A Genetic Algorithm for the Generation of Software Maintenance Release Plans without Human Bias

Tommy E. Bennett
University of Maryland University College

Michael Scott Brown
University of Maryland University College

Michael Pelosi
University of Maryland University College

Keywords: Release Plan, Software Maintenance, Genetic Algorithm, Maintenance Release

Abstract

Deciding what defect fixes and enhancements go into a maintenance release can be a difficult and complex task. This paper will discuss a technique that applies the use of a genetic algorithm (GA) during the software maintenance phase of the system development life cycle. The application of GA use in optimizing maintenance release packages will be designed to find a solution that returns the best value in terms of cost, time, and critical requests being addressed. This paper describes an algorithm that allows the input of weighted values; category of the maintenance request, severity of the maintenance request, time to develop the maintenance request, and the promised delivery date of maintenance request. More specifically, the GA will be tested by varying the mutation rate and limiting maintenance requests that can be delivered within 60 days to create optimized maintenance release package to satisfy the solution.

Introduction

One often unrecognized problem in software development is to determine what changes should be included in a release plan (RP). (Ruhe & Saliu, 2005; Carlshamre, 2002) This is often done through an ad hoc approach with no justification. At best it is done through negotiation of different stakeholders. (Ruhe & Saliu, 2005) This makes the ability to negotiation by different stakeholders a prevalent factor in what is included in a RP. Since negotiation is not an ability that all people share equally, this makes current methods to develop RP less than optimal. This negotiation is also time-consuming and costly. Studies show that some organizations spend up to 1,800 hours a year developing RPs. (Calshamre, 2002)

Software Maintenance relates to any changes made to an application or system after development is completed and delivered to the customer (Pigoski, 1997). Software Maintenance is estimated to be approximately 60% to 80% of the software life-cycle cost (Davis, 1991; Bueno & Juno, 2002). These maintenance changes can be categorized in four categories; corrective, adaptive, preventive, and perfective (Pigoski, 1997). Each type of maintenance has its place and purpose in the software maintenance phase. The categorization of maintenance request submitted by the end-users or customers, allow the software maintainers to better assess and prioritize which requests should be worked and when. Most major best practices, including PMBOK (PMBOK, 2008) and IEEE 1219 (IEEE 1219, 1998), use some type of negotiation to decide RPs, normally through the use of a Change Control Board.

With this in mind software maintainers and engineers are seeking ways to improve the way they perform maintenance from both a delivery and cost perspective (Pigoski, 1997). Researchers have noted that software maintenance of applications that have a long functional life create an interesting management dilemma (Card et al., 1987). Several organizations have

recognized the need to find ways to more efficiently perform software maintenance, which will allow a competitive edge over their competition (Moad, 1999).

Developing a good RP is important and little research has been done in the field (Carlshamre, 2002). Researchers and practitioners believe that you should not always do the most critical tasks first. Many factors should be taken into account when developing a RP. Criticality, severity and level of effort are all factors that should be considered when developing a RP.

This paper presents a customizable yet formal method for the development of RPs. It uses a Genetic Algorithm (GA) to search the large complex domain space of possible RPs to find optimal sets of changes. This method does not use negotiation, which we contend is a major drawback to traditional RP creation. Since it is an algorithm, RPs can be generated with much less time than conventional methods.

Related Work

The problem of RP is often referred to as the knapsack problem. We have to pack items into a knapsack that has a fixed volume. Unfortunately we do not have enough room for all of the items. Each item has a value and has a volume. To solve the knapsack problem we need to select a combination of items that provide the greatest value. There are a number of research papers published on the subject of RP that attempt to solve this knapsack problem.

Greer and Ruhe

The Greer and Ruhe algorithm was developed to find a way create a release plan (RP) that provided the flexibility in what requirements went into the RP and captured the requirements that met the customer's satisfaction. While developing their Genetic Algorithm (GA), they discovered that there were critical factors that need to be considered to ensure the GA was finding the most efficient RP for each iteration. The factors they felt that were significant to the success of the GA were: technical precedence, prioritization and coordination between required and available resources. The GA leverages the EVOLVE methodology to iteratively allow the GA to perform decision analysis to create RPs.

The research conducted by Greer and Ruhe was time find a solution the iterative creation of RPs through the use of GA. Their researched involved modeling the incremental software process model as an application that implements a GA to solve the problem of RP creation and management. Their approach to implementing a GA was based on the same principle in the model mentioned above; that all requirements are known before any design and development begins. Their approach allows for initial requirements to change and new requirements to be implemented during the project's life-cycle. Each RP created by the GA is considered to be a complete system that is of value to the customer.

One of the major challenges found by Greer and Ruhe in software engineering was the involvement of the customer. Their research has revealed it is not enough to just sit with a customer and gather requirements and allow the developers to go off develop application, but the necessity to have the customer actually rank the requirements in terms of priorities and the dependencies of requirements to one another. This approach can prove to be invaluable when developing with customer satisfaction as the highest priority for evaluating the success of

development project. Greer and Ruhe states that this approach may and can introduce change to requirements and introduce new requirements.

