UNF UNIVERSITY of NORTH FLORIDA

UNF Digital Commons

UNF Graduate Theses and Dissertations

Student Scholarship

2007

Dual Constraint Problem Optimization Using A Natural Approach: Genetic Algorithm and Simulated Annealing

James P. Sweeney

Suggested Citation

Sweeney, James P., "Dual Constraint Problem Optimization Using A Natural Approach: Genetic Algorithm and Simulated Annealing" (2007). *UNF Graduate Theses and Dissertations*. 283.
https://digitalcommons.unf.edu/etd/283

This Master's Thesis is brought to you for free and open access by the Student Scholarship at UNF Digital Commons. It has been accepted for inclusion in UNF Graduate Theses and Dissertations by an authorized administrator of UNF Digital Commons. For more information, please contact Digital Projects.
© 2007 All Rights Reserved

UNF UNIVERSITY of NORTH FLORIDA

DUAL CONSTRAINT PROBLEM OPTIMIZATION USING A NATURAL APPROACH: GENETIC ALGORITHM AND SIMULATED ANNEALING

by

James P. Sweeney

A thesis submitted to the
School of Computing
in partial fulfillment of the requirements for the degree of

Master of Science in Computer and Information Sciences

UNIVERSITY OF NORTH FLORIDA
SCHOOL OF COMPUTING

June 8, 2007

Copyright (©) 2007 by James P. Sweeney

All rights reserved. Reproduction in whole or in part in any form requires the prior written permission of James P. Sweeney or designated representative.

The thesis "Dual Constraint Problem Optimization Using a Natural Approach: Genetic Algorithm and Simulated Annealing" submitted by James P. Sweeney in partial fulfillment of the requirements for the degree of Master of Science in Computer and Information Sciences has been

Approved by the thesis committee: Date

Signature Deleted

7/2/07

Sanjay P. Ahuja, Ph.D.
Thesis Advisor and Committee Chairperson

Signature Deleted

7/2/07

Roger Eggen, Ph.D.

Signature Deleted

7/2/2007

Yap Chua, Ph.D.

Accepted for the School of Computing:

Signature Deleted

7/2/07

Judith L. Solano, Ph.D.
Director of the School

Accepted for the College of Computing, Engineering, and Construction:

Signature Deleted

7/2/07

Neal S. Coulter, Ph.D.
Dean of the College

Accepted for the University:

Signature Deleted

11 July 2007

David E.W. Fenner, Ph.D.
Dean of the Graduate School

ACKNOWLEDGEMENT

I want to thank my advisor, Dr. Sanjay Ahuja, for his direction, feedback, and encouragement during my research. Gratitude also goes to my thesis committee, Dr. Roger Eggen and Dr. Yap Chua, who have brought a fresh outlook and inspired new challenges during the construction of this body of work. Most importantly, I owe the opportunity to pursue this next level of education to my wife, who has graciously shared time and leant unconditional support during this lengthy endeavor.

CONTENTS

List of Figures vii

Abstract ix

Chapter 1: Introduction 1

1.1 The Dual Constraint Optimization Problem: Grid Resource Allocation 1

1.2 Grid Computing and Its Variants 3

1.3 Utility Computing: The Grids Economic Approach 6

Chapter 2: Survey of Related Work 8

2.1 The Resource Allocation Problem: How to Optimize? 8

2.2 Stochastic Algorithmic Solutions 10

2.3 Focus of Thesis 13

Chapter 3: Genetic Algorithms: Survival of the Fittest 16

3.1 Reproduction 18

3.2 Crossover 19

3.3 Mutation 22

Chapter 4: Simulated Annealing: Cooling Hot Metal 25

Chapter 5: Testing and Evaluation of the Stochastic Algorithms 30

5.1 The Optimal Solution 31

5.2 The Genetic Algorithm 34

5.2.1 Algorithmic Functionality 35

5.2.2 Variable Values 38

5.3 The Simulated Annealing 44

5.3.1 Algorithmic Functionality 44

5.3.2 Variable Values 48

Chapter 6: Conclusions and Future Work 58

6.1 Conclusions 58

6.2 Future Work 59

References 61

Appendix A: Optimal Solution Code Listings 64

Appendix B: Genetic Algorithm Code Listings 73

Appendix C: Simulated Annealing Code Listings 97

Vita 115

LIST OF FIGURES

Figure 1: Computational Jobs are Split to Run on the Grid 5

Figure 2: The Grid Resource Broker is a Part of the User-Level Middleware 7

Figure 3: Equations for the Total Cycles, Time Constraint, and Cost Constraint 9

Figure 4: Genetic Algorithm Flowchart 17

Figure 5: Weighted Roulette Wheel Representing Four Chromosomes. Chromosome #2 has the Best Fitness Value and thus the Highest Probability of Being Chosen..19

Figure 6: Crossover Example 20

Figure 7: Genetic Algorithm Propagation Pseudo Code 21

Figure 8: Wright's Adaptive Surface 23

Figure 9: Examples of Mutation Methods 24

Figure 10: Probability Function 26

Figure 11: Flowchart of a Standard Simulated Annealing Algorithm 27

Figure 12: Resource Scenarios and Their Associated Speed and Cost 32

Figure 13: Optimal Solutions for Problem Constraints #1 32

Figure 14: Optimal Solutions for Problem Constraints #2 33

Figure 15: Ten Available Resources and Their Associated Speed and Cost 34

Figure 16: Solution for Optimization Problem Using Ten Available Resources 34

Figure 17: Sample Population Size vs. Fitness Value GA Test Run 40

Figure 18: Sample Genetic Algorithm Runtime vs. Population Size GA Test Run 41

Figure 19: Sample Job Execution Time vs. Population Size GA Test Run 42

Figure 20: Sample Percentage of Optimum vs. Population Size GA Test Run 43

Figure 21: Sample Energy Value vs. Tweak Factor SA Test Run 45

Figure 22: Simulated Annealing Runtime vs. Tweak Factor Test Run 46

Figure 23: Sample Job Execution Time vs. Tweak Factor SA Test Run 47
Figure 24: Sample Percentage of Optimum vs. Tweak Factor SA Test Run 47
Figure 25: Sample Energy Value vs. Number of Iterations SA Test Run....................... 50
Figure 26: Sample Simulated Annealing Runtime vs. Number of Iterations Test Run . 51
Figure 27: Sample Percentage of Optimum vs. Number of Iterations Test Run............ 51
Figure 28: Sample Energy Value vs. Alpha SA Test Run... 52
Figure 29: Sample Simulated Annealing Runtime vs. Alpha Test Run 53
Figure 30: Sample Percentage of Optimum vs. Alpha Test Run.................................... 54
Figure 31: Sample Energy Value vs. Final Temperature SA Test Run.......................... 55
Figure 32: Sample Simulated Annealing vs. Final Temperature Test Run 56
Figure 33: Sample Percentage of Optimum vs. Final Temperature SA Test Run.......... 56
Figure 34: Results and Comparison of the GA and SA Rest Runs. 58

ABSTRACT

Constraint optimization problems with multiple constraints and a large solution domain are NP hard and span almost all industries in a variety of applications. One such application is the optimization of resource scheduling in a "pay per use" grid environment. Charging for these resources based on demand is often referred to as *Utility Computing*, where resource providers lease computing power with varying costs based on processing speed. Consumers using this resource have time and cost constraints associated with each job they submit. Determining the optimal way to divide the job among the available resources with regard to the time and cost constraints is tasked to the *Grid Resource Broker* (GRB). The GRB must use an optimization algorithm that returns an accurate result in a timely manner. The *Genetic Algorithm* and the *Simulated Annealing* algorithm can both be used to achieve this goal, although Simulated Annealing outperforms the Genetic Algorithm for use by the GRB. Determining optimal values for the variables used in each algorithm is often achieved through trial and error, and success depends upon the solution domain of the problem. Although this work outlines a specific grid resource allocation application, the results can be applied to any optimization problem based on dual constraints.

Chapter 1

INTRODUCTION

Constraint Optimization Problem's (COP) are found in many fields and span across a wide variety of industries. Examples extend from the evaluation of a business based on assets, expenses, and annual turnover, to the optimization of transportation routes. COPs can be found in chemical processing, energy systems, airlines, railroad, trucking, insurance, and all other forms of business and research.

This thesis will evaluate a COP that has two constraints and is associated with splitting a large computational job into smaller tasks to be processed on concurrent available resources running on a grid. Finding a solution using an exhaustive search algorithm would take longer to complete than the useful lifecycle of the result, or depending on the size of the solution space, could take years to calculate. We will compare the performance of two stochastic algorithms and then contrast their results to an optimal search within small solution spaces. The best performing algorithm will be recommended for use in our grid computing resource allocation COP.

1.1 The Dual Constraint Optimization Problem: Grid Resource Allocation

Grid computing is being used in a wide variety of ways throughout the educational, research, and commercial communities. There are many types of grid computing paradigms. Some of these include cluster computing, data grids, and computational

grids. The ability to pull together the power of many disparate computer systems in a heterogeneous environment makes computational grids the most dynamic use of the grid architecture. This conglomerate of computing power can be used to power High Performance Computing (HPC) applications in almost all industries, such as Aerospace, Life Sciences, Financial Services, and Automotive and Electronics. As this demand for more computing power grows, so does the commercialization of this computing power. The leasing of computer time is not a new concept; it was how the early computer users gained time to run their programs on mainframe systems. This practice of leasing time, or processing power, is once again being implemented and will only continue to grow in popularity due to the cost of purchasing equipment to run HPC applications. The business model of leasing computer resources as an *on demand* resource is commonly referred to as *Utility Computing* [Buyya02].

Guaranteeing Quality of Service (QoS) for each application is especially difficult. The resource Service Level Agreements (SLA) offered by the service providers must be mapped to the application level SLAs [Menascé04], which denote the terms required by the application. The SLAs detail quantifiable metrics that have to be met between a user application and a service provider. The availability of many different service provider options creates a very complex scheduling and optimization problem.

Attempting to find an optimal solution would require an exhaustive search, which would require more time and money than our SLA would allow. The amount of time needed to find an optimal solution climbs exponentially as the number of available

resources increase. Each resource could have allocated anywhere from 0% to 100% share of the job, as long as the sum of the shares allocated to all resources equal 100%. An optimal solution for this problem is NP hard (Non-deterministic Polynomial-time hard) and requires heuristic solutions [Menascé04A]. This optimization process becomes more complex when the constraints are added. Even though we have chosen time and cost as our constraints, the results of this thesis can be applied to any dual-constraint-based, NP-hard problem.

This research compares two stochastic algorithms as possible solutions to this optimization problem—Genetic Algorithm and Simulated Annealing. These two algorithms were created to mimic nature and the way it uses a stochastic approach to biological reproduction and the cooling of metals. Chapter 3 will further explain Genetic Algorithms and Chapter 4 will provide an overview of Simulated Annealing.

1.2 Grid Computing and Its Variants

The architecture of Grid Computing is the next advance in distributed computing [Ferriera03]. Grid computing uses many different heterogeneous resources to simulate a single computing machine that, when united with a large number of donor resources, can be extremely powerful. There are many types of grid computing architectures, and each offers redundancy, dynamic expansion, and improved performance to each respective application.

Cluster computing is a very close relation to grid computing, where resources are typically linked together by a fast Local Area Network (LAN). Cluster computing is not normally viewed as grid computing in the traditional sense, due to the way the resources are tightly coupled in a homogeneous manor. Grid computing is normally thought of as many types of heterogeneous resources, loosely coupled together across administrative domains. Clusters, on the other hand, are on a single domain with identical hardware and software configurations used across all resources. Computing clusters may be used as one of the available resources in a grid, but a grid would not be viewed as an available resource in cluster computing.

A data grid is the second most common type of grid computing [Ferriera03] and can have several uses associated with the storage of data. Data grids normally utilize space on almost any type of donor resource, although typically this space is scavenged from individual user workstations. A data grid can act as a resource for transitory data, such as what might be used by researchers who run applications that need large amounts of storage (terabytes, petabytes) to run complex computations. Researchers could also use this area as a virtual workspace, where a large amount of temporal storage is needed to study results from a test run. When data are stored on the grid using striping, then data can be accessed faster with more efficiency. Striping stores the same data at multiple locations, enabling parallel searches. A data grid also provides the ability to have data redundancy, with data at many different resources, thus eliminating a single point of contention, or failure.

In this research we concentrate on the most popular form of grid computing [Ferriera03], the computational grid. This type of grid reduces a large job to many different small sub-tasks and sends each sub-task to a different resource to process. When the resource has completed its sub-task, the result is returned to the controller, which pieces all of these sub-results together to form the answer. Figure 1 illustrates the Grid Resource Broker splitting the sub-tasks, sending them to an available resource, and then collecting the results.

