The Robust Optimization of Non-Linear Requirements Models

Gregory Gay

Thesis submitted to the
College of Engineering and Mineral Resources
at West Virginia University
in partial fulfillment of the requirements
for the degree of

Master of Science in Computer Science

Tim Menzies, Ph.D., Chair
Bojan Cukic, Ph.D.
Katerina Goseva-Popstojanova, Ph.D.

Lane Department of Computer Science and Electrical Engineering
West Virginia University
Morgantown, West Virginia
April 7, 2010

Keywords: Search-Based Software Engineering, Treatment Learning,
Minimal Contrast-Set Learning, Model-Based Software Engineering,
Meta-Heuristic Search

Abstract

The Robust Optimization of Non-Linear Requirements Models

Gregory Gay

Solutions to non-linear requirements engineering problems may be "brittle"; i.e. small changes may dramatically alter solution effectiveness. Hence, it is not enough to just generate solutions to requirements problems- we must also assess solution robustness. This thesis aims to address two concerns: (a) Is demonstrating robustness a time consuming task? and (b) Is it necessary that solution quality be traded off against solution robustness?

Using a Bayesian ranking heuristic, the KEYS2 algorithm fixes a small number of important variables, rapidly pushing the search into a stable, optimal plateau. By design, KEYS2 generates decision ordering diagrams (in time experimentally shown to be $O(N^2)$). Once generated, these diagrams can confirm solution robustness in linear time. When assessed in terms of reducing inference times, increasing solution quality, and decreasing the variance of the generated solution, KEYS2 out-performs other search algorithms (simulated annealing, A*, MaxWalkSat).

Dedication

I dedicate this work to Alan Turing , Mark Harman, Ross Quinlan, Patrick Winston, Edward Feigenbaum, Alan Newell, Barry Boehm, and all of those that established the fields of Artificial Intelligence, Machine Learning, and Search-Based Software Engineering. You are giants, and it is only by standing on your shoulders that I may feel the wind.

Acknowledgments

There are a number of people that were pivitol to the creation of this thesis, and I would like to take this moment to acknowledge them. I'll start by thanking my parents for both their unwavering support and for letting me make my own mistakes (and, consequently, my own triumphs). I would also like to thank Dr. Menzies for having faith that there would be method in my madness. I could not have asked for a better advisor, and I owe him a debt for the opportunities that he has made available to me.

Of course, this type of research is never done in a vaccuum, and I would like to thank and acknowledge Omid Jalali for the creation of the KEYS algorithm. I would also like to thank Adam Nelson, Brian Sowers, Zach Milton, and everybody else at the Modeling Intelligence Lab for being both great coworkers and great friends.

Last, I would like to acknowledge my friends for their neverending work in keeping me sane. I thank Michelle Hunt for being the most patient person on this planet, and for listening to me complain all of the time. I would also like to thank Ricky Hussmann, Andrew Matheny, Wesley Corbin and the rest of the crew for all of those great Thursday nights.

This research was conducted at West Virginia University and the Jet Propulsion Laboratory under a contract with the National Aeronautics and Space Administration. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government

Contents

List of Figures

Symbols and Abbreviations

ACO: Ant Colony Optimization
AI: Artificial Intelligence
BDD: Binary Decision Diagram
DDP: Defect Detection and Prevention Model
DOD: Decision Ordering Diagram
DM: Data Mining
GA: Genetic Algorithm
HC: Hill Climbing
JPL: NASA Jet Propulsion Laboratory
LS: Local Search
MWS: MaxWalkSat
OO: Object-Oriented Programmin
PSO: Particle Swarm Optimization
QOC: Questions-Options-Criteria Graph
RS: Random Search
SA: Simulated Annealing
SBSE: Search-Based Software Engineering
SE: Software Engineering
TS: Tabu Search
UML: Unified Modelling Language
VNS: Variable-Neighborhood Search
WCET: Worst-Case Execution Time

Chapter 1

Introduction

Consider a "requirements model" where stakeholders write:

- Their various goals for a project;
- All of the possible methods for reaching those goals;
- Their view of the possible risks that could compromise those goals;
- What mitigations they believe might reduce those risks.

A "solution" to such models is an ideal balance between the *least* cost set of mitigations that reduce the *most* risks, thereby enabling the attainment of the *most* requirements. In theory, an algorithm could be used to find the solution that best satisfies the various goals of the different stakeholders whle ensuring that the project remain within the bounds of its budget. Such tools might even find potential solutions that were missed by these stakeholders. Finding solutions to these requirements models is a non-linear optimization problem - a solution must *minimize* the sum of the mitigation costs while *maximizing* the number of achieved requirements.

According to Mark Harman [51], understanding the neighborhood of solutions is an open and pressing issue in search-based software engineering (SBSE). He argues that many software engi-

neering problems are over-constrained and, thus, no precise solution over all variables is achievable; therefore, partial solutions based on meta-heuristic search methods are preferred.

> "In some software engineering applications, solution robustness may be as important as solution functionality. For example, it may be better to locate an area of the search space that is rich in fit solutions, rather than identifying an even better solution that is surrounded by a set of far less fit solutions."

> "Hitherto, research on SBSE has tended to focus on the production of the fittest possible results. However, many application areas require solutions in a search space that may be subject to change. This makes robustness a natural second order property to which the research community could and should turn its attention [51]."

The *robustness* of solutions is a major problem for such partial heuristic search methods. There are many heuristic methods that can generate solutions to non-linear problems (see the *Related Work* chapter). Such methods can be *brittle* [50]. That is, small changes may dramatically alter the effectiveness of the generated solution. Therefore, when offering partial solutions, it is very important to also offer insight into the space of options around the proposed solution. Such neighborhood information is very useful for managers with only partial control over their projects, as it can give them confidence that, even if only some of their recommendations are enacted, then at least the range of outcomes is well understood.

A naive approach to understanding the neighborhood might be to run a system N times, then report:

- the solutions appearing in more than, for example, $\frac{N}{2}$ cases;
- Results with a $\pm 95\%$ confidence interval.

Note that both of these approaches require multiple trials of the chosen analysis method. Multiple executions are undesirable since, as experience shows [34], stakeholders often ask questions across

a large range of "scenarios" with hard-wired constraints that cannot be changed in that scenario. These scenarios might be as simple as what could be achieved assuming a maximum budget of one billion dollars? Two billion? Five billion?

Scenario analysis can become a time consuming task. Reflecting over, say, $d = 10$ possible decisions over a statistically significant number of repeats ($N = 20$) requires up to $20 * 2^{10} > 20,000$ repeats of the analysis. In order to provide useful exploration of this search space - to avoid blind spots, in other words - optimization techniques must run fast enough that humans can get feedback before they must move on to other issues. Neilson [85] reports that "the basic advice regarding response times has been about the same for thirty years; i.e.

- One second is about the limit for the user's flow of thought to stay uninterrupted, even though the user will notice the delay
- Ten seconds is the limit for keeping a user's attention focused on the dialogue."

This thesis reports a set of experiments on the search for robust solutions to a NASA requirements model called the Defect Detection and Prevention model. These experiments address two important concerns. Firstly, *is demonstrating solution robustness a time consuming task*? Secondly, is it necessary, as Harman suggests, that *solution quality must be traded off against solution robustness*? That is, in the search for the conclusions that were stable within their local neighborhood, would we have to reject better solutions because they are not stable across the same neighborhood?

At least for the NASA models used in these experiments, both of the concerns are unfounded. The KEYS2 algorithm, presented in Chapter 3, terminates in hundredths of a second (whereas prior candidate algorithms took minutes to terminate [36]). Also, the solutions found by KEYS2 were not only of highest quality of the methods benchmarked, they also exhibited the lowest variance. Further, KEYS2 generates decision ordering diagrams that can be used assess solution robustness in linear time (*decision ordering diagrams* are a visual representation of the effects of changing a

solution).

In this thesis, I demonstrate that:

- Using these diagrams, the region around a solution can be explored in linear time.
- A greedy Bayesian-based method called KEYS2 can generate the decision ordering diagrams in $O(N^2)$ time.
- KEYS2 yields solutions of higher quality that several other methods (Simulated Annealing, MaxWalkSat, ASTAR).
- Also, the variance of the solutions found by KEYS2 is less (and hence, better) than those found by the other methods.

1.1 Statement of Thesis

The KEYS2 algorithm is a robust method for the optimization of requirements models. When assessed in terms of (a) reducing inference times, (b) increasing solution quality, and (c) decreasing the variance of the generated solution, KEYS2 outperforms both standard and state-of-the-art search algorithms (including simulated annealing, A*, and MaxWalkSat).

1.2 Contributions of this Thesis

This thesis contributes a variety of findings to the literature, including:

- A new compiled form of the Defect Detection and Prevention model, a design model used by NASA's Jet Propulsion Lab.
- Implementations of standard algorithms - Simulated Annealing, A* search, and MaxWalkSat - designed to optimize settings to such models.

- A novel Bayesian learning mechanism, KEYS, that can generate robust requirements decisions in $O(N^2)$ time.
- An improved version of KEYS that generates model solutions in a shorter length of time and with less variance than other standard methods.
- A visual format - decision ordering diagrams - that models the cost and benefit effect of making certain *key* decisions.

1.3 Publications from this Thesis

- Gay, Gregory and Menzies, Tim and Jalali, Omid and Mundy, Gregory and Gilkerson, Beau and Feather, Martin and Kiper, James. Finding robust solutions in requirements models. Automated Software Engineering, 17(1): 87-116, 2010.

This paper is the first publication of the KEYS2 algorithm. It presents benchmarking experiments comparing KEYS2 against state-of-the-art search algorithms.

1.4 Structure of this Document

The remainder of this thesis is organized as follows:

- Chapter 2 presents some background material, including information on the DDP model format, an objective function for scoring model configurations, decision ordering diagrams, and a generator for building artificial models.
- Chapter 3 is a literature review of the search-based software engineering field.
- Chapter 4 describes the KEYS and KEYS2 algorithms, as well as a series of standard algorithms that are benchmarked against it.

- Chapter 5 presents a series of case studies when KEYS is compared to established algorithms. An experiment demonstrates that KEYS2 scales well to larger models. A series of experiments are also presented where attempts are made to improve the speed of KEYS.
- Chapter 6 concludes this thesis by summarizing the contributions of this algorithm as well as proposing possible future work.

Chapter 2

Background

The previous chapter has introduced the problem that this research will attempt to solve. This chapter will provide background material on the model format used in these experiments, as well as the decision ordering diagrams used for the interpretation of the search results.

2.1 The Defect Detection and Prevention Model

The Defect Detection and Prevention (DDP) requirements modeling tool [23, 34]. is used to interactively document the early life-cycle meetings conducted by ”Team X” at NASA's Jet Propulsion Laboratory (JPL). These meetings are the source of the real-world requirements models used in this thesis.

At Team X meetings, a large and diverse group of up to 30 experts from various fields (propulsion, engineering, communication, navigation, science, etc) meet for short periods of time (usually for no more than four or five days) to produce a “mission concept” document. This document may commit the current project to, for example, solar power rather than nuclear power or to some particular style of guidance software. All subsequent work on the project is guided by the initial design decisions contained in these mission concept documents.

1. Requirement goals:
    - Spacecraft ground-based testing & flight problem monitoring
    - Spacecraft experiments with on-board Intelligent Systems Health Management (ISHM)
2. Risks:
    - Obstacles to spacecraft ground-based testing & flight problem monitoring
        - Customer has no, or insufficient, money available for my use
        - Difficulty of building the models / design tools
    - ISHM Experiment is a failure (without necessarily causing flight failure)
    - Usability, User/Recipient-system interfaces undefined
    - V&V (certification path) untried and scope unknown
    - Obstacles to Spacecraft experiments with on-board ISHM
        - Bug tracking / fixes / configuration management issues, Managing revisions and upgrades (multi-center tech. development issue)
        - Concern about my technology interfering with in-flight mission
3. Mitigations:
    - Mission-specific actions
        - Spacecraft ground-based testing & flight problem monitoring
        - Become a team member on the operations team
        - Use Bugzilla and CVS
    - Spacecraft experiments with on-board ISHM
        - Become a team member on the operations team
        - Utilize xyz's experience and guidance with certification of his technology

Figure 2.1: Sample DDP requirements, risks, mitigations.

The DDP model allows for the representation of the goals, risks, and risk-removing mitigations that belong to a specific project. During a Team X meeting, users of DDP explore combinations of mitigations that will cost the least amount while still supporting the largest number of requirements. For example, here is a trivial DDP model where `mitigation1` costs $10,000 to apply and each requirement is of equal value (100). Note that the mitigation can remove 90% of the risk. Also,

unless mitigated, the risk will disable 10% to 99% of requirements one and two (respectively):

$$\overbrace{mitigation1}^{\$10{,}000} \underbrace{\rightarrow}_{0.9} risk1 \rightarrow \left\langle \begin{array}{l} \overbrace{\rightarrow}^{0.1}(requirement1 = 100) \\ \underbrace{\rightarrow}_{0.99}(requirement2 = 100) \end{array} \right. \tag{2.1}$$

The other numbers show the impact of mitigations on risks, and the impact of risks on requirements. DDP propagates a series of influences over two matrices: one for *mitigations*risks* and another for *risks*requirements*.

The DDP tool supports a graphical interface for the rapid entry of the assertions. Such rapid entry is essential, these experts are busy and it is hard to gather them together for any long period of time and, as such, no tool should slow the debate. Therefore, DDP uses a lightweight representations for its model. Such representations are essential for early life-cycle decision making since only high-level assertions can be collected in such short knowledge acquisition sessions (if the assertions get more elaborate, then experts may waste time trying to understand technical arguments from outside of their own field of expertise). DDP uses the following ontology:

- *Requirements* (free text) describe the objectives and constraints of the mission and its development process;
- *Weights* (numbers) of each requirement, reflecting their relative importance;
- *Risks* (free text) are events that damage the completion of requirements;
- *Mitigations*: (free text) are actions that reduce risks;
- *Costs*: (numbers) reflect the cost associated with activating a mitigation;
- *Mappings*: directed, weighted edges between requirements, mitigations, and risks that capture the quantitative relationships among them.
- *Part-of relations*: structure the collections of requirements, risks and mitigations;

Note that DDP models are a form of the "requirements models" that were defined in Chapter 1. For examples of risks, requirements, and mitigations, see Figure 2.1. For an example of the network of connections between risks and requirements and mitigations, see Figure 2.2.

It can be asked whether the analysis of DDP requirements models is a real problem. With these ultra-lightweight languages, aren't all open issues rendered obvious? Such a question is typically informed by the small model fragments that appear in the ultra-lightweight modeling literature. Those sample model fragments are typically selected according to their ability to fit on a page or to succinctly illustrate some point of the authors. Real world ultra-lightweight models can be much more complex, paradoxically perhaps due to their simplicity: if a model is easy to write then it is just as easy to write a lot of it. For example, the model seen in Figure 2.2 was generated in under a week by four people discussing one project. It is complex and densely-connected (a close inspection of the left and right hand sides of Figure 2.2 reveals the requirements and fault trees that inter-connect concepts in this model) and it is, by no means, the biggest or most complex DDP model that has ever been built.

This research has been structured around DDP for three reasons. Firstly, one potential drawback with ultra-lightweight models is that they are excessively lightweight and contain no useful information. DDP's models are demonstrably useful, and clear project improvements have been seen from DDP sessions at JPL. Cost savings of $100,000 have been seen in multiple sessions, and in at least two sessions, they have exceeded $1 million [34]. Cost savings are not the only benefits of these DDP sessions, numerous design improvements (such as savings of power or mass) have emerged as well. Likewise, the spectrum of risks has shifted from uncertain architectural ones to more predictable and manageable ones. At some of these meetings, non-obvious (yet significant) risks have been identified and subsequently mitigated.

The second reason for using DDP is that numerous real-world requirements models have written in this format, and many projects are likely to use these models in the future. The DDP tool can be used to document not just final decisions, but also to review the rationale that led to those de-

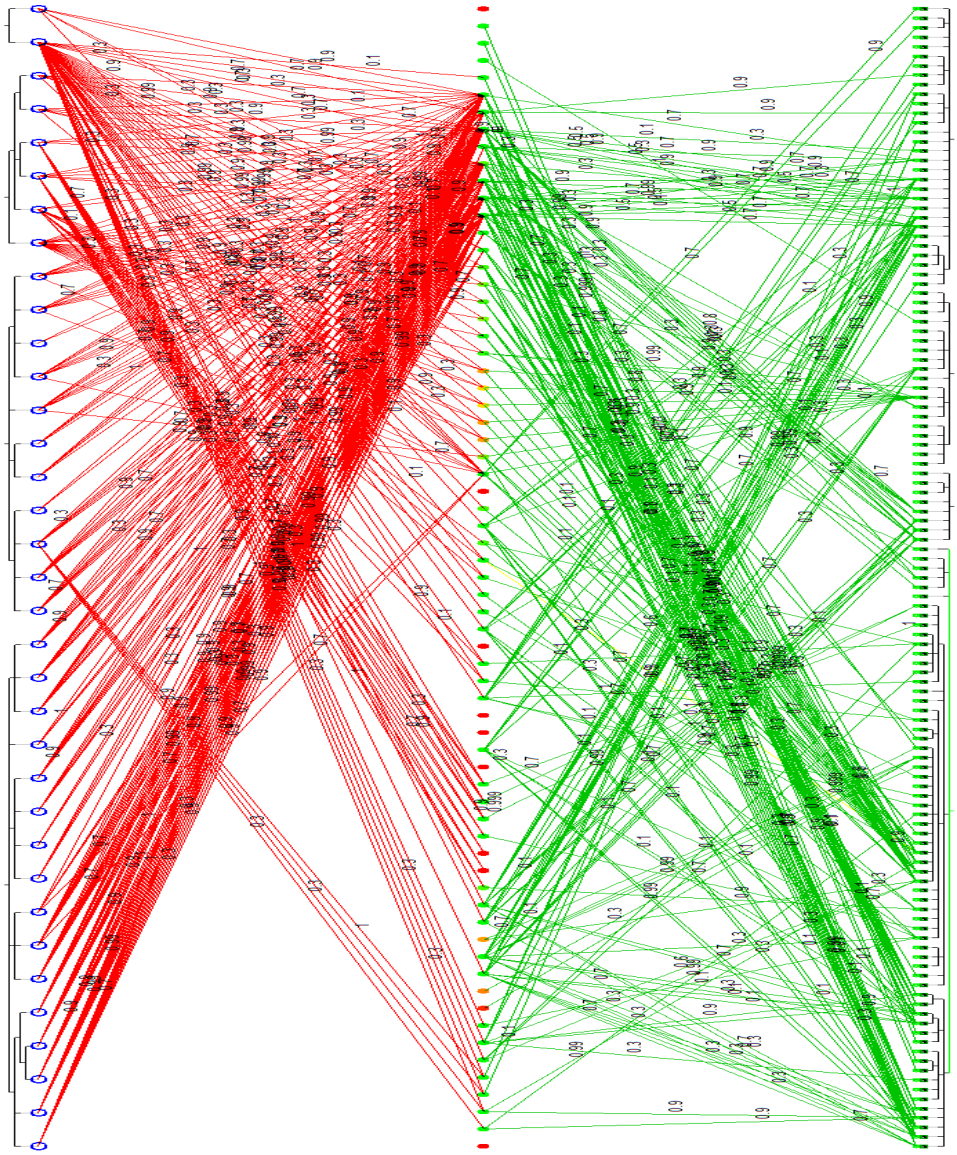

Figure 2.2: An example of a model formed by the DDP tool. Red lines connect risks (middle) to requirements (left). Green lines connect mitigations (right) to the risks.

cisions. Hence, it remains in use at JPL not only for its original purpose (group decision support), but also as a design rationale tool to document decisions. Recent DDP sessions included:

- An identification of the challenges of intelligent systems health management (ISHM) technology maturation (to determine the most cost-effective approach to achieving maturation) [37];
- A study on the selection and planning of deployment of prototype software [35].

The third, and most important, reason to use DDP in this research is that the tool is representative of other requirements modeling tools in widespread use. At its core, DDP is a set of influences expressed in a hierarchy, augmented with the occasional equation. Edges in this hierarchy have weights that strengthen or weaken the influences that flow along those edges. At this level of abstraction, DDP is just another form of QOC [101] or a quantitative variant of Mylopoulos' qualitative soft goal graphs [82].

2.2 Model Inputs and Outputs

2.2.1 Pre-Processing

To enable fast runtimes, a simple compiler exports the DDP models into a form easily accessible by optimization algorithms. This compiler stores a flattened form of the DDP requirements tree in a function usable by any program in the C language. In the compiled form, all computations are performed once and added as a constant to each reference of the requirement. For example, the compiler converts the trivial model of Equation 2.1 into `setupModel` and `model` functions similar to those in Figure 2.3. The `setupModel` function is called only once (before an algorithm performs any configuration of the mitigation settings) and sets several constant values. The `model` function is called whenever cost and attainment values are needed for a particular configuration. The topology of the mitigation network is represented as terms in equations within these functions. As the models grow more complex, so do these equations. For example, the biggest real-world

```
#include "model.h"

#define M_COUNT 2
#define O_COUNT 3
#define R_COUNT 2

struct ddpStruct
{
    float oWeight[O_COUNT+1];
    float oAttainment[O_COUNT+1];
    float oAtRiskProp[O_COUNT+1];
        float rAPL[R_COUNT+1];
    float rLikelihood[R_COUNT+1];
    float mCost[M_COUNT+1];
    float roImpact[R_COUNT+1][O_COUNT+1];
    float mrEffect[M_COUNT+1][R_COUNT+1];
};

ddpStruct *ddpData;

void setupModel(void)
{
    ddpData = (ddpStruct *) malloc(sizeof(ddpStruct));
    ddpData->mCost[1]=11;
    ddpData->mCost[2]=22;
    ddpData->rAPL[1]=1;
    ddpData->rAPL[2]=1;
    ddpData->oWeight[1]=1;
    ddpData->oWeight[2]=2;
    ddpData->oWeight[3]=3;
    ddpData->roImpact[1][1] = 0.1;
    ddpData->roImpact[1][2] = 0.3;
    ddpData->roImpact[2][1] = 0.2;
    ddpData->mrEffect[1][1] = 0.9;
    ddpData->mrEffect[1][2] = 0.3;
    ddpData->mrEffect[2][1] = 0.4;
}

void model(float *cost, float *att, float m[])
{
    float costTotal, attTotal;
    ddpData->rLikelihood[1] = ddpData->rAPL[1] * (1 - m[1] * ddpData->mrEffect[1][1])
         * (1 - m[2] * ddpData->mrEffect[2][1]);
    ddpData->rLikelihood[2] = ddpData->rAPL[2] * (1 - m[1] * ddpData->mrEffect[1][2]);
    ddpData->oAtRiskProp[1] = (ddpData->rLikelihood[1] * ddpData->roImpact[1][1])
         + (ddpData->rLikelihood[2] * ddpData->roImpact[2][1]);
    ddpData->oAtRiskProp[2] = (ddpData->rLikelihood[1] * ddpData->roImpact[1][2]);
    ddpData->oAtRiskProp[3] = 0;
    ddpData->oAttainment[1] = ddpData->oWeight[1] * (1 - minValue(1, ddpData->oAtRiskProp[1]));
    ddpData->oAttainment[2] = ddpData->oWeight[2] * (1 - minValue(1, ddpData->oAtRiskProp[2]));
    ddpData->oAttainment[3] = ddpData->oWeight[3] * (1 - minValue(1, ddpData->oAtRiskProp[3]));
    attTotal = ddpData->oAttainment[1] + ddpData->oAttainment[2] + ddpData->oAttainment[3];
    costTotal = m[1] * ddpData->mCost[1] + m[2] * ddpData->mCost[2];

    *cost = costTotal;
    *att = attTotal;
}
```

Figure 2.3: A trivial DDP model after knowledge compilation

Model	LOC	Objectives	Risks	Mitigations
model1.c	55	3	2	2
model2.c	272	1	30	31
model3.c	72	3	2	3
model4.c	1241	50	31	58
model5.c	1427	32	70	99

Figure 2.4: Details of Five DDP Models.

model used in this research, which contains 99 mitigations, generates 1427 lines of code. Figure 2.4 compares this large model to four other real-world DDP models. Without any form of compilation, any DDP assessment algorithm would have to read a separate file and import it into a usable format every time that it needed new cost and attainment values. This one-time compilation removes that pressure, leading to exponential improvements in execution time (earlier DDP algorithms would take thirty minutes or more to run).

Currently, it takes about two seconds to compile a model with 50 requirements, 31 risks, and 58 mitigations. This compilation only has to happen once, after which an algorithm can run any number of what-if scenarios. While this is not a significant bottleneck, the current compiler (written in unoptimized Visual Basic code) can certainly be sped up. Experts usually change a small portion of the model before running up to $2^{|d|}$ what-if scenarios to understand the impact of that change. Therefore, an incremental compiler (that only updates changed portions) would run much faster than a full compilation of the entire DDP model.

2.2.2 Artificial Model Generation

Although the core experiments of this thesis are based on a series of five real-world models, it is always desirable to have more data to work with. Unfortuantely, large repositories of real-world models are not always available, and they do not always meet the criteria (number of mitigations, complexity of the interconnections, etc) of a given experiment. However, artificial models are an acceptable substitute, assuming that they follow the same structures and guidelines as the real-

world models.

Thus, an artificial model generator was constructed in Python that:

- Examined the real-world DDP models of Figure 2.4;
- Extracted statistics related to the different types of nodes (mitigations or risks or requirements) and the number of edges between different types of nodes;
- Used those statistics to build random models that are both larger and more complex than the original models.

This structure of the real-world models is recreated in a set of arrays that record the internal calculations as follows:

- The raw number of goals, risks, and mitigations.
- The cost of each mitigation.
- The weight of each goal.
- The impacts of risks on goals.
- The effects of mitigations on risks.
- Other internal values.

The decisions made during the construction of a new artificial model must be constrained by the same rules that govern real-world DDP models. To ensure this, a user must choose one of the existing models as an initial starting-point. The model generator takes that real-world model and mutates it into a new one based on several user-specified parameters:

- *Model-Type*: Which existing model to base the artificial model off of.

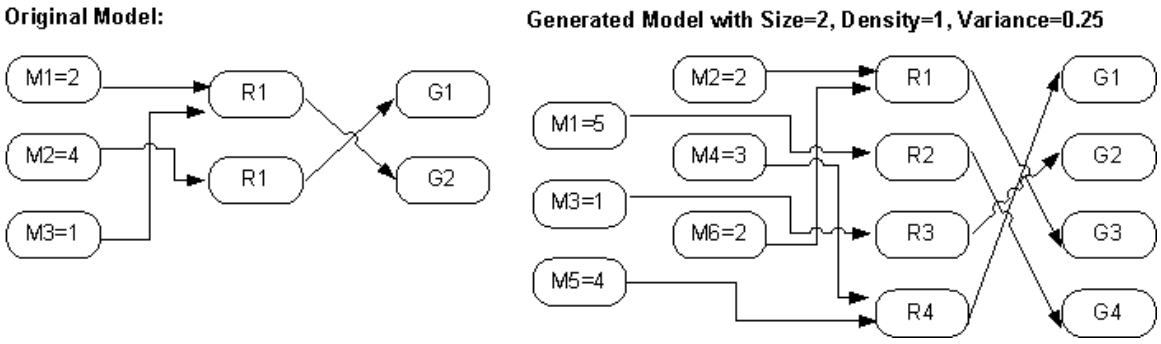

Figure 2.5: An example of creating an artificial model. The new model is twice as large as the original, with no additional connection density, and values come from a distribution within +/- 25% of the original distribution.

- *Size*: A multiplier value used to increase the number of goals, risks, mitigations, and connections between the three. If the original model had two risks, six mitigations, and three goals, setting a size multiplier of four would give the artificial model eight risks, twenty-four mitigations, and twelve goals.

- *Density*: Governs the connections between risks and goals, and between mitigations and risks. The user supplies density values for both types. The size parameter grows density linearly (a model four times the size of one of the originals will have four times the connections. This density parameter is an additional multiplier on that, which lets you produce models that are not only larger, but more complicated.

- *Variance*: When creating a new model, each mitigation, goal, and connection must have a value. These values should be informed by, but not directly copied from, the original model. The variance parameter controls the deviation from the distribution of values in the real-world model. For example, setting a variance of 0.25 would allow for values within +/- 25% of those in the original model.

After loading the specified real-world model into the arrays specified above, a coded function conducts a sampling of the model values. A second function, governed by the user-supplied

settings to each parameter, builds arrays that represent the new model. These arrays are then converted into the compiled "C" format read by the optimization algorithms and saved to the local file system.

A simple example of this process can be seen in Figure 2.5. The original model, seen on the left, has three mitigations, two risks, and two goals. The user chooses to create a new model that is twice as large, but with the same level of density. The user also wants all values to remain within 25% of the original value distribution. The new model, on the right, demonstrates all of these qualities (goal weights, risk → goal impact, and mitigation → risk impact not pictured).

This model generator is used in the scale-up study of Section 5.2 and to benchmark an algorithm variant in Section 5.3. In the absence of a large donation of real-world DDP models, artificial models will likely find more and more use as this research progresses.

2.2.3 Objective Function

When the `model` function is called, a pairing of the total cost of the selected mitigations and the number of reachable requirements (the attainment) is returned. These two values are used by an objective function to score the current configuration of mitigations. The performance values are normalized into a single score that represents the Euclidean distance to a *sweet spot* of maximum requirement attainment and minimum cost:

$$score = \sqrt{\overline{cost}^2 + (\overline{attainment} - 1)^2} \tag{2.2}$$

Here, $\bar{x}$ is a normalized value $0 \leq \frac{x - min(x)}{max(x) - min(x)} \leq 1$. Hence, my scores ranges $0 \leq score \leq \sqrt{2}$ and *lower* scores are *better*.

2.2.4 Decision Ordering Diagrams

The objective function described above summarizes *one* call to a DDP model. This section describes *decision ordering diagrams*, which are a tool for summarizing the results of thousands of calls to DDP models.