Greer and Ruhe utilized a weighted value system to capture customer ranking values. They sanitize the data by normalizing it into one weighted value. Their purpose for normalizing the data collected from the customers because each customer can have a different perspective on what requirements should be assigned what priorities. The Greer and Ruhe approach and method took the problem of building RP by change control boards and developed a GA to create incremental RP that are composed of the user articulated requirements based on rankings. Their method takes into account the possibility of re-planning of requirements throughout the RP process which is based on the assumption that the number RP is not limited or fixed in the beginning of process. Their GA is free to create as many or as little RP as necessary to encompass all of the requirements.

Greer and Ruhe do excellent job in explaining the use and/or impact of the level of effort (LOE) estimation and the constraints like precedence and dependencies have on the results of the GA. As far as LOE estimations apply to their GA, they make the assumption that effort will be equivalent to the incremental RP's total effort. They bring out a great point, that many requirements are connected in some way and these connections must be considered when creating a RP. In order to address the problem of software release planning, Greer and Ruhe have come up with six formulas/functions for the GA to use to find the best solutions.

Greer and Ruhe implement an approach that utilizes a GA's ability to search a vast space while integrating an iterative method. This approach combines the strengths of both, EVOLVE and GA. As part of the GA, Greer and Ruhe implemented Palisade's Risk Optimizer tool, which contributes different algorithms for adjusting the weighted values. Greer and Ruhe stipulates several advantages to their methodology for implementing a GA and EVOLVE as decision support for RP. EVOLVE allows for stakeholders to set priorities, software can be delivered in increments, evaluates precedence and dependencies between requirements, flexibility to change requirements, constraints and priorities, finds conflicts in stakeholders priorities, and most importantly approaches RP from a decision support point of view. A few issues we find with Greer and Ruhe's approach for RP are the following: it doesn't consider bug fixes, patches and the use of stakeholders to set the priority for each requirement. Allowing stakeholders to set priorities is a subjective activity which we believe may affect the outcome or resulting RP created by the GA.

They proposed use of EVOLVE, a method that is used during iterative software development for planning the releases. EVOLVE helps them to solve this problem by implementing the pieces of both computational and iterative methods to create a software release package. The GA is used at each iterative step to evaluate the optimal software release package for the current step. During this step of the requirements meeting the applied constraints are considered as part of the optimal solution. The algorithm makes use of the mutation and crossover to ensure the problem is being maximized for the optimal solution. This process is repeated until the solution can be no longer optimized.

Table Error! No sequence specified.: Pseudocode for Genetic Algorithm

```
BEGIN
        P = NewPopulation(seed)
        CHECK TerminateFlag U FALSE
        WHILE NOT (TerminateFlag)
                BEGIN
                        S_parent1 U SELECT (P)
                        S_parent2 U SELECT (P=S_parent1)
                        S_Offspring U CROSSOVER (S_parent1, S_parent2, crossover rate)
                        S_Offspring MUTATION (S_Offspring, mutation rate)
                        CHECK
                                If NOT IsValid (S_Offspring) THEN BackTrack(S_Offspring)
                                IF IsValid (S_Offspring)
                                        BEGIN
                                                P U P [(S_Offspring, EVALUATE (S_offspring)]
                                                Cull(P)
                                        END
                TerminateFlag = CheckTermination()
        END
        RETURN (Max(P))
END
```

Release Planner Prototype

One attempt to automate RP is the Release Planner Provotype (RPP). (Carlshamre, 2002) *Provotype* is a term referring to the provocation of current practice. The term was developed in contract to the term Prototype, which is based upon a possible solution. It is an application that accepts proposed requirements and their dependencies. The user can enter a total number of budgeted resources. The user can also select some requirements and require that they are included in the plan or force them to be excluded. RPR is also novel in the sense that it can create multiple version of RPs. A user can accept a suggested RP and then the application will assume that those requirements are implemented and seek the next RP.

Ho-Won Jung

Ho-Won Jung (Jung, 1998) paper presents a problem that several software developers and development companies are faced with when deciding what requirements should be included in a release package. One of the problems he mentions more specifically in his paper; how to best prioritize the requirements in a release package that is both within a given cost constraint while offering the greatest value. The resolution to this problem has been attempted using a variety of methods. He specifically mentions a process utilized by other researchers, J. Karlsson and K. Ryan, called Analytic Hierarchy Process (AHP) to address the packaging of releases. The AHP requires that a human inspection be performed to determine the validity of the release package. This inspection was found to be too complex for human judgment.

Ho-Won Jung in his paper proposes a variant of the 0-1 knapsack model that can modeled to remove the complexity associated with release planning on a budget that returns the best value. Jung compared the AHP to the knapsack and determined that the knapsack could be completed in two steps versus the three steps needed by AHP. Jung's goal was to create releases that provided maximum value for the least cost within an allowable cost limit. He created a variant

of the knapsack problem by adding multiples objectives to address his hypothesis. Jung reports at the time of his paper that there was no commercial code existed that could be used to solve this type of multi-objective knapsack problem. His knapsack problem had two steps; find the maximum value and find the minimum cost within the cost constraint.