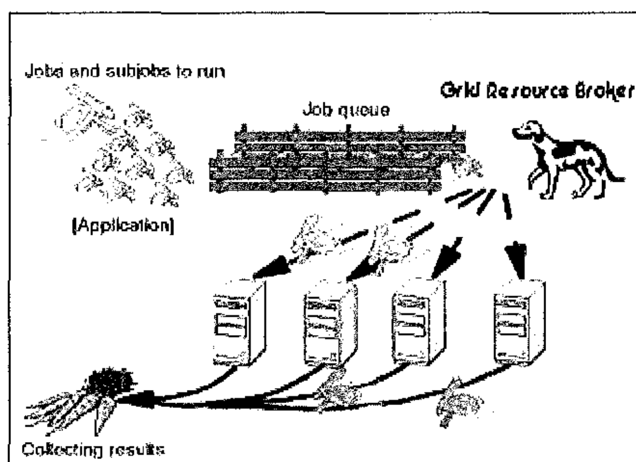


Figure 1: Computational Jobs are Split
to Run on the Grid [Ferriera03]

The Grid Resource Broker (GRB) [Buyya02] has the job of dividing the job among the available resources. The GRB also has other responsibilities, including resource discovery, resource selection, and job division, as well as task-resource matching and optimization.

1.3 Utility Computing: The Grids Economic Approach

Utility computing is fashioned after other on-demand services, such as electricity, water, and gas. The service, in our case of computing power, is offered by service providers with various levels of computing speed at different costs. No matter which industry employs utility computing, all users would like for their jobs to be completed as quickly as possible. Moreover, there are many products that manage scheduling and optimization for grid applications, such as GrADS, DAGMan, Askalon, ICENI, APST, and Pegasus [Buyya02]. These products are based on minimizing execution time, but they do not evaluate cost constraints associated with utility computing. Both cost and execution time must be considered when purchasing computing power from a service provider. Some users may need job results quickly and are able to absorb the premium cost associated with such a service, while others may wish to wait for results, as long as the job is executed within a specific time frame.

Rajkumar Buyya gave a tutorial session at the 2005 International MultiConference in Computer Science and Computer Engineering, "Grid Computing: Making the Global Cyberinfrastructure for eScience and eBusiness a Reality." He outlined many challenges of implementing a utility grid, one of the most important being how to map jobs to resources to meet QoS requirements [Buyya05]. Buyya used Figure 2 to illustrate Gridbus and its associated technologies. It shows how the GRB fits into the User-Level Middleware layer of the grid architecture.

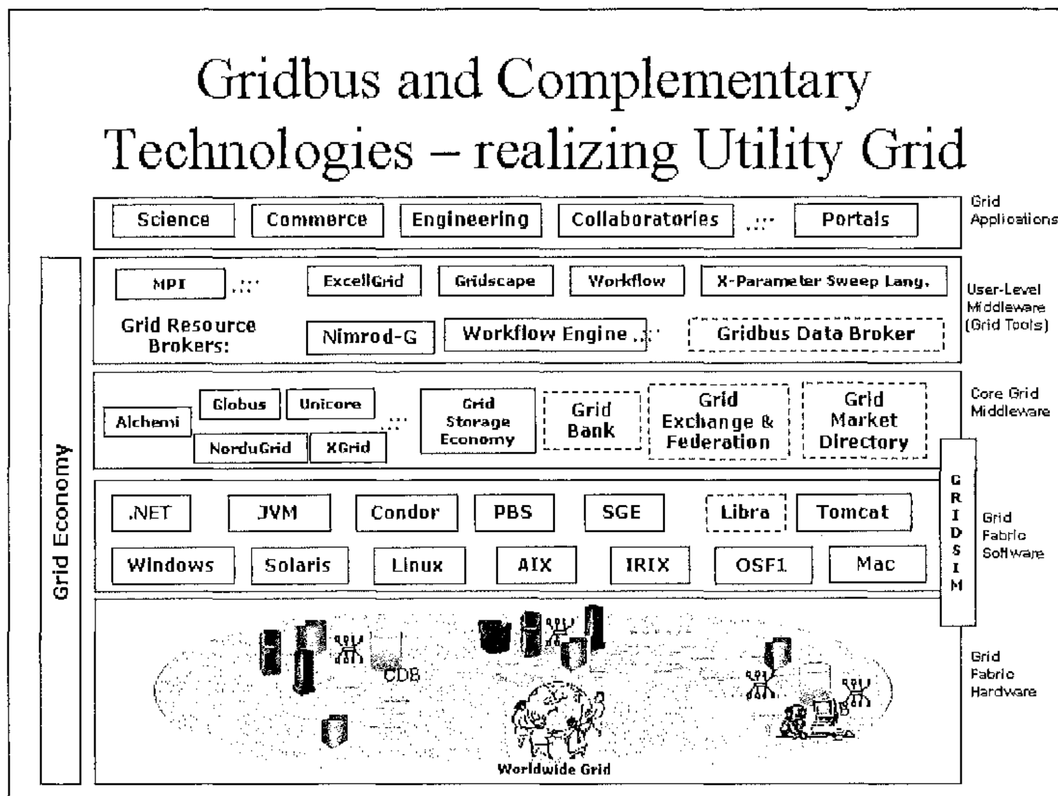


Figure 2: The Grid Resource Broker is a Part
of the User-Level Middleware [Buyya05]

The GRB communicates with the Core Grid Middleware services to find out what resources are available, how much they cost, and the processing offered by each. It uses this information to find the best solution for the jobs being submitted at the application level. The submitted jobs may have QoS requirements that have to be met, such as ensuring a job is completed in a certain time frame and ensuring it will cost no more than a certain amount. These requirements must be met for the solution to be valid and in the domain of possible solutions.

Chapter 2

SURVEY OF RELATED WORK

2.1 The Resource Allocation Problem: How to Optimize?

Daniel Menascé and Emiliano Casalicchio published an article, "QoS in Grid Computing" [Menascé04], in which they discuss a mathematical model for optimizing the selection of services and service providers to obtain a solution within the bounds of the global Service Level Agreement (SLA). The authors' mathematical model includes provisions to take into account optimization problems that have cost and time constraints. This mathematical framework is the basis for this thesis and for the grid optimization problem being explored.

Menascé and Casalicchio have written several papers discussing the QoS issue associated with grid computing resource allocation, where global SLAs must depend on local SLAs [Menascé04, Menascé04A, Menascé04B, Menascé04C]. In these papers, the authors talk at length about the factors involved when trying to select services from a service provider, so the global SLA is satisfied with a minimum of cost [Menascé04B]. Although the cost is minimized, the execution time must also fit within the guidelines of the SLA.

The main factors are as follows: the job requires NC millions of CPU cycles to finish, it has to be finished in at most T_{max} time units, and it must not cost more than C_{max} dollars

to run [Menascé04B]. For this optimization to take place, the Grid Resource Broker requires *a priori* knowledge of each resource available, along with its speed and cost. Once all of the available resources are known, the chosen algorithm must produce a near optimal solution.

The following three equations, in Figure 3, show the constraints associated with resource allocation. Equation 1 shows all of the cycles needed to complete the job, *NC*, are applied to the available computing resources, *N*. Equation 2 is the constraint associated with the maximum execution time. Since all the tasks, *N*, run in parallel, the total execution time is the task taking the longest to run. Equation 3 is the cost constraint equation.

$$\sum_{i=1}^{N} NC_i = NC \qquad (1)$$

$$T = \max_{i=1}^{N} \left\{ \frac{NC_i}{s_i} \right\} \leq T_{\max} \qquad (2)$$

$$C = \sum_{i=1}^{N} \frac{NC_i}{s_i} \times c_i \leq C_{\max} \qquad (3)$$

NC = Total number of cycles needed to complete the job
N = Number of computing resources
T = Execution time
T_{max} = Maximum allowed time to finish job
s_i = Speed of the resource ***i***
C = Cost
c_i = Cost of resource ***i***
C_{max} = Maximum allowed cost of job

Figure 3: Equations for the Total Cycles, Time Constraint, and Cost Constraint [Menascé04B]

2.2 Stochastic Algorithmic Solutions

An article published in the International Journal of Network Management discussed the use of a Genetic Algorithm (GA) for allocating network resources in a competitive electronic commerce marketplace [Ye01]. The authors, Jian Ye and Symeon Papavassiliou, use the GA to find the optimal network route, when given the ability to use multiple network providers. During their experimentation they used the following values for the GA parameters:

- Population size: 71
- Crossover rate: 0.6 (or 60%)
- Mutation rate: 0.1 (or 10%)

These researchers did not initially apply any stopping conditions and the algorithm would always run until it converged on a single solution. The results of multiple runs showed the GA would find the optimal solution in approximately 130 steps, although some runs could take more than 250 or as few as 40. The authors next applied some stopping conditions. One of these conditions halts the GA when the algorithm has not made an improvement after a certain period of time. This would give a near optimal solution, but not necessarily the very best solution. The optimization of network routing does not always require the optimal solution, but it does require a quick near-optimal solution every time. The authors concluded a GA could be used effectively when tailored to the domain associated with the optimization problem. This algorithm's goal was to find the best possible solution without any constraints, such as cost or time.

Other related work has dealt with using Simulated Annealing for scheduling distributed applications on a computational grid. YarKhan and Dongarra have compared Simulated Annealing and an Ad-Hoc Greedy scheduler as the scheduling mechanisms for a ScaLAPACK LU solver on a grid [YarKhan02]. The goal of this project was to minimize execution time, without regard to any cost or time constraints. The authors concluded the Simulated Annealing scheduler generates schedules that have better estimated execution times than those generated by the Ad-Hoc greedy scheduler.

Seonho Kim and Jon B. Weissman presented a paper at the 2004 International Conference on Parallel Processing titled, "A GA-based Approach for Scheduling Decomposable Data Grid Applications." This paper compared a GA-based algorithm with algorithms based on Divisible Load Theory (DLT), Constrained DLT (CDLT), and Tasks on Data Present (TDP). As with other reviewed works, the authors were trying to minimize optimization time, but they did not use any constraints. They found their proposed GA-based approach generally out-performed the other algorithms.

One of the authorities on grid computing, Rajkumar Buyya, has written much on scheduling jobs on a computational grid. Buyya was one of the first to use the term "utility grid." This term describes the concept of grid computing being a "pay as you go" resource, much like electricity, water, or other utilities. Teaming with Ajith Abraham and Baikunth Nath, he wrote a paper on job scheduling comparing three different heuristics; Genetic Algorithm, Simulated Annealing (SA), and Tabu Search

(TS). Also included in the comparisons were approaches using GA-SA and GA-TS hybridized algorithms [Abraham00].

The authors concluded a GA-SA hybridized solution had better convergence than a standard GA implementation [Abraham00]. A GA-TS hybrid was also tested and showed improvement in efficiency when compared to a GA solution. Although the paper stated these findings, no empirical data were given to support these conclusions. This shortcoming is most likely due to the complexity of resource allocation and the way in which the solution's efficiency is altered by the many variables associated with each instance of the problem. The authors focused on minimizing the completion time of the job, so there was no mention of cost constraints associated with a utility grid.

In 2006 Buyya co-authored a paper with Jia Yu titled, "A Budget Constrained Scheduling of Workflow Applications on Utility Grids using Genetic Algorithms" [Yu06]. Once again, the researchers explored various aspects of a "pay-per-use" grid paradigm. The work compares a slightly altered GA, with a Greedy Time (GT) scheduler. The GA uses a dual fitness function evaluation, which is divided into two parts: cost-fitness and time-fitness. Another alteration of their GA is the use of Markov decision processes to improve the convergence of the GA when given a very low budget. The authors took an approach similar to ours and tested in several areas.

- Use of varying the budget (cost) as the constraint for multiple problems.
- Use of Million Instructions (MI) to represent the length of the jobs and associated sub-tasks.

- Use of Million Instructions Per Second (MIPS) to depict the processing capabilities of available resources.
- Use of multiple testing runs and averaging the results. This method is used due to the stochastic nature of the GA (They used 10 runs, but we used 100 for our averages.)

Although there were some similarities, there were also several key differences. Buyya and Yu used cost as their only constraint and chose to only minimize time. During our testing, we compared our approach against another stochastic algorithm, Simulated Annealing, and to the optimal solution. Buyya and Yu compared their GA against a Greedy Time scheduler. The outcome of their work was displayed in a series of graphs that showed how the GA outperformed the Greedy Time scheduler for execution cost and execution time.

2.3 Focus of Thesis

The focus of this thesis was the creation of algorithms based on Menascé's mathematical model shown in Figure 3 and on the evaluation of the performance of each algorithm in relation to each other, as well as to an optimal solution. There were many steps to creating, testing, and evaluating the results for each algorithm. The main steps, goals, and contributions of this thesis are outlined below.

- Genetic Algorithm
    - The mathematical model was mapped to a Genetic Algorithm. The mapping of a Genetic Algorithm onto a string (chromosome), which is the object processed by Genetic Algorithms, is completely unique to

every problem. Once the problem description was mapped to a string, the next step was to determine how to evaluate the fitness value of that string.

  - After reproduction took place the Genetic Algorithm, the next two steps were crossover and mutation. These two steps alter the value of the string, and sometimes the resulting string no longer satisfies the constraints of the model. If the new string is no longer a valid solution, then adjustments needed to be made to alter the values within the string. That required the creation of an Adjustment Operator for crossover and mutation. These Adjustment Operators were unique to this problem.

- Simulated Annealing
  - The mathematical model were then mapped to a Simulated Annealing algorithm. The mapping into a solution set for Simulated Annealing was unique, as it was with the Genetic Algorithm. The solution was then evaluated based on its energy value, which corresponds to the fitness value in a Genetic Algorithm.
- Comparison
  - Both the Genetic Algorithm and the Simulated Annealing algorithm were compared to the optimal solution. A limited set of resources was used to create a relatively small solution space, so an optimal solution might found.