Consider some recommendation for changes to a project that requires decisions d of size $|d|$. In the general case, d is a subset of the space of all solutions D ($d \subseteq D$). When checking for solution robustness, or reflecting over modifications to d, a stakeholder may need to consider up to $d' \subseteq N^{|d|}$ possibilities (and $N = 2$ for binary decisions, like whether or not to enable a mitigation in a DDP model). This can be a slow process, especially if evaluating each decision requires invoking a slow and complex simulator.

Decision ordering diagrams are a linear-time method for studying the robustness and neighborhood of a set of decisions. The diagrams assume that some method could offer a *linear ordering* of the decisions $x \in d$ ranked from *most-important* to *least-important*. They also assume that some method offers information on the effects of applying the top-ranked $1 \leq x \leq |d|$ decisions (e.g. the median and variance seen in the model's objective function after applying solution $\{d_1..d_x\}$). For example, the decision ordering diagram of Figure 2.6 shows such a linear ordering (this figure presents $benefit$ and *cost* results). In that figure:

- The x-axis denotes the number of decisions made.
- The y-axis shows performance statistics of an objective function seen after imposing the conjunction of decisions $1 \leq i \leq x$.

To assess performance, some objective function is run, and the results report the median (50th percentile) and *spread* (the range given by the 75th percentile - the 50th percentile) of the values reported by this function. I use median and spread to avoid any parametric assumptions.

These diagrams can comment on the robustness and neighborhood of solution $\{d_1..d_x\}$ as follows:

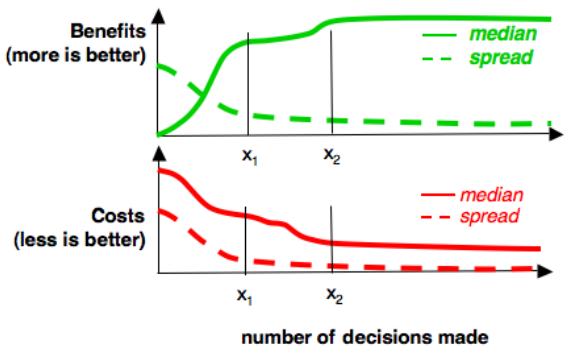

Figure 2.6: A Decision Ordering Diagram. The median and spread plots show 50%-the percentile and the (75-25)%-th percentile range (respectively) values generated from some objective function.

- By considering the variance of the performance statistics after applying $\{d_1..d_x\}$.
- By comparing the results of using the first x decisions to that of using the first $x-1$ or $x+1$ actions.

The neighborhood of a solution that uses decisions $\{d_1..d_x\}$ are solutions that use the decisions $\{d_1..d_{x\pm j}\}$. Since j is bounded $0 \leq |d|-1$, this means that *reflecting over solution neighborhoods takes time linear on the number of decisions d*.

Decision ordering diagrams are a natural representation for "trade studies," the activity of a multidisciplinary team to identify the most balanced technical solution among a set of proposed viable solutions [4]. For example, minimum costs and maximum benefits are achieved at point x_2 of Figure 2.6. However, after applying only half the decisions (see x_1) most of the benefits could be achieved, albeit at a somewhat higher cost.

Decision ordering diagrams are *useful* under at least three conditions:

- The scores output by the objective functions are *well-behaved*; i.e. move smoothly to a plateau.
- The decisions *tame* the variance; i.e. the spread falls to value much lower than then median (otherwise, it is hard to show that decisions have any effect on the system performance).

- The are generated in a *timely* manner. Fast runtimes are required in order to keep up with fast moving discussion.

According to these definitions, Figure 2.6 is a *useful* decision ordering diagram if it can be generated in a *timely* manner.

It is an open issue if real worlds requirements models generate useful decision ordering diagrams. The experiments of this thesis test if, in practice, decision ordering diagrams generated from real world requirements models are *timely* to generate while being *well-behaved* and *tame*.

2.3 Models of Requirements Engineering

The Defect Detection and Prevention model is a ultra-lightweight modeling tool. The value of ultra-lightweight ontologies in early life cycle modeling is widely recognized. For example, Mylopoulos' *soft-goal* graphs [82, 83] represent knowledge about non-functional requirements. The primitive values in soft goal modeling include statements of partial influence such as *helps* and *hurts* relationships. An example of a soft-goals graph can be seen in Figure 2.7.

Another commonly used framework in the design rationale community is a "questions-options-criteria" (QOC) graph [101]. In QOC graphs:

- *Questions* suggest *options*. Deciding on one option can raise other questions;
- Options shown in a box denote *selected options*;
- Options are assessed by *criteria*;
- Criteria represent gradual knowledge.

QOC graphs can succinctly summarize lengthy debates. For example, the four-hundred and eighty sentences stated in a debate on interface options can be displayed in a single-page QOC graph (as can be seen in Figure 2.8.

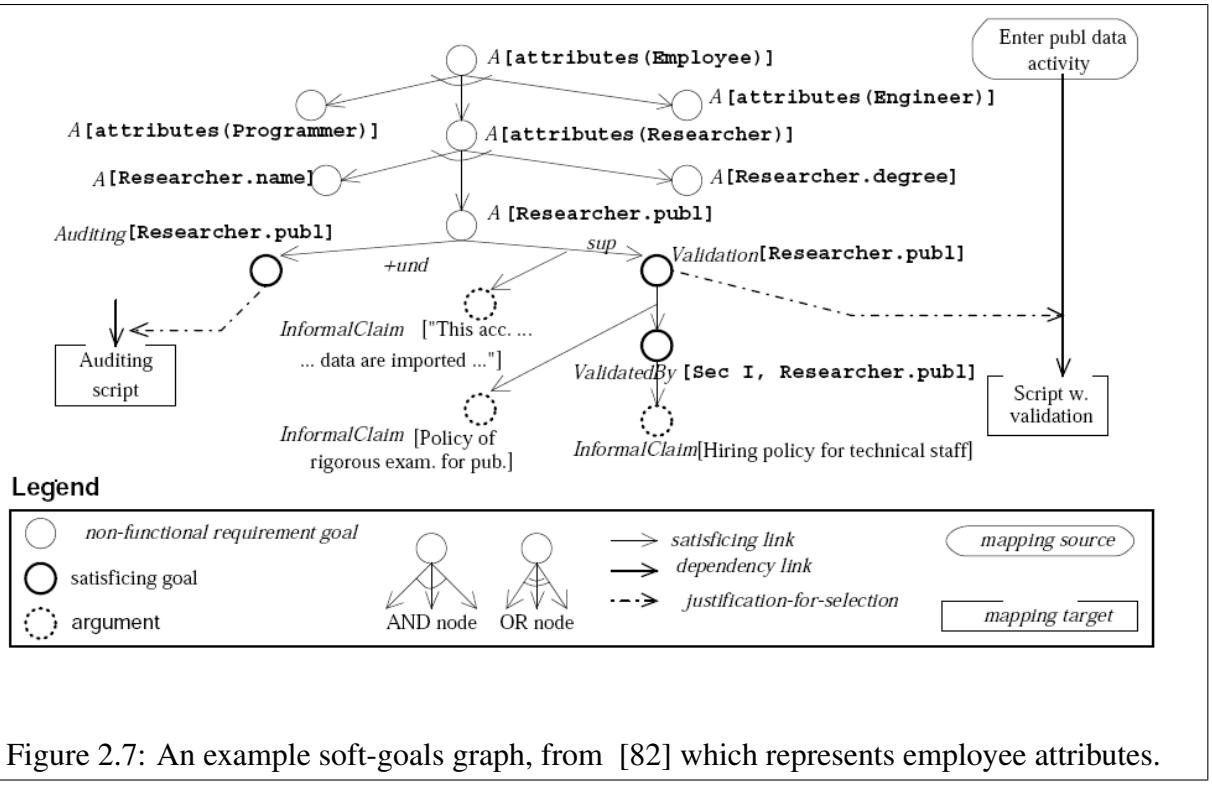

Figure 2.7: An example soft-goals graph, from [82] which represents employee attributes.

While DDP shares many of the design aspects of soft-goal and QOC graphs, it differs in its representations and inference methods. As explained above around Equation 2.1, while QOC and soft-goals propagate influences over hierarchies, DDP propagate influences over matrices.

Zave & Jackson [118] define requirements engineering as finding the specification S for the domain assumptions K that satisfies the given requirements R:

$$\text{find } S \text{ such that } S \vdash R \tag{2.3}$$

Jureta et al. [62] take issue with Equation 2.3, saying that it implicitly assumes that K, S, R are precise and complete enough for the satisfaction relation to hold. More specifically, Jureta complains that Equation 2.3 does not permit partial fulfillment of (some) non-functional requirements. Additionally, the Zave & Jackson definition does not allow any preference ordering; that is, $specification_1$ cannot be said to be more important than $specification_2$. Jureta et al. offer a

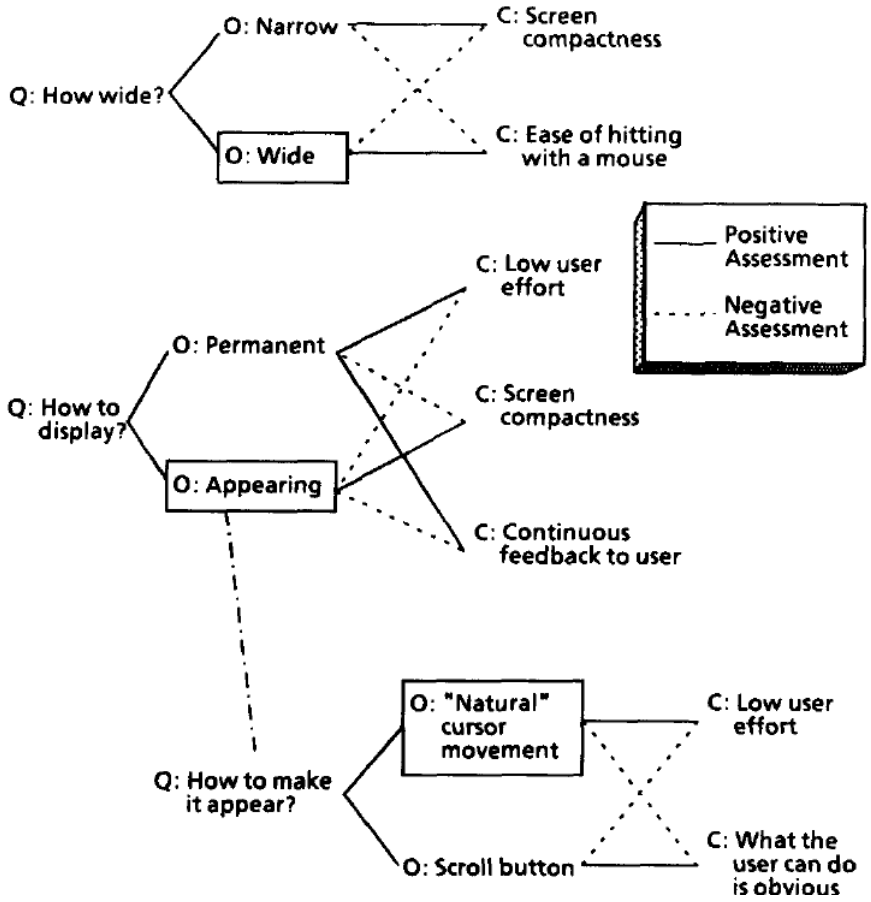

Figure 2.8: An example QOC graph, from [69].
This graph represents the design space for the XCL project. The boxed Options are the decisions made in the design of the XCL environment.

replacement ontology where classical inference is replaced with operators that support the generation and ranking of subsets of domain assumptions that lead to maximal (with respect to size) subsets of the possible goals, as well as soft-goal quality criteria.

DDP reinterprets "⊢" in Equation 2.3 as an inference across numeric quantities, rather than the inference over discrete logical variables suggested by Zave & Jackson. Hence, it can achieve the same goals as Jureta (the ranking of partial solutions with weighted goals) without requiring Jureta's ontology.

2.4 Early vs Later Life-cycle Requirements Engineering

The DDP models presented in this thesis come from the NASA Jet Propulsion Lab's Team X meetings, which conduct early life-cycle requirement discussions.

Once a system is running, released, and being maintained or extended, another problem to solve is *release planning*; i.e. what features to add to the next *N* releases. To solve this problem, some sort of inference engine must reason about which functionality extensions to current software can best satisfy outstanding stakeholder requirements. The challenge of release planning is that the benefits of added functionality must be weighed against the cost of implementing those extensions.

Several approaches have been applied to this problem. The *Next Release Problem* was elaborated on by Bagnall et al. in 2001 [11]. This problem is how to find the appropriate balance between customer satisfaction and available resources. They found that designing an optimal next release is an NP-hard problem. Their study focused on benchmarking heuristics that could find a high-quality but potentially sub-optimal solution. Best solutions for the large problems were found using variants of Simulated Annealing, which produced solutions within 1.5% of the optimal value.

Ngo-The and Ruhe [84] studied how to allocate resources towards the implementation of features such that the value gained from adding such features was maximized. They proposed a two-phase optimization approach called $OPTIMIZE_{RASORP}$ that, in the first phase, applies integer

linear programming to the problem. In the second phase, a genetic algorithm executes over the reduced search space in order to generate resource allocation plans. Their approach proved favorable over a standard greedy search when evaluated over six-hundred randomly generated problems.

Zhang et al. [119] also tackled the Next Release Problem from the multi-objective point of view. They were concerned with the quality of the solutions, the number of possible solutions, the range of solutions covered, and the number of solutions obtained that were optimal. In order to solve the problem, they employed two metaheuristics, called NSGA-II and MOCell. They found that MOCell's solutions covered a larger range, while NSGA-II obtained better solution quality on large problems. Both obtained optimal solutions with a high percentage of low-cost requirements as well as requirements that most satisfied the customers.

Without further experimentation, I cannot assert that KEYS2 (or any of the DDP optimization algorithms elaborated on in this thesis) will work as well on later life-cycle models - such as those used in release planning - as it does on the earlier life-cycle Team X models. However, at this time, I can see no reason why KEYS2 would not work as a non-linear optimizer of these later life-cycle models. Despite the differing intentions, the structure of late life-cycle models is relatively similar to that of early life-cycle models. This could be a productive area for future work.

2.5 Summary

This chapter has presented the Defect Detection and Prevention model. This model, used by NASA's Jet Propulsion Laboratory, maps the requirements of a project to a series of risks. These risks are, in turn, mapped to a collection of mitigations. These mitigations can prevent risks for a certain price. These models can be precompiled into a "C" file that contains a series of values and equations. Precompiling models leads to faster execution times for search algorithms, as it is now faster to obtain a cost and attainment score from the model for any given input. While the experiments in this thesis largely concentrate on five real-world DDP models, I would like to have

larger models in order to study the effects of scale-up on the KEYS algorithm. This chapter details a generator used to create synthetic DDP models. DDP is a form of early life-cycle modeling tool. I have discussed related tools in this chapter, as well as their uses.

I am concerned with obtaining the ideal balance between the cost of a project and the number of attained goals. For any given input, the model will generate separate cost and attainment values. This chapter presents the objective function used to transform those values into a score.

In order to study the output of any assessment algorithm, we must have a result visualization that is easy to read and understand. This chapter discusses decision ordering diagrams, which allow a user to understand the impact of each decision during the search process.

The following chapter will detail a set of candidate algorithms to assess the DDP requirements satisfaction problem.

Chapter 3

Related Work

The previous chapter has discussed the challenge of balancing budget and attainment in DDP models. This chapter will discuss the search-based software engineering field and some of the techniques used to address such non-linear optimization problems.

3.1 Search-Based Software Engineering

As one might infer from the name, search-based software engineering is concerned with the reformulation of standard software engineering problems as search problems, and the application of heuristic techniques to solve said problems. Many of the problems inherent to the software engineering field deal in the *balancing* of competing factors. Do you want to finish with money left in the budget, or is it more important to deliver a robust feature set? You could replace the programmers on an under-performing project, but would it be worth the additional man-hours necessary to get the new staff up and running?

In many of these cases, there is no single *perfect* solution. In fact, there could be a dozen solutions with the same outcome. Instead of finding a single optimal value, software engineering problems are typically concerned with near-optimal solutions, solutions that fall within a certain

tolerance threshold. While it may be impossible or impractical to attempt to find the single *best* solution, it is certainly possible to compare two candidate solutions. Because of this implied trade-off space, meta-heuristic search-based methods are ideal for producing a set of potential solutions.

According to Clark and Harman [50, 52], there are four key properties that must be met for a search-based software engineering approach to be successful:

- *A Large Search Space*: If there are only a small number of factors to compare, there is no need for a meta-heuristic approach. This is rarely the case, as software engineering typically deals in incredibly large search spaces (i.e. the space of all expressible programs written in the JAVA language).

- *Low Computational Complexity*: If the first property is met, SBSE algorithms must sample a non-trivial population. A typical run of one of these searches requires thousands of executions of a fitness evaluation (on a large model, KEYS might have to consult the DDP model ten-thousand times or more). Therefore, the computational complexity of the evaluation method has a major impact on the overall search.

- *Approximate Continuity*: While it is not necessary for a function to be continuous, too much discontinuity can mislead a search. Any search-based optimization must rely on an objective function for guidance, and some level of continuity will ensure that such guidance is accurate.

- *No Known Optimal Solutions*: If an optimal solution to a problem is already known, then there exists no need to apply search techniques.

The first and last properties are absolutely necessary for any search-based software engineering solution to succeed. The second and third *should* be met, but if they are not, it may still be possible to formulate the problem in SBSE language [7, 71]. All four of these characteristics are prevalent in software engineering. As mentioned earlier, problems typically involve a large search space, and optimal solutions are generally impossible to know.

Using these four properties, we can show that the optimization of DDP models is a SBSE problem, and that KEYS2 is a valid solution. Typical DDP models present a massive search space, encompassing hundreds to thousands of possible combinations of mitigation settings. Adding to that, one collection of mitigation settings must comment on dozens of individual mitigations. In one DDP model, there might be 2^{99} ($mitigation\ values^{number\ of\ mitigations}$) possible combinations. With such a large number of combinations in the search space, any candidate solution must execute in a short time. In Section 5.2, it is shown that KEYS2 operates at $O(N^2)$ and Section 5.1.2 demonstrates that the algorithm typically executes in less than half of a second on the largest DDP models available. DDP models do not represent a continuous search space - any model with *true/false* statements is, by definition, not continuous. However, this is not a problem, as the fitness function employed by KEYS2 (and all algorithms used to assess DDP models) represents an approximately continuous trade-off between project costs and feature attainment. Finally, DDP models have no-known optimal solution. In fact, any solution is dependent on the specific features of the project being modeled. As such, a search method must be employed to calculate a selection of *near-optimal solutions*. By meeting all four of these conditions, DDP models (and the candidate search algorithms) represent a valid search-base software engineering problem.

Although the concept of applying search to software engineering problems has existed for decades, the term SBSE was coined and the field was formalized in 2001 [50]. Since then, the SBSE research community has expanded rapidly. SBSE research has been applied successfully to requirements engineering [11, 84, 119], project cost estimation [15, 22, 28], testing [6, 9, 32, 39, 92], software maintenance [49, 80], transformation [7, 26, 71] and software evolution [14]. The following subsections detail popular approaches to solving such SBSE problems.

3.1.1 Hill Climbing and Simulated Annealing

Hill Climbing is a commonly-used form of local search [72]. The basic idea is to start with an initial solution, taken randomly from the search space, and attempt to improve it. The neighborhood

of this initial point is investigated and, if a better solution is identified, the algorithm ”climbs” to this neighbor and sets it as the new ”current” solution. Over a series of subsequent rounds, the hill climbing algorithm will repeat this process until no further improvements can be found. This process, of steady improvement, is called ”hill climbing” because the algorithms scales ”mountains” in the landscape of an objective function. The peaks of these hills represent solutions with localized maximum values, just as the lowest recorded levels represent local minima.

Hill Climbing algorithms typically employ one of two strategies at a given stage. In the first, called *steepest ascent*, all neighbors are evaluated. The neighbor showing the greatest improvement, according to the objective function, is chosen to replace the current solution. The second strategy, *random ascent*, examines neighbors at random. The first neighbor to show an improvement is selected as the replacement.

These algorithms are widely used for several reasons - they are simple to understand, easy to implement, and very fast. However, they have a key flaw - it is easy for these methods to become stuck in local maxima or any area where the score plateaus (i.e. all neighbors have the same objective value, thus there is no improvement). They will then yield results that, while optimal for that section of the search space, are inferior to the global maxima. In any non-trivial landscape (which encompasses many of those seen in SBSE), results obtained with a hill climber are highly dependent on the starting solution. A common method of dealing with this risk is to allow the algorithm a set number of ”restarts.” In that case, it will pick a new starting position if it ever becomes trapped. This results in a larger sampling of the search space, which is likely to produce a more globally optimal solution. The implementation of MaxWalkSat introduced in Section 4.2 is a hill climber (hybridized with a random search) with the ability to reset.

Hill climbing techniques are widely used, but it could be desirable for an algorithm to be less dependent on the starting position. Simulated annealing [65, 79] is one commonly used alternative. Simulated annealing (SA) is introduced in more detail in Section 4.1. Like hill climbers, a simulated annealing algorithm will choose a random starting position and move towards better

solutions in the immediate neighborhood. However, by probabilistically accepting worse solutions, SA allows for a less-restrictive traversal of the search space. As the algorithm progresses, its temperature function decreases (i.e. it cools according to the output of a controlling equation). Initially, the temperature is kept at a high level, which increases the probability of accepting a poor solution. Thus, more of the search space will be explored. As the search progresses, movement stabilizes and plateaus. One must take caution when choosing a cooling strategy - cool too quickly, and it is likely that the algorithm will return a sub-optimal solution.

Tracey et al [107,108] used simulated annealing, a genetic algorithm (GA), a hill climber, and a random search for finding the worst-case execution time (WCET) for a number of well-understood programs written in the Ada language. In the search technique executed the path through the program that yielded the already known WCET, the trial would be deemed a success. Overall, the Simulated Annealing technique completed more trials successfully than the genetic algorithm, and both of them vastly outperformed the random search and hill climber. Interestingly, their study found that varying the parameters of each optimization technique would have little effect on the end results, except in the case where the starting temperature was too low for the SA algorithm. In that case, the dependency on the starting position was too high.

Tracey [106, 108] also applied a variety of techniques, including SA and a GA, to the problem of specification conformance. The conformance of an implementation to its original specification is checked by executing the test object with some generated testing data, which is then validated against the specification. The two subsystems chosen for this experiment come from a safety-critical nuclear primary protection system, written in the Pascal language. The search techniques were wildly successful in finding the ”mutant” situations that did not conform to the specification. Both the annealer and genetic algorithm identified all of the 170 existing violations. The hill climber and random search only found about 90% of the anomalies.

Mahdavi et al [70] used hill climbing algorithms to study the problem of automated software module clustering, which is important for systems with unclear module boundaries of where struc-

ture had degraded as the system evolved. In their work, they demonstrate that the results from a set of multiple ”climbs” can be combined to form the building blocks for subsequent clustering efforts. Each climb outputs a module dependency graph, and when these graphs are looked at in sequence, common features emerge that may indicate a good cluster. When that dependency graph is used as the input for the next hill-climb, that execution tends to reach even higher score peaks. This process substantially reduces the search space and reveals certain hard-wired portions of the solution.

3.1.2 Tabu Search

Tabu search [42, 43] is a popular meta-heuristic search method. Like a hill climber, a tabu search picks a random starting position, examines its local neighborhood for locations that better satisfy an objective function, and jumps to these positions. Tabu search distinguishes itself by modifying the neighborhood structure at each stage. The solutions admitted to the new neighborhood are determined through the use of memory structures. These memory structures constrain the search space by classifying certain moves as forbidden (*taboo*), and frees the search through a mechanism that allows the algorithm to forget certain taboos over time. This ensures that a tabu search (TS) algorithm never becomes stuck in a local maxima, and that it never falls back into one of these points once it emerges from it.

The most important internal structure in a tabu search is the *tabu list*, a move list that blocks certain portions of the neighborhood deemed to violate a list of logical rules (as informed by the objective function). In the simplest form, this list is a memory of the solutions that have been recently visited. By listing these locations as taboo, the search will never spend time revisiting covered ground. Items will gradually expire from this list as their user-supplied *tabu tenure* expires. Other facets of the tabu list are dependent on the constraints of the search problem. For instance, a tabu search covering the traveling salesperson problem may have a list prohibiting certain arcs from being removed over the next N moves.

By marking certain attributes as being taboo, entire portions of the search space will be constrained. Unfortunately, these regions may still include optimal solutions. To avoid this problem, a list of *aspirational criteria* is used (this may be as simple as reconsidering any solution that is taboo, yet yields a better score when subjected to the objective function). If any of these criteria are satisfied, the tabu list will be overridden and that state may reenter the neighborhood.

In a recent article [33], Diaz et al. applied tabu search to the task of automatic generation of structural software tests. Such structural test generation is almost exclusively conducted by evolutionary algorithms (genetic algorithms, particle swarm optimization), which makes the choice of tabu search interesting. Their test generator, TSGen, manipulates its tabu list by utilizing one cost function for intensifying the search and another for diversifying the search when such intensification is unsuccessful. The algorithm combines its memory list with a backtracking process in order to avoid becoming stuck in localized areas. In order to test their tabu search, they set it loose on three structural testing problems - one a standard benchmark (the triangle classifier problem) and two more complex (line rectangle classifier and the distance between two dates). On all three problems, TSGen achieved 100% coverage in a fraction of the time that it took a random search to address the problem.

Stardom [102] has benchmarked the efficiency of several meta-heuristic techniques on the problem of finding adequate covering arrays (also called packing arrays). Covering arrays are used in software testing to account for the numerous possible configurations that a piece of software can be executed under. It is often impossible to test every possible configuration; therefore, covering arrays are used to select the ideal subset of configurations to test. In a series of experiments where the goal was to find the best covering array possible in the shortest amount of time, Stardom found that genetic algorithms were completely ineffective in comparison to simulated annealing and tabu search. The simulated annealer was very useful for finding covering arrays when an array's neighborhood was smaller, but overall, the tabu search returned the highest-quality solutions.

In a similar piece of research, Kari Nurmela used a tabu search to construct covering arrays that

would, ideally, improve on previous upper-bounds on the size of an optimal covering array [86]. Covering arrays contain a parameter t where, in any t coordinate positions, all combinations of those coordinate values occur at least once. When $t = 2$, all vectors for a covering array are binary, and the make-up of an optimal array is known; however, for all other cases, only upper and lower-bounds can be known. Nurmela's tabu search selects an uncovered t-combination at random and checks for rows that require the change of a single element such that the row will cover the selected combination. The algorithm will move based on which element requires the lowest cost to change. That element will be added to the tabu list, preventing any further changes to it for a set period of time. Their tabu search was able to improve upper bounds, with the caveat that the more time given to the algorithm, the better the results returned.

3.1.3 Genetic and Evolutionary Algorithms

Inspired by early experiments with the computational simulation of evolution [12, 55], genetic algorithms (and the broader field of evolutionary algorithms) have become one of the most famous meta-heuristics used in the search-based software engineering literature. They are inspired by Darwin's Theory of Evolution, with the general idea being to take a group of candidate solutions and mutate them over several generations - filtering out bad "genes" and promoting good ones. Genetic algorithms rely on four major attributes: population, selection, mutation, and crossover.

During each generation (that is, each step of the algorithm), multiple solutions are considered, forming a population. Because the initial population is unlikely to have the "best" solution, some form of diversity must be induced into the population. Such diversity can be maintained by using the crossover and mutation operations. During each generation, several "good" solutions (as scored by a predetermined objective function) are chosen by the selection mechanism to generate children for the next generation. The makeup of these children is influenced by the crossover operator. This function combines parts of the chromosomes, the solution presented by each parent, and inserts them into the offspring with probability $P_{crossover}$. An example of a crossover can be seen

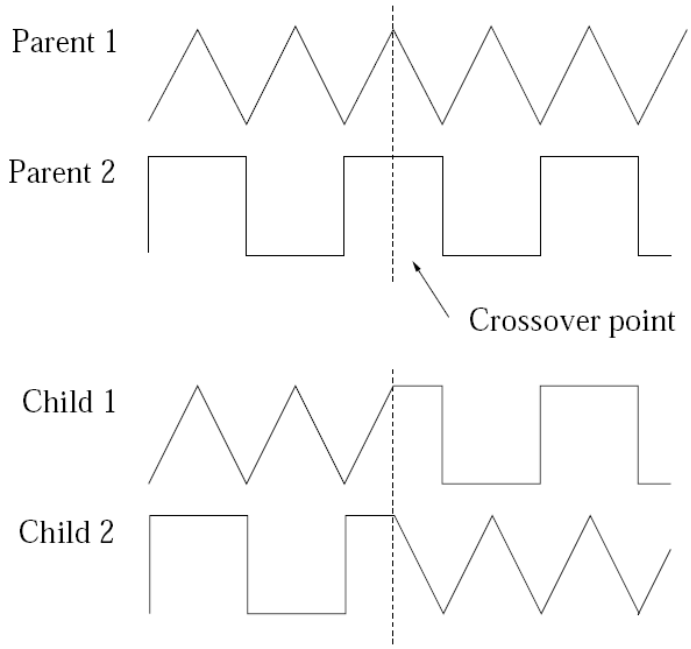

Figure 3.1: Demonstration of the crossover operation, from [91, 94].

in Figure 3.1, where the child solution inherits genes from its parents. A mutation mechanism, demonstrated in Figure 3.2, is used to instill small changes in the chromosomes of each offspring. This is necessary to prevent the algorithm from being trapped in local maxima. To avoid the loss of good solutions, a certain number of the best results will be copied to the next generation without any sort of modification. This new grouping of offspring forms a new population, and the process is repeated until a certain threshold (commonly in performance score, number of generations, or a set time period) has passed.

To summarize, a standard genetic algorithm follows this framework:

- Evaluate each member of the population.
- Create a new populations using these scores along with the crossover and mutation mutation mechanisms to generate offspring.

