A Baseline Method For Search-Based Software Engineering

Gregory Gay
University of Minnesota
Minneapolis, MN
greg@greggay.com

ABSTRACT

Background: Search-based Software Engineering (SBSE) uses a variety of techniques such as evolutionary algorithms or meta-heuristic searches but lacks a standard baseline method.
Aims: The KEYS2 algorithm meets the criteria of a baseline. It is fast, stable, easy to understand, and presents results that are competitive with standard techniques.
Method: KEYS2 operates on the theory that a small subset of variables control the majority of the search space. It uses a greedy search and a Bayesian ranking heuristic to fix the values of these variables, which rapidly forces the search towards stable high-scoring areas.
Results: KEYS2 is faster than standard techniques, presents competitive results (assessed with a rank-sum test), and offers stable solutions.
Conclusions: KEYS2 is a valid candidate to serve as a baseline technique for SBSE research.

Categories and Subject Descriptors

I.2.8 [**Problem Solving, Control Methods, and Search**]: Heuristic methods; D.2.8 [**Requirements/Specifications**]: Tools

Keywords

search-based software engineering, requirements optimization, meta-heuristic search

PROMISE2010, Sep 12-13, 2010. Timisoara, Romania

.

1. INTRODUCTION

Despite decades of research, testing, trials, and deliberation from both industry and academia, many fundamental questions of the software engineering field remain unanswered. Given the scope of many of these issues (for example, the entire space of programs developed in JAVA) or the required balancing between competing factors (i.e. the amount of money spent versus the completion of goals), many of these questions are *unanswerable*. These are the areas where search-based software engineering (SBSE) has typically excelled.

Search-based software engineering is the practice of reformulating typical software engineering issues as search problems, and applying meta-heuristic methods to find solutions. For any problem that represents a set of competing - yet equally important - factors, there will be no single perfect solution. Instead, there are likely to be several *near-optimal* solutions. In this implied space of trade-offs, meta-heuristic search techniques are ideal for sifting through a number of potential solutions for those that contain an ideal balancing of factors.

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 [25]. Since then, the SBSE research community has expanded rapidly. SBSE research has been applied successfully to requirements engineering [2,39,43], project cost estimation [7,10,33], testing [4, 17, 40], software maintenance [23, 37], transformation [3, 13, 24] and software evolution [6] (among others).

SBSE practitioners apply a variety of techniques, among them: genetic and evolutionary algorithms [5,27], hill-climbers [32, 34, 36], tabu search [20, 21], particle swarm optimization [12, 31], and ant colony optimization [11]. Yet, there exists no agreed-upon baseline technique. A number of these approaches are common - simulated annealing and genetic algorithms are heavily favored - and a "random search" is often used as a sanity measure, but there is no standard method to use as a basis for comparison when new methods are introduced.

In 1993, Robert Holte introduced the 1R classification algorithm [28]. In comparison to many of the prevalent techniques used at the time, such as the entropy-based C4.5 classifier, 1R was incredibly simple. The learning method treated each attribute of a data set as a continuous range (rather than as discrete intervals) and ranked them according to the error rate. Although Holte did not set out with the intention of defining a baseline for the data mining research field, there are several factors that perfectly positioned 1R

for such a title:

- *Simplicity*: 1R is easy to understand, using error rate as a judgment heuristic.
- *Competitive Results*: 1R produces results that are, on average, within 5% of C4.5's on the same data sets [28].
- *Fast Runtimes*: 1R is faster than more complex techniques.
- *Stable Results*: The results produced by 1R are consistent for the same data set over multiple trials.

A baseline technique for any field *must* have some level of simplicity. The algorithm must be easy to comprehend and easy to implement. A researcher's time and effort must go into the technique meant to surpass the baseline, not the baseline itself. However, it is not enough to be simple - "dumb" ideas will yield poor results. For a algorithm to stand as a valid baseline, it must meet all four of these factors; the heuristic must be easy-to-understand, fast, stable, *and* produce results that are competitive with state-of-the-art techniques.

The PROMISE conference series (and, ultimately, the entire academic world) seeks "repeatable, improvable, maybe even refutable, software engineering experiments."[1] It is not enough that some techniques, such as simulated annealing or random testing, reach near-ubiquity. The existence of a valid baseline gives a common basis of comparison, facilitating the repeatability and improvement of a wide variety of software engineering experiments. Just as the data mining field has benefited from the use of 1R as a baseline [42], the SBSE research field would benefit as well.