Jung has decided to implement a model or process that takes the simple knapsack problem and enhances it by making it a multi-objective solution. This is accomplished by finding an alternative optimal solution beyond the initially found value. His approach was to implement an algorithm that included as many requirements that can be completed in a given cost constraint, once that number is found it is then passed to the minimum cost algorithm to produce the optimal solution.

Jung's paper presented a methodology that was used to test its ability to be useful in solving this type of release planning problems. Jung verified this by running his method against that of Karlsson and Ryan's real-world problem, the PMR project, to determine if his method could solve the problem and do a better job at it.

Gupta, Soni and Jolly

Gupta, Soni and Jolly research is based on an information technology age old problem; how to put together software maintenance plans that deliver the most bang for the customer's dollar in the most efficient way, with limited resources. They state that release planning as a keystone issue. This means software maintainers must perform proper assignment of requirements that are placed in a logical order of significance to maximize return, thus reducing delays in customer satisfaction while dependencies and constraints are accounted for and met. They assert through other research that there is increased pressure to get systems, applications and enhance to the market faster and faster and in many cases there are more requirements than actual available resources that can work to implement.

Gupta, Soni and Jolly have found that many scholars researching this problem have come up with a list of variables that need to be considered when solving the problem of software release planning. Scholars' lists the following; importance or business value, preferences of the stakeholders, cost of software development, quality of customer stated requirements, development risk(s) and the dependencies among the varying requirements [Joachim & Kevin, 1997]. They found one specific risk with the development of software; time to completion. The fact that Actual Time to Completion will exceed the Estimated Time to Completion beyond an intolerable or acceptable time period. This risk and fact resulted in a lot of research and the development of several cost models. Their paper was focused on the above risk of time to completion while addressing an over budget situation within a project caused by the above stated risk related completion time.

Bagnall, Rayward-Smith and Whittley

Bagnall, Rayward-Smith and Whittley asserts that many companies engaged in software development will undoubtedly have to address the difficult task of determining which requirements or enhancements should go in what release. They call it the "next release problem (NRP)." They have listed four things that must be considered when creating the NRP. The items that must be considered are; customer demands, prerequisite requirements, variation of customer valuation of requirements and complexity of requirements. Their research uncovered

its a major challenge for companies to select the proper set of requirements that can be delivered within the customer's budget and meet their demands.

There proposed model to find the most optimal software release incorporates three main factors: ***R*** represents all the potential or requested software enhancements, ***I*** represents each customer with a set of their own requirements to be considered and W_i represents the weight, which depicts the organizational importance applied to these requirements. This algorithm also analyzes requirements for dependencies with other requirements in the dataset. This model can incorporate several other factors to find optimal software releases based on these additional values such as: *time*, *cost* and *requirement dependencies*. This algorithm uses the "*knapsack*" methodology to create the software release.

Their research into "the NRP" has been termed a NP-hard irregardless of the basic nature of the problem and requirements. They have come up with three algorithms to address the NRP; steepest ascent, first found, and sampling. Similar to Ho-Won Jung, Bagnall, Rayward-Smith and Whittley adapted the knapsack formulation to solve the NRP. They found that several algorithms have been used to solve the NRP with a varying success. Their research found that simulated annealing algorithms found the best solutions to the NRP in less than average amount of time. They recommend additional research can be done into the NRP using both heuristic (utilizes things like tabu search and neighborhood based schemes) and exact (utilizes a special structure that solved knapsack problem) techniques.

Gupta, Soni and Jolly

Gupta, Soni and Jolly states that software release planning is a major issue related to software development. They have also found through research many software development companies are scrambling to overcome the challenges of software release planning. These challenges include; resources availability, requirements with conflicting priorities, requirement dependencies and the pressures of getting releases to the stakeholders to name a few. One of the things they found that need to be addressed to overcome the software release planning problem is to develop a way to properly assign the requirements in a logical order that meets the expected profits and customer satisfaction.

Gupta, Soni and Jolly have concluded that the stakeholders are the key to solving the software release problem. Understanding the how to appropriately group the stakeholders. Each group of stakeholders will have a vastly different perspective of what should be included and when. Software release planning must focus on value and satisfaction within a given constraint, whether the constraint is money, time or need.

There proposed model uses several factors to find the optimal software release. Two of the algorithm's main variables are the number of stakeholders (N) and requirements (M). Determination of the sensitivity of the cost factor is based on $M < N$. The algorithm utilizes several other factors; such as: subsets of both stakeholders and requirements while it continues to optimize the software release for the various stakeholders.

They have also concluded that software companies' inability to lock-down requirements can have an adverse effect on the planning and delivery of software releases. In many cases they have found that many of requirements are typically communicated in an informal way which in

turns leads to requirements that are not well-defined and misunderstood. The non-formulation of requirements also makes it difficult for the stakeholders to evaluate features and requirements that are delivered, thus reducing customer satisfaction. Research has uncovered that some software projects have hundreds to thousands of requirements that need to be planned making them very complex. These types of projects are significantly more difficult based on the size alone without including constraints and customer satisfaction.