- Multiple variables were adjusted in each algorithm to find the best possible combination for the Genetic Algorithm and the Simulated Annealing algorithm.

Chapter 3

GENETIC ALGORITHMS: SURVIVAL OF THE FITTEST

The Genetic Algorithm (GA) is based on the principles of natural selection and the genetic processes associated with biological organisms. Charles Darwin discussed this progression in his book, The Origin of Species by Means of Natural Selection, and Herbert Spencer used the term "survival of the fittest" in his books about evolutionary philosophy to explain how a species evolves over time. These same basic principles are mimicked in software development when creating a GA.

The GA processes solutions represented by a string of value parameters. This solution string represents a *chromosome* and each value parameter symbolizes a *gene*. Each chromosome has a corresponding *fitness value* and this value represents the degree to which this chromosome is "good." The chromosomes with the better fitness values have a greater probability of being chosen for propagation to the next generation. GAs start with a randomly generated population pool. This pool houses chromosomes used to create the subsequent generation. Through genetic evolution, the chromosomes with the better fitness values yield better offspring and eventually converge on a near optimal solution.

Finding a stopping point for the GA is another area where a decision has to be made. In our experimentation, we chose to stop when all the chromosomes converged on a single solution. Other possible stopping conditions include (a) reaching a preset limit on the

total number of iterations processed and (b) determining the fitness value of the best chromosome has changed only slightly for a number of generations [Kim04]. All of these stopping conditions could have been used separately or in combination.

Figure 4 depicts the flow of a basic GA, which starts with the randomly generated initial population pool. The population size used has a direct impact on the performance of the GA and tends to be unique to each problem. If the population size chosen is too small, the population may lose genetic diversity, causing the GA performance to decline. Some studies suggest a good guideline to use is between 30 and 200 chromosomes [Krishnakumar89]. The next steps are *reproduction*, *crossover*, and *mutation* [Goldberg89]. The algorithm stops when all the chromosome values in a population are identical and the GA has converged to a single solution.

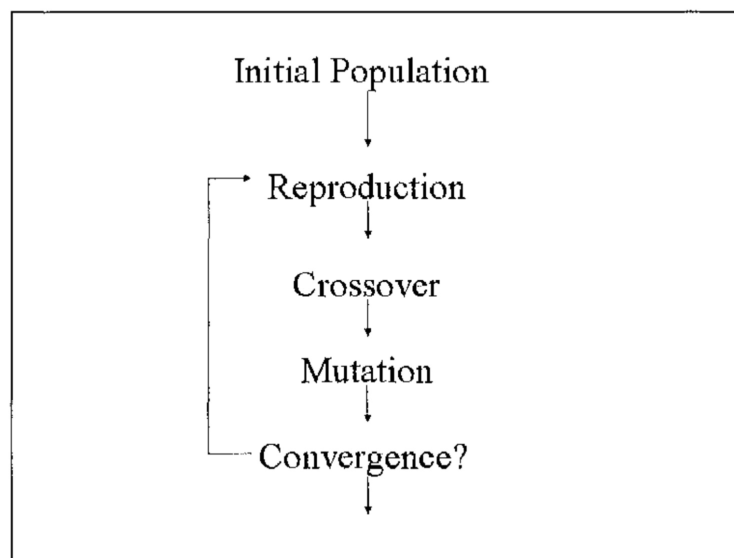
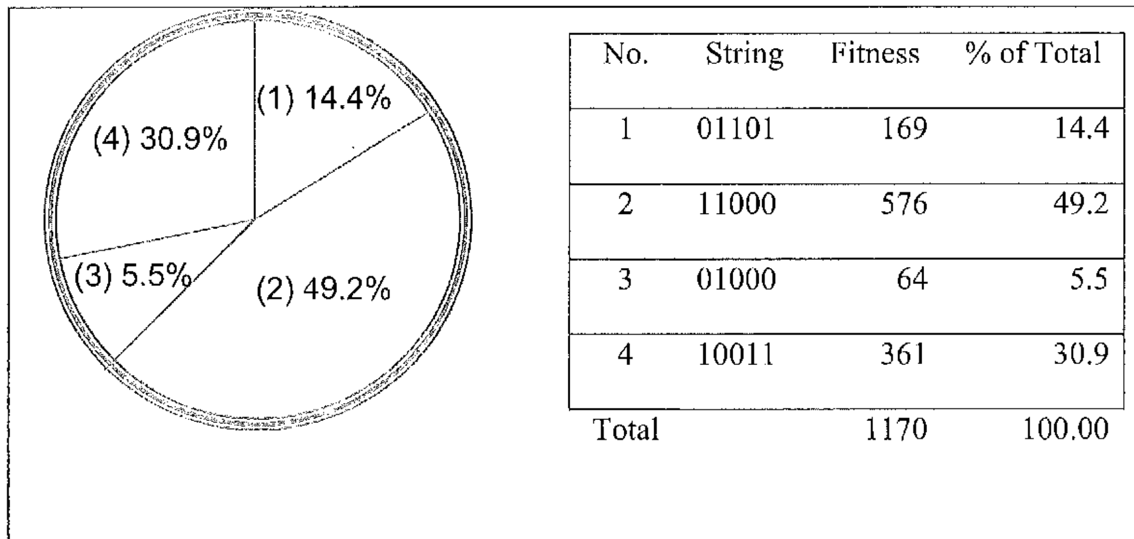


Figure 4: Genetic Algorithm Flowchart

3.1 Reproduction

During reproduction, chromosomes are evaluated and then copied based on their fitness value. The better the fitness value, the higher the probability of that chromosome contributing one or more offspring to the next generation [Goldberg89]. This process continues until a new generation is created to take the place of the current one. The reproduction selection algorithm may be implemented in many different ways. A few implementations are as follows: [Haupt04]

1. Random pairing: This process randomly chooses two parents. This method does not mimic natural selection, because the selection of mates is not uniformly random in nature.
2. Top to bottom pairing: Chromosomes are paired two at a time, beginning from the first and ending with the last. This method does not model nature, but it is easy to implement.
3. Tournament selection: This method randomly selects a small subset of chromosomes, usually between three and four. From this subset, the chromosome with the best fitness value is chosen to become a parent. The process is repeated for the next parent. This method mimics mating competition in nature.
4. Weighted roulette wheel or weighted random pairing: Using a biased roulette wheel, this method selects chromosomes using a probability weighted toward choosing those with better fitness values. In our GA, we chose the weighted roulette wheel method of selecting parent chromosomes. This method is depicted in Figure 5.



No.	String	Fitness	% of Total
1	01101	169	14.4
2	11000	576	49.2
3	01000	64	5.5
4	10011	361	30.9
Total		1170	100.00

Figure 5: Weighted Roulette Wheel Representing Four Chromosomes. Chromosome #2 has the Best Fitness Value and thus the Highest Probability of Being Chosen [Goldberg89].

3.2 Crossover

Crossover is used to create two new children chromosomes, derived from the two chosen parent chromosomes. The crossover function occurs between two parents depending on the value associated with the crossover probability, p_c, which usually falls in the range between 50% and 100% [Eiben03]. Other work [Man99] suggests that p_c has a typical value between 60% and 90%, but normal values found in nature are around 60% [Kim04, Man99, Ye01]. The setting of p_c depends on the traits of the optimization problem and is critical to the performance of the GA. There is no single value for all problems, but some guidelines have been provided [Man99]:

- For smaller populations (30), $p_c = 90\%$
- For larger populations (100), $p_c = 60\%$

As with reproduction, there are several different ways to perform a crossover operation. The method most true to biological processes, and the one used in this work, is the single point crossover. A random crossover point is selected and then the last half of each parent chromosome is swapped to yield two new children chromosome strings. This is demonstrated in Figure 6.

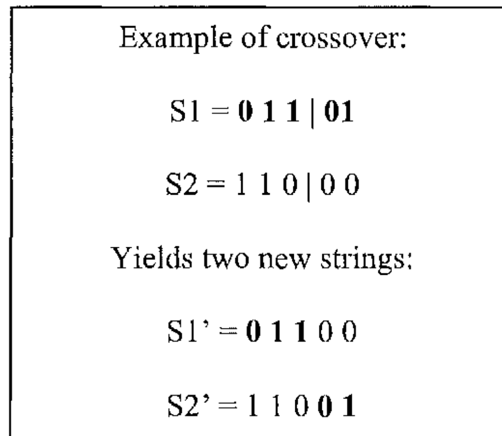


Figure 6: Crossover Example

Once a new set of chromosomes is created, rules for propagation to the next generation need to be implemented. Figure 7 displays pseudo code of the algorithm we created to determine propagation.

```
S1, S2   = Parent Chromosomes
S1', S2' = Children Chromosomes

// *** Step 1 ***
// Find the fittest child chromosome (BestChild)
If S1' Fitness Value = S2' Fitness Value
        If S1' Cost < S2' Cost
                BestChild = S1'
        Else
                BestChild = S2'
Else If S1' Fitness Value > S2' Fitness Value
        BestChild = S1'
Else
        BestChild = S2'

// *** Step 2 ***
// Find the fittest parent chromosome (BestParent)
If S1 Fitness Value = S2 Fitness Value
        If S1 Cost < S2 Cost
                BestParent = S1
        Else
                BestParent = S2
Else If S1 Fitness Value > S2 Fitness Value
        BestParent = S1
Else
        BestParent = S2

// *** Step 3 ***
// Evaluate BestChild and BestParent
If BestChild Fitness Value = BestParent Fitness Value
        If BestChild Cost < BestParent Cost
                Propagate BestChild
        Else
                Propagate Random(BestChild, BestParent)
Else If BestChild Fitness Value > BestParent Fitness Value
        Propagate BestChild
Else
        Propagate Random(BestChild, BestParent)
```

Figure 7: Genetic Algorithm Propagation Pseudo Code

3.3 Mutation

Although mutation is considered a secondary operator in a GA, it helps to keep the algorithm from converging on a single solution and failing to test new areas of the search space. This premature convergence can lead to the process of becoming stuck in a local minimum (or maximum) and not finding the global minimum (or maximum).

Figure 8 illustrates Wright's adaptive surface [Wright32]. The z plane represents the fitness value, while x and y represent values for different trait combinations. In this multimodal problem, the many peaks represent higher fitness values, where there exist many solutions better than those neighboring. Each of the smaller peaks is known as a local maximum and the highest overall peak is known as the global maximum, which is the optimum solution. This outcome differs from a unimodal problem, where there would only be a single peak that would be the optimum solution.

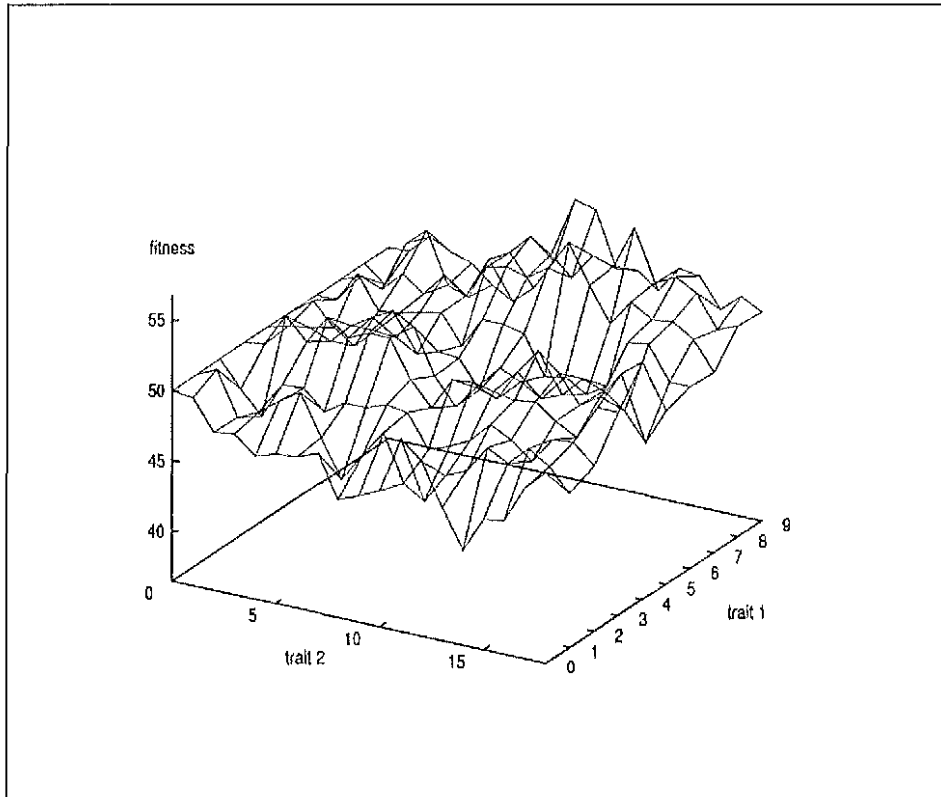


Figure 8: Wright's Adaptive Surface [Wright32]

To help avoid getting stuck in searching around a local maximum, mutation occurs using a specified probability, p_m. The probability of mutation in nature is normally low, usually on the order of one mutation per thousand position transfers [Goldberg89]. Generally, a value somewhere between 1% and 10% is used for p_m. Other work has suggested a much higher probability of mutation, around 50% [Kim04], which does not closely follow the rate represented in nature. As with the value of crossover probability and the initial number of generations, the mutation probability must be chosen with respect to the domain of the optimization problem.