In this research, I propose the KEYS2 algorithm [19] as a candidate baseline technique for SBSE problems. KEYS2 is based on a straightforward theory - if the behavior of a model is controlled by a small number of *key* variables, then ideal solutions can be quickly found by isolating those variables and exploring the range of their values. In a case study centered around the optimization of early life-cycle requirements models, KEYS2 is demonstrated to fit all four of the factors required from a baseline technique and to have certain advantages over other potential baseline methods, such as random search:

- KEYS2 is based on a simple theory; that is, the exploitation of a small set of important variables.
- KEYS2 produces results that are of higher quality than a simulated annealer and are competitive with a genetic algorithm (as assessed by a Mann-Whitney rank-sum test with a 95% confidence interval).
- KEYS2 is faster than standard SBSE techniques, executing almost four times faster than the genetic algorithm on the most complex models.
- KEYS2 produces stable results, and has the ability to produce partial solutions.

This work builds on previous optimization research with the KEYS algorithm.

- The original KEYS algorithm is presented in [30].
- KEYS2 is presented and both it and KEYS2 are benchmarked against a variety of search techniques, including Simulated Annealing, in [19].
- The contributions of this paper include a new genetic algorithm designed to optimize DDP models, new benchmarking experiments, and an explicit focus on a SBSE research context.

[1]Quoted from `http://promisedata.org/?page_id=2`

2. SEARCH-BASED SOFTWARE ENG.

Many of the problems inherent to the software engineering field deal in the *balancing* of competing factors. Consider these scenarios:

- Do you want to finish a project with money left in the budget, or is it more important to deliver a robust feature set?
- If the programmers are not meeting deadlines, you could replace them with a new team. Would the additional man-hours required for training and catch-up be worth the possible benefits?
- If a longer design cycle would result in a shorter programming period, will too many defects slip by unnoticed?

In many of these cases, there is no *perfect* solution. In fact, there could be dozens of "good" solutions. Instead of finding some theoretical catch-all answer, software engineering problems are typically concerned with near-optimal solutions. That is, the subset of 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. This comparison forms the basis of search-based software engineering.

Search-based software engineering is a research field concerned with the reformulation of standard software engineering problems as search problems, and the application of heuristic techniques to solve said problems. Because of this implied trade-off space, meta-heuristic search-based methods are ideal for producing a set of near-optimal candidate solutions.

According to Clark and Harman [22, 25], there are four key properties that must be met before attempting to apply a SBSE solution:

- *A Large Search Space*: If there are only a small number of options to optimize, there is no need for a meta-heuristic approach. This is rarely the case, as software engineering typically deals in incredibly large search spaces (for instance, the Eclipse project contains over two million lines of code and 7500 classes).
- *Low Computational Complexity*: If the first property is met, SBSE algorithms must sample a non-trivial population. Therefore, the computational complexity of any evaluation method or fitness function has a major impact on the execution time of the search algorithm.
- *Approximate Continuity*: Any search-based optimization must rely on an objective function for guidance. 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 is 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 as a search-based one [3, 24].

2.1 Simulated Annealing

Simulated annealing (SA) [32, 36] is a sophisticated hill-climbing search algorithm. Hill-climbing searches survey the immediate neighborhood and "climb" to higher-scoring states, continuing until they reach a peak (that is, an optimal area in the search space). Hill-climbers are prone to becoming stuck in local maxima, returning a sub-optimal solution. To avoid this, simulated annealing utilizes a technique from its namesake. Annealing is a metallurgy technique 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. When assessing each neighbor, SA will jump to a sub-optimal neighbor at a probability determined by the current *temperature*. When that temperature is high, SA will jump around the search space wildly. As it cools, the algorithm will stabilize into what is, ideally, an optimal area of the search space.

Simulated annealing and hill-climbers are widely used in SBSE research for several reasons. They are simple to understand, easy to implement, and are typically very fast [34]. However, they have several known flaws. While the temperature function helps to prevent the search from becoming stuck in local maxima, it does not completely mitigate this problem. In fact, past research has shown that the additional randomness in this search often leads to an unacceptable level of variance in the results [18, 19]. Thus, one must take caution when choosing a cooling strategy - cool too quickly, and it is likely that the annealer will return a sub-optimal solution.

This research uses a version of simulated annealing tuned specifically for the DDP models (discussed in Section 3). During each round, SA "picks" a neighboring set of mitigations. To calculate the configuration of this neighboring solution, a function traverses the mitigation settings of the current state and randomly flips those settings (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 following probability function:

$$prob(moving) = e^{((score - neighborscore) * \frac{time}{temp})} \quad (1)$$

$$temp = \frac{(maxtime - time)}{maxtime} \quad (2)$$

If the value of the `prob` function is greater than some randomly generated number, SA will move to that state regardless of its score. The algorithm will continue to operate until the number of tries is exhausted (i.e. the temperature drops to zero) or a score meets the threshold requirement.

