# 1 Introduction

Search Based Software Engineering (SBSE) is the name given to a body of work in which Search Based Optimisation is applied to Software Engineering. This approach to Software Engineering has proved to be very successful and generic.

SBSE seeks to reformulate Software Engineering problems as ‘search problems. Search Based Software Engineering, a search problem is one in which optimal or near optimal solutions are sought in a search space of candidate solutions, guided by a fitness function that distinguishes between better and worse solutions.

SBSE has been applied to many fields within the general area of Software Engineering, some of which are already sufficiently mature to warrant their own surveys.

Rather, this paper aims to provide those unfamiliar with SBSE with a tutorial and practical guide.

# Why SBSE?

All of these questions and many more like them, can (and have been) addressed by work on SBSE . In this section we briefly review some of the motivations for SBSE to give a feeling for why it is that this approach to Software Engineering has generated so much interest and activity.

## Generality

SBSE is very widely applicable. we can make progress with an instance of SBSE with only two definitions: a *representation* of the problem and a *fitness function* that captures the objective or objectives to be optimisedRobustness**.**

SBSE’s optimisation algorithms are robust. Often the solutions required need only to lie within some specified tolerance. That is, while it is true that a great deal of progress and improvement can be made through tuning, one may well find that all reasonable parameter choices comfortably outperform a purely random search. Therefore, if one is the first to use a search based approach, almost any reasonable (non-extreme) choice of parameters may well support progress from the current ‘state of the art’.

## Scalability Through Parallelism.

Search based optimisation techniques are often referred to as being ‘embarrassingly parallel’ because of their potential for scalability through parallel execution of fitness computations. Several SBSE authors have demonstrated that this parallelism can be exploited in SBSE work to obtain scalability through distributed computation

1. **Re-unification.**

SBSE can also create linkages and relationships between areas in Software Engineering that would otherwise appear to be completely unrelated. For instance, the problems of Requirements Engineering and Regression Testing would appear to be entirely unrelated topics; they have their own conferences and journals and researchers in one field seldom exchange ideas with those from the other.

However, using SBSE, a clear relationship can be seen between these two problem domains. That is, as *optimisation problems* they are remarkably similar: Both involve selection and prioritisation problems that share a similar structure as search problems.

# Defining a Representation and Fitness function

SBSE starts with only two key ingredients:

* 1. The choice of the representation of the problem.
  2. The definition of the fitness function.

# Commonly used algorithms

1. **Random search:** it is the simplest form of search algorithm that appears frequently in the software engineering literature. However, it does not utilise a fitness function, and is thus unguided, often failing to find globally optimal solutions. Higher quality solutions may be found with the aid of a fitness function, which supplies heuristic information regarding the areas of the search space which may yield better solutions and those which seem to be unfruitful to explore further.
2. **Hill Climbing:** The simplest form of search algorithm using fitness information in the form of a fitness function is Hill Climbing. Hill Climbing selects a point from the search space at random. It then examines candidate solutions that are in the‘neighbourhood’of the original; i.e. solutions in the search space that are similar but differ in some aspect, or are close or some ordinal scale. If a neighbouring candidate solution is found of improved fitness, the search ‘moves’ to that new solution. It then explores the neighbourhood of that new candidate solution for better solutions, and so on, until the neighbourhood of the current candidate solution offers no further improvement. Such a solution is said to be *locally optimal*, and may not represent globally optimal solutions ,and so the search is often restarted in order to find even better solutions. Hill Climbing may be restarted as many times as computing resources allow.

As can be seen, not only must the fitness function and the ‘neighbourhood’ be defined, but also the type of ‘ascent strategy’. Types of ascent strategy include ‘steepest ascent’, where all neighbours are evaluated, with the ascending move made to the neighbour offering the greatest improvement in fitness. A ‘random’ or ‘first’ ascent strategy, on the other hand, involves the evaluation of neighbouring candidate solutions at random, and the first neighbour to offer an improvement selected for the move.