Once the mutation probability is selected, the next step is to determine how to mutate the chromosome. As with the other variables associated with a GA, this process is also unique to the optimization problem. An example would be chromosomes composed of binary digits for genes. One common form of mutation would be to invert a randomly selected gene value. Figure 9 depicts three different variations of the mutation operation, Bit Flip Mutation, Swap Mutation, and Inverse Mutation.

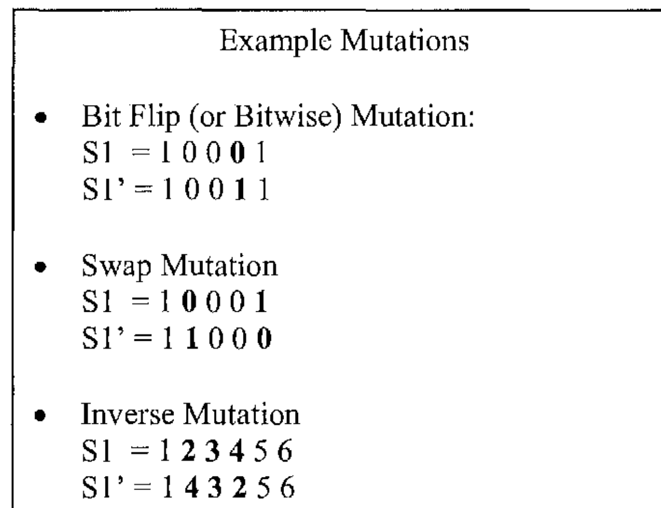


Figure 9: Examples of Mutation Methods

Chapter 4

SIMULATED ANNEALING: COOLING HOT METAL

The physical process of slowly cooling a material until it has a strong crystalline structure is known as "annealing" in metallurgy. The material is heated, giving the atoms lots of energy, and then it is slowly cooled so the atoms align and leave the material with little or no imperfections. In 1982 Kirkpatrick used the term "Simulated Annealing" (SA) to describe how to use a virtual physical process to search out solutions to optimization problems [Kirpatrick83].

Unlike a Genetic Algorithm, which maintains a pool of candidate solutions, Simulated Annealing only evaluates one candidate at a time. SA starts with a random solution and then perturbs that solution slightly, creating a new solution for comparison purposes. Since this new solution is only slightly perturbed, it is considered a neighbor and is located near the first solution in the solution space. If the new solution has a better energy value, meaning a better solution, then it is kept. If the newly created solution does not have a better energy value, then it is accepted solely on the basis of probability. The probability function is displayed in Figure 10.

$P = e^{(-\delta E/T)}$

Where,

δE = Change in energy, E, between two solutions

T = Current temperature

Figure 10: Probability Function

A random number, *r*, is generated with a value between 0 and 1. This number is then compared to the probability, *P*, and is kept if it is less than *r*. Initially the new solutions are selected, but as the temperature *T* is reduced, so is the probability of accepting solutions with worse energy values. However, as with mutations in the Genetic Algorithm, periodically accepting an inferior solution is necessary to avoid becoming trapped in a local maximum without searching other areas of the solution space.

The flowchart of a standard Simulated Annealing algorithm is shown in Figure 11.

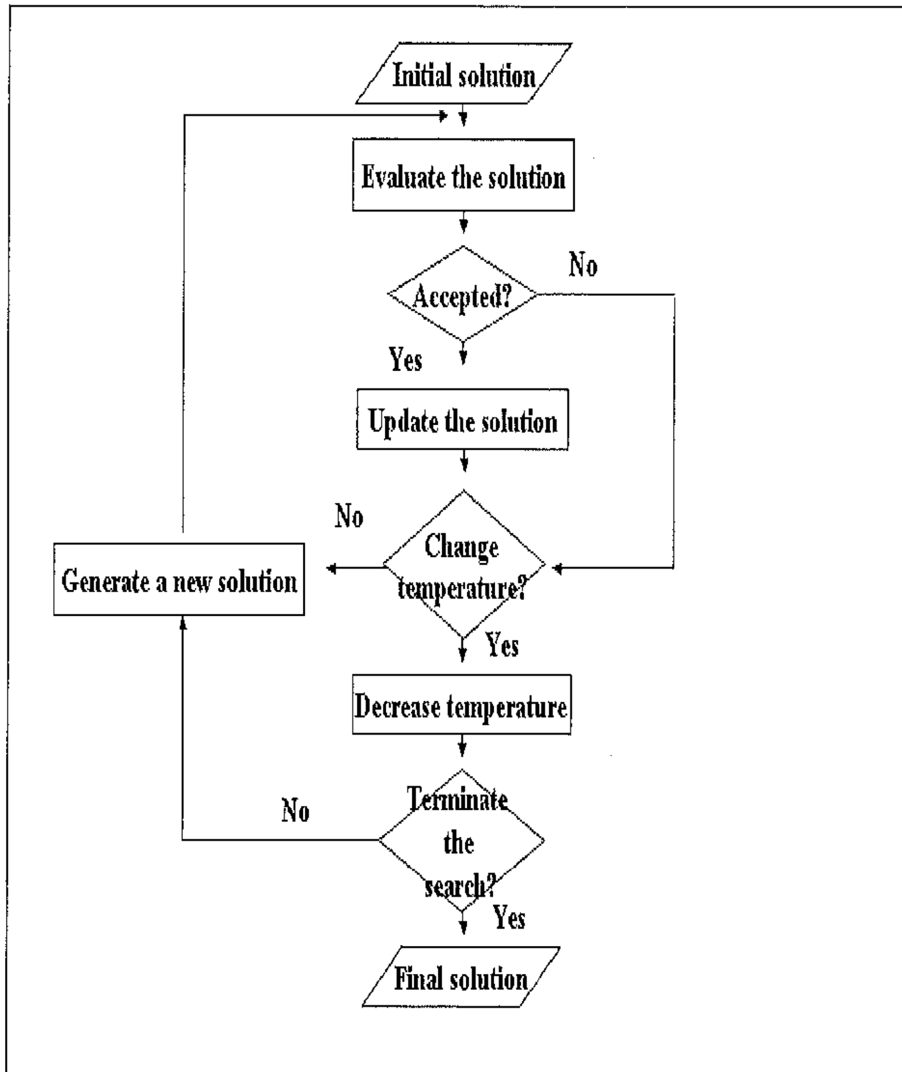


Figure 11: Flowchart of a Standard Simulated Annealing Algorithm [Pham00]

When an SA algorithm is implemented, many decisions need to be made. As with the Genetic Algorithm, the solution representation is unique to every problem. Also unique is the evaluation function, which begets an energy value. The energy value is analogous to the fitness value found in the Genetic Algorithm. This value represents the "goodness" of the solution. As the algorithm iterates, the solution must be perturbed slightly to create the new comparison solution. This perturbation is also unique to each problem and is usually adjusted through experimentation to determine how much of a perturbation is needed to achieve the desired results. The main algorithm functionality issues that must be solved are listed below:

- How to represent the solution
- How to determine the energy value, E
- How to perturb the solution in order to create a neighbor solution
- How to construct a temperature function, $T(t)$, to determine how the temperature is to be changed
- How to determine the stopping criterion to terminate the algorithm

Other choices that need to be made deal more specifically with the cooling schedule [Eglese90]. Each of the variables associated with the algorithm functionality has to be adjusted, usually through experimentation, to obtain the best possible results for the given solution domain. Although the best possible solution is desired, this solution must be arrived at in a timely manner and meet both the time and cost constraints of the problem. Some of these critical choices that deal with the cooling schedule are listed below:

- The initial value of the temperature, T
- The number of iterations, $N(t)$, to be performed at each temperature
- The value used to update to a new, lower temperature at each level, α
- The final temperature value

Cooling schedules can vary, but most remain close to those based on the physical annealing process. Kirkpatrick *et al.* started with an initial temperature high enough to ensure most of the initial solutions are accepted [Kirkpatrick83]. This approach would simulate the heating of the material until it is a liquid and all of the atoms are moving around rapidly. A temperature function is then used to decrement the current temperature in small amounts, $T(t + 1) = \alpha T(t)$. In this equation, α is a constant that usually has a value between 0.80 and 0.99 [Eglese90]. At each temperature level, a number of iterations are performed, $N(t)$. The number of iterations could be deduced by several methods, but one of the simplest is to set the value in proportion to the size of the solution space. The stopping criterion is usually that the new solution has not been altered for a specified number of temperature changes. This condition is analogous to a physical frozen state [Eglese90]. More cooling schedules are available, but they tend to stray from the original physical analogy on which the algorithm was based.

Chapter 5

TESTING AND EVALUATION OF THE STOCHASTIC ALGORITHMS

To evaluate the results of our GA and SA algorithms, we had to establish a baseline, which meant an optimal solution had to be found for a given problem with a defined set of constraints. Finding the optimal solution requires an exhaustive search and can only be achieved for a small solution space. As a result, the number of available resources would have to be limited for our optimization problem. Through some informal testing, we found five available resources would give us a manageable solution space over which to perform an exhaustive search.

All of the testing for this research was performed on a workstation with an Intel Zeon 3-GHz CPU and 2 GB of RAM. To find the optimal solution for five available resources, the run completed over 1.9 billion comparisons and took approximately 112 hours. To run comparisons using more available resources, we adapted our algorithms to solve for 10 resources. Although it would be impossible for us to run an exhaustive search for such a large search space, we ran the GA and SA algorithms several hundred times using 10 available resources. From these runs, we found both algorithms consistently derived the same near-optimal solution. We used this solution as the baseline for our testing with 10 available resources.

There are many different variables associated with the implementation of a GA or SA algorithm. The value of each variable can be adjusted to alter the performance of the algorithm, depending on the parameters associated with the problem itself. With such a large amount of changeability associated with each of the algorithm's variables, the creation of true empirical data is very difficult. Other researchers in the area of job scheduling on a grid usually make broad statements in their conclusions when comparing different algorithms. The conclusion typically states one algorithm has better convergence over another or one improves the efficiency when compared to a similar algorithm [Abraham00].

5.1 The Optimal Solution

To find the true optimal solution against which to compare the GA and SA performance would require an exhaustive search through the entire solution space. We ran an exhaustive search to find the optimal solution for five available resources, offering five different computational rates, at five different costs. The exhaustive search ran for over 112 hours and completed over 1.92 billion comparisons to arrive at the optimal solution for each of the test scenarios. Considering the parallelism of grid computing, the total execution time of a specific job would be the maximum execution time among the set of sub-tasks.

To fully exercise each algorithm, we created three different available resource scenarios. We then ran two different problems on each scenario, varying the cost constraint. We used the values listed in Figure 12 for our scenarios.

Available Resources	Speed (Millions cycles/second)	Cost ($/second)
R1(Scenario-1)	1	$ 1
R2(Scenario-1)	2	$ 3
R3(Scenario-1)	3	$ 5
R4(Scenario-1)	4	$ 6
R5(Scenario-1)	5	$ 7
R1(Scenario-2)	1	$ 1
R2(Scenario-2)	3	$ 3
R3(Scenario-2)	5	$ 5
R4(Scenario-2)	7	$ 6
R5(Scenario-2)	9	$ 7
R1(Scenario-3)	1	$ 1
R2(Scenario-3)	3	$ 2
R3(Scenario-3)	5	$ 3
R4(Scenario-3)	7	$ 4
R5(Scenario-3)	9	$ 5

Figure 12: Resource Scenarios and Their Associated Speed and Cost

We used the problem constraints listed below and ran each on the three different resource scenarios. Figures 13 and 14 reveal the optimal solution for each resource scenario, using each set of the problem constraints.

- Problem Constraints #1
    - Total cycles required to complete job: 20 million cycles
    - Maximum job execution time: 10 seconds
    - Maximum job cost: $25.50

Scenario	% job allocated to each resource R1::R2::R3::R4::R5	Time to complete job	Job Cost
S1	32%::0%::0%::0%::68%	6.4 seconds	$ 25.44
S2	4%::12%::20%::28%::36%	0.8 seconds	$ 17.60
S3	4%::12%::20%::28%::36%	0.8 seconds	$ 12.00

Figure 13: Optimal Solutions for Problem Constraints #1

- Problem Constraints #2
  - Total cycles required to complete job: 20 million cycles
  - Maximum job execution time: 10 seconds
  - Maximum job cost: $25.00

Scenario	% job allocated to each resource R1::R2::R3::R4::R5	Time to complete job	Job Cost
S1	38%::0%::0%::0%::62%	7.6 seconds	$ 24.96
S2	4%::12%::20%::28%::36%	0.8 seconds	$ 17.60
S3	4%::12%::20%::28%::36%	0.8 seconds	$ 12.00

Figure 14: Optimal Solutions for Problem Constraints #2

Once the optimal solutions were found for the five-resource problems, we needed to find the optimal solution for the ten-resource problem. To ensure the entire job could not be entirely allocated to the fastest resource, we chose constraints that would force the job to be distributed throughout the available resources. Using informal testing, we chose the following constraints:

- Total cycles required to complete job: 20 million cycles
- Maximum job execution time: 10 seconds
- Maximum job cost: $25.00

We used the values listed in Figure 15 for our ten available resources.