2.2 Genetic Algorithms

Inspired by early experiments with the computational simulation of evolution [5,27], 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. Influenced by Darwin's Theory of Evolution, genetic algorithms 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), a population of solutions is considered. Each member of the population is made up of a set of binary switches. These switches are collectively known as the chromosomes, the genetic makeup, of a particular solution state. Typically, a potential solution must be carefully translated to this binary representation. However, as the models in this research are naturally represented as a set of binary variables, no special encoding of the model states is necessary. This has an additional benefit of reducing the required setup and execution time. The GA can make use of the same fitness function that the simulated annealer and KEYS2 use.

Because the initial population is unlikely to have the "best" solution, some form of diversity must be induced into the population. This diversity is established by forming children using the crossover and mutation operations. During each generation, several "good" solutions (as scored by a predetermined fitness function) are chosen by the selection mechanism to generate new children for the next generation. The values of the binary settings in these children are determined by the crossover mechanism, which combines chromosome settings from each parent and inserts them into the offspring with probability $P_{crossover}$. A mutation mechanism takes high-quality members of the current population and moves them over to the next generation while instilling small changes in their chromosomes. Mutation 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 process is repeated with a new population each round until a certain threshold (commonly related to the 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 mechanisms.
- Discard the old population and repeat the process.
- Stop if $time > maxtime$ (in number of generations or some real-time threshold).

The genetic algorithm used in this research creates a population of size 100 each round, and selects all population members that score within 10% of the top-scoring member (as determined by the objective function defined in Section 3) to "reproduce." The chosen *parents* populate 40% of the next population (15% through crossover, 20% through mutation, and 5% are carried over unchanged). The remaining 60% of the population are randomly generated. These percentage values were chosen at the overall best values after an exhaustive search across five models, incrementing each factor by 5% each round. Each setting can be overridden by the user. This genetic algorithm stops after the number of past generations is equal to the number of variables in the model being optimized.

2.3 KEYS2

The core premise of KEYS2 is that standard SBSE techniques perform over-elaborate searches. Suppose that the

behavior of a large system is determined by a small number of *key* variables. If so, then stable near-optimal solutions can be quickly found by finding these *keys* and then exploring the range of their values.

Within a model, there are chains of *reasons* linking select inputs to the desired goals. Some of the links clash with others. However, some of those clashes 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 independent clashes are $\{e \neg e\}$, & $\{g \neg g\}$,

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

In order to optimize decision making about this model, we must first decide about these independent clashes (these are 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 irrelevant 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.

Fixing the value of these keys reduces the number of reachable states within the model. 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 example model has 2^{20} possible states. However, in the context of $input_i \vdash goal$, that massive collection of states clumps to the following two configurations: $\{input_1, f, g, h, i, j, goal\}$ or $\{input_2, k, \neg g, l, m, \neg j, goal\}$.

The KEYS algorithm finds these key variables using a greedy search and a Bayesian ranking heuristic (called BORE). 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 objective function, then the key variables are those with ranges that occur with very different frequencies in the high and the low scoring model configurations. Therefore, we need not waste time searching for the keys - rather, we just need to keep frequency counts on how often ranges appear in *best* or *rest* outputs.

The greedy search explores a space of M mitigations over the course of N 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 [30], the greedy search fixed the value of one variable per era. KEYS2, fixes an increasing number of variables as the search progresses. That is, KEYS2 fixes one variable in the first era, two in the second era, three in the third era, and so on

In 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).