Genetic Algorithms

This overview of genetic algorithm (GA) discusses what a GA is, its beginning and its applications in software engineering. The paper states that a GA is a computer application that emulates the biological process of natural evolution to find a solution to a problem (Mitchell, 1995). We find that GA were first developed by Bremermann (Bremermann, 1958) in the 1950's and later discussed in the 1960's by John Holland and later expanded through research by Holland while at the University of Michigan with the assistant of his colleagues and students during the 60's and 70's (Mitchell, 1995). Holland's focus during this time was to find a way to implement the adaptation seen in biological evolution in a computer application (Mitchell, 1995). This paper answers the following question many researchers of GA were seeking to answer: "Why use evolution as an inspiration for solving computational problems?" The overview states that problems with a vast number of possible solutions are best-suited for a computer application that could evolve to a solution through multiple iterations until a viable solution could be found.

GA in its simplest contains the following elements: a set of randomly generated solutions called chromosomes, a selection process based on a chromosome's fitness score, recombination of selected chromosomes to create a new offspring chromosome is called *crossover*, and some randomly computed probability to change an allele is called *mutation*. Mutation is significant part of every GA; it allows genes within a chromosome to be inverted so that solutions that would not be searched can be considered and evaluated. The paper describes GA's chromosome encoding as a bit string of 1's and 0's or as alphanumeric characters. Whichever method of encoding is used the GA must be able to decode the string to calculate the fitness score of the chromosome.

The paper lists a very simple GA as an example. A simple GA performs the following activities until a solution is found of the number or iterations (generations) have been met:

1. Begins with an initial population of randomly generated chromosomes
2. Each chromosome's fitness is calculated
3. Select parent pairs of chromosomes for recombination until the next generation is created
4. Perform crossover (recombination) at a randomly selected position(s) in the chromosome.
5. Perform random mutation on chromosome if the probability factor is true
6. Create new generation with chromosome
7. Repeat steps 2 through 7 until solution is found or numbers of generations have been met.

The paper lists several applications where GA could be used successfully. GA can be used for problems like optimization, automatic programming, machine learning, economic models, immune system models, ecological models, population genetic models, interactions between evolution and learning, and models of social systems (Mitchell, 1995). GA is simple enough to create and implement when a problem being solved requires the need to search a large area for a solution that is unknown. The key to every GA is the fitness function, which must be modeled correctly to yield good solutions.

Methodology

This paper's focus is to develop a GA to model the selection of a software maintenance release package, based on evolutionary computing, to optimize software maintenance release package creation.

Discussion of Methodology

As part of the research into the use of GA use in software maintenance, an application with a fully functional GA was developed in Java. This takes the research a step further than just a theoretical look at applying GA use in software maintenance. The application utilized some sample data to run through the GA to get some quantitative data. The sample data contained one-hundred and twenty-three rows of test data that was imported from a comma separated value text file into SQLite database for use in the application.

The GA application was designed with several entry fields to allow for various types of test cases to be produced by varying the parameters. The effect of each parameter can be observed to see if the changes would directly influence the results of the GA's ability to optimize the maintenance release package in a positive or negative way. Each individual will be encoded with a binary string of 1's and 0's to determine what maintenance items should be included and evaluated during the computation of their individual fitness. The higher the individual fitness score the more likely an individual will be selected as a parent producing offspring's for the next generation of individuals. An image of the user interface is shown in Figure 1.