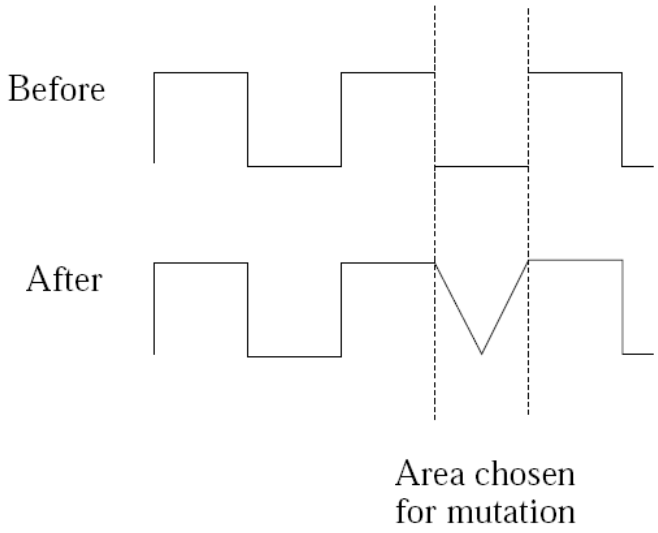

Figure 3.2: Demonstration of the mutation operation, from [91, 94].

- Discard the old population and repeat the process.
- Stop if $time > maxtime$ (in number of generations or some real-time threshold).

Genetic algorithms are a specific subset of the broader field of evolutionary algorithms, which use mechanisms inspired by biological evolution to generate offspring and test them against some pre-determined fitness function. Evolutionary algorithms have been found to be effective for finding a group of near-optimal solutions for the kind of complex problems that are not suited to a brute force approach (that is, where the search space is too large). Evolutionary algorithms cannot always find the best possible solution in a reasonable amount of time. However, they are generally able to generate a solution that is ”good enough.”

SBSE, as a formal research field, is still in its youth. There are still an unfathomable number of problems to explore, and an equal number of approaches to these problems. Genetic algorithms have become an increasingly popular approach to solving search problems. A single glance at the

”best paper” nominees for the SBSE and Search-based Testing categories at the recent 2009 ACM Genetic and Evolutionary Computation Conference reveals a number of exciting advances in the field. Some of the research presented includes:

- Gueorhuiev et al [45] explored software project planning under the hypothesis that all large-scale projects inevitably contain a degree of uncertainty, which in turn leads to overruns. The authors have formulated this issue as a multi-objective SBSE problem where project robustness and completion time are designated as competing objectives. Their genetic algorithm, SPEA II [120], searches for the pareto front, the *sweet spots*, in the solution space where there exists an optimal balance of the two opposed goals. While increased robustness generally results in longer completion times, there exist spots in localized regions where the manager can trade *small* amounts of one objective for *large* gains in the other. SPEA II seeks out these spots. The authors tested their problem and algorithm on four large real-world projects from around the world, evaluated over two different models of uncertainty. The results provided interesting insight into the projects studied, and overall accuracy was favorable compared to that of a random search.

- Phil McMinn introduced a novel approach to testability transformations [73], source-to-source program transformations intended to improve the testability of a project. In his research, transformations are used to generate a *pseudo-oracle*, an alternate version of a program whose output can be compared directly to the original. Differences in the generated output could indicate a fault in the original program. Two transformations are utilized - one highlighting numerical or roundoff errors, the second detecting race conditions in multi-threaded code. After one of these pseudo-oracles is generated, any number of SBSE techniques may be used to find differences between the programs. Both genetic algorithms and random searches were able to utilize the oracles to identify faults and assess their severity. Additionally, genetic algorithms were found to be able to explicitly maximize the differences

between the originals and the oracles, which allowed for the generation of test cases where failures are highly pronounced.

- Harman et al [53] presented an application of genetic algorithms to sensitivity analysis, which is a technique addressing problems with unreliable cost estimates on software projects. In particular, the authors use SBSE techniques to aid a theoretical decision maker in exploring sensitivity of the cost estimates for the *Next Release Problem* (several examples of this problem are discussed in Section 2.4). In short, the "Next Release Problem" is concerned with finding the appropriate balance between customer satisfaction and available resources for the $(N+1)$th version of a project. Harman's paper presents single and multi-objective formulations of the NRP with sensitivity analysis on both real-world and synthetic data, tested using a greedy algorithm and the NSGA-II [25] genetic algorithm. While some of their results were expected - such as a strong correlation between the level of inaccuracy and the impact on the selection of requirements - they also found notable exception to the trends. They show that unusually sensitive patterns occur in real-world data and how their algorithms clearly identify them.

- The majority of existing work on evolutionary testing, the automated generation of software test cases, is targeted towards procedural languages. Ferrer et al proposed an evolutionary testing method for object-oriented (OO) languages [39] that addresses the *inheritance* feature. Their approach uses information mined from the class hierarchy to help test generators better guide the search process. The authors define a branch distance for logical expressions containing the $instanceof$ operator from the Java language. They then explore two mutation operators based on this distance metric. They study the behavior of these mutation operators over nine object-oriented software projects. The results gleaned from this benchmarking prove favorable, demonstrating that information collected from the class hierarchy does help in the search for better test cases.

- Much of the research in applying SBSE to software testing has been of an empirical nature - proposed techniques are validated on software testing benchmarks. Andrea Arcuri's work [9] asks *why* these meta-heuristic techniques are effective in the testing of software projects. Such insight could theoretically be used to design new, more successful, approaches. Her work combines an empirical and theoretical analysis, and exploits the benefits of both. She analyzes the testing of Red-Black Trees, considering a white-box scenario in which the full coverage of all the branches of the software is sought, evaluated over four different search techniques (random search, hill climbing, an evolutionary algorithm, and a genetic algorithm). Rather than treating these algorithms as black-boxes, she tries to look inside of them for properties of the search problem that can be exploited. These discovered properties are used to tune the evolutionary and genetic algorithms, as well as to further constrain the random search and hill climber.

- Rhys et al. [92] are concerned with the automatic generation of test data for Matlab code. The most critical, and often error-prone, sections of software applications are code that performs complicated mathematical functions. This code is very difficult to test due to the data types used and the complex mathematics. The authors have chosen Matlab as an example of widely-used mathematical software, and they have used its matrix data types and associated relational operators to extend previous work on search-based test generation. Their paper introduces a technical framework fro instrumenting Matlab code, a method of representing matrix data types in a genetic algorithm, an extended set of branch distance functions, a rank-based selection mechanism that combines this branch distance with a cost component, and the identification of linear algebra functions that lead to difficulties when present in relational predicates. They benchmark a genetic algorithm capable of utilizing this information against a standard random search. A set of experiments proves that, despite the inherent challenges of mathematical software, automated search methods can generate feasible test data.

This snapshot of current research in using genetic algorithms on SBSE problems shows the health and popularity of this approach. KEYS and GAs have never been directly compared, but in some ways, it could be said that KEYS operates in a similar manner to a genetic algorithm. At each ”generation,” KEYS produces a set of solutions and finds one ”best” solution from the population. It uses that one solution to fix a setting (a ”chromosome”) for the next generation. There are a number of key differences - GAs produce offspring by combining all ”good” solutions for a generation while KEYS only uses one member of the population to influence a new generation - but it could be interesting to compare the two approaches. My hypothesis is that standard GAs would be too slow to compete with KEYS; however, it is possible that certain features of these algorithms could be imported into future iterations of my algorithms.

3.1.4 Particle Swarm Optimization

Particle swarm optimization (PSO), introduced in 1995 by Kennedy and Eberhart [30,64], attempts to solve search problems by mimicking the social optimization conducted in any form of group behavior. Groups of creatures - birds, fish, bees, even humans - solve problems by working with their neighbors. As they talk to others around them, their beliefs, intentions, and desires change. Individuals working in swarms tend to, over time, converge into a stable behavior in a close space. This type of optimization is the basis of PSO.

In the PSO algorithm, a swarm of particles are used to represent potential solutions. Each of those particles, i, is associated with a velocity vector $V_i = [v_i^1, v_i^2, ..., v_i^D]$ and a position vector $X_i = [x_i^1, x_i^2, ..., x_i^D]$ (D is equal to the number of dimensions in the solution space). The velocity and position of each particle is initialized to random vectors withing the range of the search space. Each round, the velocity and position of particle i on a dimension d are updated as follows:

$$v_i^d = \omega v_i^d + c_1 rand_1^d (pBest_i^d - x_i^d) + c_2 rand_2^d (nBest^d - x_i^d) \tag{3.1}$$

$$x_i^d = x_i^d + v_i^d \tag{3.2}$$

where c_1 and c_2 are acceleration coefficients. The original creators recommend a fixed value of 2.0 for each, but other researchers prefer ad-hoc values [104] or ones that vary with time [3] (with a larger c_1 at the beginning, and a larger c_2 at the end). $rand_1^d$ and $rand_2^d$ are two uniformly distributed random numbers generated for dimension d. $pBest_i$ is the position found to date with the highest score (as indicated by the problem's objective function), and *nBest* is the top-scoring position in the current neighborhood. Note that in some implementations of PSO, $pBest_i$ is split into *local* and *global* values.

ω is an inertial weight introduced into the PSO algorithm by Shi and Eberhart [98]. The inertia linearly decreases over a series of generations according to:

$$\omega = \omega_{max} - (\omega_{max} - \omega_{min})\frac{g}{G} \tag{3.3}$$

where g is the current generation, and G is a predefined maximum number of generations. [98, 99] recommend that ω_{min} and ω_{max} be set to 0.4 and 0.9 respectively. Research has also been conducted with a fuzzy adaptive inertial factor [100] and a randomized ω [31].

A user-specified parameter, V_{max}^d, is used as an upper bound on the velocity of dimension d. Thus, if the magnitude of the velocity $|v_i^d|$ exceeds V_{max}^d, then v_i^d is set equal to V_{max}^d, with the sign (positive or negative) determined by the original value of v_i^d.

Each round, the particles evaluate the fitness of each candidate solution and remember the locations of the top-scoring solutions. Each particle makes their $pBest_i$ available to the local neighborhood, and the neighborhood best (*nBest*) is updated to match the very best candidate solution in the entire local area. These "best" values are used to help guide the velocity and position functions described in equations 3.1 and 3.2. The algorithm reaches convergence when all particles approach a certain value and the fitness function stops reporting better values. PSO algorithms tend to stop either then the velocity function has slowed down to zero, a certain number of genera-

tions have passed, or if convergence has been detected. To prevent convergence at a local maxima, stagnation operators are sometimes used to reset the particles if they stop at a position known to be sub-optimal.

Particle swarm algorithms come with their own share of weaknesses. Like any other population-based evolutionary algorithm, PSO can be computationally inefficient. Further, the standard algorithm can easily become trapped in local optima. A number of approaches have been proposed to address both issues [8,67,105,117] (often by incorporating techniques from genetic algorithms [8] or local search [67]). Despite these limitations, particle swarm optimization has become a popular approach in the search-based software engineering field because it is easy to implement, easy to parallelize, and gradient-free [112].

Ferriera et al. [38] have used particle swarm optimization to study errors in network protocols. Failures in network protocols are critical, thus they must be verified in order to ensure that they meet all requirements. Such verification can be performed using model checking; however, the large number of required states limits the size of the models that are possible to check. Particle swarm optimization was chosen to optimize these models because of the algorithm's parallel nature, which allows it to outperform other approaches that would fail due to the size of the search space. PSO cannot be used to verify these network protocols, but it can efficiently find errors in the software. The authors have implemented their experiment in the Java Pathfinder model checker. The PSO algorithm was able to generate better quality results than exhaustive algorithms and it could assess protocols that exhausted the memory constraints of other techniques.

Windisch [114] researched the efficiency of particle swarm optimization in evolutionary structural testing. In such structural testing, an algorithm automatically generates test cases with the goal of achieving high code coverage. He and his coauthors performed experiments with twenty-five artificial test objects and thirteen more complex real-world projects. While both PSO and a GA needed more time to optimize as the number of parameters increased, a more detailed analysis revealed that the GA was unable to reach the global maxima within the given number of func-

tion evaluations when it had to optimize more than one parameter. The PSO algorithm reached it when optimizing one, two or three parameters. In situations where both algorithms converged successfully, PSO often reached better solutions given the same number of iterations. In thirteen of nineteen experimental cases reported, PSO outperformed the GA by orders of magnitude. In the remaining six cases, the GA reached better results, but not by any statistically significant amount. In general, GA featured a slightly faster convergence for simple functions whereas PSO outperformed GA on complex functions with larger search spaces.

Diaz-Aviles et al. recently presented an approach, using Particle Swarm Optimization, to automatically optimizing the retrieval quality of ranking functions [27]. Their method, Swarm-Rank, learns a ranking function by optimizing the combination of various types of evidence, such as content and hyperlink features. At the same time, it works to maximize the mean average precision of the results returned. In a series of experiments on benchmarking datasets, they compared Swarm-Rank against the standard BM25, as well as the sate-of-the-art RankingSVM and RankBoost algorithms. Swarm-Rank always outperformed BM25 and was found to be competitive with both RankingSVM and RankBoost on ranking relevant documents at the very top positions (often outperforming at least one of the two).

3.1.5 Ant Colony Optimization

Ant Colony Optimization, as formalized by Dorigo and Gambardella [29], is a meta-heuristic swarm intelligence method similar to the previously discussed particle swarm optimization. In an ant colony optimization (ACO) algorithm, a series of cooperating agents work to find optimal solutions to search problems in a manner similar to real-life ants. Certain species of ants are known to be able to find the shortest path from a food source to their nest without any visual guidance [90]. Instead, these ants follow a chemical trail, a *pheromone*, left by ants that have visited this food source in the past (this process is demonstrated in figure 3.3). Similarly, during ant colony optimization, the agents traverse a graph, choosing paths by incorporating information

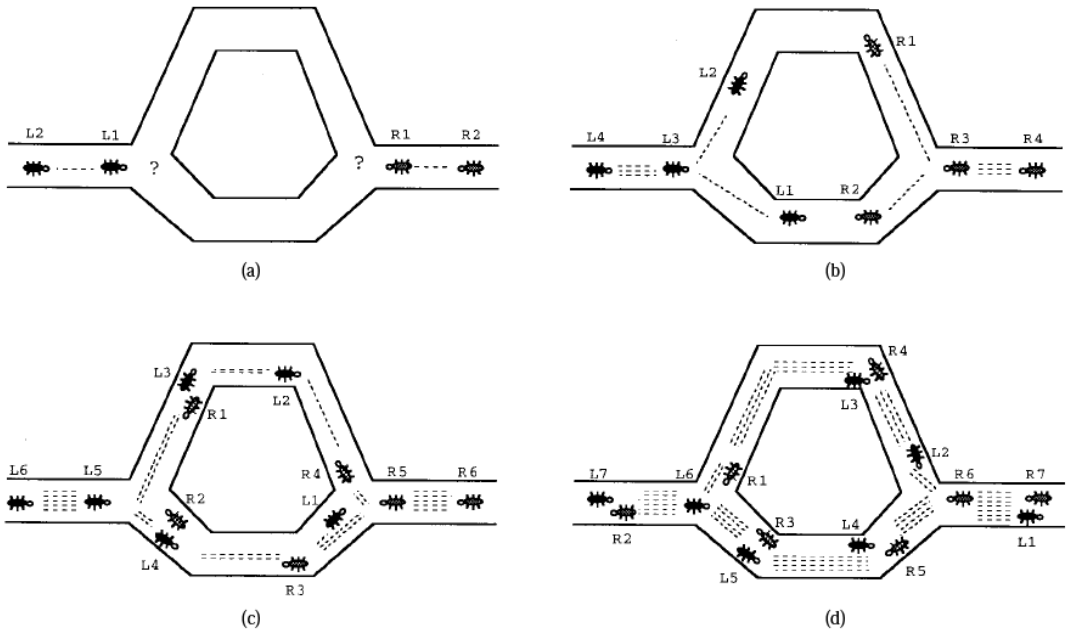

Figure 3.3: A demonstration of how real-life ants locate food, from [29].
In (a), an ant has arrived at a fork in the path. As expected, some ants will travel in one direction and some in the other - shown in (b). (c) shows that, since ants move at a constant speed, the ants that took the lower path will arrive at the food first. Thus, as seen in (d), pheromone accumulates at a higher rate on the shorter path (which means that the probability of an ant choosing that path will grow over time).

based on solutions and their qualities achieved by other agents during elapsed iterations (that is, the ”pheromones” left by past ”ants”). Through the accumulation of pheromones, the agents will be able to converge upon a set of high-scoring solutions and the shortest path to achieve said solutions.

The ACO meta-heuristic ultimately breaks down into three phases: generate solutions, update pheromones, and conduct daemon actions. The typical form of a solution depends on the type of search problem being explored, but they can be simplified as paths through some kind of graph space. At each step during the construction phase, every agent in the system adds one vertex to its

path. The ant will then choose to move from vertex i to vertex j with probability:

$$P_{i,j} = \frac{(\tau_{i,j}^{\alpha})(\eta_{i,j}^{\beta})}{\Sigma(\tau_{i,j}^{\alpha})(\eta_{i,j}^{\beta})} \tag{3.4}$$

where $\tau_{i,j}$ is the amount of pheromone left on $edge_{i,j}$ and $\eta_{i,j}$ is the *desirability* of that edge. The desirability is defined as a function of the a priori information known about the search problem. For instance, desirability could be the gain in the score from an objective function achieved by taking that step. The α and β exponents are user-supplied weights used to determine the influence of the pheromones versus the desirability.

At each stage of the problem, an ant will typically have to choose between several competing edges. This is done through a process called *roulette selection*, where all actions are given a probability (see equation 3.4) and the ant chooses a random number between zero and one. The probability of selecting an action is added to a sum, and if adding another probability to that sum would push the sum beyond the random number, that action is chosen as the ant's next step. That vertex is used to mutate the ant's current solution, and the ant steps forward in the search space.

Once every agent has selected their next action, a "pheromone update" takes place. This phase has two steps - evaporation and deposit. In the evaporation step, a certain amount of pheromone is removed on each edge according to:

$$\tau_{i,j} = (1-\rho)\tau_{i,j} \tag{3.5}$$

where ρ is a user-supplied *evaporation control rate*. The second step, the deposit, adds pheromones to each visited graph edge as follows:

$$\tau_{i,j}^{k} = \tau_{i,j}^{k} + \frac{1}{C_k} \tag{3.6}$$

where C_k is the calculated cost of the kth ant's solution. This deposit process, over time, will cause

the amount of pheromones on a ”good” edge to increase far in excess of the evaporation rate. In turn, this will make certain sections of the solution space far more attractive to each ant, guiding them towards the optimal corners.

The third phase, *daemon actions*, is a name given to the set of post-processing steps required by the search problem. This phase is optional and highly problem-specific.

Although the most obvious use of ant colony optimization is on combinatorial optimization problems such as the traveling salesperson problem [1, 29] or the quadratic assignment problem [68, 111], ACO is well-suited to any problem that is rapidly-changing or that requires that the current ”best” solution be updated regularly [51]. Until 2007, ant colony optimization saw little use in search-based software engineering, but that has changed rapidly over the past three years years.

Prandtstetter and Raidl [89] recently applied ant colony optimization and variable neighborhood search (also known as VNS, basically a hill climber with resets) to what is technically an software engineering problem, though an unconventional one - the reconstruction of shredded documents. The authors consider four different construction heuristics, favoring quick solutions over slightly more complete ones. Those heuristics were Prim's method, border similarity, and two additional heuristics based on the fact that the edges of pages (the margins) are typically left blank. The ACO algorithm then applies randomized versions of these heuristics. In experimental tests, the ACO far outperforms the VNS approach in terms of solution quality (obtaining higher-quality solutions in thirty-four of forty-five cases). However, the VNS shows shorter execution times. The solutions obtained from the ACO algorithm are some of the best to date in document reconstruction, often recreating a large portion of the original text document.

Chicano and Alba [16] applied ant colony optimization to the problem of finding safety property violations in concurrent models. Their approach, ACOhg, modifies ant behavior with internal resorts in order to search very large landscapes. ACOhg is applied to concurrent models in edge-case scenarios, where other methods have previously failed. In experiments conducted over nine

models, ACOhg had a 100% success rate in finding the error path, outperforming a standard A* search. During this search, ACOhg expanded fewer paths than A* and, as a consequence, used less memory.

Ayari et al. have applied ant colony optimization to fault-based testing [10]. Evolutionary algorithms are usually applied to this task - fault-based testing is expensive and evolutionary algorithms have been proven to lower the cost of test generation. The authors propose an ant-based scheme, augmented with a probability density estimation mechanism, for automatic test-input generation in the context of mutation testing. They compared their approach with a standard genetic algorithm, a random search, and a hill climber on two different JAVA-based testbeds. On both projects, the ACO vastly outperformed all of the other approaches (with mean mutation scores of 89% and 88% against the 35% and 42% scored by the next-best solution, the GA).

3.1.6 Other Methods

As documented by the search-based SE literature [19, 50, 52, 91] and Gu et al [44], there are many other possible optimization methods. Some of these are:

- Gradient descent methods assume that an objective function $F(X)$ is differentiable at any single point N. A Taylor-series approximation of $F(X)$ can be shown to decrease fastest if the negative gradient $(-\Delta F(N))$ is followed from point N.

- Discrete methods assume that model variables have a finite range (that may be quite small) while continuous methods assume the existence of numeric values with a very large (possibly infinite) range.

- Some methods map discrete values (true/false) into a continuous range (1/0) and then use integer programming methods like CPLEX [81] to achieve results.

- Other methods find overlaps in goal expressions and generate a binary decision diagram

(BDD) where parent nodes store the median of overlapping children nodes.

- Sequential versions of most optimization algorithms exist. These implementations run on one CPU while parallelized methods spread the work over a distributed CPU farm.

This survey of related work is hardly exhaustive - Gu et al. lists hundreds of other methods and no single researcher can experiment with them all. All the algorithms studied in this thesis are discrete and sequential. I have spent some time exploring parallel versions of the KEYS optimizers but, as of yet, the communication overhead outweighs the benefits of parallelism.

As for the general class of gradient descent methods, I do not use them because they assume the objective function being optimizing is essentially continuous. Any model with an ”if” statement in it is not continuous since, at the ”if” point, the program's behavior may become discontinuous. The DDP requirements models studied in this thesis are discontinuous at every subset of every possible mitigation.

Likewise, I do not explore the more specific class of integer programming methods for two reasons. First, Coarfa et al. [20] found that integer programming-based approaches ran an order of magnitude slower than discrete methods like the MaxWalkSat and KEYS2 algorithms, thus violating the speed requirement. Similar results have been reported by Gu et.al where discrete methods ran one hundred times faster than integer programming [44].

Harman offers another reason to avoid integer programming methods. In his search-based SE manifest [50], he argues that many SE problems are over-constrained and, thus, there may exist no precise solution that covers all constraints. This implies that a complete solution over all variables is impossible and partial solution based on heuristic search methods are preferred. Such methods may not be complete; however, as Clarke [19] remarks, “...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.”

3.2 Treatment Learning

The field of data mining uses techniques from statistics and artificial intelligence to find small, yet relevant, patterns in large sets of data. The standard practice in this field is to *classify*, to look at an object and make a guess at what category it belongs to. As new evidence is examined, these guesses are refined and improved. While this paradigm doesn't directly map into the terminology utilized by the DDP requirements models, a related example of classification might be to look at a series of projects that use DDP models during the planning phase and ask how many risks those projects mitigated. If, say, that project mitigated greater then 75% of risks, then a classifier could categorize that project as *attainment-maximized* and anything below that as *cost-minimized* (as those projects decided that some risk was acceptable in order to save part of the budget).

Treatment learning [74] focuses on a different goal. It does not try to determine *what is*, it tries to determine *what could be*. Classifiers read a collection of data and collect statistics that are then used that to place unseen data into a series of discrete categories. Treatment learners work in reverse. They take the classification of a piece of evidence (that is, the category that it belongs to) and try to reverse-engineer the statistical evidence that led a classifier to assign the data to a particular class. For instance, rather than deciding whether a project was *attainment-maximized* or *cost-minimized*, a treatment learner might try to identify which mitigations should have been enabled for an ideal cost-attainment balance (note that this is *exactly* what KEYS does). Treatment learners take that evidence and use it to produce a treatment—a small set of rules that, if imposed, will change the expected classification distribution. By filtering the data for entries that follow the rules set in the treatment, you should be able to identify *why* a particular classification was reached.

Ultimately, classifiers will strive to increase the representational accuracy. They will assess the data and grow a collection of statistical rules with the goal of making more and more accurate categorizations. As a result, if the data is complex, the decision tree output by the classifier will also be complex. Treatment learning instead focuses on minimality: what is the *smallest* rule that

can be imposed to cause the *largest* change. Often, these rules are as simple as filtering for the data where, say, the wind speed on the outside of an aircraft is between 25 and 45 knots.

Stated formally, treatment learning is a form of minimal contrast-set association rule learning. The treatments contrast undesirable situations with the desirable ones (represented by weighted classes). Treatment learning, however, is different from other contrast-set methods like STUCCO [13] because of its focus on minimal theories. Conceptually, a treatment learner explores all possible subsets of the attribute ranges looking for good treatments. Such a search is infeasible in practice so the art of treatment learning lies in quickly pruning unpromising attribute ranges (i.e. ignoring those that, when applied, lead to a class distribution where the target class is in the minority).

Earlier, I pointed out that KEYS behaves similarly to a treatment learner. KEYS is designed to provide settings for a model, and then test the model output against a continuous objective function. Settings that maximize this function are saved and imposed on subsequent rounds. A treatment learning algorithm reflects over pre-classified historical data and identifies the features that lead to a particular target class, then recommends using those features as a filter to only leave examples of the target class. KEYS can, in fact, be thought of as a treatment learner that also deals with the other steps of the process that standard treatment learners ignore. Rather than reflecting over historical data, KEYS generates a one-hundred record dataset every round when it comes up with a series of random model configurations. KEYS then enters those model configurations into an objective function, and receives a score in return. While this technically results in a series of continuous scores, those scores are effectively turned into two binary classes (*best* and *rest* around a threshold point. The KEYS algorithm then uses those classifications to identify the exact mitigation (and its value) that led to the most ”best” classifications. That setting is imposed, and subsequent rounds identify additional settings. Effectively, KEYS is a treatment learner that also generates and classifies its own data, then continues operating until all settings are fixed.

In the following subsections, I will detail two treatment learning algorithms.

3.2.1 TAR3

TAR3 [46,47,57,61] (and its predecessor TAR2 [74]) are based on two fundamental concepts—lift and support. The *lift* of a treatment is the change that some decision makes to a set of examples after imposing that decision. TAR3 is given a set of training examples E. Each example $e \in E$ contains a set of attributes, each with a specific value (which have commonly been discretized into a series of ranges). These attributes (and the range their values fall within are directly mapped to a specific classification (stated formally - $R_i, R_j, ... \rightarrow C$). The individual class symbols C_1, C_2,... are ranked and sorted based on a utility score ($U_1 < U_2 < ... < U_C$, where U_C is the target class). Within dataset E, these classes occur at certain frequencies (F_1, F_2,..., F_C) where $\sum F_i = 1$ (that is, each class occupies a fraction of the overall dataset). A treatment T of size M is a conjunction of attribute value ranges $R_1 \wedge R_2 ... \wedge R_M$ (these ranges are obtained by discretizing and combining several of the original continuous attribute values). Some subset of the dataset ($e \subseteq E$) is contained within the treatment; that is, if the treatment is used to filter E, $e \subseteq E$ is what will remain. In that subset, the classes occur at frequencies f_1, f_2,..., f_C. TAR3 seeks the smallest treatment T which induces the biggest changes in the weighted sum of the utilities multiplied by the frequencies of the classes. This score, the score of $e \subseteq E$ where T has been imposed, is divided by the score of the baseline (dataset E when no treatment has been applied). Formally, the lift is defined as

$$lift = \frac{\sum_c U_c f_c}{\sum_c U_c F_c}. \tag{3.7}$$

The classes used for treatment learning are assigned a score $U_1 < U_2 < ... < U_C$ and the learner uses this to assess the class frequencies resulting from applying a treatment by finding the subset of the inputs that falls within the reduced treatment space. In normal operation, a treatment learner conducts *controller learning*; that is, it finds a treatment which selects for better classes and rejects worse classes. By reversing the scoring function, treatment learning can also select for the worst classes and reject the better classes. This mode is called *monitor learning* because it locates the

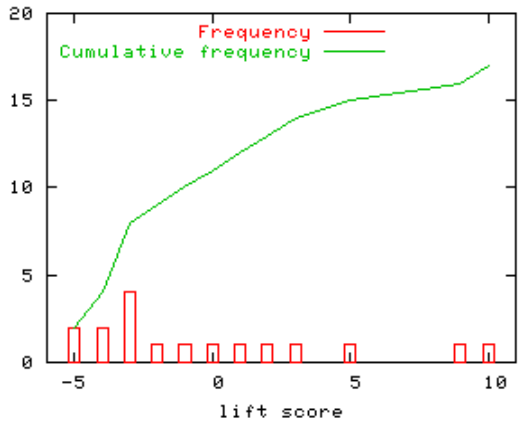

Figure 3.4: Probability distribution of individual attribute scores.

one thing we should most watch for.

Real-world datasets, especially those from hardware systems, contain some *noise* —incorrect or misleading data caused by accidents and miscalculations. If these noisy examples are perfectly correlated with failing examples, the treatment may become overfitted. An overfitted model may come with a massive lift score, but it does not accurately reflect the details of the entire dataset. To avoid overfitting, learners need to adopt a threshold and reject all treatments that fall on the wrong side of this threshold. We define this threshold as the *minimum best support*.

Given the desired class, the best support is the ratio of the frequency of that class within the treatment subset to the frequency of that class in the overall dataset. To avoid overfitting, TAR3 rejects all treatments with best support lower than a user-defined minimum (usually 0.2). As a result, the only treatments returned by TAR3 will have both a high *lift* and a high *best support*. This is also the reason that TAR3 prefers smaller treatments. The fewer rules adopted, the more evidence that will exist supporting those rules.