4: The top ranked mitigations are fixed and stored in $selected[e]$.

```
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 1: Pseudocode for KEYS

The search moves to era $e + 1$ and repeats steps 1,2,3,4. This process stops when every mitigation has a fixed value. The exact settings for $MaxTries$ and β must be set via engineering judgment. Past research has shown that, for DDP model optimization, these should be set to $MaxTries = 100$ and $\beta = 10$ [30]. For full details, see Figure 1.

KEYS ranks mitigations using a support-based Bayesian ranking measure called BORE. BORE [8] (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 for space consideration) are then normalized to calculate 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:

$$\begin{aligned} E &= (mitigation31 = false) \\ P(best) &= 1000/10000 = 0.1 \\ P(rest) &= 9000/10000 = 0.9 \\ freq(E|best) &= 10/1000 = 0.01 \\ freq(E|rest) &= 5/9000 = 0.00056 \\ like(best|E) &= freq(E|best) \cdot P(best) = 0.001 \\ like(rest|E) &= freq(E|rest) \cdot P(rest) = 0.000504 \\ P(best|E) &= \frac{like(best|E)}{like(best|E) + like(rest|E)} = 0.66 \end{aligned} \quad (3)$$

Previously [8], 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, Equation 3 is augmented 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

$$P(best|E) * support(best|E) = \frac{like(best|E)^2}{like(best|E) + like(rest|E)} \quad (4)$$

3. CASE STUDY: THE DEFECT DETECTION AND PREVENTION MODEL

The Defect Detection and Prevention (DDP) requirements modeling tool [9, 14]. is used to interactively document the early life-cycle meetings conducted by "Team X" at NASA's Jet Propulsion Laboratory (JPL).

At Team X meetings, groups of up to 30 experts from various fields (propulsion, engineering, communication, navigation, science, etc) meet for short periods of time to produce a design document. This document may commit the current project to certain design choices. For example, the experts might choose solar power rather than nuclear power, or they might decide to use some particular style of guidance software. All subsequent work on the project is guided by the initial design decisions made in these mission concept documents.

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 the combinations of mitigations that will cost the least amount of money while still allowing for the completion of a large 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:

$$\overbrace{mitigation1}^{\$10{,}000} \underbrace{\rightarrow}_{0.9} risk1 \rightarrow \left\langle \begin{array}{l} \overbrace{\rightarrow}^{0.1}(requirement1 = 100) \\ \underbrace{\rightarrow}_{0.99}(requirement2 = 100) \end{array} \right. \quad (5)$$

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*.

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;

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 and design tools
   - ISHM Experiment is a failure (without necessarily causing flight failure)
   - Usability, User/Recipient-system interfaces undefined
   - V&V (certification path) untried and the scope is 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 ISHM expert's experience and guidance with certification of his technology

Figure 2: Sample DDP requirements, risks, and mitigations.

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 3: Details of Five DDP Models.

To improve runtimes, a compiler stores a flattened form of the DDP requirements tree in a function usable by any program written in the C language. In the compiled form, all computations are performed once and added as a constant to each reference of the requirement. The topology of the mitigation network is represented as terms in equations within the `model` function, which is called each time that a configuration of the entire model needs to be scored by a fitness function. As the models grow more complex, so do these equations. The largest real-world model used in this research, which contains 99 mitigations, generates 1427 lines of code. Figure 3 contains details on five publicly-available DDP models.

When the `model` function is called, two values are returned (the total cost of the selected mitigations and the number of reachable requirements attained). These two values are input into the following objective function in order to calculate a fitness score for the current configuration of the model. This objective function normalizes the cost and attainment values into a single score that represents the Euclidean distance to a *sweet spot* of maximum requirement attainment and minimum cost:

$$score = \sqrt{\overline{cost}^2 + (\overline{attainment} - 1)^2} \quad (6)$$

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

DDP is a valid choice for early life-cycle requirements optimization for three reasons:

- One potential drawback with ultra-lightweight models is that they are excessively lightweight and contain no useful information. DDP models are demonstrably useful for NASA, and clear project improvements (such as savings in power and mass) 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 [14].