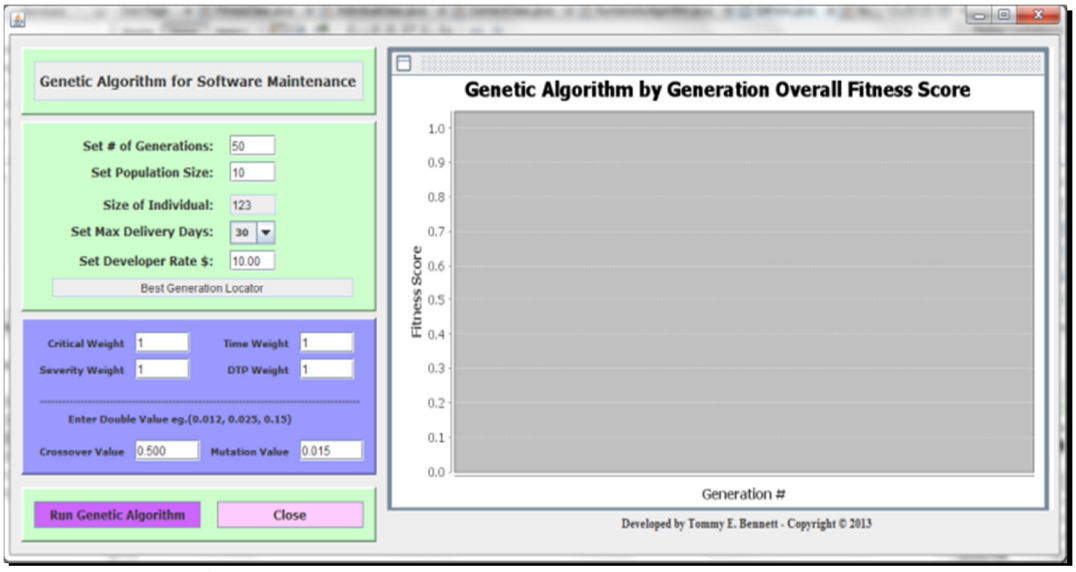


Figure 1. Genetic Algorithm Application Main Screen

Genetic Algorithm for Maintenance Release Package Optimization

The GA developed for optimization attempts to address the problem of automated creation of maintenance release packages. The traditional method of manually selecting which maintenance requests (MR) should be packaged together developed and released can be very time consuming and quite an arduous task. The GA seeks to automate the process of selection by applying various weights to add significance to requests based on the customer's expectation and needs. To accomplish the evaluation and building of the release packages the solution was modeled to address the areas of importance during the selection process.

The GA begins by randomly generating a generation of individual chromosomes modeled as maintenance release package to search for a solution. The fitness function used for this GA is shown in equation 1. In this fitness function ***i*** is each individual to be evaluated, *category(i)* is the category the MR was assigned to, *severity(i)* is the level of severity given to the MR by the help desk, *ROM(i)* is the time estimated to complete the development of the MR, and *deliverydays(i)* is the number of days for estimated to delivery.

$$f(i) = (category(i)+severity(i) / (ROM(i)+deliverydays(i)) \qquad (1)$$

The GA has several parameters it uses to produce the optimal maintenance release package; they are *Generations*, *Population Size*, *Maximum Delivery Days*, *Category Weight*, *Severity Weight*, *Time Weight*, *Delivery Weight*, and *Mutation Rate*. Each test case listed in Table 1 below will be executed ten times to normalize the results. The only value that will be varied is the Mutation Value. The test will observe the effect the mutation rate has on the creation of an optimal maintenance release package. This will allow errant data to be disregarded as an anomaly that falls outside the tolerance of the data that is consistently shown over ten passes. The resulting data will be displayed in a line graph with the various test cases listed in Table 2. The graph will show which test case yields the best setting to be used to create an optimized maintenance release package.

Table 2 Parameters for Test Cases

Test #	Generations	Population Size	Maximum Delivery Days	Weighted Values				Mutation Rate
				Category	Severity	Time	Delivery	
1	500	100	60	1	1	1	1	0.015
2	500	100	60	1	1	1	1	0.035
3	500	100	60	1	1	1	1	0.065
4	500	100	60	1	1	1	1	0.100

Maintenance release packages created by people are limited by what people can visualize and understand, whereas this GA will consider and evaluate maintenance release packages that would likely not be considered by a change control board. This is made possible by the GA performing crossover and the random probability that gene mutation will be performed. These

types of activity cannot be performed by people to accurately optimize the maintenance releases. The pseudo code is shown in Table 3.

Table 3. Pseudo Code and Genetic Algorithm

algorithm	runGeneticAlgorithm()
input	getGAFormValues() → loaddata(gaDB.db) IndividualClass.initialGeneration() createChart() evaluate → Fitness.calcFitness() repeat → IndividualClass.selectionIndividual()
output	Final Solution → Maintenance Release Package (MRP) test → (Does MRP meet requirement or Number of Generations tested? graph → plot Individual Fitness Score → runGeneticGraph()

```
public void run() {
    try {
        for (int iGenCounter=0; iGenCounter<gac.get_NOG(); iGenCounter++) {
            this.icAL = fc.calcFitness(this.mcAL, this.icAL, gac, IGenCounter+1);
            this.updateDataset(this.icAL, iGenCounter+1);
            this.icAL = ic.selectionIndividual(this.icAL, iGenCounter, gac, this.rdmNum);
        }
        Thread.sleep(5);
    } catch(InterruptionException ie) {}
    this.computeEstimatedCost();
    MyClass.displayMessage("Best"+
                                "\nGeneration: "+GAForm.iBestGenerationNumber+
                                "\nIndividual: "+GAForm.sBestIndividual+
                                "\nScore: "+GAForm.iBestIndividualScore);
}
```

Individual Chromosome Encoding

Individual Chromosomes (IC) are created randomly based on some type of predefined encoding. Encoding can be done utilizing 1's and 0's, alphanumeric and symbols. Depending upon the encoding selected, each type requires the GA's creator to design the IC's decoding. The IC's length and size is solely dependent on the rules implemented to solve the problem. The length can vary from three characters to as large as needed. Some thought and care should be given during the IC design to model the problem to be solved.

Each IC is created randomly in the initial population. For our specific problem the IC's is determined by the number maintenance requests, which can be of variable length. Each maintenance request in the maintenance log represents a specific gene (bit in a string) in the IC.

Example(s)

When the GA is initially run, an IC is randomly generated with a length that corresponds to the number of maintenance requests that are in the maintenance log. For an example, if the maintenance request log contains twelve maintenance requests, then the IC would contain twelve genes and its length would be twelve characters or bits (i.e., 101101101101) in length. The

actual maintenance log used to conduct the research initially has one-hundred and twenty-three maintenance requests. The number of maintenance requests has no effect on the performance of GA, other than additional time needed to evaluate each maintenance request. The figure below shows the construction of on individual to be evaluated for maintenance release optimization. An actual IC can be seen below in Figure 2.

