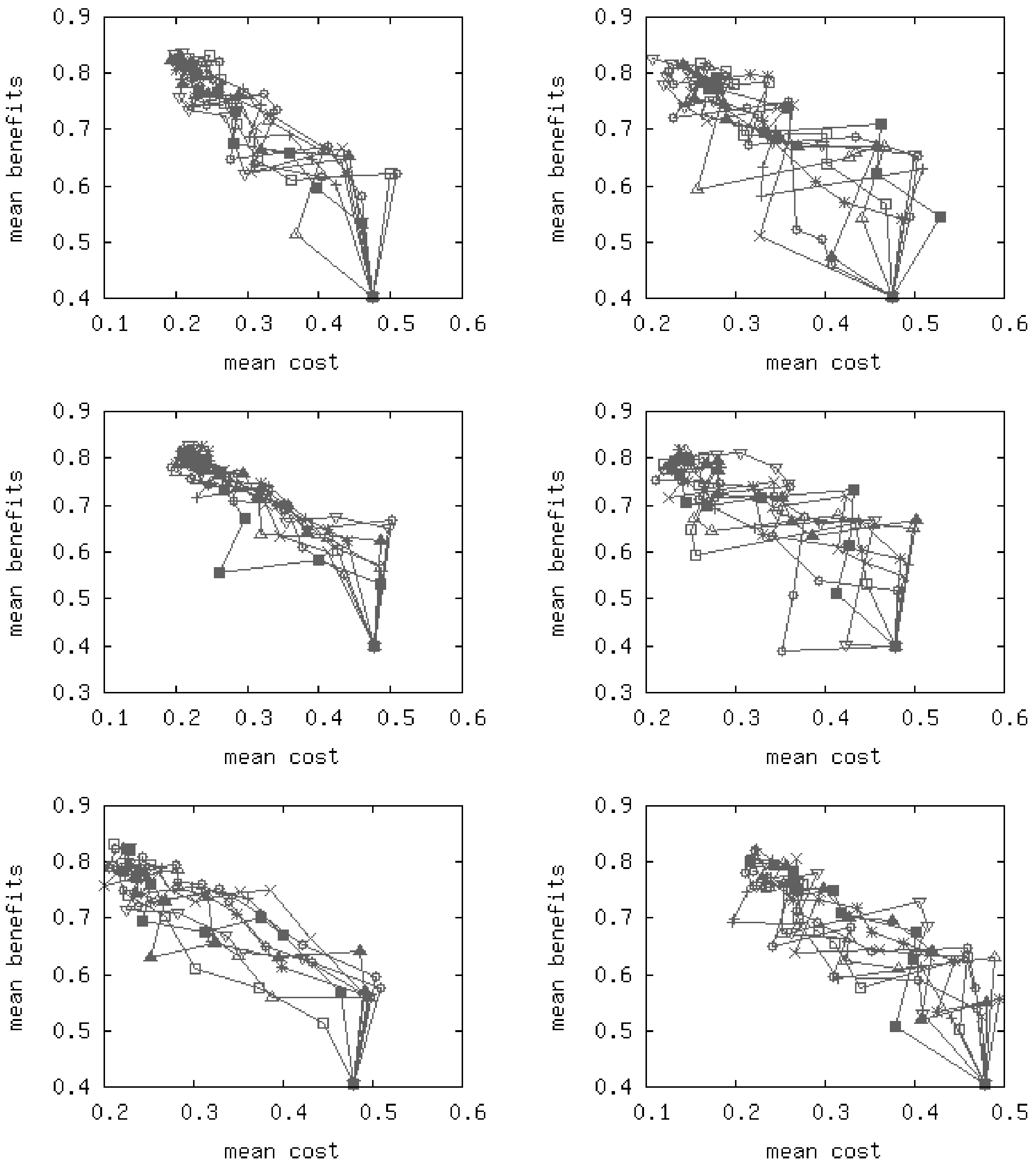


Figure A.15: Cost vs. Benefit in the *cob* model with the *wob* selector

A.3 Variance of Partial Descriptions

§4.4 did not display the varience results from all selectors and all M/N combinations. This section displays the standard deviations for all 720 trials described in §4.2.