- Numerous real-world requirements models have been written in this format, and many projects are likely to use these models in the future [16]. The DDP tool can be used to document not just final decisions, but also to review the rationale that led to those decisions. 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.
- The DDP tool is representative of other requirements modeling tools in widespread use. DDP is a set of directed influences expressed in a rigid hierarchy and controlled by a set of equations. At this level of abstraction, DDP is just another form of QOC [41] or a variant of Mylopoulos' soft goal graphs [38].

Like any interesting search-based software engineering problem, DDP models represent a space of competing factors. In this case, the trade-off is between the budget of a project and the attainment of goals. Using the four properties of SBSE problems (as discussed in Section 2), it can be shown that the optimization of DDP models is a valid application area for SBSE techniques.

- Typical DDP models present a massive search space, encompassing hundreds to thousands of possible combinations of mitigation settings. One collection of model settings must comment on dozens of individual mitigations. In one known model, there are over 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 (to meet the second condition of a SBSE problems). As will be shown in Section 4.2, standard SBSE techniques execute in one second or less, implying a low computational complexity in the model assessment method. Furthermore, past research [19] has demonstrated that the proposed baseline method, KEYS2, operates in low-order polynomial time ($O(N^2)$).
- While DDP models do not represent a continuous search space (any model with *true/false* statements is, by definition, not continuous), this is not a problem. The objective function used to score a DDP configuration represents an approximately continuous trade-off between project costs and feature attainment. Therefore, it provides the necessary guidance needed to meet the third condition.
- 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 in order to calculate some set of *near-optimal solutions*.

By meeting all four of these conditions, the optimization of these early life-cycle DDP models is a valid avenue for search-base software engineering research.

4. EXPERIMENTS & RESULTS

While the "simplicity" of the KEYS2 algorithm is arguably subjective, the other three requirements of a baseline algorithm can be assessed qualitatively. In order to demonstrate that KEYS2 is a valid candidate for a baseline technique, it must be experimentally shown to be:

- Results that are competitive with those of state-of-the-art techniques.
- As fast as competing algorithms.
- A low level of variance in its results.

In the following experiments, three algorithms (KEYS2, a simulated annealer, and a genetic algorithm) will be used to optimize three of the five publicly-available DDP models[2]. Note that:

- Models one and three are trivially small. They were used to debug source code, but have been excluded from the following experiments.
- Model 4 was previously discussed in detail [35]. Model 5 was optimized in [15].
- All five models were optimized by the original KEYS algorithm in [30]. However, that paper presented no comparison results.
- KEYS2 was benchmarked against a variety of algorithms, including simulated annealing, on models 2,4, and 5 in [19]. However, no results were reused. All results presented in this paper are from new trials.

4.1 Result Quality

By definition, the purpose of a baseline is to provide a *starting goal*. It should set some kind of bar that a newly-developed technique must pass before it can be considered for real-world use. However, while the goal of any experiment will be to "beat" the baseline score, that does not mean that a baseline should be some sort of "straw man." A baseline technique might not yield results that surpass the state-of-the-art, but it should at least be *competitive* with them.

In order to assess the overall result quality, each algorithm was executed 1000 times per model. These results are plotted in Figure 4. In each graph, the x-axis represents the "attainment" (the number of project objectives that were successfully completed in the final configuration of the model).

[2]These models are available from the PROMISE repository at `http://promisedata.org/?cat=133`

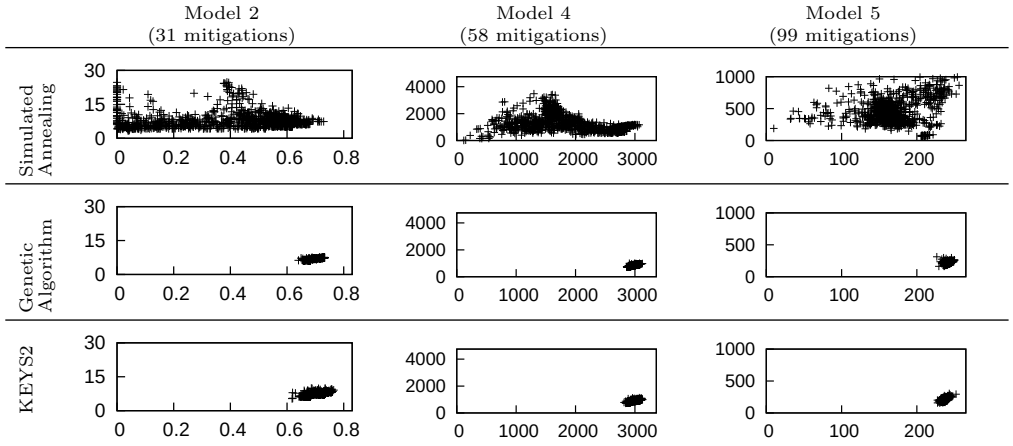

Figure 4: Results of each algorithm for each model (1000 trials per algorithm/model pairing). The y-axis shows cost (in thousands) and the x-axis shows attainment. Better solutions fall towards the bottom right of each plot (low costs and high attainment) and are clumped tightly together (low variance).

The y-axis is the cost, in thousands, of the mitigations employed by that model configuration. Better results are those that appear in the bottom-right corner. These are the configurations resulting in a low cost and high attainment of goals. It follows that the worst results are in the top-left, the area of high cost and low attainment.