Available Resources	Speed (Millions cycles/second)	Cost ($/second)
R1	1	$ 1
R2	2	$ 3
R3	3	$ 5
R4	4	$ 6
R5	5	$ 7
R6	6	$ 8
R7	7	$ 9
R8	8	$ 10
R9	9	$ 11
R10	10	$ 12

Figure 15: Ten Available Resources and Their Associated Speed and Cost

Through the hundreds of test runs performed using our GA and SA algorithms, we found the optimal solution given in Figure 16:

% job allocated to each resource R1::R2::R3::R4::R5::R6::R7::R8::R9::R10	Time to complete job	Job Cost
2%::0%::0%::0%::2%::14%::17%::19%::22%::24%	0.49 seconds	$ 24.95

Figure 16: Solution for Optimization Problem Using Ten Available Resources

We used this solution as our baseline for testing the accuracy of our GA and SA solutions for ten available resources.

5.2 The Genetic Algorithm

There are many choices to be made when creating a GA solution. Each solution is unique to the particular problem of interest. We chose to mimic nature as closely as possible and concentrate on the pure GA solution, rather than trying to artificially tweak

the GA operations. GA creation involves choices that fell into two main categories: functionality of the algorithm and variable values to be used during execution.

5.2.1 Algorithmic Functionality

The first choice made dealt with the creation of the initial population to be used for the GA, so it fell into the category of algorithm functionality. A GA's initial population can be created in several ways, so we chose to begin with a randomly generated population of chromosomes that fit within our time and cost constraints. Another approach would have been to use a seeded population, where a certain percentage of the chromosomes with the best fitness values are used for the initial population, for example the top 50%. Depending on the population size and the size of the solution domain, seeding the initial population may cause the GA to converge rapidly on a local maximum. In order to better cover the solution landscape and more closely resemble nature, we did not seed our initial population.

Unique to every GA is the fitness function, which is used to determine the "goodness" or "worth" of the particular chromosome. Once programmed, the fitness function will provide a way to compare the chromosome solutions to each other. Historically, GAs have been minimizing algorithms, which are algorithms that depict the chromosomes with smaller fitness values as being more desirable. We chose to make a maximizing algorithm out of our GA and place precedence on larger fitness values. To do this, we subtracted the chromosome solution time from the time constraint for the job and assigned that result to the fitness value. This approach would ensure the chromosome

with the fastest completion time would have the highest fitness value. The side benefit is any solution resulting in a fitness value of less than zero can immediately be discounted as a valid solution, since it would take longer to run than our time constraint would allow.

The next choice in functionality was the selection of a reproduction method. In section 3.1 several of the different ways to program the reproduction method were discussed. We chose the weighted roulette wheel, sometimes called "weighted random pairing." This method selects pairs of chromosomes based on a biased roulette wheel, which uses a probability weighted toward choosing chromosomes with better fitness values. This method seems to more closely match biological reproduction.

As with the other GA functions, crossover between two parent chromosomes can be performed in several ways. Crossover can be done from a single point, from multiple points, or using a randomly generated crossover mask. Each method has strengths and weaknesses, depending on the type of optimization problem. We chose the simplest and the one inspired by biological processes, single point crossover.

The pseudo code we used to propagate chromosomes to the next generation after crossover was presented in Section 3.2 (Figure 7). To avoid rapid convergence to a local maximum, we did not always propagate the chromosomes with the best fitness values. We chose the more fit value between the two children and propagated that child who had a better fitness value than both parents. If the child chromosome did not have

a better fitness value than both parents, then the propagation was based on randomization. Allowing a less fit chromosome to sometimes propagate facilitated our coverage of more solution space and helped prevent the run from becoming stuck in a local maximum.

Mutation is considered a secondary operator in a GA, but it also helps to guard against becoming stuck in a local maximum. Usually a low occurrence operation, mutation makes a minor random change to one of the genes in the chromosome. In Section 3.3 we discussed how mutation takes place during a GA generation. We chose to reduce a random gene in the selected chromosome by 5% of the assigned job and then to add that 5% to another randomly selected gene in the chromosome.

After crossover or mutation, the chromosome may violate the job constraints and become invalid. As with a cell's DNA repair system, we created a repair mechanism [Man99]. This mechanism is also sometimes referred to as "constraint handling" [Eiben03] or an "adjustment operator." Our repair mechanisms ensure the chromosome adheres to these main restrictions:

- 100% of the job is allocated among the genes for that particular chromosome.
- The newly created chromosome is not in violation of the job constraints.

The last decision to make about the functionality of a GA is when should it stop? There are many schemes that can be used separately or in conjunction with one other. A GA can be stopped when a certain number of generations have been reached, when there is

no change in the fittest chromosome over a certain number of generations, when the percentage of change is very slight over a certain number of generations, or when all chromosomes converge to be the same. We chose to stop when all chromosomes converged, so we would be able to compare the absolute final outcome of the GA.

5.2.2 Variable Values

The main variables that compose a GA are mutation rate, crossover rate, and initial population. The rate of mutation and crossover are normally selected by trial and error, but there are some guidelines. Goldberg quotes a study of Genetic Algorithms in function optimization completed in 1975 by De Jong, which states, "…good GA performance requires the choice of a high crossover probability, a low mutation probability (inversely proportional to the population size), and a moderated population size" [Goldberg89]. In many of Goldberg's examples, he chooses to use a crossover probability of 60% and a mutation probability of 3%. Sections 3.2 and 3.4 supply further information about crossover and mutation from our literature survey.

To determine values for the three different variables in our GA, we performed a series of test runs. For the probability of mutation, we found 2–5% yielded the best results for our test problems. The crossover probability performed best between 60–80% for our tests. We needed to find a balance between excess processing, which increases calculation time, and finding a near-optimal solution.

Using our selected values for the probability of mutation and crossover, we ran similar tests in which the population size was varied. We started with a population size of 10 and increased by 10 until we had a population size of 100. We ran the GA ten times at each population size and took the average. This process was completed for both five and ten available resources.

Figure 17 shows the fitness value of the GA in relation to the population size. This graph depicts how the population size has a direct effect on the fitness value of the solution. The optimum solution has a fitness value of 3.6 for five available resources and 9.51 for ten resources. Figures 17–20 are examples of the GA's behavior using one of the resource scenarios and a single set of variable values while varying the population size.

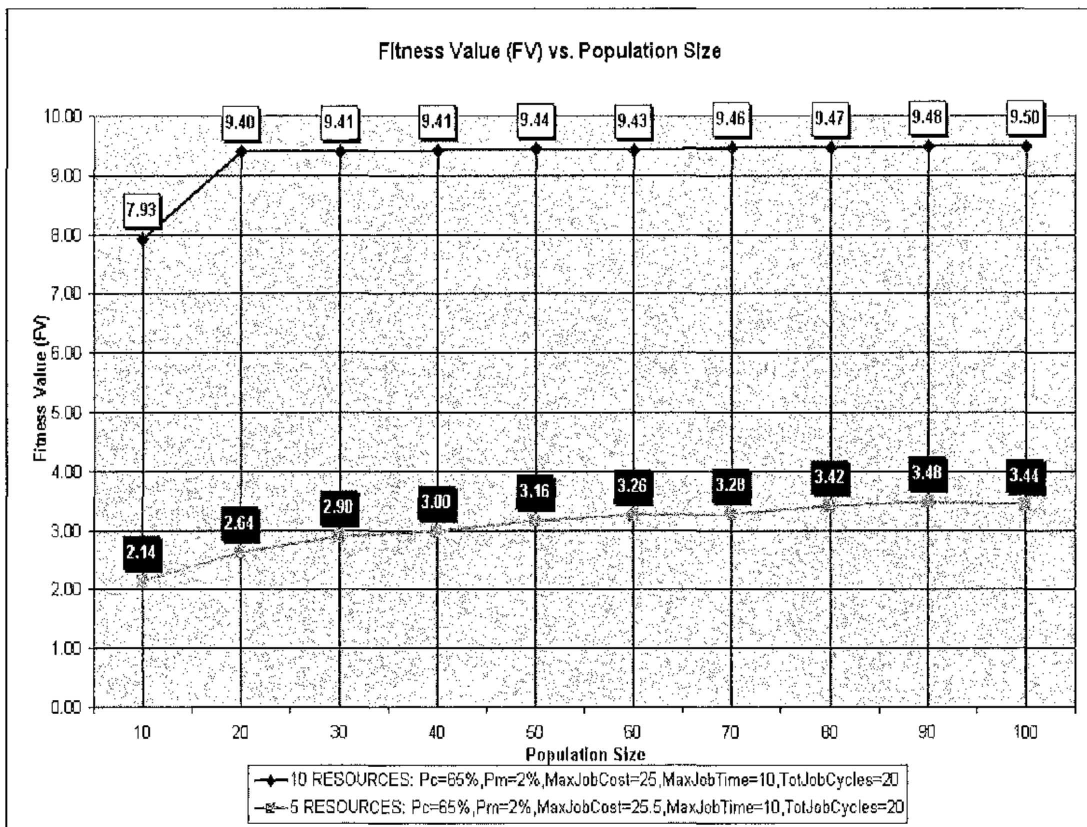


Figure 17: Sample Population Size vs. Fitness Value GA Test Run

Figure 18 shows the time it takes for the GA to run in relation to the population size. From this graph we can see that as the population size grows, so does runtime of the GA, but the runtime does not increase at the same rate. The more resources involved, the faster the rate of increase in time for subsequently larger populations. For five resources, the GA runtime increased 3650% from using an initial population of 10, compared to using a population pool of 100. The GA runtime increased 4035% for the same test when we used 10 available resources.

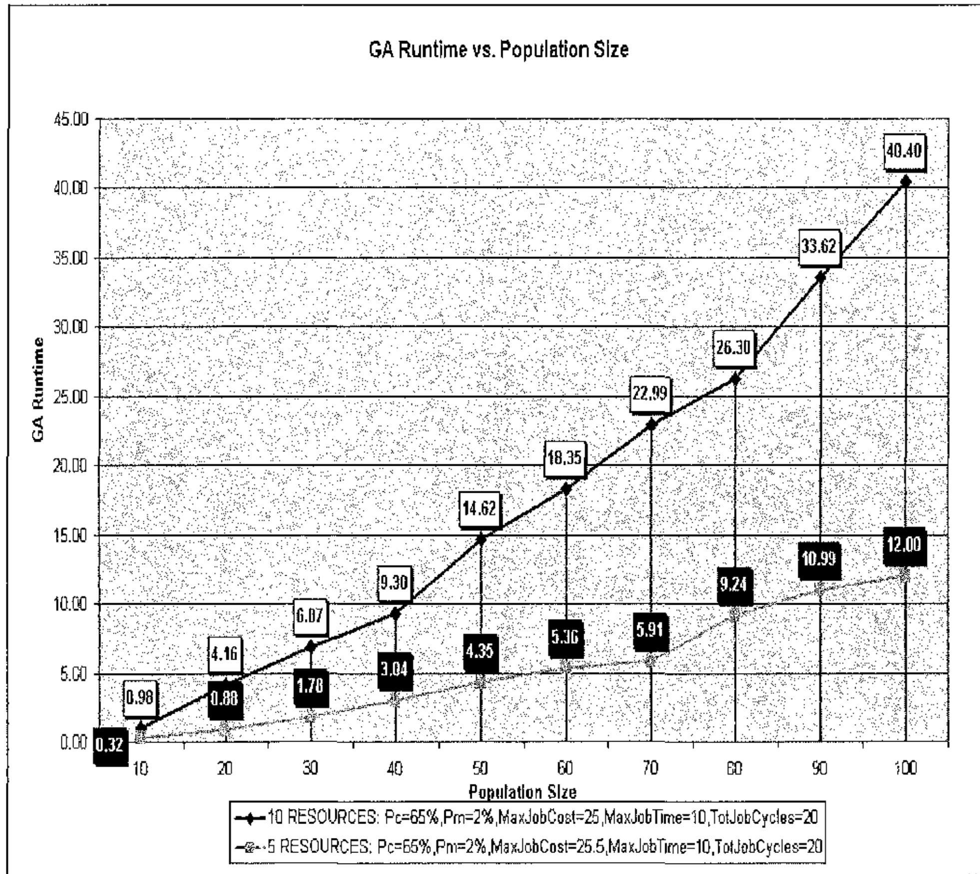


Figure 18: Sample Genetic Algorithm Runtime vs. Population Size GA Test Run

Figure 19 shows the calculated execution time of the job found by the GA in relation to the population size. Since the fitness value of a solution is computed by subtracting the GA calculated job execution time from the job time constraint, this graph is inversely proportional to the fitness value graph.