TAR3's lift and support calculations can assess the effectiveness of a treatment, but they are not what generates the treatments themselves. A naive treatment learner might attempt to test all subsets of all ranges of all of the attributes. Because a dataset of size N had 2^N possible subsets, this type of brute force attempt is inefficient. The art of a good treatment learner is in finding good

heuristics for generating candidate treatments.

The algorithm begins by discretizing every continuous attribute into smaller ranges by sorting their values and dividing them into a set of equally-sized bins. It then assumes the small-treatment effect; that is, it only builds treatments up to a user-defined size. Past research [46, 47] has shown that a treatment's size should be less than four attributes. TAR3 will then only build treatments from the discretized ranges with a high heuristic value.

TAR3 determines which ranges to use by first determining the lift score of each attribute's value ranges (that is, the score of the class distribution obtained by filtering for the data instances that contain a value in that particular range for that particular attribute). These individual scores are then sorted and converted into a cumulative probability distribution, as seen in Figure 3.4. TAR3 randomly selects values from this distribution, meaning that low-scoring ranges are unlikely to be selected. To build a treatment, n (random from 1...max treatment size) ranges are selected and combined. These treatments are then scored and sorted. If no improvement is seen after a certain number of rounds, TAR3 terminates and returns the top treatments.

3.2.2 TAR4.1

TAR3, while effective at generating informative treatments, is not a very efficient algorithm. It stores all examples from the dataset in RAM and requires three scans of the data in order to discretize, build theories, and rank the generated treatments. The TAR4.1 treatment learner was designed to address these inefficiencies [41]. Modeled after the SAWTOOTH [87] incremental Naive Bayes classifier, TAR4.1's scoring heuristic allows for an improved runtime, lower memory usage, and a better ability to scale to large datasets.

Naive Bayes classifiers offer a relationship between fragments of evidence E_i, a prior probabil-

ity for a class $P(H)$, and a posteriori probability $P(H|E)$ defined by

$$P(H|E) = \prod_i P(E_i|H)\frac{P(H)}{P(E)}. \tag{3.8}$$

For numeric features, a features mean μ and standard deviation σ are used in a Gaussian probability function [115]:

$$f(x) = 1/(\sqrt{2\pi}\sigma)e^{-\frac{(x-\mu)^2}{2\sigma^2}}. \tag{3.9}$$

TAR4.1 still requires two passes through the data, for discretization and for building treatments. These two steps function in exactly the same manner as the corresponding steps in the TAR3 learner. TAR4.1, however, eliminates the final pass by building a scoring cache during the BORE classification stage. As explained previously, examples are placed in a U-dimensional hypercube during classification, with one dimension for each utility. Each example $e \in E$ has a normalized distance $0 \leq D_i \leq 1$ from an *apex*, an area where the best examples reside. When BORE classifies examples into *best* and *rest*, that normalized distance is added as a score, called D_i (the euclidean distance from 0), to the *down* table and a seperate score, $1 - D_i$ (or, the distance from the best), is entered into the *up* table.

When treatments are scored by TAR4.1, the algorithm does a linear-time table lookup instead of scanning the entire dataset. Each range $R_j \in example_i$ adds scores $down_i$ and up_i to counters $F(R_j|rest)$ and $F(R_j|best)$. These counters are a summation of scores for a range R_j across the dataset, and represent how often data examples containing that range appear in the *best and rest*.

These summations are then used to compute the following probability and likelihood equations:

$$P(best) = \frac{\sum_i up_i}{\sum_i up_i + \sum_i down_i}, \tag{3.10}$$

$$P(rest) = \frac{\sum_i down_i}{\sum_i up_i + \sum_i down_i}, \tag{3.11}$$

$$P(R_j|best) = \frac{F(R_j|best)}{\sum_i up_i}, \tag{3.12}$$

$$P(R_j|rest) = \frac{F(R_j|rest)}{\sum_i down_i}, \tag{3.13}$$

$$L(best|R_k \wedge R_l \wedge ...) = \prod_x P(R_x|best) * P(best), \tag{3.14}$$

$$L(rest|R_k \wedge R_l \wedge ...) = \prod_x P(R_x|rest) * P(rest). \tag{3.15}$$

TAR4.1 finds the *smallest* treatment T that *maximizes*

$$P(best|T) = \frac{L(best|T)^2}{L(best|T) + L(rest|T)}. \tag{3.16}$$

Note the squared term in the top of the equation, $L(best|T)^2$. The standard Naive Bayes design assumes independence between all attributes and keeps singleton counts. By not squaring that term, TAR4.1 adds redundant information, which alters the generated probabilities. In effect, it produced treatments with high scores, but without the *support* required by the TAR3 algorithm. By squaring that term, the likelihood that a range appears in an area of top scores, those treatments that lack support are pruned in favor of those that have both a good score and support.

It should also be pointed out that Equation 3.16, the formula that TAR4.1 uses to choose treatments, is the same formula used by KEYS to choose optimal model settings (see Section 4.4. Both algorithms judge the scores output by an objective function according to a Bayesian ranking measure. Therefore, they would likely make comparable decisions when applied to the same problems. However, as pointed out earlier, the two algorithms are not directly comparable as TAR4.1 is generally only used in a single-round process and requires data to be pre-classified.

3.3 Summary

In this chapter, I have discussed algorithms and fields of research that are related to the work conducted in this thesis.

The requirements satisfaction problem that I have studied can be though of as a search-based software engineering problem. We want to *search* for the set of mitigations that will allow us to attain the most requirements at the lowest price. KEYS and the algorithms benchmarked against it all reflect aspects of the SBSE field. We use MaxWalkSat, a hybrid random search/ hill climber that utilizes resets to escape local maxima. Our experiments also make use of simulated annealing, which is extremely popular in SBSE research. I have also discussed tabu search, genetic algorithms, particle swarm optimization, and ant colony optimization in detail.

This chapter has discussed the concept of treatment learning, a machine learning method that takes a backlog of classified data, weights the classes by *desirability*, and identifies which features (and which values of those features) are the most pivotal in predicting for a target class. The KEYS algorithm can be thought of as a treatment learner that generates and classifies its own data, and then continues operating until all features have fixed settings.

In fact, the entire treatment learning field can be thought of as an over-elaboration of a very simple idea - that there exist a small number of important variables that, if controlled, lead to optimal solutions. This idea is the very basis of the KEYS algorithm, which I will present in detail in the next chapter.

I will now shift from the general to the specific. The next chapter presents a series of algorithms - taken from various aspects of the SBSE paradigms presented in this chapter - that will be used to address the DDP problem.

Chapter 4

Algorithms

The previous chapters have set forth the DDP requirements satisfaction problem (that is, how to attain the most goals for the lowest price) and provided a survey of the search-based software engineering field. In this chapter, I will discuss the algorithms used to search and optimize the space of model configurations.

Numerous researchers have stressed the difficulties associated with comparing radically different algorithms. For example, Uribe and Stickel [110] doubted that it was fair to compare algorithms that perform constraint satisfaction and no search (like binary decision diagrams) and methods that perform search and no constraint satisfaction (like Davis-Putnam proofs). For this reason, model checking researchers like Holzmann eschew comparisons of tools like SPIN [56], which are search-based, with tools like NuSMV [17], which are BDD-based.

Hence, I take care to only compare algorithms which are similar to my primary approach, KEYS. In terms of Gu's exhaustive optimization survey [44], the algorithms that I have selected (simulated annealing, A* and MaxWalkSat) share certain properties with KEYS and KEYS2. They are each discrete, sequential, *unconstrained* algorithms (constrained algorithms work towards a pre-determined number of possible solutions while unconstrained methods are allowed to adjust to the goal space). These shared properties ensure that all algorithm comparisons are appropriate.

```
1. Procedure SA
2. MITIGATIONS:= set of mitigations
3. SCORE:= score of MITIGATIONS
4. while TIME < MAX_TIME && SCORE < MIN_SCORE //minScore is a constant score (threshold)
5.    find a NEIGHBOR close to MITIGATIONS
6.    NEIGHBOR_SCORE:= score of NEIGHBOR
7.    if NEIGHBOR_SCORE > SCORE
8.        MITIGATIONS:= NEIGHBOR
9.        SCORE:= NEIGHBOR_SCORE
10.   else if prob(SCORE, NEIGHBOR_SCORE, TIME, temp(TIME, MAX_TIME)) > RANDOM)
11.       MITIGATIONS:= NEIGHBOR
12.       SCORE:= NEIGHBOR_SCORE
13.   TIME++
14. end while
15. return MITIGATIONS
```

Figure 4.1: Pseudocode for SA

For full source code of these techniques, please see the appendices.

4.1 Simulated Annealing

Simulated annealing (SA) is a classic stochastic search algorithm. It was first described in 1953 [79] and refined in 1983 [65]. The algorithm's namesake, annealing, is a technique from metallurgy, where a material is heated, then cooled. The heat causes the atoms in the material to wander randomly through different energy states and the cooling process increases the chances of finding a state with a lower energy than the starting position.

During each round, SA "picks" a neighboring set of mitigations. To calculate this neighbor, a function traverses the mitigation settings of the current state and randomly flips those mitigations (at a 5% chance). If the neighbor has a better score, SA will move to it and set it as the current state. If no score improvement is seen, the algorithm will decide whether or not to move based on the probability function:

$$prob(w,x,y,temp(y,z)) = e^{((w-x)*\frac{y}{temp(y,z)})} \tag{4.1}$$

$$temp(y,z) = \frac{(z-y)}{z} \tag{4.2}$$

If the value of the `prob` function is greater than a randomly generated number, SA will move to that state anyways. This randomness is the very cornerstone of the Simulated Annealing algorithm. Initially, the "atoms" (current solutions) will take large random jumps, sometimes to even sub-optimal new solutions. These random jumps allow simulated annealing to sample a large part of the search space, while avoiding being trapped in local minima. Eventually, the "atoms" will cool and stabilize and the search will converge to a simple hill climber.

As shown in line 4 of Figure 4.1, the algorithm will continue to operate until the number of tries is exhausted or a score meets the threshold requirement.

Other uses for the simulated annealing algorithm in the search-based software engineering field are discussed in Section 3.1.1.

4.2 MaxFunWalk (MaxWalkSat)

The design of simulated annealing dates back to the 1950s. In order to benchmark KEYS against a more state-of-the-art algorithm, I implemented the variant of MaxWalkSat described below.

WalkSat is a local search method designed to address the problem of boolean satisfiability [63]. MaxWalkSat is a variant of that algorithm that applies weights to each clause in a conjunctive normal form equation [96]. While WalkSat tries to satisfy the entire set of clauses, MaxWalkSat tries to maximize the sum of the weights of the satisfied clauses.

In one respect, both algorithms can be viewed as a variant of simulated annealing (or the hill climbing approaches seen in Section 3.1.1). Whereas simulated annealing always selects the next solution randomly, the WalkSat algorithms will *sometimes* perform random selection while, other times, conduct a local search to find the next best setting to one variable.

MaxFunWalk is a generalization of MaxWalkSat:

```
1. Procedure MaxFunWalk
2. for TRIES:=1 to MAX-TRIES
3.    SELECTION:=A randomly generate assignment of mitigations
4.    for CHANGED:=1 to MAX-CHANGES
5.        if SCORE satisfies THRESHOLD return
6.        CHOSEN:= a random selection of mitigations from SELECTION
7.        with probability P
8.            flip a random setting in CHOSEN
9.        with probability (P-1)
10.           flip a setting in CHOSEN that maximizes SCORE
11.    end for
12. end for
13. return BESTSCORE
```

Figure 4.2: Pseudocode for MaxFunWalk

- MaxWalkSat is defined over CNF formulae. The success of a collection of variable settings is determined by how many clauses are "satisfiable" (defined using standard boolean truth tables).

- MaxWalkFun, on the other hand, assumes that there exist an arbitrary function that can assess a collection of variable settings. Here, we use the DDP model as a assessment function.

Note that $MaxWalkFun = MaxWalkSat$ if the assessment is conducted via a logical truth table.

The MaxFunWalk procedure is shown in Figure 4.2. When run, the user supplies an ideal cost and attainment. This setting is normalized, scored, and set as a goal threshold. If the current setting of mitigations satisfies that threshold, the algorithm terminates.

MaxFunWalk begins by randomly setting every mitigation. From there, it will attempt to make a *single* change until the threshold is met or the allowed number of changes runs out (100 by default). A random subset of mitigations is chosen and a random number P between 0 and 1 is generated. The value of P will decide the form that the change takes:

- P$\leq \alpha$: A stochastic decision is made. A setting is changed completely at random within the set CHOSEN.

- P$> \alpha$: Local search is utilized. Each mitigation in CHOSEN is tested until one is found that

improves the current score.

The best setting of α is domain-specific. For this study, we used $\alpha = 0.3$.

If the threshold is not met by the time that the allowed number of changes is exhausted, the set of mitigations is completely reset and the algorithm starts over. This measure allows the algorithm to avoid becoming trapped in local maxima. For the DDP models, I found that the number of retries has little effect on solution quality.

If the threshold is never met, MaxFunWalk will reset and continue to make changes until the maximum number of allowed resets is exhausted. At that point, it will return the best settings found.

As an additional measure to improve the results found by MaxFunWalk, a heuristic was implemented to limit the number of mitigations that could be set at one time. If too many are set, the algorithm will turn off a few in an effort to bring the cost factor down while minimizing the effect on the attainment.

4.3 A* Search

A* is a best-first path finding algorithm that uses distance from origin (G) and estimated cost to goal (H) to find the best path [54]. The algorithm is widely used in a multitude of research areas [58, 88, 93, 103].

A* is a natural choice for DDP optimization since the objective function used to assess DDP model configurations (see Section 2.2.3) is actually a Euclidean distance measure to the desired goal of maximum attainment and minimum costs. Hence, for the second potion of the A* heuristic, we can make direct use of Equation 2.2.

The A* algorithm keeps a *closed* list in order to prevent backtracking. We begin by adding the starting state to the closed list. In each "round," a list of neighbors is populated from the series of possible states reached by making a change to a single mitigation. If that neighbor is not on the

```
1. Procedure A*
2. CURRENT_POSITION:= Starting assignment of mitigations
3. CLOSED[0]:= Add starting position to closed list
4.
5. while END:= false
6.    NEIGHBOR_LIST:=list of neighbors
7.    for each NEIGHBOR in NEIGHBOR_LIST
8.        if NEIGHBOR is not in CLOSED
9.            G:=distance from start
10.           H:=distance to goal
11.           F:=G+H
12.           if F<BEST_F
13.               BEST_NEIGHBOR:=NEIGHBOR
14.   end for
15.   CURRENT_POSITION:= BEST_NEIGHBOR
16.   CLOSED[++]:=Add new state to closed list
17.   if STUCK
18.       END:= true
19. end while
20. return CURRENT_POSITION
```

Figure 4.3: Pseudocode for A*

closed list, two calculations are made:

- G = Distance from the start to the current state plus the additional distance between the current state and that neighbor.
- H = Distance from that neighbor to the goal (an ideal spot, usually 0 cost and a high attainment). For DDP models, we use Equation 2.2 to compute H.

The *best* neighbor is the one with the lowest $F = G + H$. The algorithm "travels" to that neighbor and adds it to the closed list. Part of the optimality of the A* algorithm is that the distance to the goal is underestimated. Thus, the final goal is never actually reached by A*. My implementation terminates once it stops finding better solutions for a total of ten rounds. This number was chosen to give ample time for A* to become "unstuck" if it hits a corner early on.

4.4 KEYS and KEYS2

The core premise of KEYS and KEYS2 is that the above algorithms perform over-elaborate searches. Suppose that the behavior of a large system is determined by a small number of *key* variables. If so, then a very rapid search for solutions can be found by (a) finding these *keys* then (b) explore the ranges of the key variables.

The following small example illustrates this idea. Within a model, there are chains of *reasons* linking inputs to desired goals. Some of the links clash with others. Some of those clashes are most upstream; they are not dependent on other clashes. In the following chains of reasoning the clashes are $\{e, \neg e\}$, $\{g, \neg g\}$ & $\{j, \neg j\}$; the most upstream clashes are $\{e\neg e\}$, & $\{g\neg g\}$,

$$
\begin{aligned}
& a \longrightarrow b \longrightarrow c \longrightarrow d \longrightarrow e \\
input_1 \longrightarrow & f \longrightarrow g \longrightarrow h \longrightarrow i \longrightarrow j \longrightarrow goal \\
input_2 \longrightarrow & k \rightarrow \neg g \longrightarrow l \longrightarrow m \rightarrow \neg j \longrightarrow goal \\
& \neg e
\end{aligned}
$$

In order to optimize decision making about this model, we must first decide about these *most upstream clashing reasons* (the "keys"). Returning to the above reasoning chains, any of $\{a, b, ..q\}$ are subject to discussion. However, most of this model is completely irrelevant to the task of $input_i \vdash goal$. For example, the $\{e, \neg e\}$ clash is unimportant to the decision making process as no reason uses e or $\neg e$. In the context of reaching $goal$ from $input_i$, the only important discussions are the clashes $\{g, \neg g, j, \neg j\}$. Further, since $\{j, \neg j\}$ are dependent on $\{g, \neg g\}$, then the core decision must be about variable g with two disputed values: true and false.

Setting the keys reduces the number of reachable states within the model. Formally, the reachable states reduce to the cross-product of all of the ranges of the collars. This is called the *clumping* effect. Only a small fraction of the possible states are actually reachable. The effects of clumping can be quite dramatic. Without knowledge of these keys, the above model has $2^{20} > 1,000,000$

possible consistent states. However, in the context of $input_i \vdash goal$, those 1,000,000 states clumps to just the following two states: $\{input_1, f, g, h, i, j, goal\}$ or $\{input_2, k, \neg g, l, m, \neg j, goal\}$.

As documented in the Section 4.6, this notion of *keys* has been discovered and rediscovered many times by many researchers. The KEYS algorithm finds the keys uses a Bayesian sampling method. If a model contains keys then, by definition, those variables must appear in all solutions to that model. If model outputs are scored by some oracle, then the key variables are those with ranges that occur with very different frequencies in high/low scoring model outputs. Therefore, we need not search for the keys- rather, we just need to keep frequency counts on how often ranges appear in *best* or *rest* outputs.

KEYS has two main components - a greedy search and the BORE (best or rest) ranking heuristic. The greedy search explores a space of M mitigations over the course of M "eras." Initially, the entire set of mitigations is set randomly. During each era, one more mitigation is set to $M_i = X_j$, $X_j \in \{true, false\}$. In the original version of KEYS [59], the greedy search fixes one variable per era. A newer variant, KEYS2, fixes an increasing number of variables as the search progresses (see below for details).

In KEYS (and KEYS2), each era e generates a set $< input, score >$ as follows:

1: $MaxTries$ times repeat:

- $Selected[1 \ldots (e-1)]$ are settings from previous eras.
- *Guessed* are randomly selected values for unfixed mitigations.
- $Input = selected \cup guessed$.
- Call `model` to compute $score = ddp(input)$;

2: The $MaxTries$ *score*s are divided into β% "best" and remainder become "rest".

3: The *input* mitigation values are then scored using BORE (described below).

```
1. Procedure KEYS
2. while FIXED_MITIGATIONS != TOTAL_MITIGATIONS
3.    for I:=1 to 100
4.       SELECTED[1...(I-1)] = best decisions up to this step
5.       GUESSED = random settings to the remaining mitigations
6.       INPUT = SELECTED + GUESSED
7.       SCORES= SCORE(INPUT)
8.    end for
9.    for J:=1 to NUM_MITIGATIONS_TO_SET
10.      TOP_MITIGATION = BORE(SCORES)
11.      SELECTED[FIXED_MITIGATIONS++] = TOP_MITIGATION
12.   end for
13. end while
14. return SELECTED
```

Figure 4.4: Pseudocode for KEYS

4: The top ranked mitigations (the default is one, but the user may fix multiple mitigations at once) are fixed and stored in $selected[e]$.

The search moves to era $e+1$ and repeats steps 1,2,3,4. This process stops when every mitigation has a setting. The exact settings for $MaxTries$ and β must be set via engineering judgment. After some experimentation, we used $MaxTries = 100$ and $\beta = 10$. For full details, see Figure 4.4.

KEYS ranks mitigations using a support-based Bayesian ranking measure called BORE. BORE [18] (short for "best or rest") divides numeric scores seen over K runs and stores the top 10% in *best* and the remaining 90% scores in the set *rest* (the *best* set is computed by studying the delta of each score to the best score seen in any era). It then computes the probability that a value is found in *best* using Bayes theorem. The theorem uses evidence E and a prior probability $P(H)$ for hypothesis $H \in \{best, rest\}$, to calculate a posteriori probability $P(H|E) = P(E|H)P(H)\ /\ P(E)$. When applying the theorem, *likelihoods* are computed from observed frequencies. These likelihoods (called "like" below) are then normalized to create probabilities. This normalization cancels out $P(E)$ in Bayes theorem. For example, after $K = 10,000$ runs are divided into 1,000 *best* solutions and 9,000 *rest*, the value $mitigation31 = false$ might appear 10 times in the *best* solutions, but only 5 times in the *rest*. Hence:

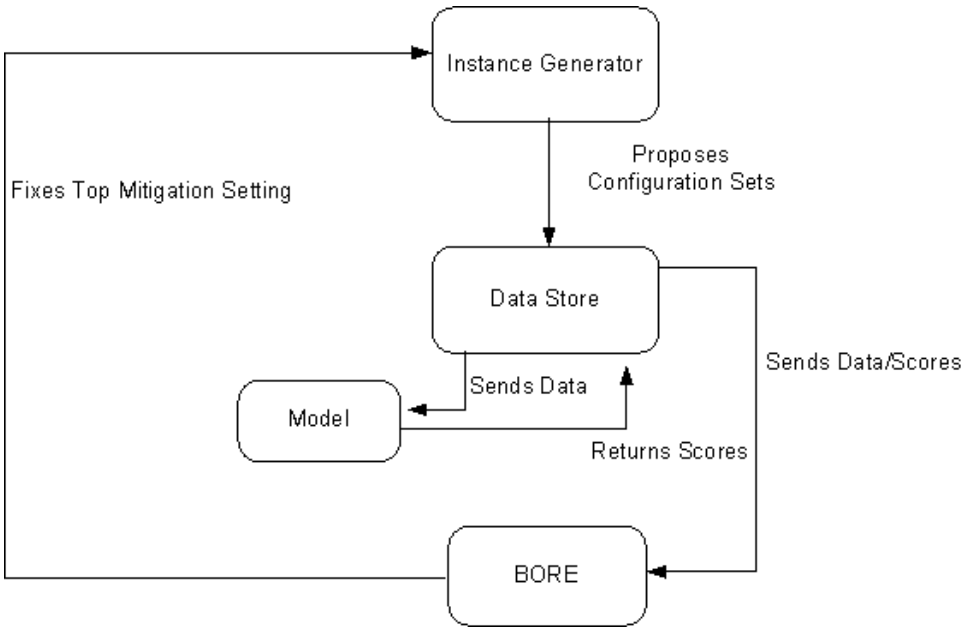

Figure 4.5: The data architecture of the KEYS algorithm.

$$\begin{aligned}
E &= (mitigation31 = false) \\
P(best) &= 1000/10000 = 0.1 \\
P(rest) &= 9000/10000 = 0.9 \\
freq(E|best) &= 10/1000 = 0.01 \\
freq(E|rest) &= 5/9000 = 0.00056 \\
like(best|E) &= freq(E|best) \cdot P(best) = 0.001 \\
like(rest|E) &= freq(E|rest) \cdot P(rest) = 0.000504 \\
P(best|E) &= \frac{like(best|E)}{like(best|E) + like(rest|E)} = 0.66
\end{aligned} \tag{4.3}$$

Previously [18], it has been found that Bayes theorem is a poor ranking heuristic since it is easily distracted by low frequency evidence. For example, note how the probability of E belonging to the best class is moderately high even though its support is very low; i.e. $P(best|E) = 0.66$ but $freq(E|best) = 0.01$.

To avoid the problem of unreliable low frequency evidence, I augment Equation 4.3 with a

support term. Support should *increase* as the frequency of a value *increases*, i.e. $like(best|E)$ is a valid support measure. Hence, step 3 of the greedy search ranks values via

$$P(best|E) * support(best|E) = \frac{like(best|E)^2}{like(best|E) + like(rest|E)} \tag{4.4}$$

During each era, KEYS samples the DDP models and fixes the top $N = 1$ settings. KEYS2 assigns progressively larger values. In era 1, KEYS2 behaves exactly the same as KEYS while in (say) era 3, KEYS2 will fix the top 3 ranked ranges. Then, in era 4, KEYS2 will fix the top four ranges. Since it sets more variables at each era, KEYS2 terminates earlier than KEYS. A summary of the architecture of KEYS can be seen in Figure 4.5. Note that decision ordering diagrams could be directly generated during execution, just by collection statistics from the `SCORES` array used in line 7 of Figure 4.4.

4.5 Other KEYS Variants

KEYS was orders of magnitude faster than prior treatment learning solutions to the DDP requirements problem [78]. While other algorithms would take minutes to execute, KEYS is capable of optimizing the largest models in a fraction of a second. While this is adequate for current requirements models, it is always desirable to find faster solutions. Therefore, it became necessary to research methods that would speed up KEYS' current performance.

Several different methods and rates have been tested for fixing multiple settings per round. KEYS2, presented in the previous section, was found to be the best of these for its balance of execution time and low result variance. KEYS2 was not the first attempt to improve the performance of the original KEYS, nor was it the last. Some of the other heuristics benchmarked were:

- Fix all mitigations with a performance score within 5% of the top score for that round.
- Set all mitigations that are correlated (i.e. that have the same frequency in the "best" in-

stances) to the top-performing configuration.

- Generate fewer random configurations each round. That is, on the first round, it would generate N configurations ($N >=$ number of mitigations). On the second round, it would generate $N-1$ instances.

- Introduce a sloping threshold for the split between the *best* and *rest* instances.

- On the first round, fix a mitigation as KEYS normally would. At the same time, this variant would populate a list of similar-scoring instances. The following round, it would only look at those instances instead of generating new ones. On the third round, it would form a new list of ”neighboring” instances and consider those for the fourth round. This continues until all mitigations have fixed settings.

- A KEYS variant combining the above *neighbor* concept with KEYS2, setting a larger number of mitigation values each round.

In benchmarking tests, some of these were faster than KEYS2, but those variants returned either sub-par performance or too much variance in their judgments (in the language of the decision ordering diagrams, they were *well-behaved* but not *tame*). Other variants represented too small of a runtime improvement over KEYS. Ultimately, KEYS2's heuristic was selected as the most effective.

4.5.1 KEYS-R

While KEYS2 represents a large runtime improvement over the original KEYS, it contains a flaw that limits future performance improvements. By setting multiple mitigations each round, the algorithm loses the ”support” built by the one-hundred instances generated each round. Therefore, while KEYS2 is faster than the original KEYS, its results are less tame.

The small amount of additional variance in KEYS2 is still within acceptable boundaries; therefore, it has found some use in DDP-related experiments. However, this has limited the further improvements that could be made to the KEYS algorithm. Numerous other variations on how to set multiple mitigations per round have shown result variance far outside acceptable values. Other methods have not shown variance problems, but these versions were slower than KEYS2, limiting their possible use. This issue prompted a new round of research. If the key to speeding up the KEYS algorithm was not in setting additional mitigations each round, where was it?

When a software profiler was used on KEYS2, one interesting point was revealed. Nearly *70%* of the runtime was spent in calling the model (i.e. sending a new set of mitigations to the precompiled model and receiving back a new cost and attainment value). Each round, both KEYS and KEYS2 generate one-hundred sets of mitigations and score them. They use these instances to find the best set of mitigations to fix for that round. The instance generation step is the chief bottleneck in the KEYS algorithm. However, a large number of instances is needed to provide the *support* necessary to accurately rank mitigations. If fewer instances are generated, you have the same issue as when too many mitigations are set. There is a larger variance in results, and those results tend to contain higher cost and lower attainment values. Therefore, the question is how to call the model *fewer* times while retaining *high* support for the instance set.

A new variant of the KEYS algorithm, KEYS-R addresses this issue by retaining certain subsets of previously-generated instances and reusing them in later rounds. KEYS-R begins by calculating two *threshold* values. These values are the highest possible cost (i.e. the cost if all mitigations are used) and the highest possible attainment. The central loop (instance generation, ranking, and mitigation fixing) commence in the same way as the original KEYS. One hundred new instances are generated and stored. The key difference comes in the next step. Once an instance is scored, its cost and attainment values are compared to the previously-calculated threshold values. If an instance scores within ten percent of these values (if its cost is less than 10% of the highest possible cost and its attainment is greater than 90% of the model's possible attainment), it is stored in a

collection of the *best* instances. If the instance's cost value is within 10% to 35% of the cost threshold and its attainment is from 65% to 90% of the possible attainment value, it is stored in a collection of *middle* instances.

Each round, before a new set of instances is generated, a random subset of the *top* and *middle* sets are used instead of generating new instances. In order to ensure that support is maintained, new instances must be generated. Therefore, up to 33 instances can come from each set, and the remaining 34 will be randomly generated new instances.

As individual mitigations are fixed, these previously used instances will begin to lose their support value. That is to say, results will lose their rigidity and variance will increase. Therefore, it is still important that we not reuse an instance too many times. A balance must be maintained between the stored instances and the newly-generated ones. To address this, a decay timer is attached to each instance when it is added to either the *top* or *middle* subsets. Whenever a new instance is added to once of these subsets, it will immediately replace an instance whose decay timer has expired. This allows support to be maintained while reducing the number of new instances that must be generated each round.

Benchmarking experiments assessing KEYS-R can be seen in Section 5.3.

4.6 Other Work on "Keys"

The core premise of the KEYS algorithm is that a small number of important variables effectively set the rest, greatly reducing the search space. Elsewhere [77], dozens of papers that have reported this same effect under a variety of names, including *narrows, master-variables, back doors,* and *feature subset selection*.

Amarel [5] observed that search problems contain narrow sets of variables or collars that must be used in any solution. In such a search space, what matters is not so much how you get to these collars, but what decision you make when you get there. Amarel defined macros encoding paths

between *narrows*, effectively permitting a search engine to jump between them.