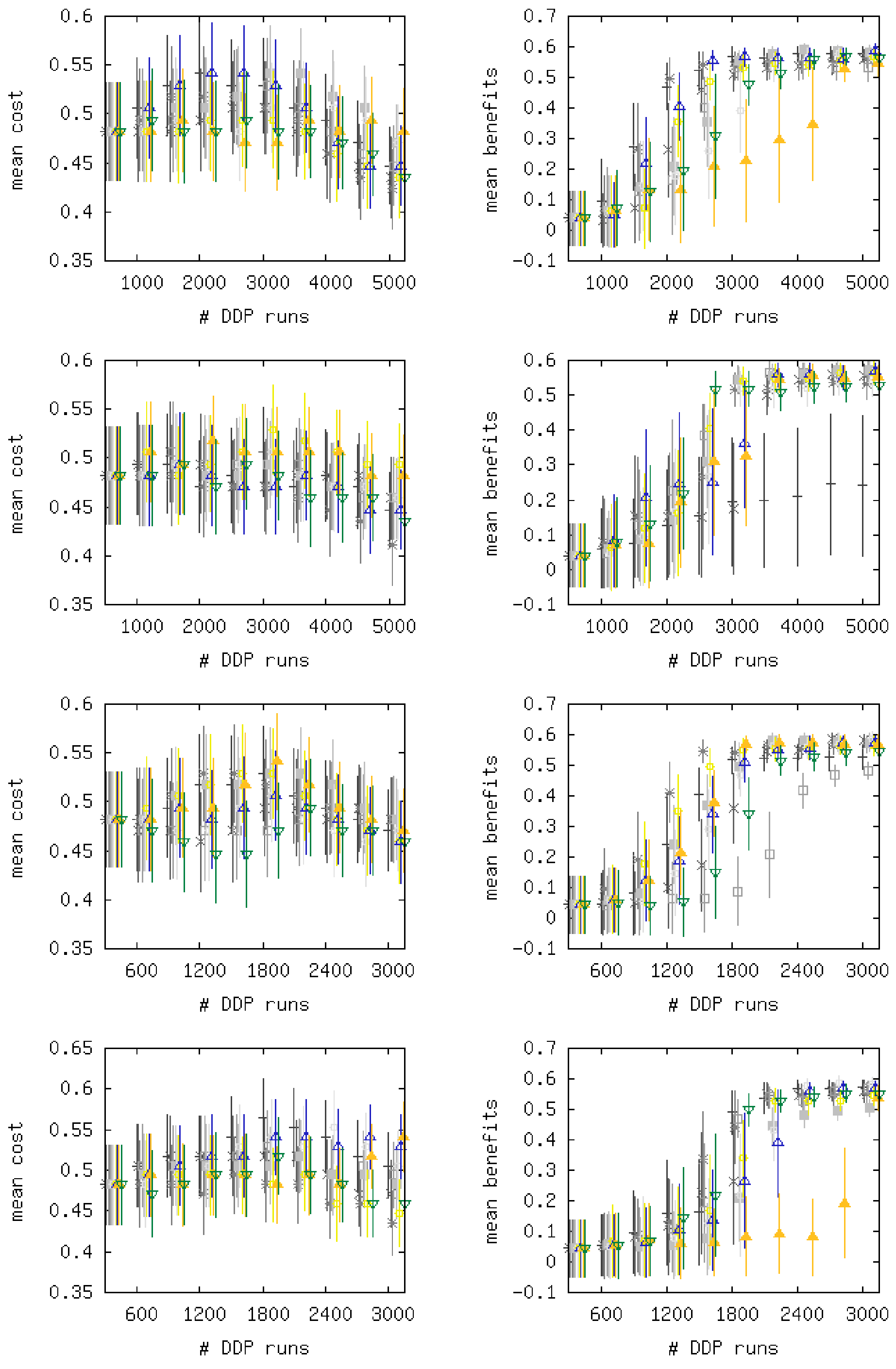


Figure A.16: Cost and Benefit in the *aero* model with the *bore* selector

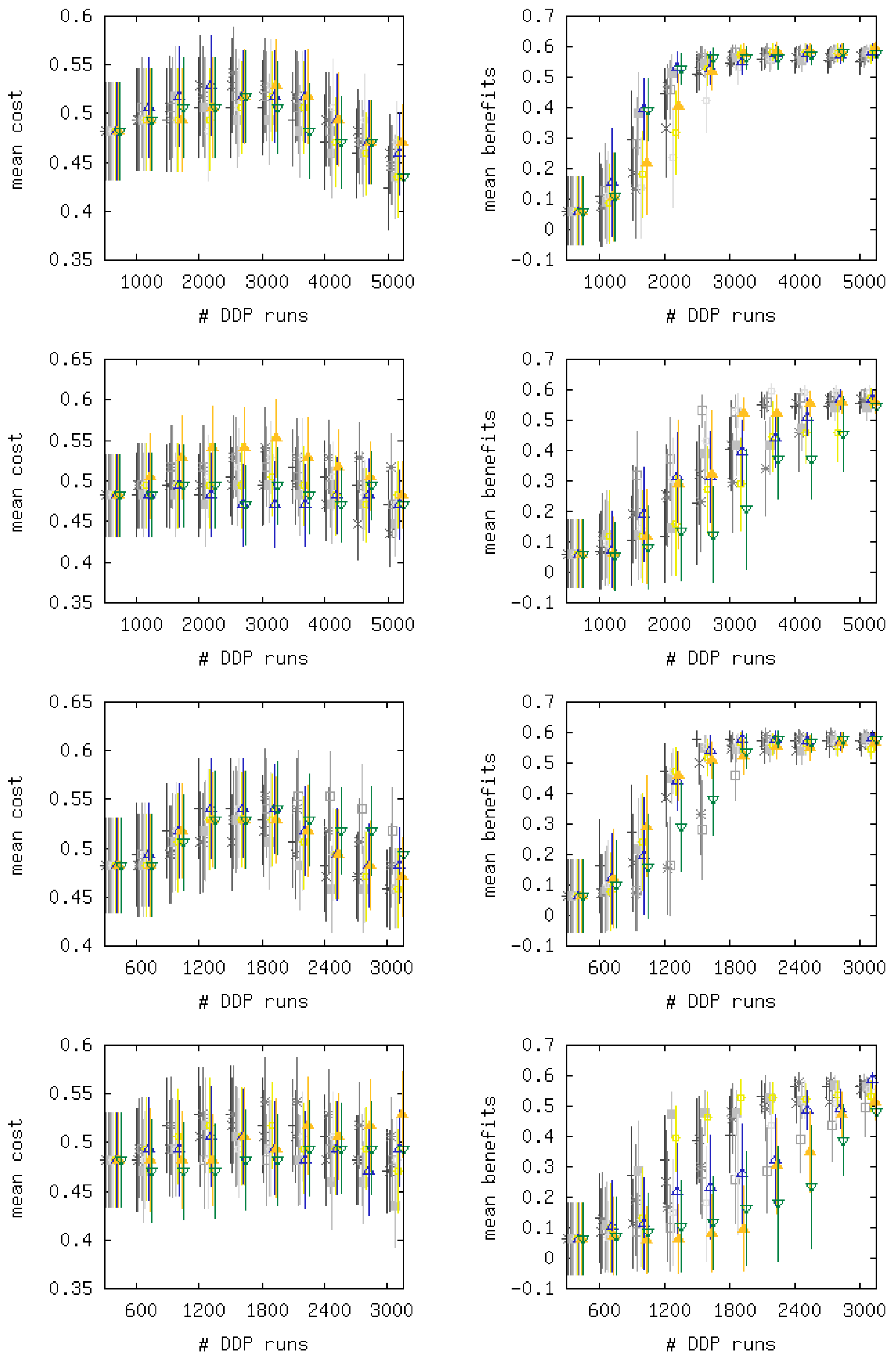