A visual inspection gives a very clear idea of the result quality. The simulated annealing method can instantly be eliminated from the comparison. Its results, for every model, fall all over the graph. Very few of its conclusions result in an ideal balance of the cost and attainment. Even if some of its results are ideal, the massive level of variance eliminates it from the competition. Both KEYS2 and the genetic algorithm fare better, with the vast majority of their results concentrated in the ideal low cost, high attainment area.

To confirm this visual analysis, the distribution of values for each algorithm and each model were compared using a non-parametric rank-sum test (Mann-Whitney) with a 95% confidence interval. Separate tests were completed for cost and attainment for each of the three models. The wins, ties, and losses were aggregated and are listed in Figure 5.

Algorithm	Ties	Wins	Losses	Score (wins-losses)
Genetic Algorithm	0	7	5	2
KEYS2	0	7	5	2
Simulated Annealing	0	4	8	-4

Figure 5: Win-loss-tie results from a Mann-Whitney rank-sum test with a 95% confidence interval.

As expected, the wide range of results for simulated annealing hurt its overall result quality. The algorithm lost twice as often as it won in comparison tests. KEYS2 and the genetic algorithm yielded the exact same number of wins and losses. While they never tied, the fact that they traded wins and losses at such equal parity shows that KEYS2 is extremely competitive with the most commonly used SBSE techniques.

4.2 Runtimes

For both researchers and industrial practitioners, time is limited and expensive. By asking any of these individuals to run a baseline algorithm for comparison purposes, we are implicitly asking them to spend more time considering a problem. Therefore, it is crucial that any baseline technique be fast, even real-time if possible.

In order to compare execution times, each algorithm was executed for each model 100 times. The total time for these 100 trials was recorded using the Unix `time` command, and the "real time" value was divided by 100 in order to calculate the average execution time.

All three algorithms were executed under the same operating conditions on the same machine - a Dell XPS workstation with a 2.40 GHz Intel Core2 Duo CPU and 2 GB of memory running the Ubuntu 8.04 operating system.

Algorithm	Model 2	Model 4	Model 5
simulated annealing	0.40967	0.94350	0.64050
genetic algorithm	0.00976	0.04625	0.09948
KEYS2	0.00430	0.01407	0.02695

Figure 6: Average runtimes for each algorithm on each model, in seconds, averaged over 100 runs.

The results of this experiment can be seen in Figure 6. The simulated annealer clearly experiences some difficulty in optimizing these models. It takes orders of magnitude more time to complete a single run than other algorithms.

The remaining two algorithms are incredibly fast, both returning results in under a tenth of a second on average. A closer look shows that KEYS2 is still much faster than the genetic algorithm, completing each trial between 2.27 to 3.69 times faster than its competitor.

4.3 Stability

The results shown in Section 4.1 give some idea of the variance in the performance of each of the compared techniques. From a visual inspection of Figure 4, it is clear that the annealer's final values are unpredictable. Very few of

its results fall in the ideal bottom-right corner (where the project leads pay a little to achieve a lot). The final score values of both the genetic algorithm and KEYS2 fall in a much smaller cluster. To gain a clearer view of the result variance of each algorithm, we will examine the range of values output by each algorithm for the largest model (model 5 from Section 3).

	quartiles				
	min		med		max
	0	25%	50%	75%	100%
SA	163000	239025	248525	709025	1079000
GA	162369	205525	215197	227525	312052
KEYS2	154025	198025	211525	224525	305525

Figure 7: Quartile charts of the final "cost" results, taken from 1000 executions of each algorithm on model 5. Lines represent the full range of values, with the dot representing the median.

	quartiles				
	min		med		max
	0	25%	50%	75%	100%
SA	46.3	201.1	207.4	217.0	255.2
GA	226.4	239.5	241.3	242.4	249.9
KEYS2	227.3	236.2	237.9	239.4	252.0

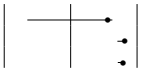

Figure 8: Quartile charts of the final "attainment" results, taken from 1000 executions of each algorithm on model 5. Lines represent the full range of values, with the dot representing the median.

Figures 7 and 8 show quartile breakdowns of the value distributions for the cost and attainment results on model 5. Simulated annealing reaches median values that are weaker than, but not too far from, the other two algorithms. However, the full range of its results is much wider than that of KEYS2 or the genetic algorithm. Simulated annealing demonstrates an unacceptable level of variance. KEYS2 and the GA yield similar results. The GA has a slightly tighter range of results with a spread (75th quartile - 25th) of 22000 to 26500 for cost and 2.9 to 3.2 on attainment.

From this, it can be concluded that KEYS2 demonstrates the consistency required of a baseline method. However, its very design provides one additional degree of stability. Recall that each round, KEYS2 seeks out the most important variable and fixes its value. Thus, by providing a linear ranking of model variables, KEYS2 could be stopped at any point during its execution. Unlike the other two algorithms, KEYS2 can provide partial results. Because it generates 100 random configurations at each stage of execution, the variance of those partial decisions can be measured.