Individual:
1100101000111000101000111100010001110101111000000111110101010111100011100101

Figure 2. Individual Chromosome

Fitness Function

The fitness function is the cornerstone of every GA. The complexity of any GA is determined by the fitness function (Coley, 1999). The fitness function is developed to model the problem or solution. The fitness function is considered to be the most difficult part of the GA to develop. The GA applies the fitness function against the vast possibilities to find an optimal solution to the problem. The fitness function is continually used to evaluate all the individuals in each generation assigning a fitness value to determine an individual's probability of reproducing; searching for a solution suitable or until the number of generations has been exhausted.

The fitness of an individual chromosome in the application for software maintenance release packages was developed to assess the maintenance package ability to meet customer's expectation for meeting the following metrics; severity, delivery time, criticality, and time to develop. These metrics are measured by the fitness function. The fitness function's input comes from the maintenance request log stored in database file that is read from the disk. The fitness function takes an encoded individual, creates a maintenance release package, and checks it against the maximum delivery time rule. Maintenance request whose time to deliver exceeds the user inputted maximum delivery time (default value of 60 days) are given a fitness score of zero. The GA's values can be weighted to place the user's desired emphasis on the categories they deem important for the creation of a maintenance release package. This is covered in more detail in the next section.

Weighted Values

The GA includes input fields that allow the GA to weight the four categories to give significant strength to a specific value. The weighted values are multiplied against its corresponding values assigned to each maintenance request. The weights can influence the fitness value of an individual by either increasing or decreasing its resulting score. The other effects are the number of 1's and 0's that can be found in the resulting individual. When the values in the ROM (time to develop) and Delivery Days (delivery time promised) are weighted the number of 0's increase and likewise when the Category and Severity are weighted the number of 1's increase. This was observed through testing.

For the purposes of this problem, no values were weighted giving any one category more importance than the others. This approach was used to keep the data consistent to observe the effect of changing the mutation rate would have on finding an optimal maintenance release package. Varying too many values would make base-lining the GA difficult. It was important to form a base-line for the data to ensure the GA was performing as designed. Utilizing the

weighted parameters would be more appropriate in future research that could observe the effect the weights actually have on creating a more effective maintenance release package.

Algorithm Parameters

Generation

A generation size of 500 was chosen to ensure the GA had a chance to evolve to search all possibilities to find the optimal maintenance release package. Studies have shown that small generation sizes don't yield an optimal solution (Coley, 1999). Researchers have found that a generation size of 700 to thousand is a good starting point. This value can be tweaked to find a generation size that works for a particular GA. I found through testing that a generation size of 500 works well for this particular problem to find a maintenance release package that would meet the desired outcome. The generation size value can be seen in Table 1 above.

Population

The population size chosen must be divisible by two for the GA to work. A population size of 100 was used for this GA. With a population size of 100 and a generation size of 500, the GA will analyze and evaluate 50,000 binary encoded individual chromosomes that have a length of 123. This means the GA has a vast search space with up to 1.06×10^{37} possibilities to find the optimal maintenance release package.

Results

The test to create an optimized maintenance release package was run through four different test cases ten passes each to ensure we could obtain the best results. Each test case was varied by the mutation rate from 0.015 up to 0.100, as seen in Table 1 above. The mutation rate was varied to see if any significant changes to the best results could be achieved. The results of each test case can be seen in Figure 8 along with a 3-D bar graph showing them plotted graphically. Table 4 shows the graphs for all ten passes from Test Case #4.

Table 4. Best Individual Fitness per Run Results

	Test Case #1	**Test Case #2**	**Test Case #3**	**Test Case #4**
1	8.949968446	8.943755216	9.187640538	9.340128852
2	8.583087648	9.383418896	9.056752871	9.693541550
3	8.578239163	9.227248360	9.088779646	9.172995630
4	8.577756285	9.468571581	9.411319329	9.771752229
5	8.594948592	9.143339531	9.545683321	9.291428724
6	9.101592971	8.807149697	9.445107207	9.412169002
7	9.269291467	9.319471200	9.546832237	9.547327240
8	8.392243492	9.244231410	8.989176472	9.070065087
9	8.431615140	9.036218319	9.183202446	9.185265443
10	8.881794432	9.727985995	9.412985995	9.147646890
Average Fitness	8.736053764	9.230139020	9.286748006	9.363232065