Figure A.17: Cost and Benefit in the *aero* model with the *bore′* selector

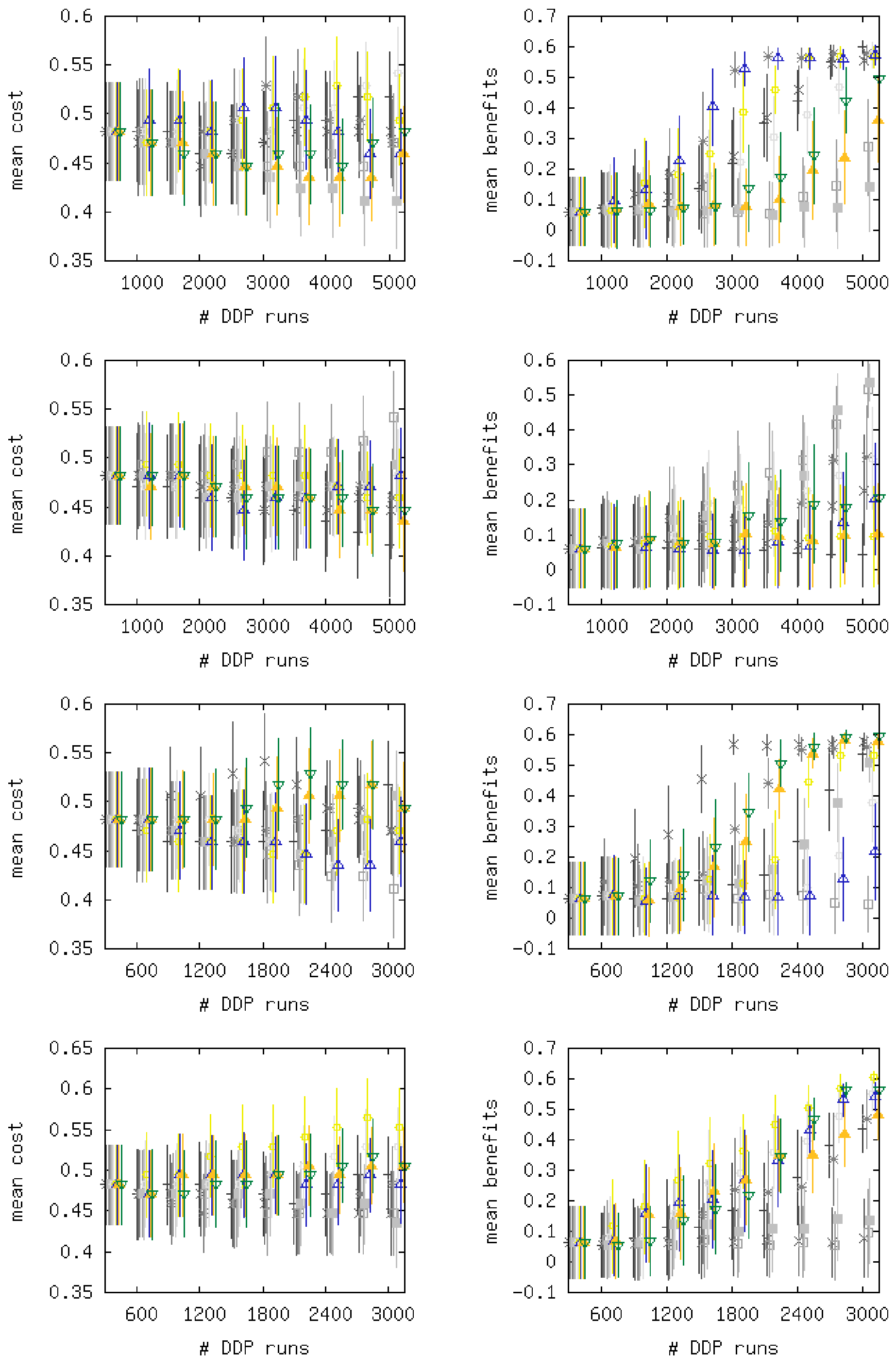