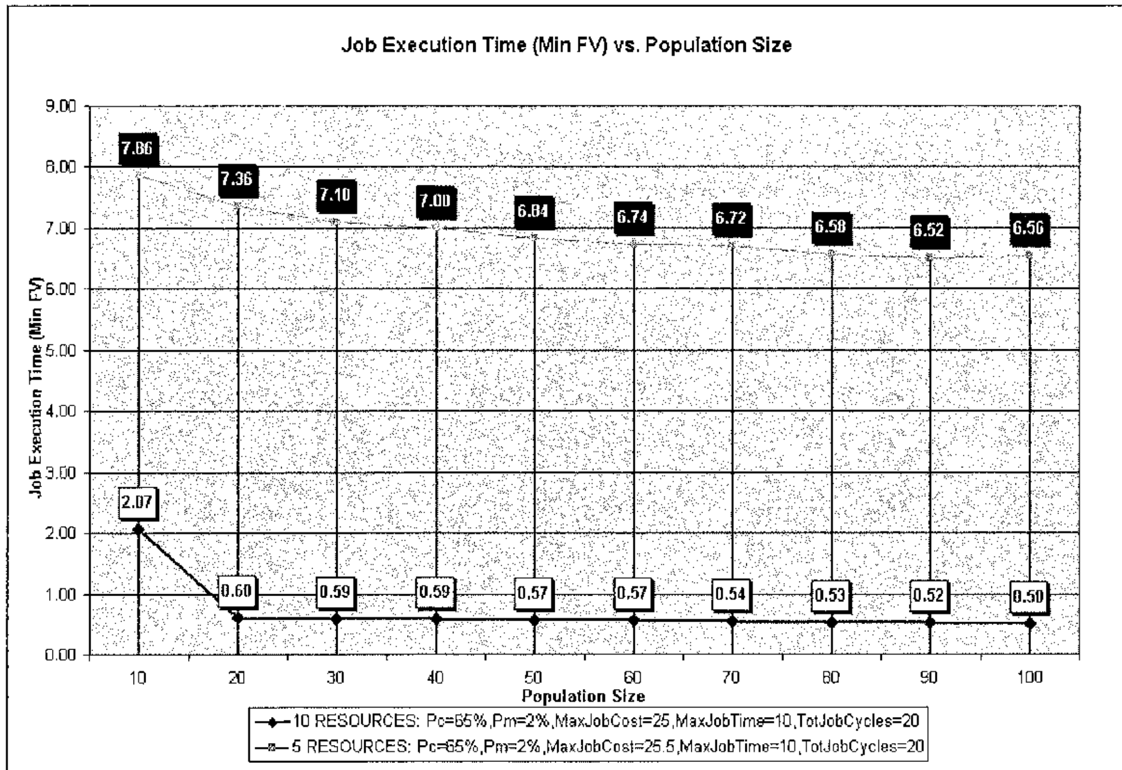


Figure 19: Sample Job Execution Time vs. Population Size GA Test Run

Figure 20 shows the percentage of optimum for the solution found by the GA in relation to the population size. This graph illustrates the accuracy of the solutions created by the GA. As the number of available resources decreases, the population size must be increased to find a near-optimal solution. We wanted our GA to create solutions that were at least 95% of the optimum. To achieve this goal for five resources would require a population of 80 or more chromosomes for this example. For ten resources, our GA was able to achieve our 95% goal with a population of 20 for this test scenario.

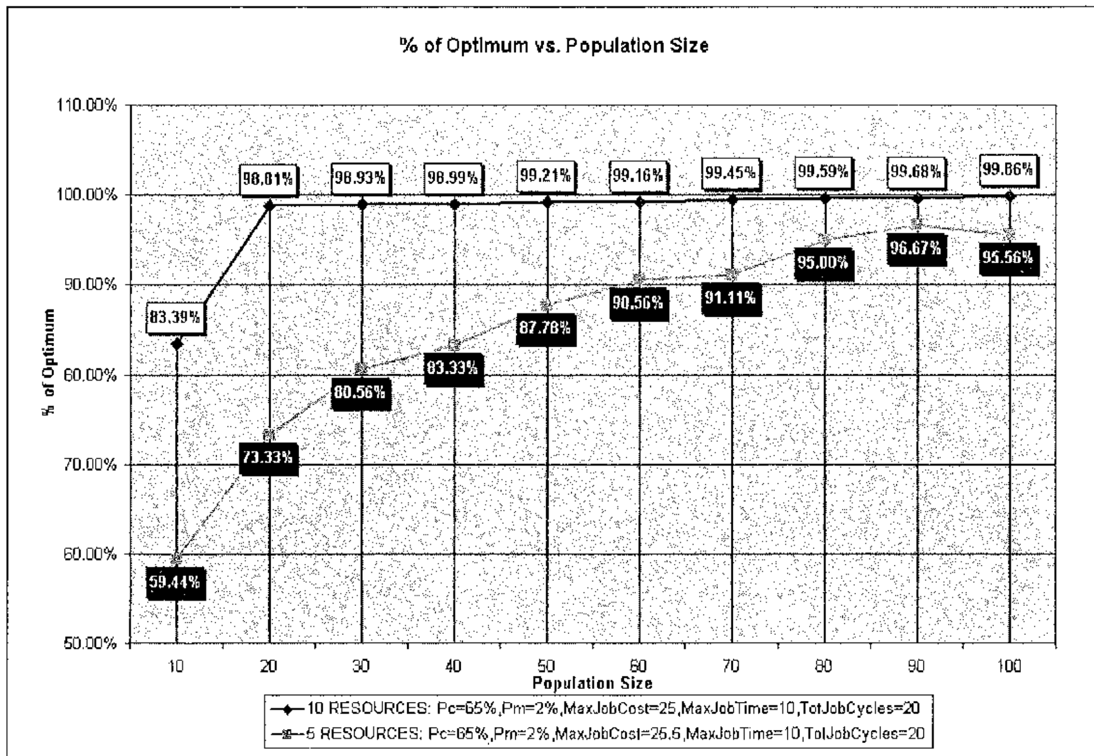


Figure 20: Sample Percentage of Optimum vs. Population Size GA Test Run

We found the value ranges listed below worked best with our solution domain and problem characteristics for the GA:

- Probability of Crossover: 60–80%
- Probability of Mutation: 2–5%
- Mutation Amount: 5%

These results are inline with values we found during our Survey of Related Work [Eiben03, Goldberg89, Kim04, Man99, Ye01].

5.3 The Simulated Annealing Algorithm

As with the creation of a GA, the SA algorithm is also a unique solution for each new problem. The SA algorithm also requires the determination of key variable values, as well as determining the functionality of several areas within the algorithm.

5.3.1 Algorithmic Functionality

The GA and SA algorithms are similar in many ways. We attempted to have the two algorithms use the same structure and functionality whenever possible to facilitate more accurate comparisons between the solutions.

Each solution in an SA algorithm has the same structure as a chromosome in a GA. The solutions created by the SA algorithm contain the number of available resources and the percentage of the job assigned to each. This structure is equivalent to the chromosome-gene structure used by our GA. We also used the same "goodness" evaluation methods used for a chromosome's fitness value as the energy value in our SA algorithm. This method subtracts the solution job execution time from the time constraint associated with that particular job. By being able to transfer the same evaluation methods and structure from our GA to our SA algorithm, we were able to better evaluate the basic algorithmic functions of each optimization method.

An SA algorithm compares a single solution with one of its neighbors. To find this neighboring solution, we tweak the current solution slightly and then make a comparison. The tweaking of the solution requires reducing the percentage assigned to

one resource and adding to another to create a new solution. To find the best percentage to use for the tweak factor, we ran a series of tests. We started with a tweak factor of 1%, ran ten tests, and then took the average. We increased the tweak factor by 1% after each series of tests until we reached 10%. Figures 21–24 are examples of the SA algorithm's behavior using one of the resource scenarios and a single set of variable values while the tweak factor is being varied.

Figure 21 shows the results of our tests for a varying tweak factor and the resultant energy value. Figure 22 shows how long it took for the SA took to find a solution.

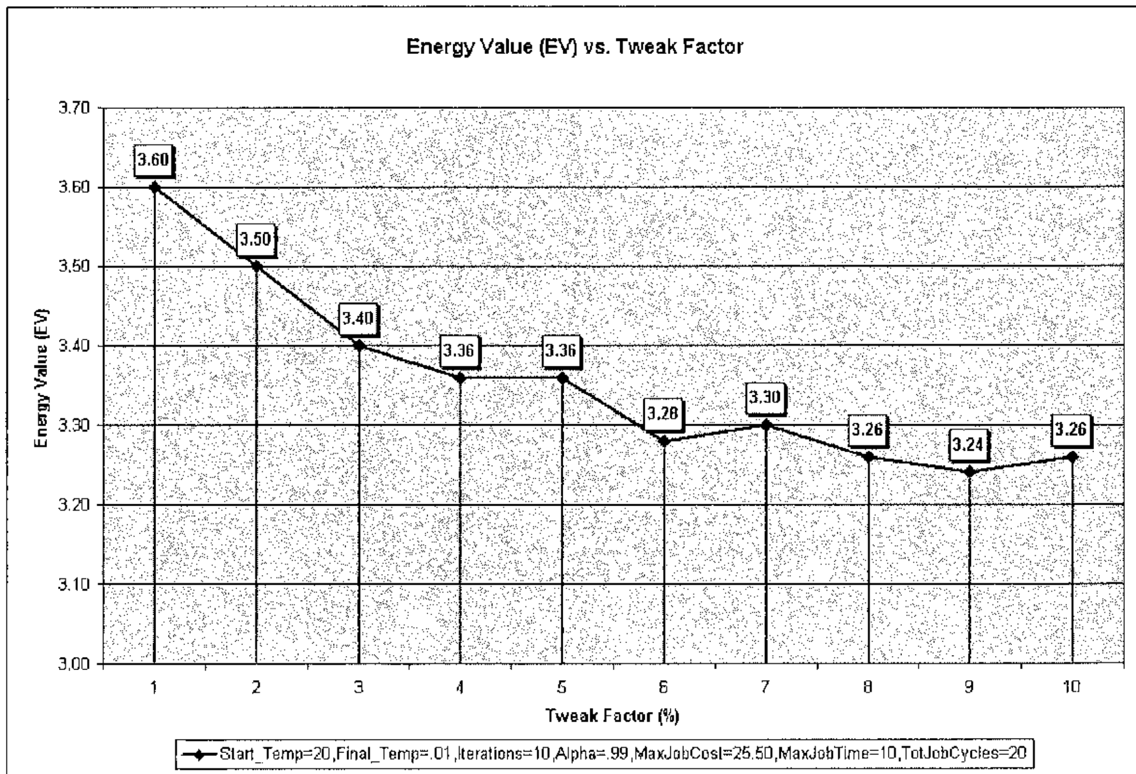


Figure 21: Sample Energy Value vs. Tweak Factor SA Test Run

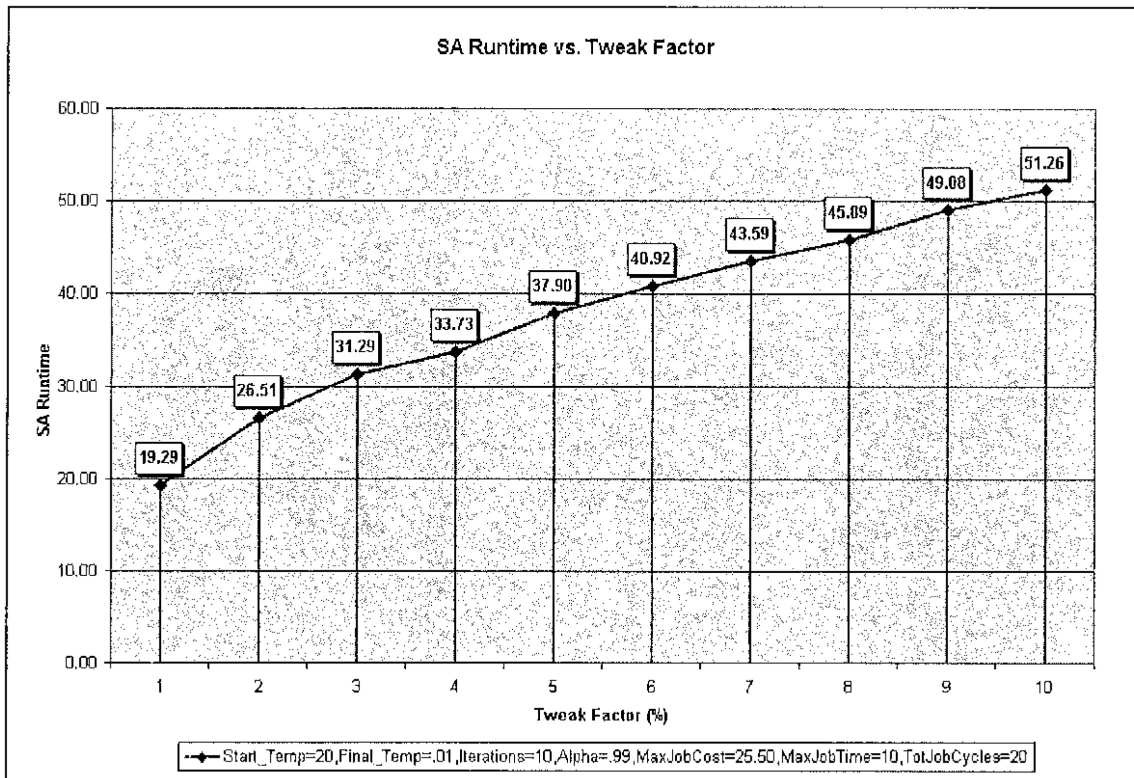


Figure 22: Sample Simulated Annealing Runtime vs. Tweak Factor Test Run

Figures 23 and 24 show the execution time of the SA created solution and the percentage of optimum for each solution, respectively.