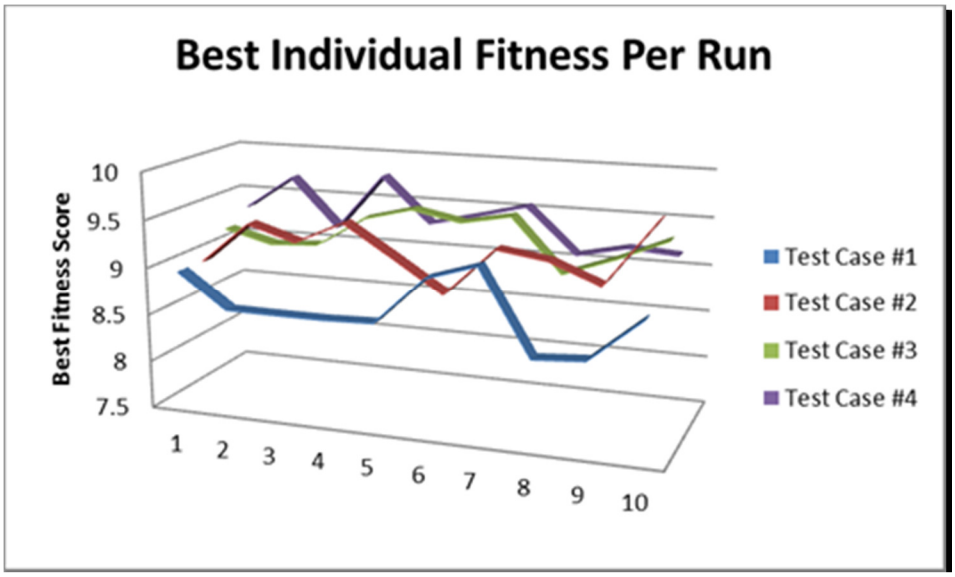
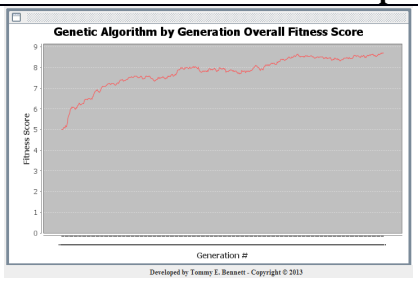
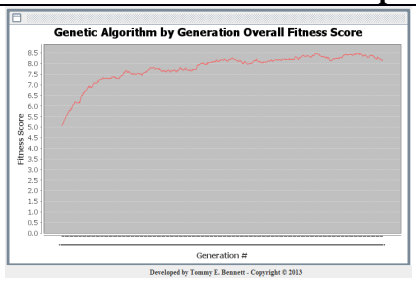
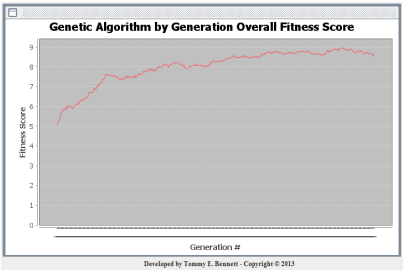
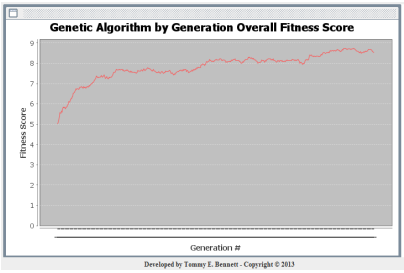
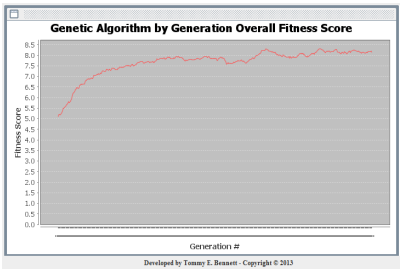
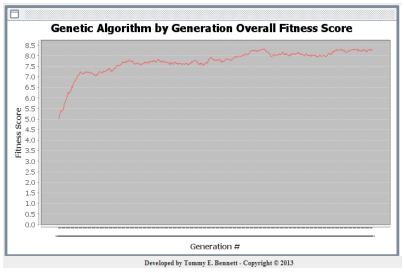
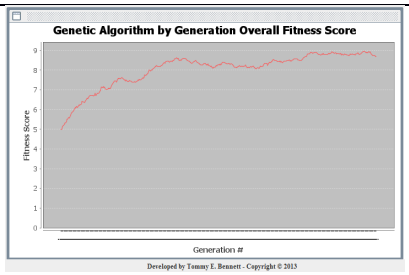
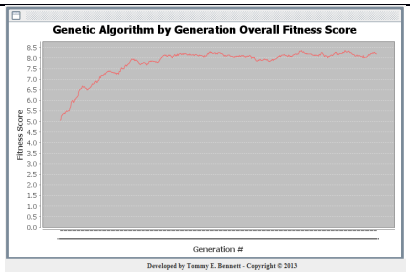
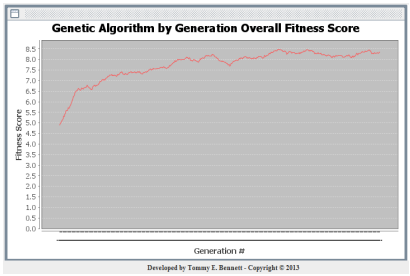
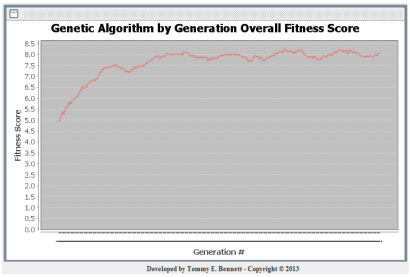