In a similar theoretical analysis, Menzies & Singh [77] computed the odds of a system selecting solutions to goals using complex, or simpler, sets of preconditions. In their simulations, they found that a system will naturally select for tiny sets of preconditions (a.k.a. the keys) at a very high probability.

Numerous researchers have examined *feature subset selection*; i.e. what happens when a data miner deliberately ignores some of the variables in the training data. For example, Kohavi and John [66] showed in numerous datasets that as few as 20% of the variables are *key* - the remaining 80% of variables can be ignored without degrading a learner's classification accuracy.

Williams et.al. [113] discuss how to use keys (which they call "back doors") to optimize search. Constraining these back doors also constrains the rest of the program. So, to quickly search a program, they suggest imposing some set values on the key variables. They showed that setting the keys can reduce the solution time of certain hard problems from exponential to polytime, provided that the keys can be cheaply located, an issue on which Williams et.al. are curiously silent.

Crawford and Baker [24] compared the performance of a complete TABLEAU prover to a very simple randomized search engine called ISAMP. Both algorithms assign a value to one variable, then infer some consequence of that assignment with forward checking. If contradictions are detected, TABLEAU backtracks while ISAMP simply starts over and re-assigns other variables randomly. Incredibly, ISAMP took *less* time than TABLEAU to find *more* solutions using just a small number of tries. Crawford and Baker hypothesized that a small set of *master variables* set the rest and that solutions are not uniformly distributed throughout the search space. TABLEAU's depth-first search sometimes drove the algorithm into regions containing no solutions. On the other hand, ISAMP's randomized sampling effectively searches in a smaller space.

In summary, this "keys" assumption, that a small subset of variables are far more important than all others, is supported in many different domains.

4.7 Summary

This chapter has discussed a series of algorithm implementations intended to offer ideal solutions to the DDP problem (that is, a balance between the budget and the attainment of goals). Each algorithm chosen for my experiments is discrete, sequential, and unconstrained, and thus, is appropriate for comparison to the others. These methods all operate in different ways - A* treats the search space as a graph, while simulated annealing jumps between solutions based on a probability function - but they all utilize the same objective function and offer compatible results.

In this chapter, I have also presents my own approach to the problem - the KEYS (and KEYS2) algorithm. This technique focuses on the idea that a small number of variables have a disproportionate influence over the search space, and by setting those variables, you can quickly force the results into a somewhat-optimal area of the search space. There is much evidence for such a notion in the data mining field, as shown in Section 4.6.

The next chapter details a series of case studies benchmarking the algorithms presented in this chapter.

Chapter 5

Case Studies

Previous chapters have discussed the DDP models and a series of algorithms designed to comment on an optimal combination of settings to these models. Chapter 4 has also discussed external research relevant to this thesis.

This chapter presents a series of experiments where we benchmark KEYS and KEYS2 against andard techniques for solving the DDP requirements satisfaction problem. I will also detail scale-up results that show KEYS' ability to handle models multiple times larger than those seen in the present day. Lastly, I will show the results of one attempt to develop a more powerful variant of KEYS2.

5.1 Benchmarking KEYS and KEYS2

Each of the algorithms described in Chapter 3 was tested on the five models of Figure 2.4. Note that:

- Models one and three are trivially small. They were used them to debug source code, but have been left out the core experiments. I report my results using models two, four and five since they are large enough to stress test real-time optimization.

- Model 4 was discussed in [75] in detail. The largest, model 5 was optimized previously in [36]. At that time (2002), it took 300 seconds to generate solutions using an old, very slow, rule learning method.

- All five models were assessed by the KEYS algorithm in [59]. However, that paper presented no comparison results.

I have also studied how well KEYS and KEYS2 scale to larger models. Further, I have instrumented KEYS and KEYS2 to generate decision ordering diagrams. The results from all of these experiments are shown below.

5.1.1 Attainment and Costs

All of the comparison algorithms (KEYS, KEYS2, simulated annealing, A*, and MaxFunWalk) were run 1000 times on each model. Such a high number of repeats was chosen because it yielded enough data points to give a clear picture of the span of results, and ensured that outlying values would have no effect on the experiment outcomes. At the same time, it is a low enough number that I can still generate a complete set of results in a fairly short time span.

The results are pictured in Figure 5.1. Attainment is along the x-axis and cost (in thousands) is along the y-axis. Note that better solutions fall towards the bottom right of each plot; that is, the area of lower costs and higher attainment. Also, better solutions exhibit less variance and, as such, the results are clumped closely together.

These graphs give a clear picture of the results obtained by the various algorithms. Two methods are clearly inferior:

- Simulated annealing exhibits the worst variance, lowest attainments, and highest costs.

- MaxFunWalk is better than SA (less variance, lower costs, higher attainment) but its variance is still far too high to use in any sort of critical situation.

As for the other techniques:

- On larger models such as model4 and model5, KEYS and KEYS2 exhibit lower variance, lower costs, and higher attainments than A*.
- On smaller models such as model2, A* usually produces higher attainments and lower variance than the KEYS algorithms (this advantage disappears on the larger models). However, observe the results near the $(0,0)$ point of model2's A* results: sometimes A*'s heuristic search failed completely for that model.

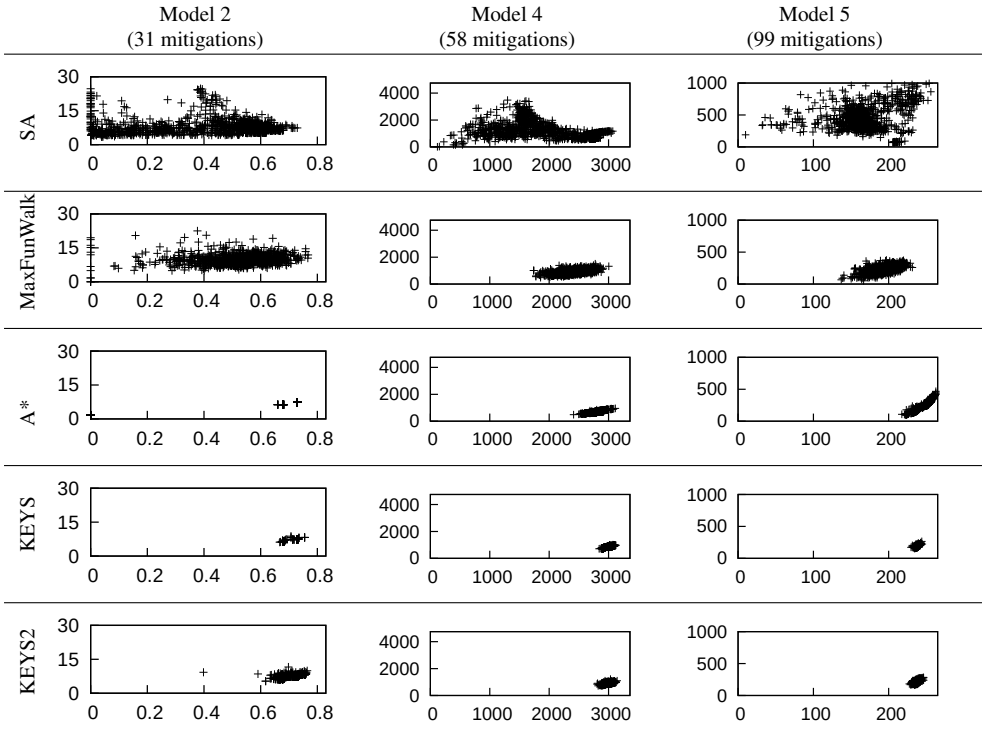

Figure 5.1: 1000 results of running five algorithms on three models (15,000 runs in all). The y-axis shows cost and the x-axis shows attainment. The size of each model is measured in number of mitigations. Note that better solutions fall towards the bottom right of each plot; i.e. lower costs and higher attainment. Also better solutions exhibit less variance, i.e. are clumped tighter together.

	Model 2 (31 mitigations)	Model 4 (58 mitigations)	Model 5 (99 mitigations)
SA	0.577	1.258	0.854
MaxFunWalk	0.122	0.429	0.398
A*	0.003	0.017	0.048
KEYS	0.011	0.053	0.115
KEYS2	0.006	0.018	0.038

Figure 5.2: Runtimes in seconds, averaged over 100 runs, measured using the "real time" value from the Unix `times` command. The size of each model is measured in number of mitigations (and for more details on model size, see Figure 2.4).

5.1.2 Runtime Analysis

Measured in terms of attainment and cost, there is little difference between KEYS and KEYS2. However, as shown by Figure 5.2, KEYS2 runs twice to three times as fast as its predecessor. Interestingly, Figure 5.2 ranks two of the algorithms in a similar order to Figure 5.1:

- Simulated annealing is clearly the slowest;
- MaxFunWalk is somewhat better but not as fast as the other algorithms.

As to A* versus KEYS or KEYS2:

- A* is faster than KEYS;
- and KEYS2 runs in time comparable to A*.

Measured purely in terms of runtimes, there is little to recommend KEYS2 over A*. However, A*'s heuristic guesses were sometimes observed to be sub-optimal (recall the above discussion on the $(0,0)$ results in model2's A* results). Such sub-optimality was never observed for KEYS2.

5.1.3 Decision Ordering Algorithms

The decision ordering diagrams of Figure 5.3 show the effects of the decisions made by KEYS and KEYS2. For both algorithms, at $x = 0$, all of the mitigations in the model are set at random. During each subsequent era, more mitigations are fixed (KEYS sets one at a time, KEYS2 freezes

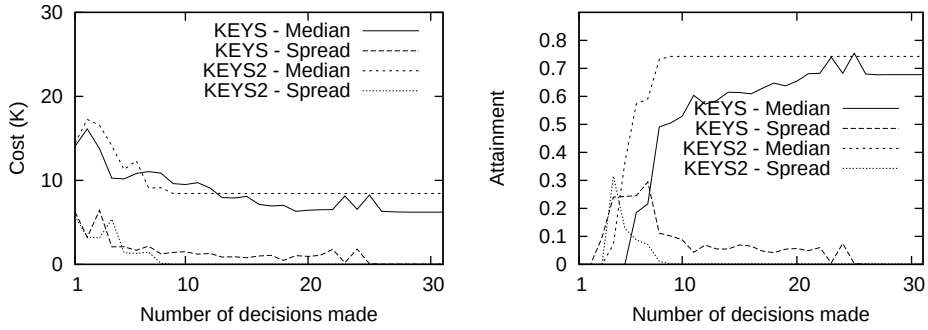

Figure 5.3a: Internal Decisions on Model 2.

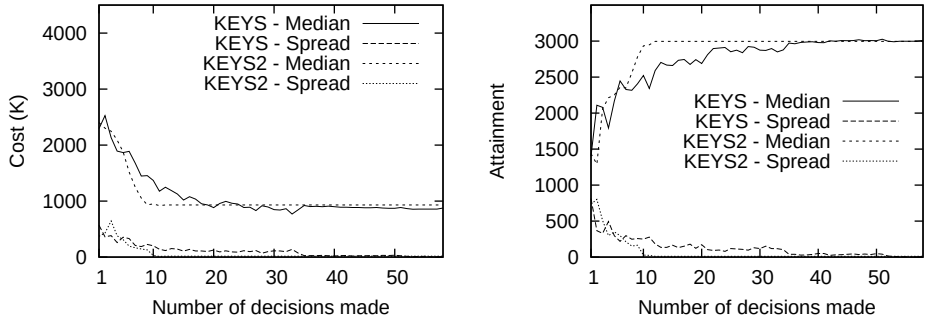

Figure 5.3b: Internal Decisions on Model 4.

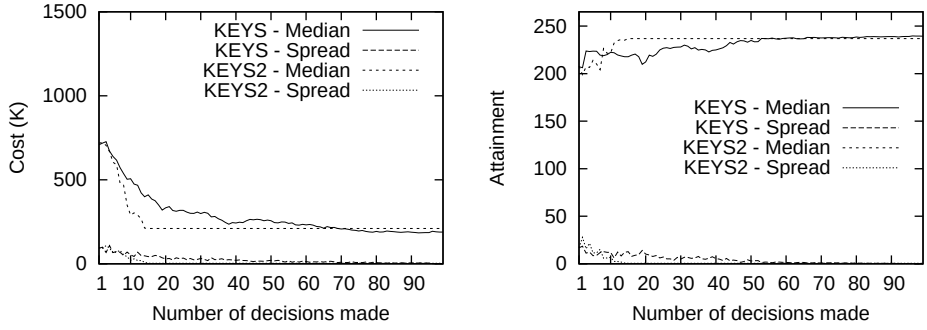

Figure 5.3c: Internal Decisions on Model 5.

Figure 5.3: Median and spread of partial solutions learned by KEYS and KEYS2. X-axis shows the number of decisions made. "Median" shows the 50th percentile of the measured value seen in 100 runs at each era. "Spread" shows the difference between the 75th and 50th percentile.

an incrementally increasing number). The lines in each of these plots show the median and spread seen in the one-hundred calls to the `model` function during each round.

Note that the these diagrams are both *tame* and *well-behaved*, as defined in Chapter 2:

- *Tame:* The "spread" values quickly shrink to a small fraction of the median.
- *Well-behaved:* The median values move smoothly to a plateau of best performance (high attainment, low costs).

On termination (at maximum value of x), KEYS and KEYS2 arrive at nearly identical median results (caveat: for model2, KEYS2 attains slightly more requirements at a slightly higher cost than KEYS). The spread plots for both algorithms are almost indistinguishable except, again, for model 2. In that model, the KEYS2 spread is less than KEYS. Based on these diagrams, we can observe that KEYS2 achieves approximately the same results as KEYS, but (as shown in Figure 5.2 and Figure 5.5) it does so in less time.

A core assumption of this work is the "keys" concept - that a small minority of important model variables restrict the remaining majority. Figure 5.3 offers significant support for this assumption. It can be observed that most of the improvement in costs and attainments were achieved after KEYS and KEYS2 made only a handful of decisions (often ten or fewer).

It is insightful to reflect on the effectiveness of different algorithms at generating these decision ordering diagrams. KEYS2 is the most direct and fastest method, and by its very design, automatically generates these diagrams. As mentioned above, all of the required information can be collected during one execution of KEYS2. On the other hand, simulated annealing, A*, and MaxFunWalk would require a post-processor to generate the diagrams:

- Given D possible decisions, At each era, KEYS and KEYS2 collects statistics on partial solutions where $1, 2, 3, ..|d|$ variables are fixed (where d is the set of decisions) while the remaining $D - d$ decisions are made at random.

Year	Num. Variables
2004	30
2008	100
2010	300
2013	800

Figure 5.4: Growth trends; number of variables in DDP's data dictionary.

- A*, Simulated Annealing, and MaxFunWalk work with full solutions since at each step they offer settings to all $d_i \in D$ variables. In the current form, they cannot comment on partial solutions. Modified forms of these algorithms could theoretically add in extra instrumentation and extra post-processing to comment on partial solutions using methods like feature subset selection [48] or a sensitivity analysis [95].

5.2 Scale-Up Studies

In order to provide useful blind-spot exploration, our DDP optimization tools must run fast enough that humans can get feedback before they must move on to other issues. Our previous solutions to the DDP optimization problem are too slow. Ideally, we wish for requirements optimization to occur in "real-time"; i.e. to offer advice before an expert's attention wanders to other issues.

Figure 5.4 shows the growth rate (historically and projected) of the number of variables within a DDP model. It is expected that, by 2013, DDP models will be roughly eight times larger than they were in 2008 and twice as large as they are today. As shown in Figure 5.2, KEYS and KEYS2 report results that could be considered "real-time." However, it remains unclear whether they scale to the complex models that we expect to see in the near future?

In order to study how well the scalability of KEYS and KEYS2, an *instance generator* was created that:

- Examined the real-world DDP models of Figure 2.4;
- Extracted statistics related to the different types of nodes (mitigations or risks or require-

model	expansion	model size	Runtime (secs)		$\frac{KEYS}{KEYS2}$
			KEYS	KEYS2	
2	1	62	0.01	0.01	1.07
2	2	124	0.03	0.02	1.23
4	1	139	0.04	0.02	2.29
5	1	201	0.13	0.04	3.18
2	4	248	0.10	0.05	2.09
4	2	278	0.17	0.05	3.48
5	2	402	0.50	0.12	4.26
2	8	496	0.44	0.14	3.21
4	4	556	0.73	0.16	4.66
5	4	804	1.98	0.38	5.21
4	8	1112	2.97	0.52	5.71
5	8	1608	8.06	1.35	5.96

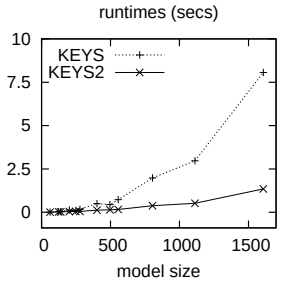

Figure 5.5: Runtimes KEYS vs KEYS2 (medians over 1000 repeats) as models increase in size. The "model" number in column one corresponds to Figure 2.4. The "expansion factor" of column two shows how much the instance generator expanded the model. The "model sizes" of column three are the sum of mitigations, requirements, and risks seen in the expanded model.

ments) and the number of edges between different types of nodes;

- Used those statistics to build random models that were 2,4,8, and 16 times larger than the original models.

For more on the model generator, see Section 2.2.2. It is worth noting that, for the purposes of this scale-up study, I have actually stepped ahead of the expected growth curve. The artificial models generated for this contain up to 1600 mitigations (twice as large as those that JPL is expected to be designing in 2013, and sixteen times larger than the largest models studied in this thesis).

Figure 5.5 and Figure 5.6 show the effect of changing the size of the model on the number of times that the model is asked to generate a score for both KEYS and KEYS2.

Figure 5.7 shows the results of curve fitting to the plots of Figure 5.5 and Figure 5.6. The KEYS and KEYS2 performance curves fit a low-order polynomial (of degree two) with very high coefficients of determination ($R^2 \geq 0.98$). That is to say, the KEYS2 algorithm is of complexity size $O(N^2)$.

The evidence in Figure 5.7 suggests that one could scale *either* KEYS or KEYS2 to larger models without an inordinate increase in execution time. However, I would still have to recommend KEYS2. The column marked $\frac{KEYS}{KEYS2}$ in Figure 5.6 shows the ratio of the number of calls made by

model	expansion	model size	Calls to `model`		$\frac{\text{KEYS}}{\text{KEYS2}}$
			KEYS	KEYS2	
2	1	62	3100	800	3.9
2	2	124	6200	1100	5.6
4	1	139	5800	1100	5.3
5	1	201	9900	1400	7.1
2	4	248	12400	1600	7.8
4	2	278	11600	1500	7.7
5	2	402	19800	2000	9.9
2	8	496	24800	2200	11.3
4	4	556	23200	2200	10.5
5	4	804	39600	2800	14.1
4	8	1112	46400	3000	15.5
5	8	1608	79200	4000	19.8

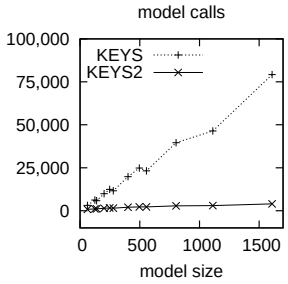

Figure 5.6: Number of model calls made by KEYS vs KEYS2 (medians over 1000 results) as models increase in size. This figure uses the same column structure as Figure 5.5.

	KEYS		KEYS2	
	runtimes	model calls	runtimes	model calls
exponential	0.82	0.83	0.88	0.93
polynomial (of degree 2)	0.99	0.99	0.99	0.98

Figure 5.7: Coefficients of determination R^2 of KEYS/KEYS2 performance figures, fitted to two different functions: exponential or polynomial of degree two. Higher values indicate a better curve fit. In all cases, the best fit is not exponential.

KEYS vs KEYS2. As models get larger, the number of calls to the model are an order of magnitude greater in KEYS than in KEYS2. If applied to models with slower runtimes than those current DDP models, then this order of magnitude is highly undesirable.

5.3 Benchmarking KEYS-R

While KEYS2 represents a speed upgrade over the original KEYS, it still has one clear limitation. By setting multiple mitigations each round, the algorithm loses the ”support” built by the one-hundred instances generated each round. Therefore, while KEYS2 is faster than the original KEYS, its results contain slightly more *variance*. Other attempts to increase the number of mitigations set per round have resulted in variance outside of useful bounds. When a software profiler was used on KEYS2, one interesting point was revealed. Nearly *70%* of the runtime was spent in asking the model for a new $(cost, attainment)$ performance pairing. Each round, both KEYS and KEYS2

generate one-hundred random sets of mitigation settings and score them, using these scores to find mitigations to fix for subsequent rounds (thus, the model is called one hundred times each round). This instance generation step is the chief bottleneck in the KEYS algorithm. However, one cannot just cut the number of random configurations generated at each step - a large number of instances is needed to provide the *support* necessary to accurately rank mitigations. If fewer instances are generated, the same issue arises as when too many mitigations are fixed. There is a larger variance in results, and those results tend to contain higher cost and lower attainment values. KEYS-R, as explained in Section 4.5.1, was developed as an attempt to find a balance between speed and support. It retains certain high-scoring instances and attempts to reuse them in later rounds.

For two of the models from the original experiment (models 2 and 4), as well as several of the artificially-generated models from Sections 2.2.2 and 5.2, I ran each KEYS, KEYS2, and KEYS-R one-hundred times in order to filter for outliers and obtain stable results. For each algorithm and model, the final attainment, cost, and runtimes were recorded. The results are presented in the following subsections.

5.3.1 Attainment and Costs

As expected, the original KEYS obtains the absolute best balance between cost and attainment, with KEYS2 performing similarly. Unfortunately, KEYS-R does not do very well in comparison to KEYS2. KEYS-R returns similar attainments, but much higher cost values. In certain extreme cases, like artificial model 5-8, it returns a cost estimate three times higher the value given by the original KEYS. These median results were compared using the Mann-Whitney U tests, only to reach the same conclusion. KEYS slightly outperforms KEYS2, and both outperform KEYS-R.

The results that consistently emerged for each model, as output by a Mann-Whitney test were:

- Two wins for KEYS;
- One win and one loss for KEYS2;

Original:

Model	KEYS	KEYS2	KEYS-R
model2.c	7325	7325	6250
model4.c	876500	921500	711500

Artificial:

Model	KEYS	KEYS2	KEYS-R
model2-2.c	0	140	140
model2-4.c	125	710	1055
model2-8.c	1080	3435	5540
model2-16.c	10815	22030	35050
model4-2.c	1000	36500	39500
model4-4.c	35500	146000	226000
model4-8.c	220000	577000	670000
model4-16c	1422500	2733000	5285000
model5-2.c	13575	43050	59500
model5-4.c	93000	228000	325500
model5-8.c	430050	796650	1233575

Figure 5.8: Median cost results for each of our algorithms run on each model.

Original:

Model	KEYS	KEYS2	KEYS-R
model2.c	0.7	0.7	0.7
model4.c	3013	3011	2858

Artificial:

Model	KEYS	KEYS2	KEYS-R
model2-2.c	54	54	54
model2-4.c	112	112	112
model2-8.c	228	228	228
model2-16.c	460	460	460
model4-2.c	3835	3835	3835
model4-4.c	8727	8727	8727
model4-8.c	18492	18492	18492
model4-16c	34975	34975	34975
model5-2.c	523	523	523
model5-4.c	845	845	845
model5-8.c	1788	1788	1788

Figure 5.9: Median attainment results for each of our algorithms run on each model.

- Two losses for KEYS-R.

What is occurring here is that KEYS-R, though designed to maintain support, clearly fails to do so. In the best case, it is likely that the rules governing instance decay were too simple. It is also equally possible that the entire concept of instance retention is flawed - configurations that scored well at one point just might not be useful later in the search process.

One point of interest is that KEYS-R performed very differently on the original models as opposed to the artificially generated ones. The results are not necessarly better, they have just gone in the opposite direction. When executed on the original models, KEYS-R returns pairings with lower costs than KEYS or KEYS2 (this is good) but lower attainments as well (this is bad). The rules governing instance retention were perhaps too loose for the artificially-generated models and too strict on the original models. Very few instances are scored as ”best” (a subset of instances retained within the KEYS-R algorithm). For example, in one trial involving model 4 from the original set, only three instances were ever stored in the best subset. In the artificial version of that model, one-hundred (the limit before decayed instances are replaced) examples are stored. This implies that KEYS-R is very dependent on the actual construction of the model. Things like the number of connections between specific risks and mitigations seem to drastically affect the results that appear.

5.3.2 Runtime Analysis

As Figure 5.10 shows, KEYS-R falls right between KEYS and KEYS2 in terms of runtime. This is crucial, as standard KEYS is too slow to provide realtime results on the largest of our artificial models (in one case, taking almost nine seconds).

KEYS2 and KEYS-R outperform the original KEYS on every model. However, the fact that is *does not* run faster than KEYS2, and that it returns sub-optimal cost and attainment pairings, leaves little reason to recommend it over the existing solutions to DDP model optimization. Un-

Original:

Model	KEYS	KEYS2	KEYS-R
model2.c	0.00881	0.00497	0.00838
model4.c	0.04724	0.01523	0.03763

Artificial:

Model	KEYS	KEYS2	KEYS-R
model2-1.c	0.00833	0.00775	0.00671
model2-2.c	0.02720	0.02216	0.0197
model2-4.c	0.10398	0.04964	0.07067
model2-8.c	0.44088	0.13736	0.30278
model2-16.c	1.76741	0.48072	1.29550
model4-1.c	0.04375	0.01911	0.02502
model4-2.c	0.17386	0.05003	0.09178
model4-4.c	0.73049	0.15686	0.39623
model4-8.c	2.97024	0.59174	1.83779
model4-16c	12.60780	1.88267	8.8363
model5-1.c	0.12882	0.04055	0.07276
model5-2.c	0.49749	0.11666	0.29788
model5-4.c	1.97696	0.37918	1.34306
model5-8.c	8.05896	1.35119	5.97226

Figure 5.10: Runtime results for each of our algorithms run on each model.

less lingering issues with its instance-retention model were addressed, it is unlikely that KEYS-R could be considered as a replacement for KEYS2. Future directions for KEYS research will likely involve quick-caching methods (elaborated on in Section 6.2) rather than any form of full instance retention.

5.4 Summary

The experiments shown in this chapter have shown that when assessed in terms of (a) reducing inference times, (b) increasing solution quality, and (c) decreasing the variance of the generated solution, KEYS2 out-performs other search algorithms including simulated annealing, A*, MaxWalkSat.

KEYS and KEYS2 are extremely fast algorithms, and I created a series of increasingly large synthetic models to test their ability to scale to the complex projects of the future. The second

experiment shows that KEYS2 will scale to models up to sixteen times larger than those currently in existence with no discernible issues.

Finally, I have compared KEYS and KEYS2 to a new version of KEYS designed to retain certain ”good” instances (called KEYS-R). Unfortunately, the KEYS-R experiment yielded poor results, with both KEYS and KEYS2 demonstrating higher-quality $(cost, attainment)$ pairings (and KEYS2 demonstrating faster execution times).

The next chapter offers a more detailed analysis of these results, as well as potential plans for future work.

Chapter 6

Conclusions

6.1 Conclusions

Requirements tools such as the Defect Detection and Prevention model (used at NASA for early life-cycle discussions), contain a shared group memory that stores all of the requirements, risks, and mitigations of each member of this group. Software tools can explore this shared memory to find consequences and interactions that may have been overlooked.

Studying that group memory is a non-linear optimization task: possible benefits (like the attainment of requirements) must be traded off against the increased cost of applying various mitigations. Harman [50] cautions that solutions to non-linear problems may be "brittle." That is, small changes to the search results may dramatically alter the effectiveness of the solution. Hence, when reporting an analysis of this shared group memory, it is vitally important to comment on the robustness of the solution.

Decision ordering diagrams (Sections 2.2.4 and 5.1.3) are one such solution robustness assessment method. These diagrams rank all of the possible decisions from most-to-least influential. Each point x on the diagrams shows the effects on imposing the conjunction of decisions $1 \leq j \leq x$. These diagrams can comment on the robustness and neighborhood of solution $\{d_1..d_x\}$ using two

operators:

1. By considering the variance of the performance statistics after applying $\{d_1..d_x\}$.

2. By comparing the results of using the first x decisions to that of using the first $x-1$ or $x+1$ actions.

Since the decisions are sorted by importance, this analysis of robustness and neighborhood takes, at most, time linear of the number of decisions made. That is, theoretically, it takes linear time to *use* a decision ordering diagram (see 2.2.4).

Empirically, it take low-order polynomial time to *generate* a decision ordering diagram using the KEYS2 algorithm. This algorithm makes the "keys" assumption - that a small group of variables control all of the others - and uses Bayesian ranking mechanism to quickly find those keys. As discussed in the Section 4.6, this assumption holds over a wide range of models used in a wide range of domains. The keys assumption can be remarkably effective: empirical results show that KEY2 can generate decision ordering diagrams faster than both standard and state-of-the art techniques in the search field (Section 5.1). Better yet, curve fits to our empirical results in Section 5.2 show that KEYS runs in $O(N^2)$ and, thus, should scale to very large models.

Prior to this work, two pre-experimental concerns were that:

- An algorithm would need to trade solution robustness against solution quality. More robust solutions may not have the highest quality.

- Demonstrating solution robustness requires multiple calls to an analysis procedure.

At least for the models studied here, neither concern was realized. KEYS2 generated the highest quality solutions (lowest cost, highest attainments) and did so more quickly than the other methods (full results in Section 5.1).

In Section 2.2.4 it was argued that decision ordering diagrams are useful when they are *timely* to generate while being *well-behaved* and *tame*. KEYS2's results are the most *timely* (fastest to

generate) of all of the methods studied here. As to the other criteria, Figure 5.3 shows that KEYS2's decision ordering diagrams:

- Move smoothly to a plateau with only a small amount of variance;
- Have very low spreads, compared to the median results.

That is, at least for the models explored here, KEYS2 generated decision ordering diagrams they are both *well-behaved* and *tame*.