Figure A.18: Cost and Benefit in the *aero* model with the *wob* selector

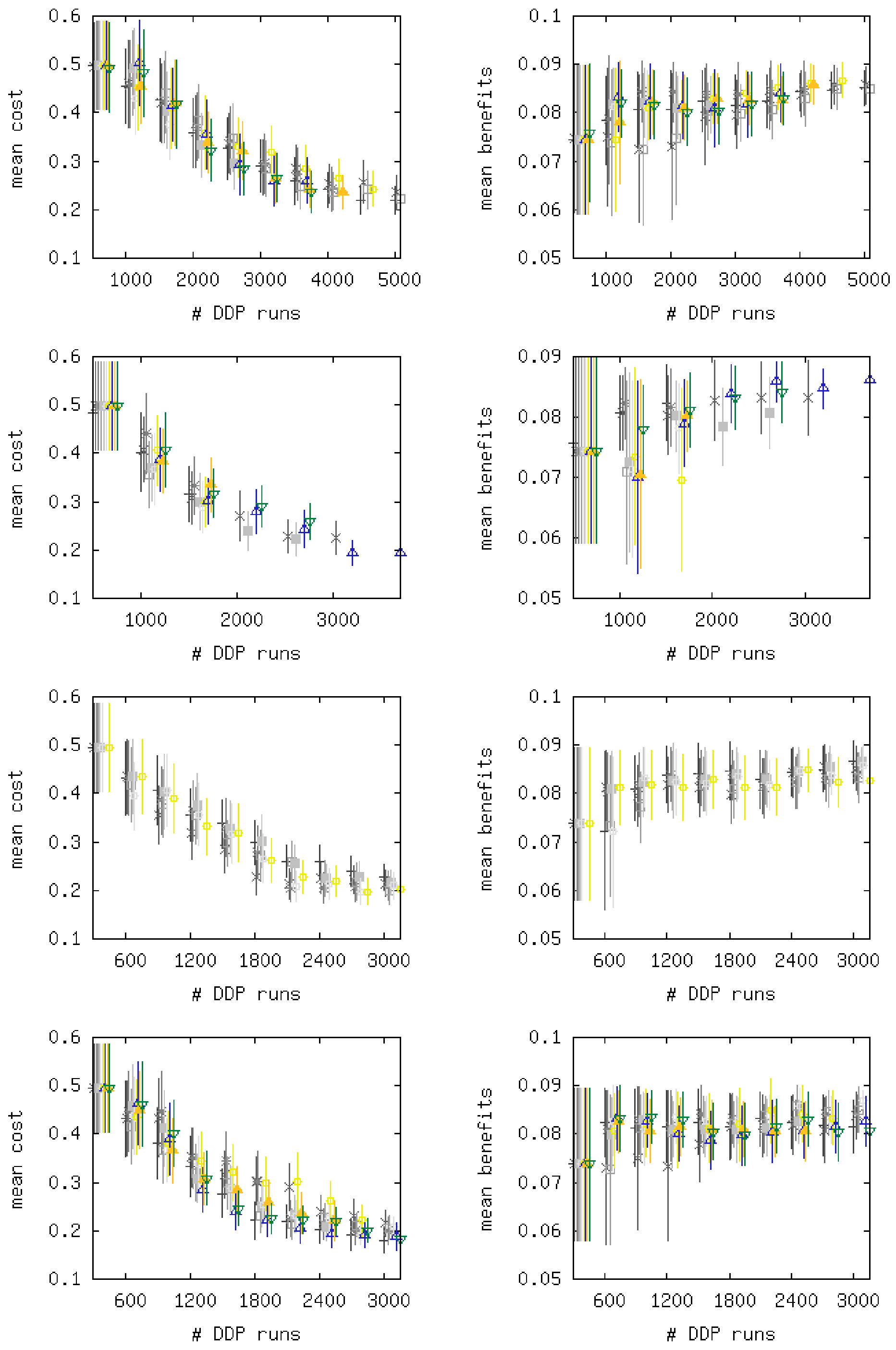


Figure A.19: Cost and Benefit in the *holo* model with the *bore* selector