Select a starting solution *s S*

Repeat

∈

Select *s*j ∈ *N* (*s*) such that *fit*(*s*j) *> fit*(*s*) according to ascent strategy

*s* ← *s*j

Until *fit*(*s*) ≥ *fit*(*s*j)*,* ∀*s*j ∈ *N* (*s*)

High level description of a hill climbing algorithm, for a problem with solution space *S*; neighbour- hood structure *N* ; and *fit*, the fitness function to be maximised (adapted from McMinn [65])

1. **Simulated Annealing**: it is similar to Hill Climbing in that it too attempts to improve one solution. However, Simulated Annealing attempts to escape local optima without the need to continually restart the search. It does this by tem- porarily accepting candidate solutions of poorer fitness, depending on the value of a variable known as the *temperature*. Initially the temperature is high, and free movement is allowed through the search space, with poorer neighbouring solutions representing potential moves along with better neighbouring solutions. As the search progresses, however, the temperature reduces, making moves to poorer solutions more and more unlikely. Eventually, *freezing point* is reached, and from this point on the search behaves identically to Hill Climbing. The probability of acceptance *p* of an inferior solution is calculated as *p* = *e*− *t* , where *δ* is the difference in fitness value between the current solution and the neighbouring inferior solution being considered, and *t* is the current value of the temperature control parameter.

*δ*

Select a starting solution *s S*

Select an initial temperature *t >* 0 Repeat

←

∈

*It* 0

Repeat

Select *s*j *N* (*s*) at random

← −

∈

*∆e fit*(*s*) *fit*(*s*j)

If *∆e <* 0

*s s*j

←

Else

Generate random number *r*, 0 ≤ *r <* 1 If *r < e*− *t* Then *s s*j

*δ*

←

End If

*it it* + 1

←

Until *it* = *num solns*

Decrease *t* according to cooling schedule Until Stopping Condition Reached

**Fig. 6.** High level description of a simulated annealing algorithm, for a problem with solution space *S*; neighbourhood structure *N* ; *num solns*, the number of solutions to consider at each temperature level *t*; and *fit*, the fitness function to be maximised (adapted from McMinn [65])

1. **Genetic Algorithms:** it issaid to be *global searches*, sampling many points in the search space at once offering more robustness to local optima. The set of candidate solutions currently under consideration is referred to as the current *population*, with each successive population considered referred to as a *generation*. Genetic Algorithms are inspired by Darwinian Evolution, in keeping with this analogy, each candidate solution is represented as a vector of components referred to as *individuals* or *chromosomes*. Typically, a Genetic Algorithm uses a binary representation, i.e. candidate solutions are encoded as strings of 1s and 0s; yet more natural representations to the problem may also be used, for example a list of floating point values.

The first generation is made up of randomly selected chromosomes, although the population may also be ‘seeded’ with selected individuals representing some domain information about the problem, which may increase the chances of the search converging on a set of highly-fit candidate solutions. Each individual in the population is then evaluated for fitness.

On the basis of fitness evaluation, certain individuals are selected to go forward to the following stages of crossover, mutation and reinsertion into the next generation. Usually selection is biased towards the fitter individuals, however the possibility of selecting weak solutions is not removed so that the search does not converge early on a set of locally optimal solutions.

In the crossover stage, elements of each individual are recombined to form two offspring individuals. Different choices of crossover operator are available, including ‘one-point’ crossover, which splices two parents at a randomly-chosen position in the string to form two offspring. Other operators may recombine using multiple crossover points, while ‘uniform’ crossover treats every position as a potential crossover point.

Subsequently, elements of the newly-created chromosomes are mutated at random, with the aim of diversifying the search into new areas of the search space. For GAs operating on binary representation, mutation usually involves randomly flipping bits of the chromosome. Finally, the next generation of the population is chosen in the ‘reinsertion’ phase, and the new individuals are evaluated for fitness. The GA continues in this loop until it finds a solution known to be globally optimal, or the resources allocated to it (typically a time limit or a certain budget of fitness evaluations) are exhausted.

Randomly generate or seed initial population *P*

Repeat

Evaluate fitness of each individual in *P*

Select parents from *P* according to selection mechanism Recombine parents to form new offspring

Construct new population *P* j from parents and offspring Mutate *P* j

*P P* j

←

Until Stopping Condition Reached

**Fig. 8.** High level description of a Genetic Algorithm