In summary, I recommend KEYS2 for generating decision ordering diagrams since, apart from the (slightly slower) KEYS algorithm, I am currently unaware of other search-based software engineering methods that enable such a rapid reflection of solution robustness.

6.2 Recommendations for Future Work

The scale-up study conducted in Section 5.2 states that, by 2013, DDP models will be more than twice their current size. As these DDP models grow, it is not just the number of variables that grow. Larger models are more densely interconnected. More mitigations affect more risks, and more risks affect more requirements. Our current top optimization techniques terminate in a matter of milliseconds. They effectively report real-time results for current models. However, further optimization is required in order to support the models that we expect to see in the future.

Furthermore, by setting multiple mitigations per round, KEYS2's judgments are based on less *evidence* and thus contain more *variance* (that is, less tame). The small amount of variance in KEYS2 is within acceptable bounds, but other attempts to fix a higher number of configurations per round have fallen outside of these limits.

Research into how to improve the KEYS algorithm revealed that over 70% of the execution time is spent polling the model for the cost and attainment performance values. Each round, both KEYS and KEYS2 generate one-hundred sets of mitigation configurations and score them. They

use these instances to set the optimal mitigations to fixed values. This instance generation step is the chief bottleneck in the KEYS algorithm. However, a large number of instances is needed to provide the *support* necessary to accurately rank mitigations. If fewer instances are generated, the algorithm's result variance will still fall outside of appropriate limits. Therefore, the basis for future research should be in how to retain high support while executing the model fewer times.

One attempt, KEYS-R, focused on reusing existing configuration sets that recieved high scores from the objective function (for more, see Section 4.5.1). This KEYS variation was fast, but its results were notably poor. It failed because it tried to reuse high-scoring items without regard for whether or not they still provided an adequate level of support. One possible future extension to the KEYS formula could solve the issues of the original algorithm by using *active learning* [21] to reflect over the space of past decisions. Think of a natural thought process; someone who is looking for a specific book would not search every shelf of a store in order of appearance. Instead, they would walk directly to the appropriate section. The same idea guides active learning. Most machine learning algorithms are passive, they simply read over (or randomly generate) a set of examples with no regard for the contents of that sample population. In contrast, an algorithm that makes use of active learning is able to exercise some level of control over the input that it trains on.

In short, the performance of KEYS could be improved by - rather than attempting to *reuse* instances - saving them and providing a quick method of *looking up* previously assessed model configurations. This proposed improvement clusters an initial set of configurations, sorts these clusters into a cover tree, and only consults the model for a score when an instance's performance cannot be assessed from the space of past judgments.

KEYS (and its extension, KEYS2)is a powerful algorithm because it only explores a small fraction of the total number of configurations. It is a safe expectation that, out of the sheer number of possible settings, that not all possible configurations of the mitigations will lead to different effects. By extension, it becomes possible that KEYS can assess new configurations of the model

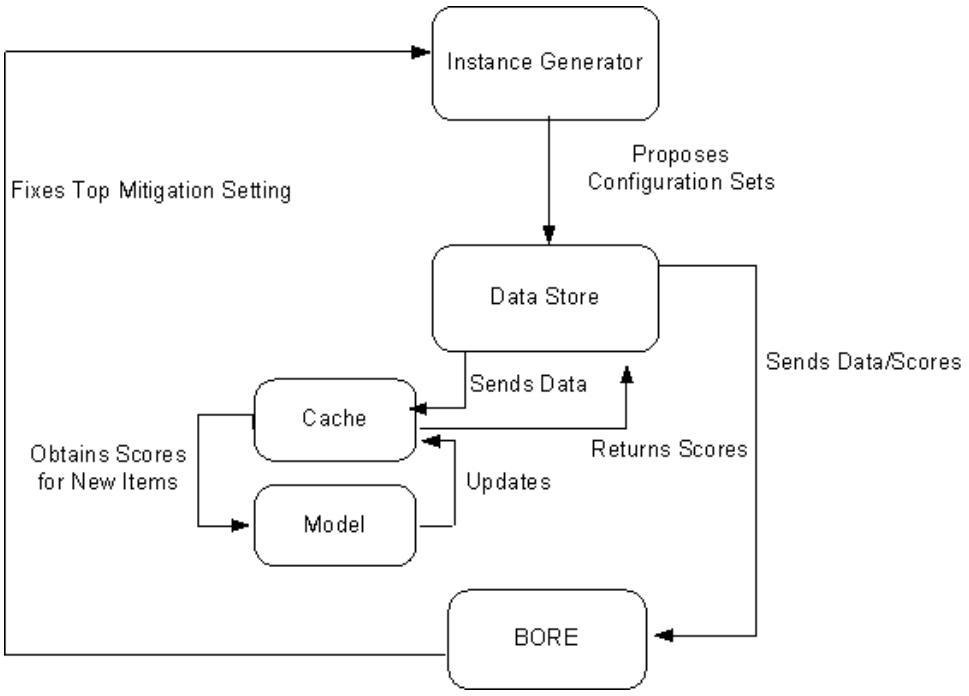

Figure 6.1: The data architecture of a KEYS variant using a cache.

without actually executing the model, but rather by remembering what happened with past settings.

Figure 6.1 shows the dataflow of a KEYS variant extended with an automated cache. In this system, KEYS will track the configurations used in the past as well as their unique score. Past data can be efficiently grouped, and a nearest-neighbor technique could extrapolate new combinations. By clustering the old results by their performance scores, KEYS could reflect over old settings in order to offer potentially ideal new combinations. Likewise, KEYS can pull out performance scores of configurations that fall into regions explored in the past (that is, the *smooth* regions where results can be extrapolated).

A cluster can be said to be linear if, on average, performance scores can be interpolated from neighboring instances. If a cluster is linear and contains three items x, y, z and y, z are the nearest neighbors to x, then the $(cost, attainment)$ score pair of x should be the same as a linear interpolation between y and z. The assumption behind this theory is based on analogy-based effort estimation [97] - the *kth* nearest neighbors to the test instance are linear. The performance score

for one instance can be averaged from its neighbors. Consequently, if a new proposed set of mitigation settings falls into a linear cluster, then KEYS needs not waste CPU time polling the model for a score. Instead, it can just interpolate the result in linear time.

For this cache-enabled variant of KEYS to outperform the original algorithm, it is essential that the clustering and processing of cache entries takes less time than executing all possible configurations during each round. For this to succeed, KEYS must be able to quickly and incrementally cluster a large vector space. Without efficient clustering, KEYS will not scale well to models with a large number of mitigations. Numerous studies [2, 60, 76, 109] have looked at the clustering of large vectors of settings, and they have found that many existing tools require complex and costly computations that must be repeated with every new set of configurations.

One way to avoid some of the computational overhead when dealing with new data is to do a single clustering on the initial set of one-hundred configurations that KEYS proposes. Once these clusters are computed, the algorithm can assign a unique identifier to each one. A cover-tree algorithm can be run on these clusters in order to generate a hierarchical distance tree. The generation of this tree will be much faster if the data is in a set number of clusters, rather than having to process every initial set of mitigation settings. From this point, every unseen set of configurations can be quickly mapped to the most appropriate cluster by dropping it through the tree.

Interestingly, once linear clusters are identified, KEYS could be trained to avoid them. New configurations that fall into linear clusters, especially those clusters associated with worse performance scores, add no information about the effects of different configurations. Once these regions are known, the KEYS algorithm can rule out large sections of the search space by ignoring regions which are linear or, on average, produce results that are significantly worse than other regions.

An interesting, and open, research question is how many clusters are appropriate and how many data items should *"fall"* into a region before any sort of re-clustering occurs. The fewer the clusters, the more approximate the reasoning. Until this hypothetical version of KEYS is

implemented, it is impossible to know the space of trade-offs between efficiency and cluster size; however, one potential approach is to use the PID incremental discretizer, as proposed by Gama and Pinto [40]. This discretizer maintains two arrays - a *lower* and *upper* - each with different discretizers. The upper array is filled with new data, and if the frequency counts grow too large, PID splits each bin into two new bins. At certain intervals, the upper array migrates its contents into a summarized set of breakpoints in the lower array. This simple scheme has been recommended by Yang and Webb [116] because it is simple to implement and because, by only propagating frequency counts to the lower array, PID has been shown to be seven to ten times more efficient than other algorithms.

Unfortunately, any change in the bin boundaries of the data will require the recreation of clusters. This is an expensive process, as it also requires a rebuilding of the cover tree and a complete pass through all of the data. Incremental re-clustering is necessary to maintain the accuracy of KEYS' predictions, but how often this should occur remains an open question.

Research following the development of KEYS2 has revealed a limiting factor in further attempts to improve the algorithm. Over 70% of the execution time is spent polling the model for the cost and attainment values associated with a particular configuration. This proposed extension to KEYS solves such issues by utilizing active learning. An initial set of configurations is generated and clustered, these clusters are sorted into a cover tree, and the model is only consulted when a configuration's performance cannot be assessed from the space of past judgments. This caching concept should see a performance gain for the KEYS algorithm, which could be further increased by training KEYS to avoid linear (i.e. smooth) areas of data where performance is less than ideal.

Bibliography

[1] M. Dorigo A. Colorni and V. Maniezzo. Distributed optimization by ant colonies. In *ECAL'91 - European Conference on Artificial Life*, pages 134–142, New York, NY, USA, 1991. Elsevier.

[2] D. Poshyvanyk A. Marcus and R. Ferenc. Using the conceptual cohesion of classes for fault prediction in object oriented systems. *IEEE Transactions on Software Engineering*, 34:287–300, 2 2008.

[3] S. Halgamuge A. Ratnaweera and H. Watson. Self-organizing hierarchical particle swarm optimizer with time-varying acceleration coefficients. *IEEE Transactions on Evolutionary Computing*, 8:240–255, 2004.

[4] Federal Aviation Administration. System engineering manual version 3.1, section 4.6: Trade studies, 2006. Available from `http://www.faa.gov/about/office_org/headquarters_offices/ato/service_units/operations/sysengsaf/seman/SEM3.1/Section%204.6.pdf`.

[5] S. Amarel. Program synthesis as a theory formation task: Problem representations and solution methods. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach: Volume II*, pages 499–569. Kaufmann, Los Altos, CA, 1986.

[6] Harmen Sthamer Andre Baresel and Michael Schmidt. Fitness function design to improve evolutionary structural testing. In *GECCO '02: Proceedings of the 4th Annual conference on Genetic and evolutionary computation*, pages 1329–1336. Morgan Kaufmann Publishers, 2002.

[7] Mark Harman Andre Baresel, David Wendell Binkley and Bogdan Korel. Evolutionary testing in the presence of loop-assigned flags: A testability transformation approach. In *International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 108–118, 2004.

[8] P.S. Andrews. An investigation into mutation operators for particle swarm optimization. In *IEEE Congress on Evolutionary Computing*, pages 1044–1051, 2006.

[9] Andrea Arcuri. Insight knowledge in search based software testing. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1649–1656, New York, NY, USA, 2009. ACM.

[10] Kamel Ayari, Salah Bouktif, and Giuliano Antoniol. Automatic mutation test input data generation via ant colony. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1074–1081, New York, NY, USA, 2007. ACM.

[11] A. Bagnall, V. Rayward-Smith, and I. Whittley. The next release problem. *Information and Software Technology*, 43(14), December 2001.

[12] Nils Aall Barricelli. Esempi numerici di processi di evoluzione. *Mehodos*, pages 45–68, 1954.

[13] 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`.

[14] Terry Van Belle and David Ackley. Code factoring and the evolution of evolvability. In *GECCO '02: Proceedings of the 4th Annual conference on Genetic and evolutionary computation*, pages 1383–1390. Morgan Kaufmann Publishers, 2002.

[15] Colin Burgess and Martin Lefley. Can genetic programming improve software effort estimation? a comparitive evaluation. *Information and Software Technology*, 43(14):863–873, 2001.

[16] Francisco Chicano and Enrique Alba. Ant colony optimization with partial order reduction for discovering violations in concurrent models. *Information Processing Letters*, 106:221–231, 2008.

[17] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification*, 2002.

[18] R. Clark. Faster treatment learning, Computer Science, Portland State University. Master's thesis, 2005.

[19] 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.

[20] Cristian Coarfa, Demetrios D. Demopoulos, Alfonso San, Miguel Aguirre, Devika Subramanian, and Moshe Y. Vardi. Random 3-sat: The plot thickens. In *In Principles and Practice of Constraint Programming*, pages 143–159, 2000. Availabe from `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.3662`.

[21] David Cohn, Les Atlas, and Richard Ladner. Improving generalization with active learning. *Mach. Learn.*, 15(2):201–221, 1994.

[22] Martin Shepperd Colin Kirsopp and John Hart. Search heuristics, case-based reasoning and software project effort prediction. In *GECCO '02: Proceedings of the 4th Annual conference on Genetic and evolutionary computation*, pages 1367–1374. Morgan Kaufmann Publishers, 2002.

[23] 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.

[24] J. Crawford and A. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *AAAI '94*, 1994.

[25] Pratap A. Agarwal S. Deb, K. and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[26] Mark Harman Deji Fatiregun and Rob Hierons. Evolving transformation sequences using genetic algorithms. In *In 4th International Workshop on Source Code Analysis and Manipulation (SCAM 04)*, pages 65–71, Los Alamitos, CA, USA, 2004. IEEE Computer Society Press.

[27] Ernesto Diaz-Aviles, Wolfgang Nejdl, and Lars Schmidt-Thieme. Swarming to rank for information retrieval. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 9–16, New York, NY, USA, 2009. ACM.

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

[29] M. Dorigo and L. Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computing*, 1:53–66, 1997.

[30] R.C. Eberhart and J. Kennedy. A new optimizer using particle swarm theory. In *6th International Symposium on Micromachine Human Science*, pages 39–43, 1995.

[31] R.C. Eberhart and Y. Shi. Tracking and optimizing dynamic systems with particle swarms. In *IEEE World Congress on Evolutionary Computing*, pages 94–97, 2001.

[32] Maria Claudia Figueiredo Pereira Emer and Silva Regina Vergilio. Gptest: A testing tool based on genetic programming. In *GECCO '02: Proceedings of the 4th Annual conference on Genetic and evolutionary computation*, pages 1343–1350. Morgan Kaufmann Publishers, 2002.

[33] Raquel Blanco Jos Javier Dolado Eugenia Daz, Javier Tuya. A tabu search algorithm for structural software testing. *Computers & Operations Research*, 35:3052–3072, 2008.

[34] M. Feather, S. Cornford, K. Hicks, J. Kiper, and T. Menzies. Application of a broad-spectrum quantitative requirements model to early-lifecycle decision making. *IEEE Software*, 2008. Available from `http://menzies.us/pdf/08ddp.pdf`.

[35] M.S. Feather, K.A. Hicks, R.M. Mackey, and S. Uckun. Guiding technology deployment decisions using a quantitative requirements analysis technique. In *IEEE International Conference on Requirements Engineering, Industrial Practice and Experience track Barcelona, Spain*, 2008.

[36] 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`.

[37] M.S. Feather, S. Uckun, and K.A. Hicks. Technology maturation of integrated system health management. In *Space Technology and Applications International Forum (STAIF-2008) Albuquerque, USA*, February 2008.

[38] Marco Ferreira, Francisco Chicano, Enrique Alba, and Juan A. Gmez-Pulido. Detecting protocol errors using particle swarm optimization with java pathfinder. In *High Performance Computing & Simulation Conference (HPCS '08)*, 2008.

[39] Javier Ferrer, Francisco Chicano, and Enrique Alba. Dealing with inheritance in oo evolutionary testing. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1665–1672, New York, NY, USA, 2009. ACM.

[40] Joao Gama and Carlos Pinto. Discretization from data streams: applications to histograms and data mining. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 662–667, New York, NY, USA, 2006. ACM Press. Available from `http://www.liacc.up.pt/~jgama/IWKDDS/Papers/p6.pdf`.

[41] Gregory Gay, Tim Menzies, Misty Davies, and Karen Gundy-Burlet. A new monte carlo filtering method for the diagnosis of mission-critical failures. *Submitted to Automated Software Engg.*, 2010.

[42] F. Glover. Tabu search - part 1. *ORSA Journal on Computing*, 1:190–206, 1989.

[43] F. Glover. Tabu search - part 2. *ORSA Journal on Computing*, 2:4–32, 1990.

[44] Jun Gu, Paul W. Purdom, John Franco, and Benjamin W. Wah. Algorithms for the satisfiability (sat) problem: A survey. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 19–152. American Mathematical Society, 1997.

[45] Stefan Gueorguiev, Mark Harman, and Giuliano Antoniol. Software project planning for robustness and completion time in the presence of uncertainty using multi objective search based software engineering. In *GECCO '09: Proceedings of the 11th Annual conference*

on Genetic and evolutionary computation, pages 1673–1680, New York, NY, USA, 2009. ACM.

[46] K. Gundy-Burlet, J. Schumann, T. Barrett, and T. Menzies. Parametric analysis of antares re-entry guidance algorithms using advanced test generation and data analysis. In *9th International Symposium on Artifical Intelligence, Robotics and Automation in Space*, 2007.

[47] K. Gundy-Burlet, J. Schumann, T. Barrett, and T. Menzies. Parametric analysis of a hover test vehicle using advanced test generation and data analysis. In *AIAA Aerospace*, 2009.

[48] 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. Available from `http://www.cs.waikato.ac.nz/~mhall/HallHolmesTKDE.pdf`.

[49] M. Harman, R. Hierons, and M. Proctor. A new representation and crossover operator for search-based optimization of software modularization. In *GECO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1351–1358. Morgan Kaufmann, July 2002.

[50] M. Harman and B.F. Jones. Search-based software engineering. *Journal of Information and Software Technology*, 43:833–839, December 2001.

[51] Mark Harman. The current state and future of search based software engineering. In *Future of Software Engineering, ICSE'07*. 2007.

[52] Mark Harman and John Clark. Metrics are fitness functions too. In *10th International Software Metrics Symposium (METRICS 2004)*, 2004), pages = 58–69, location = Chicago, IL, USA, publisher = IEEE Computer Society Press, address = Los Alamitos, CA, USA,.

[53] Mark Harman, Jens Krinke, Jian Ren, and Shin Yoo. Search based data sensitivity analysis applied to requirement engineering. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1681–1688, New York, NY, USA, 2009. ACM.

[54] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968.

[55] J. Holland. *Adaptation in natural and artificial systems*. University of Michigan Press, Ann Arbor, MI, USA, 1975.

[56] G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[57] Y. Hu. Treatment learning: Implementation and application. Master's thesis, Department of Electrical Engineering, University of British Columbia, 2003. Masters Thesis.

[58] Y.C Hui, E.C. Prakash, and N.S. Chaudhari. Game ai: artificial intelligence for 3d path finding. In *TENCON 2004. 2004 IEEE Region 10 Conference*, volume 2, pages 306–309, 2004.

[59] O. Jalali, T. Menzies, and M. Feather. Optimizing requirements decisions with keys. In *Proceedings of the PROMISE 2008 Workshop (ICSE)*, 2008. Available from `http://menzies.us/pdf/08keys.pdf`.

[60] Y. Jiang, B. Cukic, and T. Menzies. Does transformation help? In *Defects 2008*, 2008. Available from `http://menzies.us/pdf/08transform.pdf`.

[61] Corina Pasareanu Tim Menzies Johann Schumann, Karen Gundy-Burlet and Anthony Barrett. Software v&v support by parametric analysis of large software simulation systems. In *2009 IEEE Aerospace Conference*, 2009.

[62] I.J. Jureta, J. Mylopoulos, and S. Faulkner. Revisiting the core ontology and problem in requirements engineering. In *International Requirements Engineering, 2008. RE '08. 16th IEEE*, pages 71–80, Sept. 2008.

[63] Henry Kautz and Bart Selman. Pushing the envelope: Planning, propositional logic and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, pages 1194–1201, Menlo Park, August 4–8 1996. AAAI Press / MIT Press. Available from `http://www.cc.gatech.edu/~jimmyd/summaries/kautz1996.ps`.

[64] J. Kennedy and R.C. Eberhart. Particle swarm optimization. In *IEEE International Conference on Neural Networks*, pages 1942–1948, 1995.

[65] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983.

[66] Ron Kohavi and George H. John. Wrappers for feature subset selection. *Artificial Intelligence*, 97(1-2):273–324, 1997.

[67] J.J. Liang and P.N. Suganthan. Dynamic multi-swarm particle swarm optimizer with local search. In *IEEE Congress on Evolutionary Computing*, pages 522–528, 2005.

[68] L. Gambardella M. Dorigo and E. Taillard. Ant colonies for the quadratic assignment problem. *Journal of the Operational Research Society*, 50:167–176, 1999.

[69] A. MacLean, R.M. Young, V. Bellotti, and T.P. Moran. Questions, options and criteria: Elements of design space analysis. In T.P. Moran and J.M. Carroll, editors, *Design Rationale: Concepts, Techniques, and Use*, pages 53–106. Lawerence Erlbaum Associates, 1996.

[70] Harman M. Mahdavi, K. and R.M. Hierons. A multiple hill climbing approach to software module clustering. In *International Conference on Software Maintenence 2003*, pages 315–324, 2003.

[71] Robert Mark Hierons Joachim Wegener Harmen Sthamer Andre Baresel Mark Harman, Lin Hu and Marc Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, 2004.

[72] Phil McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14:105–156, 2004.

[73] Phil McMinn. Search-based failure discovery using testability transformations to generate pseudo-oracles. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1689–1696, New York, NY, USA, 2009. ACM.

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

[75] T. Menzies, J. Kiper, and M. Feather. Improved software engineering decision support through automatic argument reduction tools. In *SEDECS'2003: the 2nd International Workshop on Software Engineering Decision Support (part of SEKE2003)*, June 2003. Available from `http://menzies.us/pdf/03star1.pdf`.

[76] T. Menzies and A. Marcus. Automated severity assessment of software defect reports. In *ICSM'08*, 2008. Available from `http://menzies.us/pdf/08severis.pdf`.

[77] T. Menzies and H. Singh. Many maybes mean (mostly) the same thing. In M. Madravio, editor, *Soft Computing in Software Engineering*. Springer-Verlag, 2003. Available from `http://menzies.us/pdf/03maybe.pdf`.

[78] Tim Menzies, Omid Jalali, and Martin Feather. Optimizing requirements decisions with keys. In *Proceedings PROMISE '08 (ICSE)*, 2008.

[79] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, and E. Teller. Equation of state calculations by fast computing machines. *J. Chem. Phys*, 21:1087–1092, 1953.

[80] Brian S. Mitchell and Spiros Mancoridis. Using heuristic search techniques to extract design abstractions from source code. In *GECCO '02: Proceedings of the 4th Annual conference on Genetic and evolutionary computation*, pages 1375–1382. Morgan Kaufmann Publishers, 2002.

[81] H.D. Mittelmann. Recent benchmarks of optimization software. In *22nd Euorpean Conference on Operational Research*, 2007.

[82] J. Mylopoulos, L. Cheng, and E. Yu. From object-oriented to goal-oriented requirements analysis. *Communications of the ACM*, 42(1):31–37, January 1999.

[83] J. Mylopoulos, L. Chung, and B. Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE Transactions of Software Engineering*, 18(6):483–497, June 1992.

[84] An Ngo-The and G. Ruhe. Optimized resource allocation for software release planning. *Software Engineering, IEEE Transactions on*, 35(1):109–123, Jan.-Feb. 2009.

[85] J. Nielson. *Usability Engineering*. Academic Press, 1993.

[86] Kari J. Nurmela. Upper bounds for covering arrays by tabu search. *Discrete Applied Mathematics*, 138:143–152, 2004.

[87] A.S. Orrego. Sawtooth: Learning from huge amounts of data. Master's thesis, Computer Science, West Virginia University, 2004.

[88] Judea Pearl. *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.

[89] Matthias Prandtstetter and Günther R. Raidl. Meta-heuristics for reconstructing cross cut shredded text documents. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 349–356, New York, NY, USA, 2009. ACM.

[90] J.L. Deneubourg R. Beckers and S. Gross. Trails and u-turns in the selection of the shortest path by the ant lasius niger. *Journal of Theoretical Biology*, 159:397–415, 1992.

[91] Leo Rela. Evolutionary computing in search-based software engineering. Master's thesis, Department of Information Technology, Lappeenranta University of Technology, 2004. Available from `http://now.unbox.org/all/trunk/doc/06/jeremy/dtyo_Leo_Rela(GAforSE).pdf`.

[92] Sion Ll Rhys, Simon Poulding, and John A. Clark. Using automated search to generate test data for matlab. In *GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1697–1704, New York, NY, USA, 2009. ACM.

[93] Stuart J. Russell, Peter Norvig, John F. Candy, Jitendra M. Malik, and Douglas D. Edwards. *Artificial intelligence: a modern approach*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2003.

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

[95] A. Saltelli, K. Chan, and E.M. Scott. *Sensitivity Analysis*. Wiley, 2000.

[96] Bart Selman, Henry A. Kautz, and Bram Cohen. Local search strategies for satisfiability testing. In Michael Trick and David Stifler Johnson, editors, *Proceedings of the Second DIMACS Challange on Cliques, Coloring, and Satisfiability*, Providence RI, 1993.

[97] 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`.

[98] Y. Shi and R.C. Eberhart. A modified particle swarm optimizer. In *IEEE World Congress on Computer Intelligence*, pages 69–73, 1998.

[99] Y. Shi and R.C. Eberhart. Empirical study of particle swarm optimization. In *IEEE World Congress on Evolutionary Computing*, pages 1945–1950, 1999.

[100] Y. Shi and R.C. Eberhart. Fuzzy adaptive particle swarm optimization. In *IEEE World Congress on Evolutionary Computing*, pages 101–106, 2001.

[101] 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.

[102] J. Stardom. Metaheuristics and the search for covering and packing arrays. Master's thesis, Simon Fraser University, B.C., Canada., 1998. Available from `http://www.collectionscanada.gc.ca/obj/s4/f2/dsk3/ftp04/MQ61608.pdf`.

[103] Bryan Stout. Smart moves: Intelligent pathfinding. *Game Developer Magazine*, (7), 1997.

[104] P.N. Suganthan. Particle swarm optimizer with a neighborhood operator. In *IEEE Congress on Evolutionary Computing*, pages 1958–1962, 1999.

[105] W.H. Liahu S.Y. Ho, H.S. Lin and S.J. Ho. Opso: orthogonal particle swarm optimization and its application to task assihnment problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 38:288–298, 2008.

[106] Clark J. Tracey, N. and K. Mander. Automated program flaw finding using simulated annealing. In *International Symposium on Software Testing and Analysis (ISSTA 1998)*, pages 73–81, 1998.

[107] Clark J. Tracey, N. and K. Mander. The way forward for unifying dynamic test-case generation: The optimisation-based approach. In *Workshop on Dependable Computing and Its Applications*, pages 169–180. Dept of Computer Science, University of Witwatersrand, 1998.

[108] N. Tracey. *A Search-Based Automated Test-Data Generation Framework for Safety Critical Software*. PhD thesis, 2000.

[109] B. Turhan, T. Menzies, A. Bener, and J. Distefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering (to appear)*, 2009. Available from `http://menzies.us/pdf/08ccwc.pdf`.

[110] Tomas E. Uribe and Mark E. Stickel. Ordered binary decision diagrams and the davis-putnam procedure. In *In Proc. of the 1st International Conference on Constraints in Computational Logics*, pages 34–49. Springer-Verlag, 1994.

[111] A. Colorni V. Maniezzo and M. Dorigo. The ant system applied to the quadratic assignment problem. Technical report, Universite Libre de Bruxelles, Belgium, 1994.

[112] Daniel N. Wilke. Evolutionary computing in search-based software engineering. Master's thesis, Department of Mechanical and Aeronautical Engineering, University of Pretoria, 2005. Available from http://upetd.up.ac.za/thesis/available/etd-01312006-125743/.

[113] R. Williams, C.P. Gomes, and B. Selman. Backdoors to typical case complexity. In *Proceedings of IJCAI 2003*, 2003. http://www.cs.cornell.edu/gomes/FILES/backdoors.pdf.

[114] Andreas Windisch, Stefan Wappler, and Joachim Wegener. Applying particle swarm optimization to software testing. In *9th annual Conference on Genetic and Evolutionary Computation (GECCO '07)*, 2007.

[115] Ian H. Witten and Eibe Frank. *Data mining. 2nd edition*. Morgan Kaufmann, Los Altos, US, 2005.

[116] Y. Yang and G. Webb. Weighted proportional k-interval discretization for naive-bayes classifiers. In *Proceedings of the 7th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD 2003)*, 2003. Available from http://www.csse.monash.edu/~webb/Files/YangWebb03.pdf.

[117] Y. Li Z-H Zhan, J. Zhang and HS-H Chung. Adaptive particle swarm optimization. *IEEE Transactions on Systems, Man, and Cybernetics - Part B: Cybernetics*, 39:1362–1381, 2009.

[118] Pamela Zave and Michael Jackson. Four dark corners of requirements engineering. *ACM Trans. Softw. Eng. Methodol.*, 6(1):1–30, 1997.

[119] Y. Zhang, M. Harman, and S.A. Mansouri. The multi-objective next release problem. In *In ACM Genetic and Evolutionary Computation Conference (GECCO 2007*, page 11, 2007.

[120] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: A comparative case study and the strength pareto approach. *IEEE Transactions on Evolutionary Computation*, 3(4):257–271, 1999.

Appendix A

Obtaining the System

We have placed on-line all the materials required for other researchers to conduct further investigation into this problem. All the code, Makefiles, scripts, and so on used in this paper are available at `http://unbox.org/wisp/tags/ddpExperiment/install`. For security reasons, all the available JPL requirements models have been "sanitized"; i.e. all words replaced with anonymous variables.