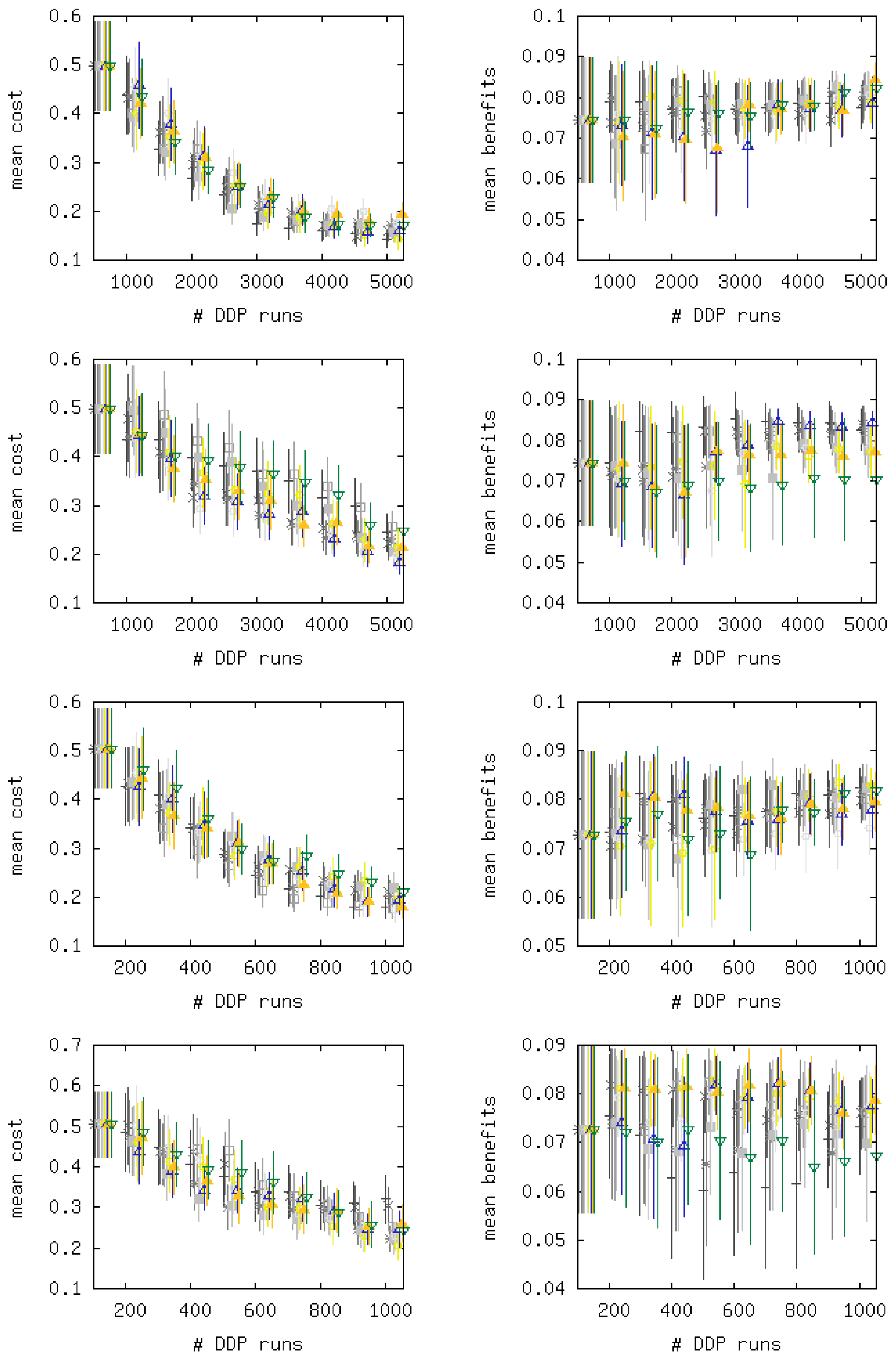


Figure A.20: Cost and Benefit in the *holo* model with the *bore*′ selector