Figure 9 demonstrates the internal stability during one execution of KEYS2 on model 5. At any stage, the spread (i.e. the variance) of KEYS internal decisions can be measured. It can clearly be seen that KEYS2 rapidly plateaus towards stable, high-scoring results for both the cost and attainment of goals. The ability to generate partial *or* complete solutions, both with a low degree of variance, lend credence to the use of KEYS2 as an experimental baseline.

5. DISCUSSION & CONCLUSIONS

Despite the extensive body of research that has been invested in the search-based software engineering field over

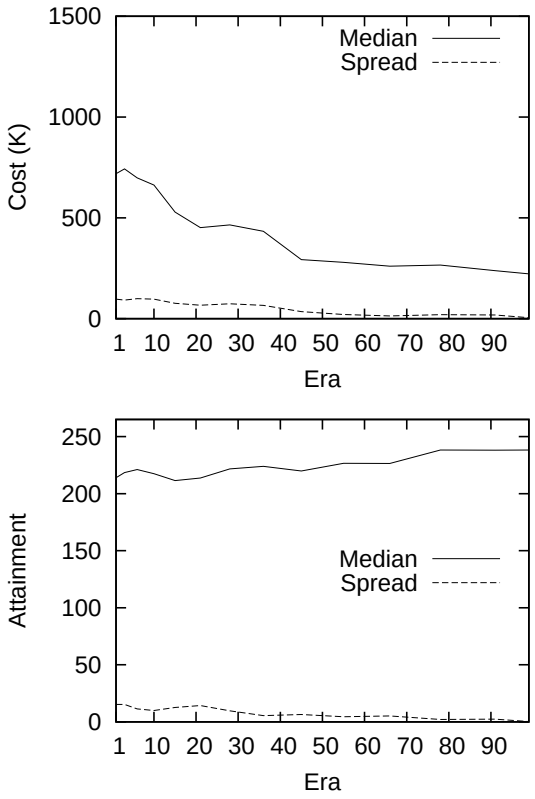

Figure 9: Median and spread of internal decisions made by KEYS2 during one trial. X-axis shows the number of rounds passed. "Median" shows the 50th percentile of the measured value seen in the 100 random configurations made during each round. "Spread" shows the difference between the 75th and 50th percentile.

the past few years, there is still no agreed-upon basis for comparison between the disparate research tracts. With all of the different improvements to and implementations of the *"standard"* techniques, it has become difficult to achieve the PROMISE community's goal of "repeatable, improvable, maybe even refutable, software engineering experiments."

For the further expansion of the body of knowledge, it is necessary for the SBSE field to adopt some sort of baseline technique. Just as data mining research has benefited from Holte's 1R algorithm [28], the SBSE community would benefit from a common baseline. Such a technique would allow comparisons between seemingly incomparable algorithms, would more easily allow for the repetition and improvement of existing experiments, and would give a performance goal to be beaten by newly-developed methods.

However, not just any algorithm can be chosen as a baseline. A baseline technique must meet several criteria:

- *Simplicity*: The method must be easily understood and easily implemented. If not, researchers will waste crucial time and brainpower that would be better spent on their own ideas.
- *Competitive Results*: Although a baseline is meant to be "beaten," is should not be a straw man. It must

yield results within the same neighborhood as the state-of-the-art.
- *Fast Runtimes*: By asking a researcher to execute a baseline algorithm, you are implicitly asking them to devote more time to their experiments. Thus, an ideal baseline technique should execute in an almost unnoticeable amount of time.
- *Stable Results*: A baseline technique must be reliable and trustworthy. It must return results that fall within a small range of fitness values.

It is not enough for a technique to meet one or two of these criteria. A fast algorithm is pointless if its results are seemingly random. Likewise, a simple method is not useful if its results are poor. A baseline algorithm should be fast, reliable, competitive, and be straight-forward to implement.

Out of the commonly-used approaches in the SBSE field, random search is likely the closest to an agreed-upon baseline. Random search is used by may researchers as a "sanity check," a de facto bar to beat with their algorithm of choice [26]. However, random search is a weak approach. Its very nature implies that it will eventually violate either the second or third of the previously-specified properties of a baseline. Although it *may* find an ideal solution in a short amount of time, there is no way to ensure that it *always* will. Because it picks inputs at random, there will be a massive amount of variance in its final results (for a similar effect, see the simulated annealing results in Sections 4.1 and 4.3). There is some empirical evidence that, given enough time, random testing will deliver predictable results [29]. However, such a time threshold will violate the second guideline, that any baseline execute quickly. Many researchers seem to have little confidence in random search, treating it as a straw man. As Harman notes, "It is something of a 'sanity check'; in any optimization problem worthy of study, the chosen technique should be able to convincingly outperform random search. [26]" The use of random search is an ongoing debate in the SBSE community [1]. I would suggest, despite the popularity of random search, a stronger baseline algorithm should be chosen.

The KEYS2 algorithm [19, 30] meets all of these criteria. It is based on a straightforward theory, that a small number of important variables control the vast majority of the search space. A search that rapidly isolates these variables and fixes their values will quickly plateau towards stable, optimal solutions. In a case study centered around the DDP early lifecycle requirements models [9,14], KEYS2 was demonstrated to fulfill the other three requirements.

- In Section 4.1, it is shown that KEYS2 and a genetic algorithm are statistically identical in terms of quality. They yielded the same number of wins and losses in a series of rank-sum tests.