Appendix B

Source Code

B.1 KEYS/KEYS2

B.1.1 keys.h

```
#ifndef _keys_h
#define _keys_h

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <unistd.h>
#include "model.h"

#define TotalMitigations MITIGATION

void reportMedianAndSpread(float** Data);
void findMedianAndSpread(float inputArray[], int size, float *median, float *spread);
void rankMitigations(float** Data, float* Distance);
void sweetSpot(float** Data, float* Distance);
int selectValue(int val1, int val2);
void addInstance(float costVar, float attVar, float** Data);
float minValue(float val1, float val2);
float findBestDistance(float inputArray[], int size);
void model(float *cost, float *att, float m[]);

#endif
```

B.1.2 keys.c

```
#include "keys.h"

/*
########################################################################
#
#  KEYS: DDP model optimization tool
#  Copyright (C) 2007-2008 Omid Jalali, Gregory Gay
#
#  This program is free software: you can redistribute it and/or modify
#  it under the terms of the GNU General Public License as published by
#  the Free Software Foundation, either version 3 of the License.
#
#  This program is distributed in the hope that it will be useful,
#  but WITHOUT ANY WARRANTY; without even the implied warranty of
#  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#  GNU General Public License for more details.
#
#  You should have received a copy of the GNU General Public License
#  along with this program.  If not, see <http://www.gnu.org/licenses/>.
########################################################################
*/

int costFlag, attFlag, displayFlag, randomFlag, runFlag, maxFutileFlag;
float costLimit, attLimit;
```

```
int Seed;
int RunTotal;

float FixedMitigations[TotalMitigations+1];
float mArray[TotalMitigations+1];

int mCounter,era,run,TotalInstances,instanceCounter,displayMode;
float MinCost,MaxCost,MinAtt,MaxAtt;
float infinity,small;

int recentSetMitigation;
float recentSetMitigationStatus;

float LastMinCost,LastMaxAtt;

int main(int argc, char **argv)
{
        //this is required to set up the ddp model.
        setupModel();

        Seed = 0;

        char *costValue = NULL;
        char *attValue = NULL;
        char *displayValue = NULL;
        char *randomValue = NULL;
        char *runValue = NULL;
        char *futileValue = NULL;
        int c;
        int futile = 0;
        int MaxFutile;

        displayFlag = 0;
        costFlag = 0;
        attFlag = 0;
        randomFlag = 0;
        runFlag = 0;
        maxFutileFlag = 0;

        infinity = pow(10,20);
        small = pow(10,-20);

        opterr = 0;

        while ((c = getopt(argc, argv, "a:c:d:f:h:r:t:")) != -1)
        {
                switch (c)
                {
                        case 'a':
                                attFlag = 1;
                                attValue = optarg;
                                break;
                        case 'c':
                                costFlag = 1;
                                costValue = optarg;
                                break;
                        case 'd':
                                displayFlag = 1;
                                displayValue = optarg;
                                break;
                        case 'f':
                                maxFutileFlag = 1;
                                futileValue = optarg;
                                break;
                        case 'r':
                                randomFlag = 1;
                                randomValue = optarg;
                                break;
                        case 't':
                                runFlag = 1;
                                runValue = optarg;
                                break;
                        case 'h':
                        case '?':
                                printf("\nThe options must have the format -a AttainmentLowerLimit -c CostUpperLimit"+
                                        "-d DisplayMode -f Futile -r Seed -t TotalRuns.\n");
                                printf("\nAttainmentLowerLimit:\n\tThis is the lower limit"+
                                        "that the user can set for attainment.\n");
                                printf("\tThe tool tries to find a mitigation set that gives an attainment"+
                                        "equal to or higher than this value.\n");
                                printf("\tHowever, this is not guaranteed especially for limits set too high.\n");
                                printf("\nCostUpperLimit:\n\tThis is the upper limit that the user can set for cost.\n");
                                printf("\tThe tool tries to find a mitigation set that gives a cost equal to"+
                                        "or lower than this value.\n");
                                printf("\tHowever, this is not guaranteed especially for limits set too low.\n");
                                printf("\nDisplay Mode:\n\tDisplay mode 1 prints the final results.\n");
                                printf("\tDisplay mode 2 prints median and spread of each mitigation-fixing round.\n");
```

```
                        printf("\tDisplay mode 3 is for debugging purposes only and prints everything needed.\n");
                        printf("\tDisplay mode 4 is for debugging purposes only and prints cost and"+
                                "attainment of each run.\n");
                        printf("\nFutile:\n\tThis is the number of trials before the tool decides"+
                                "that no improvements were seen.\n");
                        printf("\tIt can be a number between 1 and the number of mitigations used.\n");
                        printf("\tIt can be turned off by setting it the number of mitigations"+
                                "used. The default value is 10.\n");
                        printf("\nSeed:\n\tSeed is used in random number generation."+
                                "By default, it is generated using the system time.\n");
                        printf("\nTotalRuns:\n\tThe number of total runs that is used internally."+
                                "The default value is 100.\n");
                        return(1);
                        break;
                default:
                        break;
        }
}

if (attFlag == 1 && attValue != NULL)
        attLimit = (float)atof(attValue);
else
        attLimit = -infinity;

if (costFlag == 1 && costValue != NULL)
        costLimit = (float)atof(costValue);
else
        costLimit = infinity;

if (randomFlag == 1 && randomValue != NULL)
        Seed = atoi(randomValue);
else
        Seed = 1;

if (runFlag == 1 && runValue != NULL)
        RunTotal = atoi(runValue);
else
        RunTotal = 100;

if (displayFlag == 1 && displayValue != NULL)
        displayMode = atoi(displayValue);
else
        displayMode = 1;

if (maxFutileFlag == 1 && futileValue != NULL)
        MaxFutile = atoi(futileValue);
else
        MaxFutile = 99;

float att, cost;

float *Distance = new float[RunTotal+1];

float **Data = new float*[RunTotal+1];
for (int i = 0; i < RunTotal + 1; i++)
        Data[i] = new float[TotalMitigations+2];

if (randomFlag == 1)
        srand(Seed);
else
        srand((unsigned int)time(NULL));

//set all mitigations to non-fixed value of -1
for (mCounter = 1; mCounter <= TotalMitigations; mCounter++)
        FixedMitigations[mCounter] = -1;

if (displayMode == 2)
        printf("Era, Mitigation Number, Mitigation Value, Median of Cost,"+
                "Spread of Cost, Median of Attainment, Spread of Attainment\n");

MinCost = infinity;
MaxCost = -infinity;

MinAtt = infinity;
MaxAtt = -infinity;

LastMinCost = MinCost;
LastMaxAtt = MaxAtt;

//Temp. Find the max cost, max att
int x;
for(x=1;x<=TotalMitigations;x++){
        mArray[x]=1;
}
model(&cost,&att,mArray);
// printf("%.1f,%.5f\n",cost,att);
```

```
	for(x=1;x<=TotalMitigations;x++){
		mArray[x]=0;
	}


	for (era = 1; era <= TotalMitigations; era++)
	{
		TotalInstances = 0;

		for (run = 1; run <= RunTotal; run++)
		{
			for (mCounter = 1; mCounter <= TotalMitigations; mCounter++)
			{
				//if in the previous runs the mitigation is fixed to a certain value (0 or 1) then use that value.
				//otherwise, select it at random
				if (FixedMitigations[mCounter] == -1)
					mArray[mCounter] = selectValue(0,1);
				else
					mArray[mCounter] = FixedMitigations[mCounter];
			}
			//find the cost and att using these mitigations
			model(&cost, &att, mArray);

			if (displayMode == 4)
				printf("%.1f,%.5f\n", cost, att);

			//store the current instance
			addInstance(cost, att, Data);
		}

		//find the sweet spot and distances from sweet spot for each distance
		sweetSpot(Data, Distance);

		//rank the mitigations and find the mitigation that should be fixed
		rankMitigations(Data, Distance);

		if (displayMode == 2)
			reportMedianAndSpread(Data);

		if (MinCost < LastMinCost && MaxAtt >= LastMaxAtt)
		{
			LastMinCost = MinCost;
			LastMaxAtt = MaxAtt;
			futile = 0;
		}
		else if (futile > MaxFutile)
		{
			if (displayMode == 1 || displayMode == 3)
				printf("Terminated at era: %d\n",era);
			era = TotalMitigations + 1;
			futile++;
		}
		else
			futile++;
	}

	//show the final results
	if (displayMode == 1 || displayMode == 3)
	{
		for (mCounter = 1; mCounter <= TotalMitigations; mCounter++)
		{
			mArray[mCounter] = FixedMitigations[mCounter];
			if (FixedMitigations[mCounter] == -1)
				mArray[mCounter] = 0;
		}
		model(&cost,&att,mArray);
		for (mCounter=1; mCounter<=TotalMitigations; mCounter++)
			printf("m[%d],",mCounter);
		printf("cost,attainment\n");
		for (mCounter=1; mCounter<=TotalMitigations; mCounter++)
			printf("%.0f,", mArray[mCounter]);
		printf("%.1f,%.5f\n",cost,att);
	}
}

void reportMedianAndSpread(float** Data)
{
	float tempCostArray[TotalInstances], tempAttArray[TotalInstances];
	float costMedian,costSpread,attMedian,attSpread;

	//sort the cost and att (individually) and find the median and spread
	for (instanceCounter = 1; instanceCounter <= TotalInstances; instanceCounter++)
	{
		tempCostArray[instanceCounter] = Data[instanceCounter][1];
		tempAttArray[instanceCounter] = Data[instanceCounter][2];
	}
	findMedianAndSpread(tempCostArray,TotalInstances,&costMedian,&costSpread);
```

```
        findMedianAndSpread(tempAttArray,TotalInstances,&attMedian,&attSpread);

        printf("%d,%d,%.0f,%.5f,%.5f,%.5f,%.5f\n",era,recentSetMitigation,
                recentSetMitigationStatus,costMedian,costSpread,attMedian,attSpread);
}

void findMedianAndSpread(float inputArray[], int size, float *median, float *spread)
{
        float tempValue;
        int i,j;
        float tempArray[size];

        for (i = 1; i <= size; i++)
                tempArray[i] = inputArray[i];

        //sort
        for (i = 1; i <= size; i++)
        {
                tempValue = tempArray[i];
                j = i;

                while ((j > 1) && (tempArray[j-1] > tempValue))
                {
                        tempArray[j] = tempArray[j-1];
                        j = j - 1;
                }
                tempArray[j] = tempValue;
        }

        *median = tempArray[size/2];
        *spread = tempArray[3*size/4] - tempArray[size/2];
}

float findBestDistance(float inputArray[], int size)
{
        float tempValue;
        int i,j;
        float tempArray[size];

        for (i = 1; i <= size; i++)
                tempArray[i] = inputArray[i];

        //sort
        for (i = 1; i <= size; i++)
        {
                tempValue = tempArray[i];
                j = i;

                while ((j > 1) && (tempArray[j-1] > tempValue))
                {
                        tempArray[j] = tempArray[j-1];
                        j = j - 1;
                }
                tempArray[j] = tempValue;
        }

        return tempArray[int(0.1*size)];
}

void rankMitigations(float** Data, float* Distance)
{
        float tempScoreOff[TotalMitigations],tempScoreOn[TotalMitigations],
                tempBestFreqCount[TotalMitigations][2],tempRestFreqCount[TotalMitigations][2];

        for (mCounter = 1; mCounter <= TotalMitigations; mCounter++)
        {
                tempBestFreqCount[mCounter][0] = 0;
                tempBestFreqCount[mCounter][1] = 0;
                tempRestFreqCount[mCounter][0] = 0;
                tempRestFreqCount[mCounter][1] = 0;
        }

        float bestValue = findBestDistance(Distance,TotalInstances);

        for (instanceCounter = 1; instanceCounter <= TotalInstances; instanceCounter++)
        {
                if (displayMode == 3)
                {
                        for (mCounter=1; mCounter<=TotalMitigations; mCounter++)
                                printf("%.0f,",Data[instanceCounter][mCounter+2]);
                        printf("%.3f,%.3f,\t", Data[instanceCounter][1],Data[instanceCounter][2]);
                }

                //if it is in the Best distance from the sweet spot,
                //count the frequency of each mitigation (0 and 1) for the best instances
                if (Distance[instanceCounter] <= bestValue)
                {
```

```
                if (displayMode == 3) printf("%.3f best from %.3f\n", Distance[instanceCounter],bestValue);
                for (mCounter = 1; mCounter <= TotalMitigations; mCounter++)
                {
                        //keep track of the mitigation's counts if it is not already fixed
                        if (FixedMitigations[mCounter] == -1)
                        {
                                if (Data[instanceCounter][mCounter+2] == 0)
                                        tempBestFreqCount[mCounter][0]++;
                                else if (Data[instanceCounter][mCounter+2] == 1)
                                        tempBestFreqCount[mCounter][1]++;
                        }
                }
        }
        //else it is in the Rest distance from the sweet spot and so
        //count the frequency of each mitigation (0 and 1) for the rest instances
        else
        {
                if (displayMode == 3) printf("%.3f rest from %.3f\n", Distance[instanceCounter],bestValue);
                for (mCounter = 1; mCounter <= TotalMitigations; mCounter++)
                {
                        //keep track of the mitigation's counts if it is not already fixed
                        if (FixedMitigations[mCounter] == -1)
                        {
                                if (Data[instanceCounter][mCounter+2] == 0)
                                        tempRestFreqCount[mCounter][0]++;
                                else if (Data[instanceCounter][mCounter+2] == 1)
                                        tempRestFreqCount[mCounter][1]++;
                        }
                }
        }
}

float maxScore = -infinity;
int maxScoredMitigation = 0;
float maxScoredMitigationStatus = -1;
float best,rest;

//normalize each frequency count by dividing it by the total number of instances and score each mitigation
// using the best^2/(best+rest) and keep min and max
for (mCounter = 1; mCounter <= TotalMitigations; mCounter++)
{
        //do this only if mitigation is not fixed already
        if (FixedMitigations[mCounter] == -1)
        {
                //find the score of the mitigation when it is off
                best = tempBestFreqCount[mCounter][0]/TotalInstances;
                rest = tempRestFreqCount[mCounter][0]/TotalInstances;

                if (best == 0 && rest == 0)
                        tempScoreOff[mCounter] = 0;
                else
                        tempScoreOff[mCounter] = pow(best,2)/(best+rest);

                if (displayMode == 3) printf( "m%d with best:%.3f and rest:%.3f\n",mCounter,best,rest);

                //keep its information if it is the max score seen so far
                if (tempScoreOff[mCounter] > maxScore)
                {
                        maxScore = tempScoreOff[mCounter];
                        maxScoredMitigation = mCounter;
                        maxScoredMitigationStatus = 0;
                }

                //find the score of the mitigation when it is on
                best = tempBestFreqCount[mCounter][1]/TotalInstances;
                rest = tempRestFreqCount[mCounter][1]/TotalInstances;

                if (best == 0 && rest == 0)
                        tempScoreOn[mCounter] = 0;
                else
                        tempScoreOn[mCounter] = pow(best,2)/(best+rest);

                if (displayMode == 3) printf( "m%d with best:%.3f and rest:%.3f\n",mCounter,best,rest);

                //keep its information if it is the max score seen so far
                if (tempScoreOn[mCounter] > maxScore)
                {
                        maxScore = tempScoreOn[mCounter];
                        maxScoredMitigation = mCounter;
                        maxScoredMitigationStatus = 1;
                }
                if (displayMode == 3) printf( "score of m%d 0:%.3f 1:%.3f\n",mCounter,
                        tempScoreOff[mCounter],tempScoreOn[mCounter]);
        }
}

if (displayMode == 3)
```

```
        printf( "chosen mitigation is m%d with status %.0f has score %.3f\n",
                maxScoredMitigation,maxScoredMitigationStatus,maxScore);

        FixedMitigations[maxScoredMitigation] = maxScoredMitigationStatus;
        recentSetMitigation = maxScoredMitigation;
        recentSetMitigationStatus = maxScoredMitigationStatus;
}

void sweetSpot(float** Data, float* Distance)
{
        if (costFlag == 1)
                MaxCost = costLimit;
        if (attFlag == 1)
                MinAtt = attLimit;

        if (displayMode == 3) printf("MIN and MAX %.3f,%.3f,%.3f,%.3f\n",MinCost,MaxCost,MinAtt,MaxAtt);

        float normalizedCost,normalizedAtt;
        //normalize the att and cost using their
        for (instanceCounter = 1; instanceCounter <= TotalInstances; instanceCounter++)
        {
                normalizedCost = (Data[instanceCounter][1] - MinCost)/(MaxCost - MinCost + small);
                normalizedAtt = (Data[instanceCounter][2] - MinAtt)/(MaxAtt - MinAtt + small);
                Distance[instanceCounter] = pow(pow((normalizedCost - 0),2) + pow((normalizedAtt - 1),2),0.5);
        }
}

int selectValue(int val1, int val2)
{
        double randomValue = (double)rand()/((double)(RAND_MAX)+(double)(1));
        int returnValue;

        if (randomValue < 0.5)
                returnValue = val1;
        else
                returnValue = val2;
        return returnValue;
}

void addInstance(float costVar, float attVar, float** Data)
{
        TotalInstances++;
        Data[TotalInstances][1] = costVar;
        Data[TotalInstances][2] = attVar;

        for (mCounter = 1; mCounter <= TotalMitigations; mCounter++)
                Data[TotalInstances][mCounter+2] = mArray[mCounter];

        if (MinCost > Data[TotalInstances][1])
                MinCost = Data[TotalInstances][1];
        if (MaxCost < Data[TotalInstances][1])
                MaxCost = Data[TotalInstances][1];

        if (MinAtt > Data[TotalInstances][2])
                MinAtt = Data[TotalInstances][2];
        if (MaxAtt < Data[TotalInstances][2])
                MaxAtt = Data[TotalInstances][2];
}

float minValue(float val1, float val2)
{
        if (val1 < val2)
                return val1;
        else
                return val2;
}
```

B.1.3 keys2.c

```
#include "keys.h"

/*
########################################################################
#
#  KEYS2: DDP model optimization tool
#  Copyright (C) 2008-2009 Omid Jalali, Gregory Gay
#
#  This program is free software: you can redistribute it and/or modify
#  it under the terms of the GNU General Public License as published by
#  the Free Software Foundation, either version 3 of the License.
#
#  This program is distributed in the hope that it will be useful,
#  but WITHOUT ANY WARRANTY; without even the implied warranty of
#  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#  GNU General Public License for more details.
```

```
#
#  You should have received a copy of the GNU General Public License
#  along with this program.  If not, see <http://www.gnu.org/licenses/>.
#############################################################################
*/

int costFlag, attFlag, displayFlag, randomFlag, runFlag, maxFutileFlag;
float costLimit, attLimit;
int Seed;
int RunTotal;

float FixedMitigations[TotalMitigations+1];
float mArray[TotalMitigations+1];

int mCounter,era,run,TotalInstances,instanceCounter,displayMode;
float MinCost,MaxCost,MinAtt,MaxAtt;
float infinity,small;

int recentSetMitigation;
float recentSetMitigationStatus;

float LastMinCost,LastMaxAtt;

int mitigationsRemain=1;

int main(int argc, char **argv)
{
        //this is required to set up the ddp model.
        setupModel();

        Seed = 0;

        char *costValue = NULL;
        char *attValue = NULL;
        char *displayValue = NULL;
        char *randomValue = NULL;
        char *runValue = NULL;
        char *futileValue = NULL;
        int c;
        int futile = 0;
        int MaxFutile=1000000;
        int round=1;
        int ecntr=0;
        int tic=1;

        displayFlag = 0;
        costFlag = 0;
        attFlag = 0;
        randomFlag = 0;
        runFlag = 0;
        maxFutileFlag = 0;

        infinity = pow(10,20);
        small = pow(10,-20);

        opterr = 0;

        while ((c = getopt(argc, argv, "a:c:d:f:h:r:t:")) != -1)
        {
                switch (c)
                {
                        case 'a':
                                attFlag = 1;
                                attValue = optarg;
                                break;
                        case 'c':
                                costFlag = 1;
                                costValue = optarg;
                                break;
                        case 'd':
                                displayFlag = 1;
                                displayValue = optarg;
                                break;
                        case 'f':
                                maxFutileFlag = 1;
                                futileValue = optarg;
                                break;
                        case 'r':
                                randomFlag = 1;
                                randomValue = optarg;
                                break;
                        case 't':
                                runFlag = 1;
                                runValue = optarg;
                                break;
                        case 'h':
                        case '?':
```

```
                        printf("\nThe options must have the format -a AttainmentLowerLimit -c CostUpperLimit"+
                                "-d DisplayMode -f Futile -r Seed -t TotalRuns.\n");
                        printf("\nAttainmentLowerLimit:\n\tThis is the lower limit"+
                                "that the user can set for attainment.\n");
                        printf("\tThe tool tries to find a mitigation set that gives an attainment"+
                                "equal to or higher than this value.\n");
                        printf("\tHowever, this is not guaranteed especially for limits set too high.\n");
                        printf("\nCostUpperLimit:\n\tThis is the upper limit that the user can set for cost.\n");
                        printf("\tThe tool tries to find a mitigation set that gives a cost equal to"+
                                "or lower than this value.\n");
                        printf("\tHowever, this is not guaranteed especially for limits set too low.\n");
                        printf("\nDisplay Mode:\n\tDisplay mode 1 prints the final results.\n");
                        printf("\tDisplay mode 2 prints median and spread of each mitigation-fixing round.\n");
                        printf("\tDisplay mode 3 is for debugging purposes only and prints everything needed.\n");
                        printf("\tDisplay mode 4 is for debugging purposes only and prints cost and"+
                                "attainment of each run.\n");
                        printf("\nFutile:\n\tThis is the number of trials before the tool decides"+
                                "that no improvements were seen.\n");
                        printf("\tIt can be a number between 1 and the number of mitigations used.\n");
                        printf("\tIt can be turned off by setting it the number of mitigations"+
                                "used. The default value is 10.\n");
                        printf("\nSeed:\n\tSeed is used in random number generation."+
                                "By default, it is generated using the system time.\n");
                        printf("\nTotalRuns:\n\tThe number of total runs that is used internally."+
                                "The default value is 100.\n");
                        printf("\n");
                        return(1);
                        break;
                default:
                        break;
        }
}

if (attFlag == 1 && attValue != NULL)
        attLimit = (float)atof(attValue);
else
        attLimit = -infinity;

if (costFlag == 1 && costValue != NULL)
        costLimit = (float)atof(costValue);
else
        costLimit = infinity;

if (randomFlag == 1 && randomValue != NULL)
        Seed = atoi(randomValue);
else
        Seed = 1;

if (runFlag == 1 && runValue != NULL)
        RunTotal = atoi(runValue);
else
        RunTotal = 100;

if (displayFlag == 1 && displayValue != NULL)
        displayMode = atoi(displayValue);
else
        displayMode = 1;

if (maxFutileFlag == 1 && futileValue != NULL)
        MaxFutile = atoi(futileValue);
else
        MaxFutile = 1000000;

float att, cost;

float *Distance = new float[RunTotal+1];

float **Data = new float*[RunTotal+1];
for (int i = 0; i < RunTotal + 1; i++)
        Data[i] = new float[TotalMitigations+2];

if (randomFlag == 1)
        srand(Seed);
else
        srand((unsigned int)time(NULL));

//set all mitigations to non-fixed value of -1
for (mCounter = 1; mCounter <= TotalMitigations; mCounter++)
        FixedMitigations[mCounter] = -1;

if (displayMode == 2)
        printf("Era, Mitigation Number, Mitigation Value, Median of Cost,"+
                "Spread of Cost, Median of Attainment, Spread of Attainment\n");

MinCost = infinity;
MaxCost = -infinity;
```

```
        MinAtt = infinity;
        MaxAtt = -infinity;

        LastMinCost = MinCost;
        LastMaxAtt = MaxAtt;

        era=0;

        //for(mCounter=1; mCounter<=TotalMitigations; mCounter++)
        //{
        //      mArray[mCounter]=1;
        //}
        //model(&cost, &att, mArray);
        //printf("%.1f,%.5f\n",cost,att);

        while(mitigationsRemain)
        {
                TotalInstances = 0;

                for (run = 1; run <= RunTotal; run++)
                {
                        for (mCounter = 1; mCounter <= TotalMitigations; mCounter++)
                        {
                                //if in the previous runs the mitigation is fixed to a certain value (0 or 1) then use that value.
                                //otherwise, select it at random
                                if (FixedMitigations[mCounter] == -1)
                                        mArray[mCounter] = selectValue(0,1);
                                else
                                        mArray[mCounter] = FixedMitigations[mCounter];
                        }
                        //find the cost and att using these mitigations
                        model(&cost, &att, mArray);

                        if (displayMode == 4)
                                printf("%.1f,%.5f\n", cost, att);

                        //store the current instance
                        addInstance(cost, att, Data);
                }

                //find the sweet spot and distances from sweet spot for each distance
                sweetSpot(Data, Distance);

                //rank the mitigations and find the mitigation that should be fixed
                //printf("%i,%i\n",round,tic);
                for(ecntr=1;ecntr<=round;ecntr++)
                {
                        rankMitigations(Data, Distance);
                        era++;
                        if(era>=TotalMitigations)
                                break;
                }
                round++;

                if (displayMode == 2)
                        reportMedianAndSpread(Data);

//              printf("%d,futile:%d,%f,last:%f,%f,last:%f\n",era,futile,MinCost,LastMinCost,MaxAtt,LastMaxAtt);
                if (MinCost < LastMinCost && MaxAtt >= LastMaxAtt)
                {
                        LastMinCost = MinCost;
                        LastMaxAtt = MaxAtt;
                        futile = 0;
                }
                else if (futile > MaxFutile)
                {
                        if (displayMode == 1 || displayMode == 3)
                                printf("Terminated at era: %d\n",era);
                        era = TotalMitigations + 1;
                        futile++;
                }
                else
                        futile++;

                if(era>=TotalMitigations)
                        mitigationsRemain=0;

        }

        //show the final results
        if (displayMode == 1 || displayMode == 3)
        {
                for (mCounter = 1; mCounter <= TotalMitigations; mCounter++)
                {
                        mArray[mCounter] = FixedMitigations[mCounter];
                        if (FixedMitigations[mCounter] == -1)
                                mArray[mCounter] = 0;
```

```
                }
                model(&cost,&att,mArray);
                for (mCounter=1; mCounter<=TotalMitigations; mCounter++)
                        printf("m[%d],",mCounter);
                printf("cost,attainment\n");
                for (mCounter=1; mCounter<=TotalMitigations; mCounter++)
                        printf("%.0f,", mArray[mCounter]);
                printf("%.1f,%.5f\n",cost,att);
        }
}

void reportMedianAndSpread(float** Data)
{
        float tempCostArray[TotalInstances], tempAttArray[TotalInstances];
        float costMedian,costSpread,attMedian,attSpread;

        //sort the cost and att (individually) and find the median and spread
        for (instanceCounter = 1; instanceCounter <= TotalInstances; instanceCounter++)
        {
                tempCostArray[instanceCounter] = Data[instanceCounter][1];
                tempAttArray[instanceCounter] = Data[instanceCounter][2];
        }
        findMedianAndSpread(tempCostArray,TotalInstances,&costMedian,&costSpread);
        findMedianAndSpread(tempAttArray,TotalInstances,&attMedian,&attSpread);

        printf("%d,%d,%.0f,%.5f,%.5f,%.5f,%.5f\n",era,recentSetMitigation,
                recentSetMitigationStatus,costMedian,costSpread,attMedian,attSpread);
}

void findMedianAndSpread(float inputArray[], int size, float *median, float *spread)
{
        float tempValue;
        int i,j;
        float tempArray[size];

        for (i = 1; i <= size; i++)
                tempArray[i] = inputArray[i];

        //sort
        for (i = 1; i <= size; i++)
        {
                tempValue = tempArray[i];
                j = i;

                while ((j > 1) && (tempArray[j-1] > tempValue))
                {
                        tempArray[j] = tempArray[j-1];
                        j = j - 1;
                }
                tempArray[j] = tempValue;
        }

        *median = tempArray[size/2];
        *spread = tempArray[3*size/4] - tempArray[size/2];
}

float findBestDistance(float inputArray[], int size)
{
        float tempValue;
        int i,j;
        float tempArray[size];

        for (i = 1; i <= size; i++)
                tempArray[i] = inputArray[i];

        //sort
        for (i = 1; i <= size; i++)
        {
                tempValue = tempArray[i];
                j = i;

                while ((j > 1) && (tempArray[j-1] > tempValue))
                {
                        tempArray[j] = tempArray[j-1];
                        j = j - 1;
                }
                tempArray[j] = tempValue;
        }

        return tempArray[int(0.1*size)];
}

void rankMitigations(float** Data, float* Distance)
{
        float tempScoreOff[TotalMitigations],tempScoreOn[TotalMitigations];
        float tempBestFreqCount[TotalMitigations][2],tempRestFreqCount[TotalMitigations][2];
```

```
for (mCounter = 1; mCounter <= TotalMitigations; mCounter++)
{
        tempBestFreqCount[mCounter][0] = 0;
        tempBestFreqCount[mCounter][1] = 0;
        tempRestFreqCount[mCounter][0] = 0;
        tempRestFreqCount[mCounter][1] = 0;
}

float bestValue = findBestDistance(Distance,TotalInstances);

for (instanceCounter = 1; instanceCounter <= TotalInstances; instanceCounter++)
{
        if (displayMode == 3)
        {
                for (mCounter=1; mCounter<=TotalMitigations; mCounter++)
                        printf("%.0f,",Data[instanceCounter][mCounter+2]);
                printf("%.3f,%.3f,\t", Data[instanceCounter][1],Data[instanceCounter][2]);
        }

        //if it is in the Best distance from the sweet spot,
        //count the frequency of each mitigation (0 and 1) for the best instances
        if (Distance[instanceCounter] <= bestValue)
        {
                if (displayMode == 3) printf("%.3f best from %.3f\n", Distance[instanceCounter],bestValue);
                for (mCounter = 1; mCounter <= TotalMitigations; mCounter++)
                {
                        //keep track of the mitigation's counts if it is not already fixed
                        if (FixedMitigations[mCounter] == -1)
                        {
                                if (Data[instanceCounter][mCounter+2] == 0)
                                        tempBestFreqCount[mCounter][0]++;
                                else if (Data[instanceCounter][mCounter+2] == 1)
                                        tempBestFreqCount[mCounter][1]++;
                        }
                }
        }
        //else it is in the Rest distance from the sweet spot and so count
        //the frequency of each mitigation (0 and 1) for the rest instances
        else
        {
                if (displayMode == 3) printf("%.3f rest from %.3f\n", Distance[instanceCounter],bestValue);
                for (mCounter = 1; mCounter <= TotalMitigations; mCounter++)
                {
                        //keep track of the mitigation's counts if it is not already fixed
                        if (FixedMitigations[mCounter] == -1)
                        {
                                if (Data[instanceCounter][mCounter+2] == 0)
                                        tempRestFreqCount[mCounter][0]++;
                                else if (Data[instanceCounter][mCounter+2] == 1)
                                        tempRestFreqCount[mCounter][1]++;
                        }
                }
        }
}

float maxScore = -infinity;
int maxScoredMitigation = 0;
float maxScoredMitigationStatus = -1;
float best,rest;

//normalize each frequency count by dividing it by the total number of instances
//and score each mitigation using the best^2/(best+rest)
// and keep min and max
for (mCounter = 1; mCounter <= TotalMitigations; mCounter++)
{
        //do this only if mitigation is not fixed already
        if (FixedMitigations[mCounter] == -1)
        {
                //find the score of the mitigation when it is off
                best = tempBestFreqCount[mCounter][0]/TotalInstances;
                rest = tempRestFreqCount[mCounter][0]/TotalInstances;

                if (best == 0 && rest == 0)
                        tempScoreOff[mCounter] = 0;
                else
                        tempScoreOff[mCounter] = pow(best,2)/(best+rest);

                if (displayMode == 3) printf( "m%d with best:%.3f and rest:%.3f\n",mCounter,best,rest);

                //keep its information if it is the max score seen so far
                if (tempScoreOff[mCounter] > maxScore)
                {
                        maxScore = tempScoreOff[mCounter];
                        maxScoredMitigation = mCounter;
                        maxScoredMitigationStatus = 0;
                }
```

```
                        //find the score of the mitigation when it is on
                        best = tempBestFreqCount[mCounter][1]/TotalInstances;
                        rest = tempRestFreqCount[mCounter][1]/TotalInstances;

                        if (best == 0 && rest == 0)
                                tempScoreOn[mCounter] = 0;
                        else
                                tempScoreOn[mCounter] = pow(best,2)/(best+rest);

                        if (displayMode == 3) printf( "m%d with best:%.3f and rest:%.3f\n",mCounter,best,rest);

                        //keep its information if it is the max score seen so far
                        if (tempScoreOn[mCounter] > maxScore)
                        {
                                maxScore = tempScoreOn[mCounter];
                                maxScoredMitigation = mCounter;
                                maxScoredMitigationStatus = 1;
                        }
                        if (displayMode == 3) printf( "score of m%d 0:%.3f 1:%.3f\n",
                                mCounter,tempScoreOff[mCounter],tempScoreOn[mCounter]);
                }
        }

        if (displayMode == 3) printf( "chosen mitigation is m%d with status %.0f has score %.3f\n",maxScoredMitigation,
        maxScoredMitigationStatus,maxScore);

        FixedMitigations[maxScoredMitigation] = maxScoredMitigationStatus;
        recentSetMitigation = maxScoredMitigation;
        recentSetMitigationStatus = maxScoredMitigationStatus;
}

void sweetSpot(float** Data, float* Distance)
{
        if (costFlag == 1)
                MaxCost = costLimit;
        if (attFlag == 1)
                MinAtt = attLimit;

        if (displayMode == 3) printf("MIN and MAX %.3f,%.3f,%.3f,%.3f\n",MinCost,MaxCost,MinAtt,MaxAtt);

        float normalizedCost,normalizedAtt;
        //normalize the att and cost using their
        for (instanceCounter = 1; instanceCounter <= TotalInstances; instanceCounter++)
        {
                normalizedCost = (Data[instanceCounter][1] - MinCost)/(MaxCost - MinCost + small);
                normalizedAtt = (Data[instanceCounter][2] - MinAtt)/(MaxAtt - MinAtt + small);
                Distance[instanceCounter] = pow(pow((normalizedCost - 0),2) + pow((normalizedAtt - 1),2),0.5);
        }
}

int selectValue(int val1, int val2)
{
        double randomValue = (double)rand()/((double)(RAND_MAX)+(double)(1));
        int returnValue;

        if (randomValue < 0.5)
                returnValue = val1;
        else
                returnValue = val2;
        return returnValue;
}

void addInstance(float costVar, float attVar, float** Data)
{
        TotalInstances++;
        Data[TotalInstances][1] = costVar;
        Data[TotalInstances][2] = attVar;

        for (mCounter = 1; mCounter <= TotalMitigations; mCounter++)
                Data[TotalInstances][mCounter+2] = mArray[mCounter];

        if (MinCost > Data[TotalInstances][1])
                MinCost = Data[TotalInstances][1];
        if (MaxCost < Data[TotalInstances][1])
                MaxCost = Data[TotalInstances][1];

        if (MinAtt > Data[TotalInstances][2])
                MinAtt = Data[TotalInstances][2];
        if (MaxAtt < Data[TotalInstances][2])
                MaxAtt = Data[TotalInstances][2];
}

float minValue(float val1, float val2)
{
        if (val1 < val2)
                return val1;
        else
```

```
        return val2;
}
```