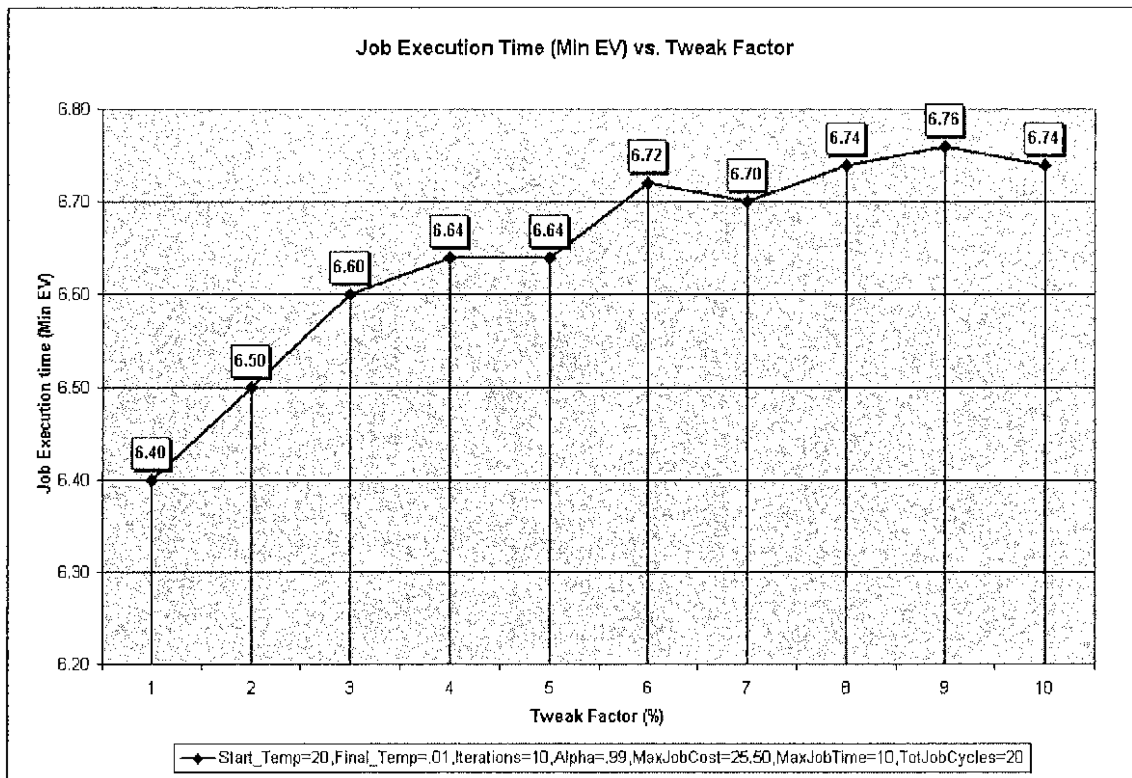


Figure 23: Sample Job Execution Time vs. Tweak Factor SA Test Run

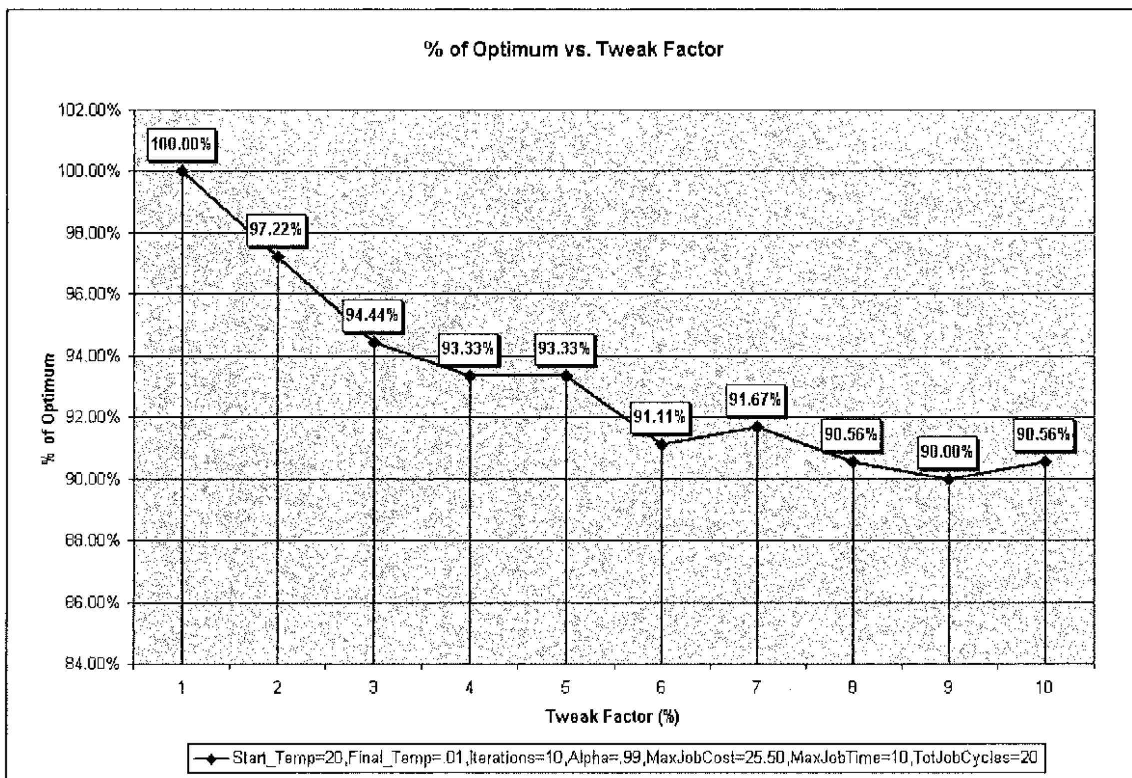


Figure 24: Sample Percentage of Optimum vs. Tweak Factor SA Test Run

Through a series of tests, we found 1% to be the best value for our tweak factor, so we used this value for the rest of our testing of the SA algorithm.

The SA algorithm performs a number of iterations at varying temperature levels. The temperature is reduced, a number of iterations are performed, and the temperature is dropped again. Decrementing the temperature to the next lower level is accomplished by multiplying the current temperature by a constant. Through our literature survey, we found this constant varies between 0.80 and 0.99 [Braun01, Eglese90].

Unless other stopping criteria are put into place, the SA algorithm will complete when the current temperature reaches zero. We chose to have the algorithm run until the current temperature reaches zero, without introducing extra measures for early termination. In this regard, as with the GA algorithm, the SA algorithm closely follows the laws of nature.

5.3.2 Variable Values

There are four main cooling schedule variables associated with an SA algorithm: the initial temperature value, the number of iterations performed at each temperature level, the α constant, and the final temperature value used to determine the stopping point. To find the best value for each of these variables, we performed a series of test runs.

The value for the initial temperature is based on the size of the solution space for each specific problem. The value must be large enough to allow the algorithm to search

other parts of the solution space and not be trapped locally. However, if the value is too large, then no better solution is derived, and the algorithm is inefficient because of long processing time. We found a value in the range of 5–30 for the initial temperature gave us the best balance between processing time and solution quality.

The number of iterations performed at each temperature level is another variable that needs to be fine tuned for each problem. As with the other variables, the number of iterations is determined by the size of the solution space. When the process first begins and has a higher temperature, the SA algorithm searches more of the solution landscape for the global optimum. As the algorithm progresses and the temperature cools, the solution search space is narrowed while it searches for the local optimum. Many researchers suggest manually performing experiments with the number of iterations to find the best values [Jones03]. To determine this value, we used the average of ten test runs at each of the following iteration values: 1, 5, 10, 15, 20, 25, 30, 35, 40, and 45.

We performed the iteration value testing for both five and ten available resources. Figure 25 shows the energy value for each iteration value. Figures 25–27 depict some of the test runs we performed with the SA algorithm using one of the resource scenarios while varying the number of iterations.

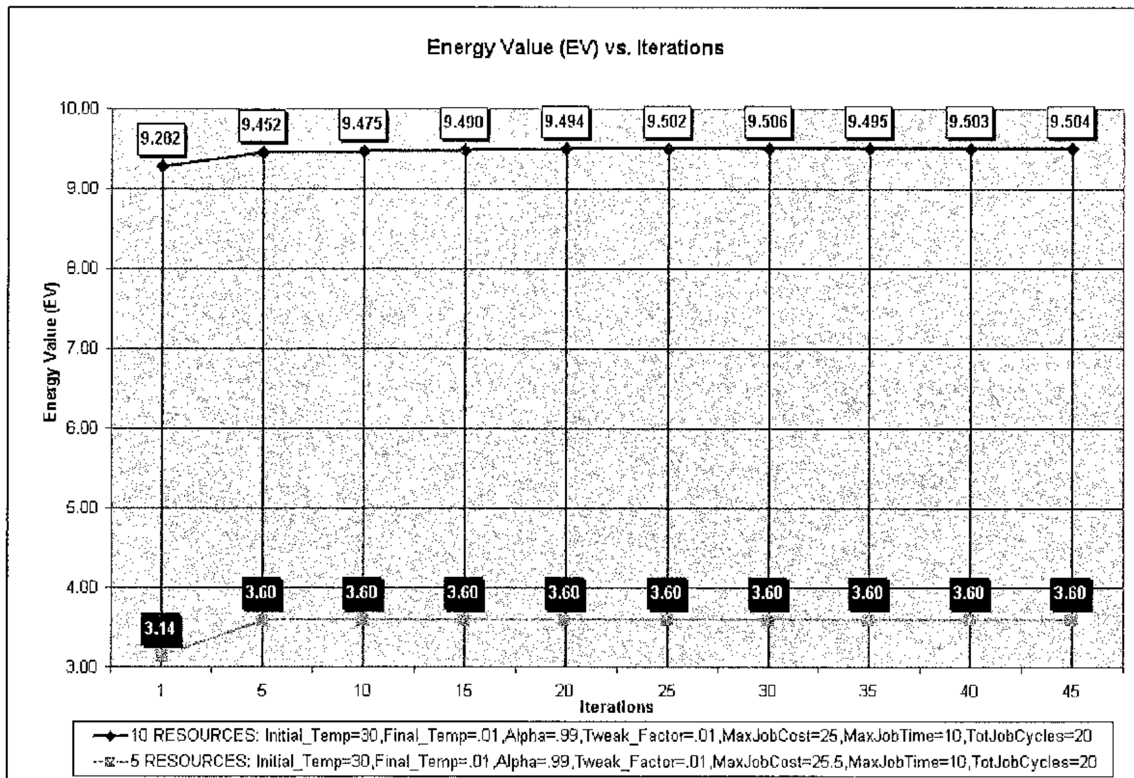


Figure 25: Sample Energy Value vs. Number of Iterations SA Test Run

Figure 26 shows SA algorithm runtime for each iteration value and Figure 27 gives the percentage of the optimum solution.