- Section 4.2 shows that KEYS2 executes three or more times faster than other SBSE techniques.
- Finally, Section 4.3 demonstrates that KEYS2 yields stable results. KEYS2 can also output partial results, and the variance and spread of these can be examined as well.

Thus, KEYS2 is a candidate to serve as a baseline technique for the SBSE field.

I am not presumptuous enough to insist that KEYS2 be the absolute baseline or even that it is the best candidate. However, it is crucially important to debate this topic. Even amongst "standard" techniques like simulated annealing, there are thousands of slightly different implementations and tweaks. Direct comparisons are almost impossible. For the goal of open research to become possible, a simple (yet competitive) technique should be adopted and made available to the research community. Such a baseline is necessary for the growth of this research field, and the PROMISE community is in an ideal position to promote its use.

6. REFERENCES

[1] A. Arcuri, M. Iqbal, and L. Briand. Formal analysis of the effectiveness and predictability of random testing. In *ISSTA '10: Proceedings of the 19th international symposium on Software testing and analysis*, pages 219–230, New York, NY, USA, 2010. ACM.

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

[3] A. Baresel, D. Wendell Binkley, M. Harman, and B. 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.

[4] A. Baresel, H. Sthamer, and M. 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.

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

[6] T. Van Belle and D. 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.

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

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

[9] 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.

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

[11] 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.

[12] 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.

[13] D. Fatiregun, M. Harman, and R. 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.

[14] 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`.

[15] 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`.

[16] 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.

[17] M. Claudia Figueiredo, P. Emer, and S. 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.

[18] G. Gay, T. Menzies, M. Davies, and K. Gundy-Burlet. Automatically finding the control variables for complex system behavior. *Accepted for Automated Software Engg.*, 17(4), 2010.

[19] G. Gay, T. Menzies, O. Jalali, G. Mundy, B. Gilkerson, M. Feather, and J. Kiper. Finding robust solutions in requirements models. *Automated Software Engg.*, 17(1):87–116, 2010.

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

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

[22] M Harman and J. 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,.

[23] 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.

[24] M. Harman, L. Hu, R. Mark Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, 2004.

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

[26] M. Harman and P. McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Trans. Softw. Eng.*, 36(2):226–247, 2010.

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

[28] R.C. Holte. Very simple classification rules perform well on most commonly used datasets. *Machine Learning*, 11:63, 1993.

[29] Ciupa I., A. Pretschner, M. Oriol, A. Leitner, and B. Meyer. On the number and nature of faults found by random testing. *Software Testing, Verification and Reliability*, 2009.

[30] 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`.

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

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

[33] C. Kirsopp, M. Shepperd, and J. 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.

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

[35] 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`.

[36] 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.

[37] B. S. Mitchell and S. 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.

[38] 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.

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

[40] S.L. Rhys, S. Poulding, and J.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.

[41] 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.

[42] I. H. Witten and E. Frank. *Data mining. 2nd edition.* Morgan Kaufmann, Los Altos, US, 2005.

[43] 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.

APPENDIX

Acquiring the Algorithms

In the spirit of the PROMISE conference series, all of the algorithm implementations and data used in this research are available for public use.

All five DDP models are available from the PROMISE repository at `http://www.promisedata.org/?cat=133`.

KEYS2 can be downloaded from `http://www.unbox.org/wisp/var/greg/keys/2.6`. The genetic algorithm can be found at `http://www.unbox.org/wisp/var/greg/genefreeze` and the simulated annealing is located at `http://www.unbox.org/wisp/tags/ddpExperiment`.