B.2 ASTAR

B.2.1 astar.h

```
#ifndef _astar_h
#define _astar_h

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <unistd.h>
#include "model.h"

#define TotalMitigations MITIGATION

int selectValue(int val1, int val2);
float minValue(float val1, float val2);
void model(float *cost, float *att, float m[]);
void loop();
double score(float att, float cost);
void reportMedianAndSpread(float closedCost[], float closedAttain[], int c);
void findMedianAndSpread(float inputArray[], int size, float *median, float *spread);


#endif
```

B.2.2 astar.c

```
/*
#########################################################################
#
#  astar: Low-cost path search for DDP models
#  Copyright (C) 2008 Gregory Gay <greg@4colorrebellion.com>
#
#  This program is free software: you can redistribute it and/or modify
#  it under the terms of the GNU General Public License as published by
#  the Free Software Foundation, either version 3 of the License.
#
#  This program is distributed in the hope that it will be useful,
#  but WITHOUT ANY WARRANTY; without even the implied warranty of
#  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#  GNU General Public License for more details.
#
#  You should have received a copy of the GNU General Public License
#  along with this program.  If not, see <http://www.gnu.org/licenses/>.
#########################################################################
*/

#include "astar.h"

float costLimit=0.0;
float attLimit=0.0;

float maxCost,minCost,maxAtt,minAtt;
float mArray[TotalMitigations+1];

int mCounter;
float infinity = pow(10,20);
float small= pow(10,-20);

int startmode=0;
int dispMode=0;

int main(int argc, char *argv[])
{
        setupModel();
        int i;
        float bestScore;
        unsigned int seed= (unsigned int)time(NULL);

        for (i=1; i<argc; i++)
        {
                if(strcmp(argv[i],"-cost")==0)
                {
                        costLimit=atof(argv[i+1]);
                        i++;
```

```
                }
                else if(strcmp(argv[i],"-attainment")==0)
                {
                        attLimit=atof(argv[i+1]);
                        i++;
                }
                else if(strcmp(argv[i],"-seed")==0)
                {
                        seed=(unsigned int) atoi(argv[i+1]);
                        i++;
                }
                else if(strcmp(argv[i],"-start")==0)
                {
                        startmode=1;
                }
                else if(strcmp(argv[i],"-mode")==0)
                {
                        dispMode=1;
                }
                else
                {
                        printf("Invalid option entered. Valid flags include:\n -cost [value] : cost goal\n"+
                                "-attainment [value] : attainment goal\n -seed [value] : random number seed\n"+
                                "-start : Use a fixed starting point\n -mode : Verbose mode\n");
                        exit(1);
                }
        }

        srand(seed);

        loop();
}

void loop()
{
        int i, j, k, l, n, o, p,bestNeighbor,sameNeighbor;
        int tempN=0;
        float att,cost,bestCost, bestAtt,f,g,h,bestF;
        double threshold, currentScore, bestScore;
        float bestM[TotalMitigations+1];
        float tempM[TotalMitigations+1];

        float closed[TotalMitigations+20][TotalMitigations+1];
        float closedCost[TotalMitigations+20];
        float closedAttain[TotalMitigations+20];

        //Initialize the closed array to 0
        for(i=0;i<TotalMitigations+20;i++)
        {
                closed[i][1]=5.0;
        }

        int c=0;

        float neighbor[TotalMitigations][TotalMitigations+1];
        float neighborCost[TotalMitigations];
        float neighborAttain[TotalMitigations];
        double neighborDistance[TotalMitigations];
        double randomValue;
        int found=0;
        maxCost=infinity;
        minCost=-infinity;
        maxAtt=-infinity;
        minAtt=infinity;

        //make initial random assignment
        for (mCounter = 1; mCounter <= TotalMitigations; mCounter++)
        {
                if(startmode==0)
                        mArray[mCounter] = selectValue(0,1);
                else if(startmode==1)
                        mArray[mCounter]=0;
        }

        //Add initial state to the closed list
        for(mCounter=1;mCounter<=TotalMitigations;mCounter++)
        {
                closed[c][mCounter]=mArray[mCounter];
        }
        model(&cost,&att,mArray);
        float startingDistance=score(att,cost);
        closedCost[c]=cost;
        closedAttain[c]=att;
        c++;

        //Traverses neighbors until best is found
        while(found==0)
```

```
{
        //Compute neighbors by storing every possible migiation combination with only one change
        for(j=0;j<TotalMitigations;j++)
        {
                for(mCounter=1;mCounter<=TotalMitigations;mCounter++)
                {
                        if(mCounter==j+1)
                        {
                                if(mArray[mCounter]==0.0)
                                        neighbor[j][mCounter]=1.0;
                                else
                                        neighbor[j][mCounter]=0.0;
                        }
                        else
                                neighbor[j][mCounter]=mArray[mCounter];
                }
        }

        //Make computations for each neighbor
        for(l=0;l<TotalMitigations;l++)
        {
                int same=1;
                //Check to see if it is on the closed list.
                for(n=0;n<TotalMitigations;n++)
                {
                        if(closed[n][1]==5.0)
                                break;

                        same=1;
                        for(mCounter=1;mCounter<=TotalMitigations;mCounter++)
                        {
                                if(same==0)
                                        break;
                                else if(neighbor[l][mCounter]!=closed[n][mCounter])
                                        same=0;
                        }
                }
                if(same==1)
                {
                        //printf("On the closed list\n");
                        continue;
                }

                //Compute cost and attainment for that neighbor

                for(mCounter=1;mCounter<=TotalMitigations;mCounter++)
                {
                        tempM[mCounter]=neighbor[l][mCounter];
                }
                model(&neighborCost[l],&neighborAttain[l],tempM);

                if(minCost==-infinity)
                        minCost=neighborCost[l];
                if(maxCost==infinity)
                        maxCost=neighborCost[l];

                if(minAtt==infinity)
                        minAtt=neighborAttain[l];
                if(maxAtt==-infinity)
                        maxAtt=neighborAttain[l];

                //Compute distance for each neighbor
                neighborDistance[l]=score(neighborAttain[l],neighborCost[l]);
        }

        //Compute "threshold" distance for h(x)
        threshold=score(attLimit,costLimit);
        startingDistance=score(closedAttain[0],closedCost[0]);

        //Computer att,cost,distance for current state
        model(&cost,&att,mArray);
        currentScore=score(att,cost);

        //Compare with each neighbor and choose neighbor with best g+h
        bestF=infinity;
        bestNeighbor=0;

        for(o=0;o<TotalMitigations;o++)
        {
                g=neighborDistance[o]-(startingDistance-currentScore);
                h=neighborDistance[o]-threshold;
                f=g+h;

                if(f<bestF)
                {
                        bestF=f;
                        bestNeighbor=o;
```

```
                        }
                }

                if(tempN==bestNeighbor)
                        sameNeighbor++;
                else
                        sameNeighbor=0;

                //Sets best neighbor as new state and adds to closed list

                if((costLimit/neighborCost[bestNeighbor]>0.8)&&(costLimit/neighborCost[bestNeighbor]<1.2)
                        &&(attLimit/neighborAttain[bestNeighbor]>0.8)&&(attLimit/neighborAttain[bestNeighbor]<1.2))
                {
                        printf("%f, %f\n",neighborCost[bestNeighbor],neighborAttain[bestNeighbor]);
                        found=1;
                }

                for(mCounter=1;mCounter<=TotalMitigations;mCounter++)
                {
                        mArray[mCounter]=neighbor[bestNeighbor][mCounter];
                        closed[c][mCounter]=mArray[mCounter];
                }
                closedCost[c]=neighborCost[bestNeighbor];
                closedAttain[c]=neighborAttain[bestNeighbor];
                c++;
                tempN=bestNeighbor;

                if(sameNeighbor>=10)
                {
                        printf("%f, %f\n",cost,att);
                        found=1;
                }
                else
                {
                        if(dispMode==1)
                        {
                                printf("%f, %f\n",neighborCost[bestNeighbor],neighborAttain[bestNeighbor]);
                        }
                }
        }
}

double score(float att, float cost)
{
        if(cost<minCost)
                minCost=cost;
        else if(cost>maxCost)
                maxCost=cost;

        if(att<minAtt)
                minAtt=att;
        else if(att>maxAtt)
                maxAtt=att;

        double normalizedCost,normalizedAtt, distance;

        normalizedCost=(double) ((cost-minCost)/(maxCost-minCost+small));
        normalizedAtt= (double) ((att-minAtt)/(maxAtt-minAtt+small));

        distance= pow(pow((normalizedCost),2)+pow((normalizedAtt-1),2),0.5);
        return distance;
}

float minValue(float val1, float val2)
{
        if (val1 < val2)
                return val1;
        else
                return val2;
}

int selectValue(int val1, int val2)
{
        double randomValue = (double)rand()/((double)(RAND_MAX)+(double)(1));
        int returnValue;

        //printf("%d\n", randomValue);

        if (randomValue < 0.5)
                returnValue = val1;
        else
                returnValue = val2;

        //printf("%i\n", returnValue);
        return returnValue;
}
```

```
void reportMedianAndSpread(float closedCost[], float closedAttain[], int c)
{
        float tempCostArray[c], tempAttArray[c];
        float costMedian,costSpread,attMedian,attSpread;
        int i;

        //sort the cost and att (individually) and find the median and spread
        for (i = 0; i <c; i++)
        {
                tempCostArray[i] = closedCost[i];
                tempAttArray[i] = closedAttain[i];
        }
        findMedianAndSpread(tempCostArray,c,&costMedian,&costSpread);
        findMedianAndSpread(tempAttArray,c,&attMedian,&attSpread);

        printf("%.5f,%.5f,%.5f,%.5f\n",costMedian,costSpread,attMedian,attSpread);
}

void findMedianAndSpread(float inputArray[], int size, float *median, float *spread)
{
        float tempValue;
        int i,j;
        float tempArray[size];

        for (i = 0; i < size; i++)
                tempArray[i] = inputArray[i];

        for (i = 0; i < size; i++)
        {
                tempValue = tempArray[i];
                j = i;

                while ((j > 0) && (tempArray[j-1] > tempValue))
                {
                        tempArray[j] = tempArray[j-1];
                        j = j - 1;
                }
                tempArray[j] = tempValue;
        }

        *median = tempArray[size/2];
        *spread = tempArray[3*size/4] - tempArray[size/2];
}
```

B.3 MaxFunWalk

B.3.1 maxfunwalk.h

```
#ifndef _maxfunwalk_h
#define _maxfunwalk_h

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <unistd.h>
#include <string.h>
#include "model.h"

#define TotalMitigations MITIGATION
#define RunTotal 100
#define Best 10

int selectValue(int val1, int val2);
float minValue(float val1, float val2);
void model(float *cost, float *att, float m[]);
float loop();
float score(float att, float cost);
int numYes(float mArray[]);

#endif
```

B.3.2 maxfunwalk.c

```
/*
#############################################################################
#
#  MaxFunWalk: Local search for DDP models
#  Copyright (C) 2008 Gregory Gay <greg@4colorrebellion.com>
#
```

```
#  This program is free software: you can redistribute it and/or modify
#  it under the terms of the GNU General Public License as published by
#  the Free Software Foundation, either version 3 of the License.
#
#  This program is distributed in the hope that it will be useful,
#  but WITHOUT ANY WARRANTY; without even the implied warranty of
#  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#  GNU General Public License for more details.
#
#  You should have received a copy of the GNU General Public License
#  along with this program.  If not, see <http://www.gnu.org/licenses/>.
########################################################################
*/


#include "maxfunwalk.h"

int maxtries=100;
int maxchanges=100;

float costLimit=0.0;
float attLimit=0.0;

float maxCost,minCost,maxAtt,minAtt;
float mArray[TotalMitigations+1];

int mCounter;
float infinity = pow(10,20);
float small= pow(10,-20);


int main(int argc, char *argv[])
{
        setupModel();
        int i;
        float bestScore;
        unsigned int seed= (unsigned int)time(NULL);

        for (i=1; i<argc; i++)
        {
                if(strcmp(argv[i],"-maxtries")==0)
                {
                        maxtries=atoi(argv[i+1]);
                        i++;
                }
                else if(strcmp(argv[i],"-maxchanges")==0)
                {
                        maxchanges=atoi(argv[i+1]);
                        i++;
                }
                else if(strcmp(argv[i],"-cost")==0)
                {
                        costLimit=atof(argv[i+1]);
                        i++;
                }
                else if(strcmp(argv[i],"-attainment")==0)
                {
                        attLimit=atof(argv[i+1]);
                        i++;
                }
                else if(strcmp(argv[i],"-seed")==0)
                {
                        seed=(unsigned int) atoi(argv[i+1]);
                        i++;
                }
                else
                {
                        printf("Invalid flag. Valid flags include:\n -cost [value] : Desired Cost\n -attainment [value] :"+
                                "Desired Value\n -maxtries [value] : Max Retries\n -maxchanges [value] : "+
                                "Max Changes per Retry\n -seed [value] : Random Number Seed\n");
                        exit(1);
                }
        }

        srand(seed);

        bestScore= loop();
}

float loop()
{
        int j, k, l, n, randomValue2, maximized, numY;
        int maximizedTries=0;
        float att,cost,currentScore,newScore,threshold, bestScore, bestCost, bestAtt, tempAtt;
        int randomSelection[(TotalMitigations/2)+1];
        float bestM[TotalMitigations+1];
        double randomValue;
```

```
maxCost=infinity;
minCost=-infinity;
maxAtt=-infinity;
minAtt=infinity;

for(j=0; j<maxtries; j++)
{
        n=0;
        //make random assignment
        for (mCounter = 1; mCounter <= TotalMitigations; mCounter++)
        {
                mArray[mCounter] = selectValue(0,1);
                n++;
        }

        for(k=0; k<maxchanges; k++)
        {
                model(&cost,&att,mArray);

                if(minCost==-infinity)
                        minCost=cost;
                if(maxCost==infinity)
                        maxCost=cost;

                if(minAtt==infinity)
                        minAtt=att;
                if(maxAtt==-infinity)
                        maxAtt=att;

                if(cost<minCost)
                        minCost=cost;
                else if(cost>maxCost)
                        maxCost=cost;

                if(att<minAtt)
                        minAtt=att;
                else if(att>maxAtt)
                        maxAtt=att;

                //Scores and exits if threshold met

                currentScore=score(att,cost);
                if(bestScore==0)
                {
                        bestScore=currentScore;
                        bestAtt=att;
                        bestCost=cost;
                        for(mCounter=1;mCounter<=TotalMitigations; mCounter++)
                        {
                                bestM[mCounter]=mArray[mCounter];
                        }
                }
                threshold = score(attLimit,costLimit);
                if(((threshold/currentScore)>=0.9999999)&&((threshold/currentScore)<=1.0000001))
                {
                        if(((threshold/currentScore)>=(threshold/bestScore))&&((threshold/currentScore)<=1.0))
                        {
                                bestScore=currentScore;
                                bestCost=cost;
                                bestAtt=att;
                                for(mCounter=1;mCounter<=TotalMitigations; mCounter++)
                                {
                                        bestM[mCounter]=mArray[mCounter];
                                }
                        }
                        printf("%f,%f\n",cost,att);

                        return bestScore;
                }
                if(((threshold/currentScore)>=(threshold/bestScore))&&((threshold/currentScore)<=1.0))
                {
                        bestScore=currentScore;
                        bestCost=cost;
                        bestAtt=att;
                        for(mCounter=1;mCounter<=TotalMitigations; mCounter++)
                        {
                                bestM[mCounter]=mArray[mCounter];
                        }
                }

                //Random part of selection

                for(l=0;l<(TotalMitigations/2)+1;l++)
                {
                        randomValue = (double)rand()/((double)(RAND_MAX)+(double)(1)) * (TotalMitigations+1);
                        if( (int) randomValue ==0)
                                randomValue=1;
```

```
                        randomSelection[1]= (int) randomValue;
                }

                randomValue = (double)rand()/((double)(RAND_MAX)+(double)(1));

                if(randomValue <= 0.3)
                {
                        //with probability p, change a setting in c suggested by best

                        randomValue2 = (int) ((double)rand()/((double)(RAND_MAX)+(double)(1)) * ((TotalMitigations/2)+1));
                        if(randomValue2==0)
                                randomValue2=1;
                        if(mArray[randomSelection[randomValue2]]==0)
                                mArray[randomSelection[randomValue2]]=1;
                        else
                                mArray[randomSelection[randomValue2]]=0;

                }
                else
                {
                        //else change a setting in c that maximizes score
                        for(maximizedTries=0;maximizedTries<(TotalMitigations/2)+1;maximizedTries++)
                        {
                                numY=numYes(mArray);
                                if((mArray[randomSelection[maximizedTries]]==0)&&(numY<=(TotalMitigations/3)))
                                        mArray[randomSelection[maximizedTries]]=1;
                                else
                                        mArray[randomSelection[maximizedTries]]=0;

                                tempAtt=att;
                                model(&cost,&att,mArray);
                                newScore=score(att,cost);
                                if(((threshold/newScore)>=(threshold/currentScore))&&((threshold/newScore)<=1.0)
                                        &&(att>=tempAtt))
                                {
                                        currentScore=newScore;
                                        break;

                                }
                                else
                                {
                                        if((mArray[randomSelection[maximizedTries]]==0)&&(numY<=(TotalMitigations/3)))
                                                mArray[randomSelection[maximizedTries]]=1;
                                        else
                                                mArray[randomSelection[maximizedTries]]=0;
                                }

                        }
                }

        }

        printf("%f,%f\n",bestCost,bestAtt);
        return bestScore;
}

float score(float att, float cost)
{
        float normalizedCost,normalizedAtt,distance;

        normalizedCost=(cost-minCost)/(maxCost-minCost+small);
        normalizedAtt= (att-minAtt)/(maxAtt-minAtt+small);

        distance= pow(pow((normalizedCost),2)+pow((normalizedAtt-1),2),0.5);
        return distance;
}

int numYes(float mArray[])
{
        int cntr,numY;
        for(cntr=1;cntr<=TotalMitigations;cntr++)
        {
                if(mArray[cntr]==1.0)
                {
                        numY++;
                }
        }

        return numY;
}

float minValue(float val1, float val2)
{
        if (val1 < val2)
                return val1;
        else
```

```
                return val2;
}

int selectValue(int val1, int val2)
{
        double randomValue = (double)rand()/((double)(RAND_MAX)+(double)(1));
        int returnValue;


        if (randomValue <= 0.5)
                returnValue = val1;
        else
                returnValue = val2;

        return returnValue;
}
```

B.4 Simulated Annealing

B.4.1 sa.h

```
#ifndef _sa_h
#define _sa_h

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <unistd.h>
#include "model.h"

#define TotalMitigations MITIGATION

void neighbor(float inputArray[], float outputArray[]);
double prob(float distance, float neighborDistance, int k, double T);
double temp(int k, int kmax);
float findScore(float FixedMitigations[]);
int selectValue(int val1, int val2);
float minValue(float val1, float val2);
void model(float *cost, float *att, float m[]);

#endif
```

B.4.2 sa.c

```
#include "sa.h"

int costFlag, costLimitFlag, attFlag, stepFlag, scoreFlag, randomFlag;
float costFactor, costLimit, attFactor, minScore;
int kmax;

int main(int argc, char **argv)
{
        //this is required to set up the ddp model.
        setupModel();

        int Seed;

        char *randomValue = NULL;
        char *costValue = NULL;
        char *costLimitValue = NULL;
        char *attValue = NULL;
        char *stepValue = NULL;
        char *scoreValue = NULL;
        int c;

        randomFlag = 0;
        costFlag = 0;
        costLimitFlag = 0;
        attFlag = 0;
        stepFlag = 0;
        scoreFlag = 0;

        opterr = 0;

        while ((c = getopt(argc, argv, "a:c:l:m:r:s:")) != -1)
        {
                switch (c)
                {
                        case 'a':
```

```
                        attFlag = 1;
                        attValue = optarg;
                        break;
                case 'c':
                        costFlag = 1;
                        costValue = optarg;
                        break;
                case 'l':
                        costLimitFlag = 1;
                        costLimitValue = optarg;
                        break;
                case 'm':
                        stepFlag = 1;
                        stepValue = optarg;
                        break;
                case 'r':
                        randomFlag = 1;
                        randomValue = optarg;
                        break;
                case 's':
                        scoreFlag = 1;
                        scoreValue = optarg;
                        break;
                case '?':
                        printf("You entered an unknown option. The options must have the format -a AttainmentScaleFactor
                                -c CostScaleFactor -l UpperCostLimit -m MaxStep -r Seed -s MaxScore.\n");
                        break;
                default:
                        break;
        }
    }

    if (attFlag == 1 && attValue != NULL)
        attFactor = atof(attValue);
    else
        attFactor = 1; //2000;

    if (costLimitFlag == 1 && costLimitValue != NULL)
        costLimit = atof(costLimitValue);
    else
        costLimit = 250000; //300000;

    if (costFlag == 1 && costValue != NULL)
        costFactor = atof(costValue);
    else
        costFactor = 0; //1;

    if (randomFlag == 1 && randomValue != NULL)
        Seed = atoi(randomValue);
    else
        Seed = 1;

    if (stepFlag == 1 && stepValue != NULL)
        kmax = atoi(stepValue);
    else
        kmax = 500000; //200000; //100000;

    if (scoreFlag == 1 && scoreValue != NULL)
        minScore = atof(scoreValue);
    else
        minScore = 200; //150; //100000;

    if (randomFlag == 1)
        srand(Seed);
    else
        srand((unsigned int)time(NULL));

    float FixedMitigations[TotalMitigations+1];
    float NeighborFixedMitigations[TotalMitigations+1];
    float score, neighborScore;
    int k = 0;
    int mCounter;

    //initialize the current state initially
    for (mCounter = 1; mCounter <= TotalMitigations; mCounter++)
        FixedMitigations[mCounter] = 0;

    //find the initial energy (score)
    score = findScore(FixedMitigations);

    while (k < kmax && score < minScore)
    {
//          printf("%d,",k);
        neighbor(FixedMitigations, NeighborFixedMitigations);
        neighborScore = findScore(NeighborFixedMitigations);
        //if improvement seen switch to the current one
        if (neighborScore > score)
```

```
                {
                        for (mCounter = 1; mCounter <= TotalMitigations; mCounter++)
                                FixedMitigations[mCounter] = NeighborFixedMitigations[mCounter];
                        score = neighborScore;
                }
                else
                {
                        double randomValue = (double)rand()/((double)(RAND_MAX)+(double)(1));
                        if (prob(score, neighborScore, k, temp(k, kmax)) > randomValue)
                        {
                                for (mCounter = 1; mCounter <= TotalMitigations; mCounter++)
                                        FixedMitigations[mCounter] = NeighborFixedMitigations[mCounter];
                                score = neighborScore;
                        }
                }
                k++;
        }

        //print FixedMitigations
        float att, cost;
        for (mCounter=1; mCounter<=TotalMitigations; mCounter++)
                printf("m[%d],",mCounter);
        printf("cost,attainment\n");
        for (mCounter=1; mCounter<=TotalMitigations; mCounter++)
                printf("%.0f,", FixedMitigations[mCounter]);
        model(&cost,&att,FixedMitigations);
        printf("%.1f,%.5f\n",cost,att);
}

void neighbor(float inputArray[], float outputArray[])
{
        int mCounter;
        double randomValue;
        for (mCounter = 1; mCounter <= TotalMitigations; mCounter++)
        {
                randomValue = (double)rand()/((double)(RAND_MAX)+(double)(1));
                if (randomValue < 0.05)
                        //flip it from 0 to 1 and from 1 to 0
                        outputArray[mCounter] = 1 - inputArray[mCounter];
                else
                        outputArray[mCounter] = inputArray[mCounter];
        }
}

double prob(float score, float neighborScore, int k, double T)
{
        return (double)exp(((double)score - (double)neighborScore) * k / T);
}

double temp(int k, int kmax)
{
        return (double)(kmax - k)/kmax;
}

float findScore(float FixedMitigations[])
{
        int mCounter;
        float att, cost, score;
        float mArray[TotalMitigations+1];

        for (mCounter = 1; mCounter <= TotalMitigations; mCounter++)
                        mArray[mCounter] = FixedMitigations[mCounter];

        //find the cost and att using these mitigations
        model(&cost,&att,mArray);

        score = attFactor*att - costFactor*cost;

        //this is to penalize the ones with too high of a cost
        if (cost > costLimit)// && costFactor == 0)
                score = att - cost;

//      printf("%.1f,%.3f,%.1f\t%.1f\n",cost, att,score,cost/att);

        return score;
}

int selectValue(int val1, int val2)
{
        double randomValue = (double)rand()/((double)(RAND_MAX)+(double)(1));
        int returnValue;

        if (randomValue < 0.5)
                returnValue = val1;
        else
                returnValue = val2;
        return returnValue;
```

```
}

float minValue(float val1, float val2)
{
        if (val1 < val2)
                return val1;
        else
                return val2;
}
```