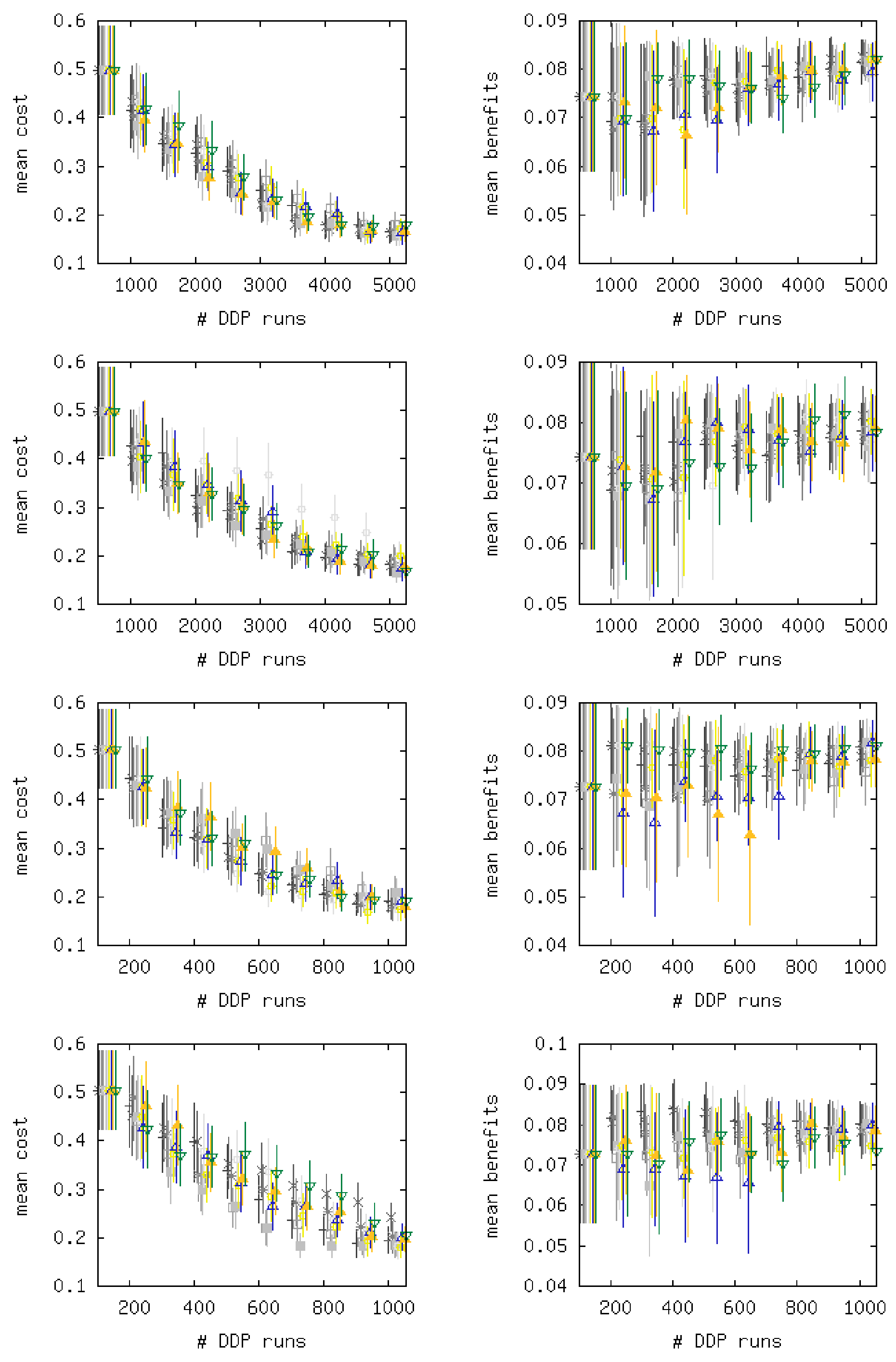


Figure A.21: Cost and Benefit in the *holo* model with the *wob* selector

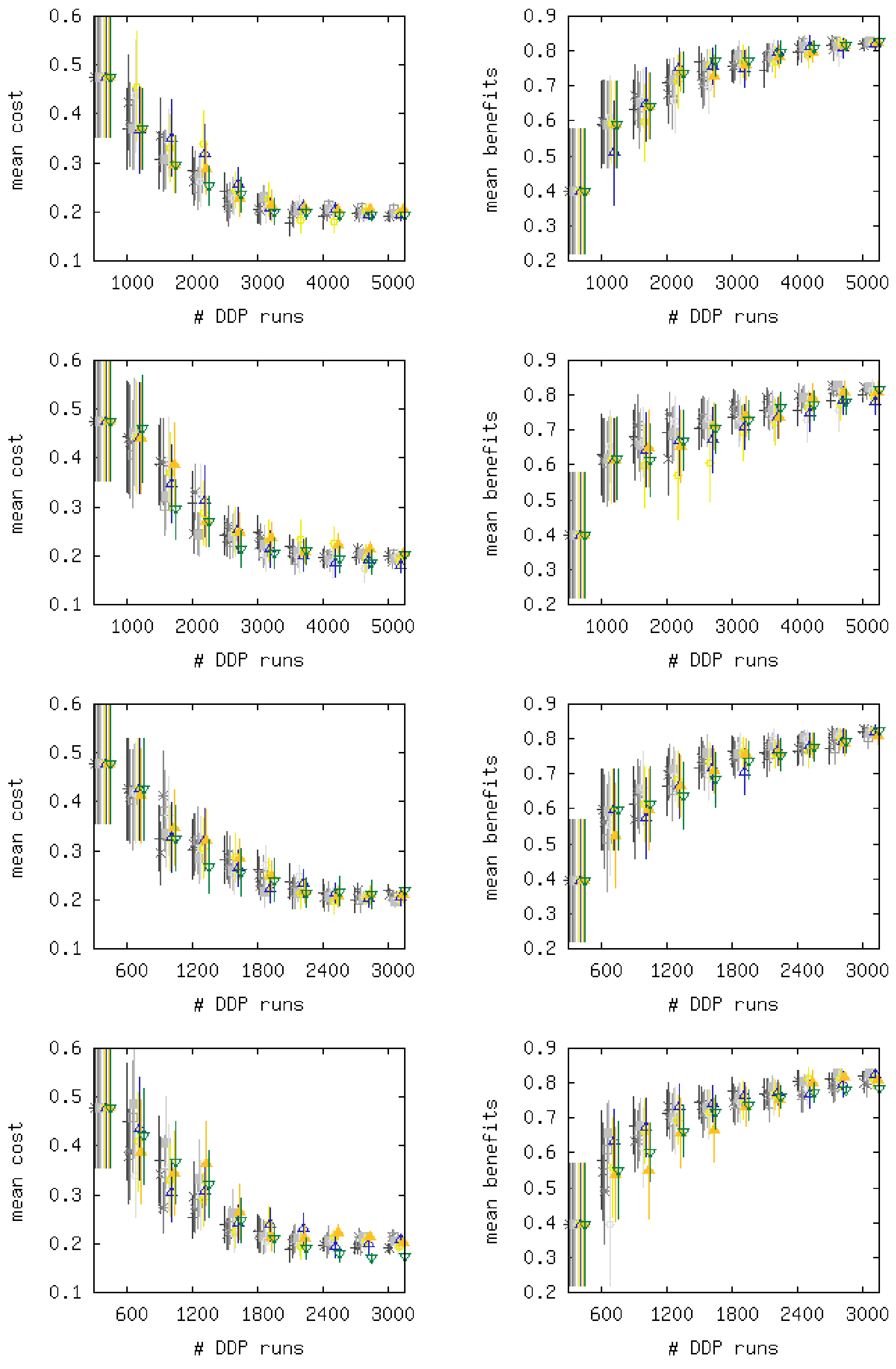