Figure 3. Best Individual Fitness Chart and Graph

Table 5. Test Case #4 – Best Individual per Generation Graph

Test Run #	Best Individual Fitness Score/Generation Fitness Graph	Test Run #	Best Individual Fitness Score/Generation Fitness Graph
1	 9.34	6	 9.41
2	 9.69	7	 9.54
3	 9.17	8	 9.07

4		9	
	9.77		9.18
5		10	
	9.29		9.14

Conclusion

The results confirm the use and application of how a GA can be applied to maintenance release package creation. Based on other research performed with GA and their ability to be used for optimization problems such as test-suite reduction (Nachiyappan et al., 2010), the findings show GA can be used successfully for software maintenance. In each of our test cases and increasing the mutation rate, we experienced an increase in individual fitness scores and the average fitness score for each test case. This tells us that with additional increases in mutation rate we should continue to see improvements in the average fitness score of each test case.

From our results, we can see the application of GA are very promising. Given the time to perform more experimenting with the parameters, the GA could continue to evolve individuals producing increasingly better solutions. The current research did not allow the time to adjust the weighted values to see what affect, if any, they would have on locating the optimal solution faster or slower. This is definitely something that needs to be explored. The results of testing would benefit greatly from validation against a real-world example. The results in the current study utilizing sample data could only be hypothesized to its accuracy.

This GA poses several benefits to software engineering in the area of maintenance. The GA could be used as the foundation for creating maintenance release packages to be vetted during a change control board (CCB) meeting. Software engineers can use the solutions provided by the GA to estimate cost of various scenario based maintenance releases in relatively small time-frame. The resulting maintenance release can be used to evaluate maintenance release packages created by CCB, developers and business owners for training purposes. The GA can provide various benefits to software applications in the maintenance phase of the SDLC.

References

Mitchell, M. (1995). *Genetic Algorithms: An Overview* [PDF]. Santa Fe: Santa Fe Institute.

Baqias, A. A., Alshayeb, M., & Baig, Z. A. (2013). Hybrid Intelligent Model for Software Maintenance Prediction. *Proceedings of the World Congress on Engineering, 1*.

Pargas, R. P., Harrold, M. J., & Peck, R. R. (1999). Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability, 9*(4), 263-282. doi: 10.1002/(SICI)1099-1689(199912)9:43.0.CO;2-Y

Forrest, S., Le Goues, C., Nguyen, T., & Weimer, W. (2009). A Genetic Programming Approach to Automated Software Repair. *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation,* 947-954.

Brown, M. S., Neptune, F., Bohl, G. L., Cifuentes, M. A., Bennett, T. E., Tsobgni, Y., ... Sissay, A. (2013). A Survey of the use of Genetic Algorithms in Structural Testing. *International Journal of Advanced Research in Computer Science and Software Engineering, 3*(10), 1-6.

Weimer, W., Forrest, S., Le Goues, C., & Nguyen, T. (2010). Automatic program repair with evolutionary computation. *Communications of the ACM, 53*(5), 109-116.

Nachiyappan, S.; Vimaladevi, A.; SelvaLakshmi, C. B., "An evolutionary algorithm for regression test suite reduction," *Communication and Computational Intelligence (INCOCCI), 2010 International Conference on* , vol., no., pp.503,508, 27-29 Dec. 2010

Moataz A. Ahmed, Irman Hermadi, GA-based multiple paths test data generator, Computers & Operations Research, Volume 35, Issue 10, October 2008, Pages 3107-3124, ISSN 0305-0548.

P. M. Bueno and M. Juno, "Automatic test data generation for program paths using GA", *International Journal of Software Engineering and Knowledge Engineering,* vol. 12 (6), pp. 691-709, 2002.

Pigoski, T. M. (1997). *Practical Software Maintenance: Best Practice for Manging your Software Investment*. New York: John Wiley & Sons, Inc.

Davis, L. (1991). *Handbook of GA*. New York: Van Nostrand Reinhold.

Coley, D. A. (1999). *An introduction to GA for scientists and engineers*. Singapore: World Scientific.

Stark, G. (1997) *Measurements to Manage Software Maintenance.* Colorado Springs: The MITRE Corporation.

Card, D. N., Cotnoir, D. V., Goorevich, C. E., "Managing Software Maintenance Cost and Quality," *Proceedings of International Conference on Software Maintenance,* 1987.

Moad, J., "Maintaining the Competitive Edge," *Datamation*, February 1990, pp. 61-66.

Joachim K., Kevin R., (1997) "A cost-value approach for prioritizing requirements." IEEE Software, 14(5): pp. 67-74.

Bremermann HJ: The Evolution of Intelligence: The Nervous System as a Model of its Environment. (Technical Report, No.1, Contract No. 477, Issue 17). Seattle WA: Department of Mathematics, University of Washington. 1958.

Bagnall, Rayward-Smith, Whittley (2001). *The next release problem*