Figure A.22: Cost and Benefit in the *cob* model with the *bore* selector

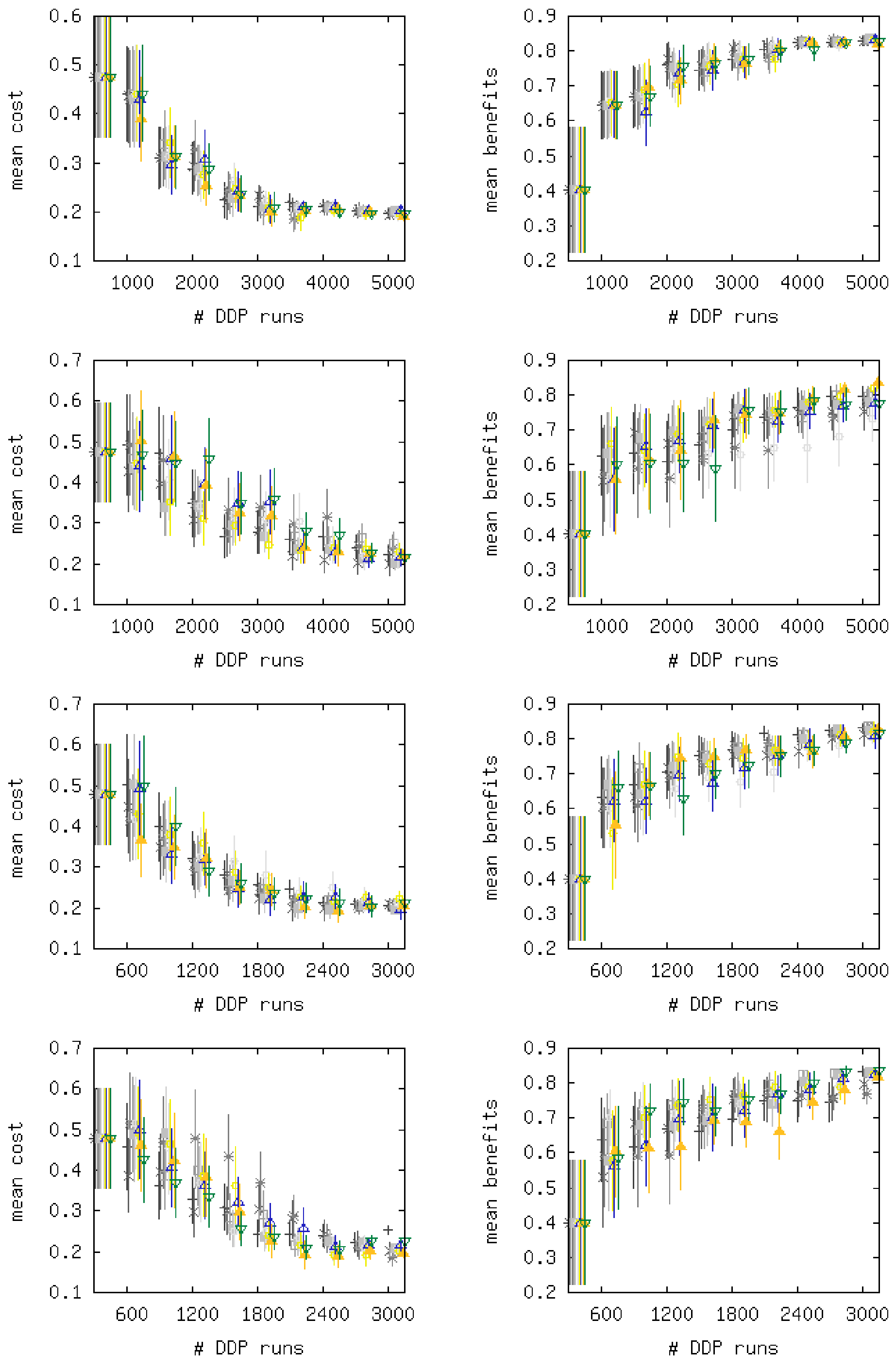


Figure A.23: Cost and Benefit in the *cob* model with the *bore*′ selector

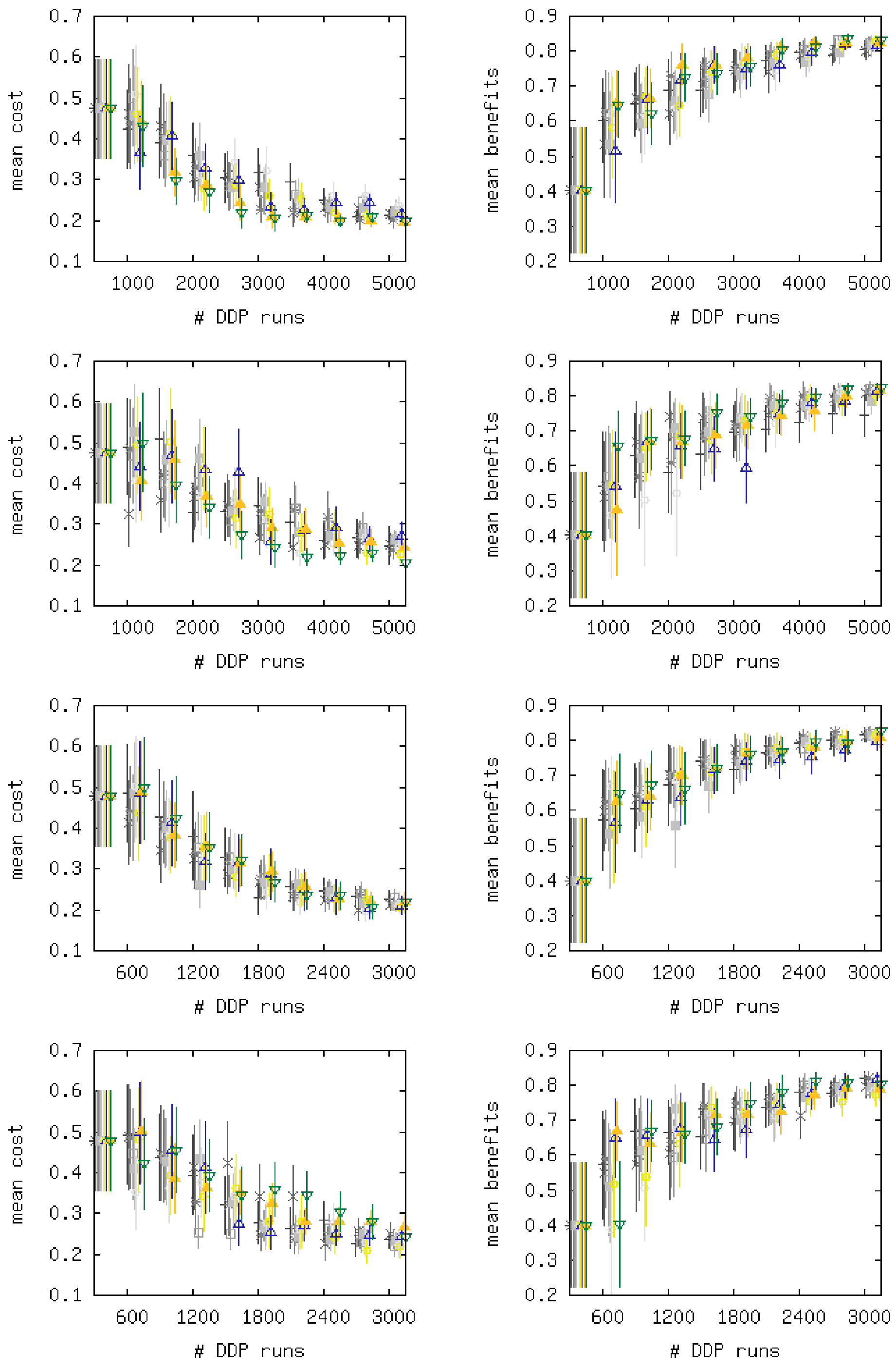


Figure A.24: Cost and Benefit in the *cob* model with the *wob* selector