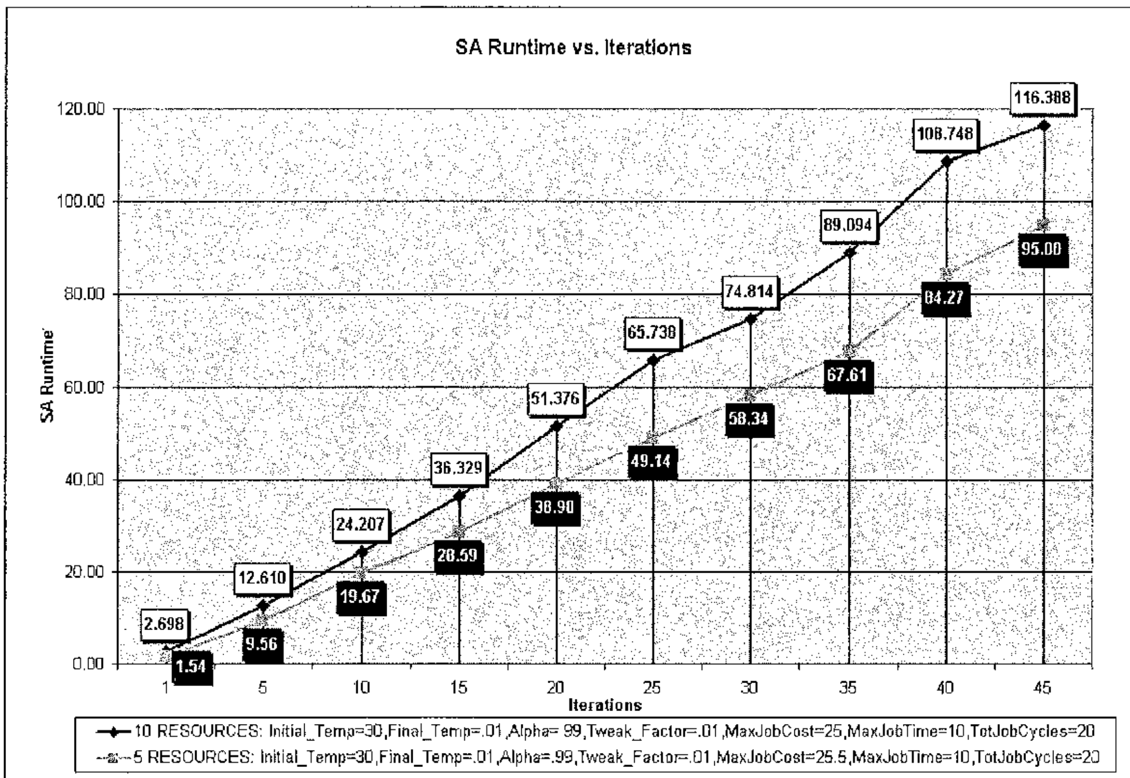


Figure 26: Sample Simulated Annealing Runtime vs. Number of Iterations Test Run

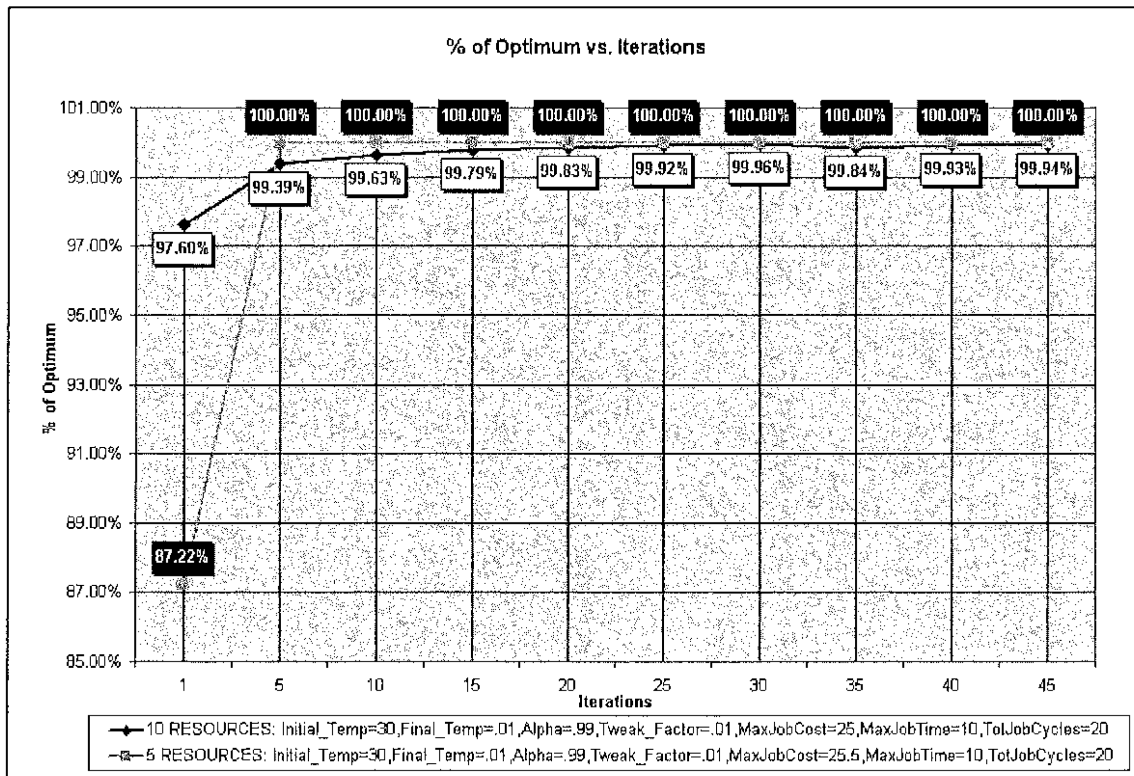


Figure 27: Sample Percentage of Optimum vs. Number of Iterations Test Run

From our testing, we found using 10 iterations at each temperature level seemed to be a good balance between the produced energy value and the time needed for the SA algorithm to find a solution.

The temperature function is used to decrement the current temperature by a small amount and produce the temperature for the next level. We did this by using the equation $T(t + 1) = \alpha T(t)$, where α represents a constant value, normally between 0.90 and 0.99. To find the best value to use for this constant, we performed a series of tests. As with our iteration testing, we took the average of ten test runs, varying the α value by 0.01, starting at 0.90 and ending at 0.99. We ran tests for five and ten available resources. Figures 28–30 show the energy value for each α value for one series of tests.

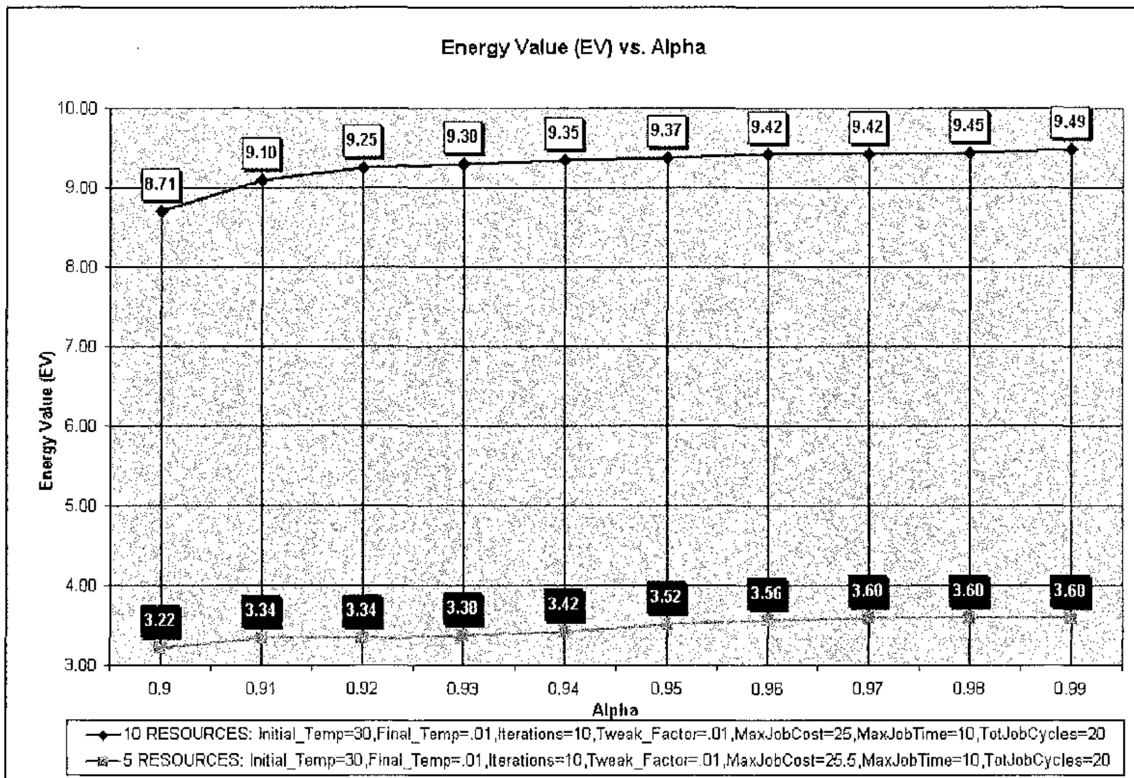


Figure 28: Sample Energy Value vs. Alpha SA Test Run

Figure 29 shows the amount of time it took for the SA algorithm to find the solution and Figure 30 gives the percentage of optimum for each value of α.

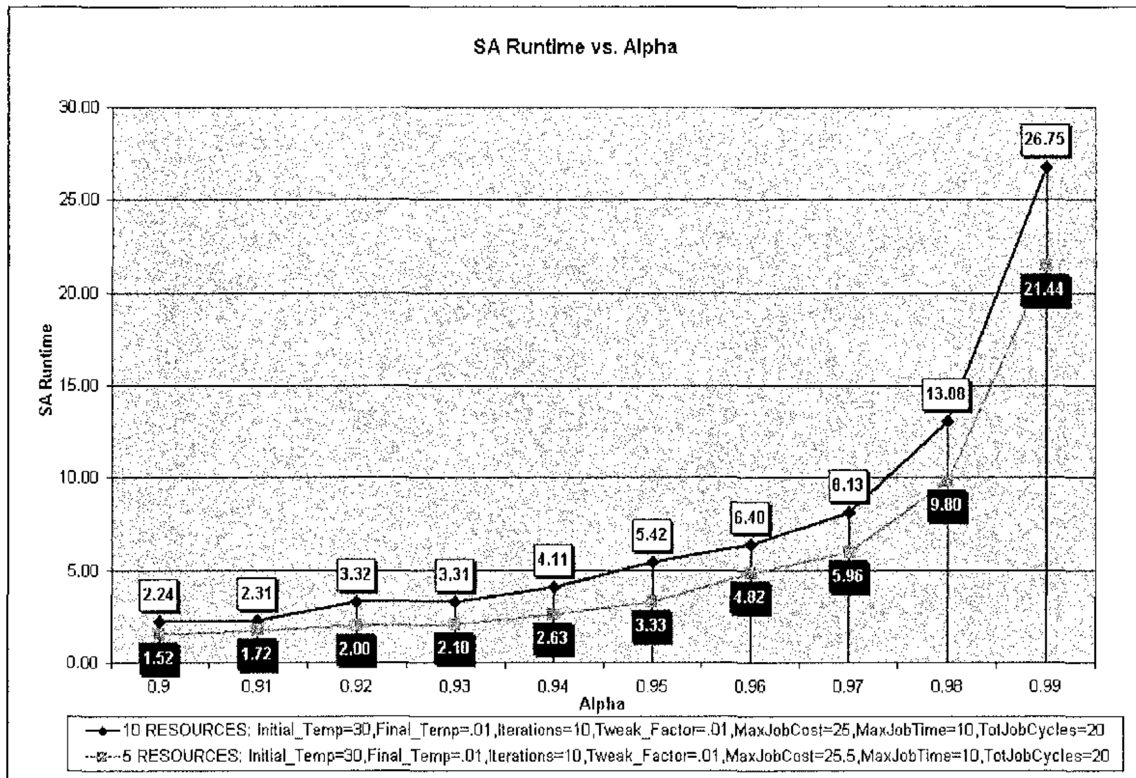


Figure 29: Sample Simulated Annealing Runtime vs. Alpha Test Run

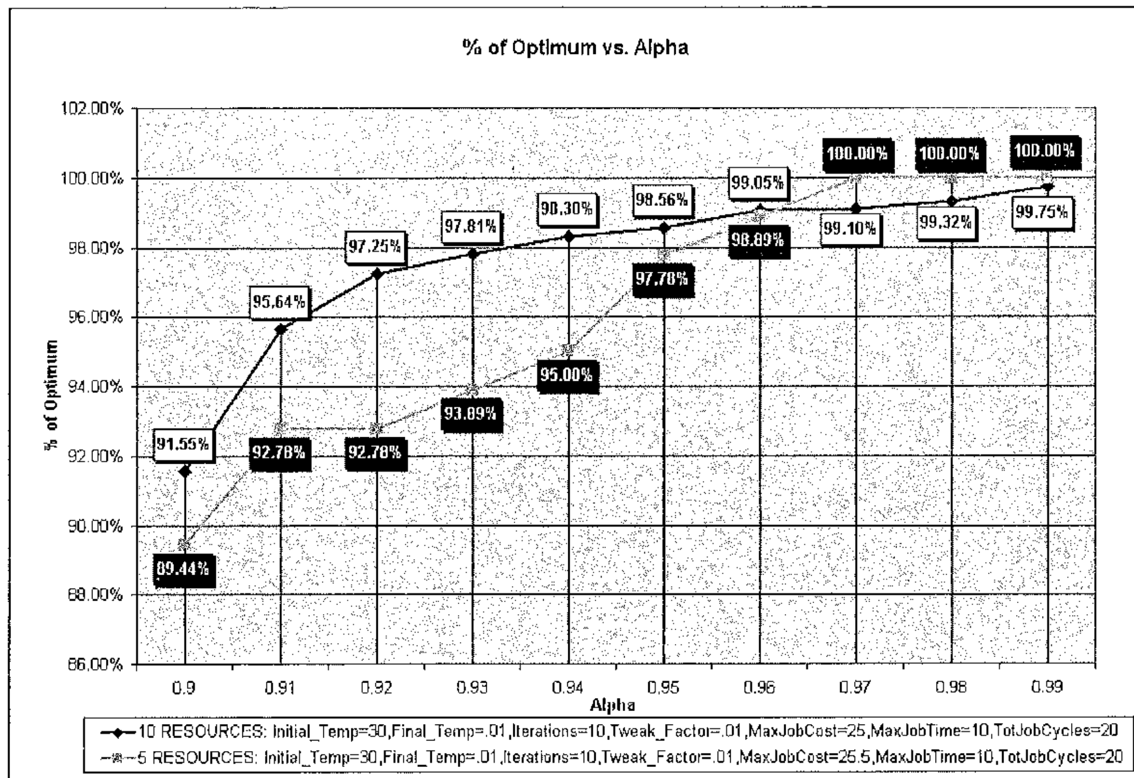


Figure 30: Sample Percentage of Optimum vs. Alpha Test Run

The final temperature value was the last of the cooling schedule variables we needed to determine. This value is the one used to stop the SA algorithm and cause the run to present the solution. We performed tests using the same format as used in the iteration and α experiments. Although the tests showed a higher value could have been used for the final temperature, we chose to use 0.01. This value would give us a more accurate solution and still keep us within the time range observed during our GA experiments. This choice was also made with regard to the fact the solution space was smaller than would be likely in a real world solution. If this were a real world problem, then our SA algorithm could run longer than would be practically useful, and the value would have to be raised to reflect the size of the solution space. According to published research, the usual value for the final temperature is near 0.5 degrees [Jones03].

The following graphs, Figures 31–33, show the energy value, SA algorithm runtime, and percentage of optimum for differing values used for the final temperature. Once again, these graphs represent one series of testing.

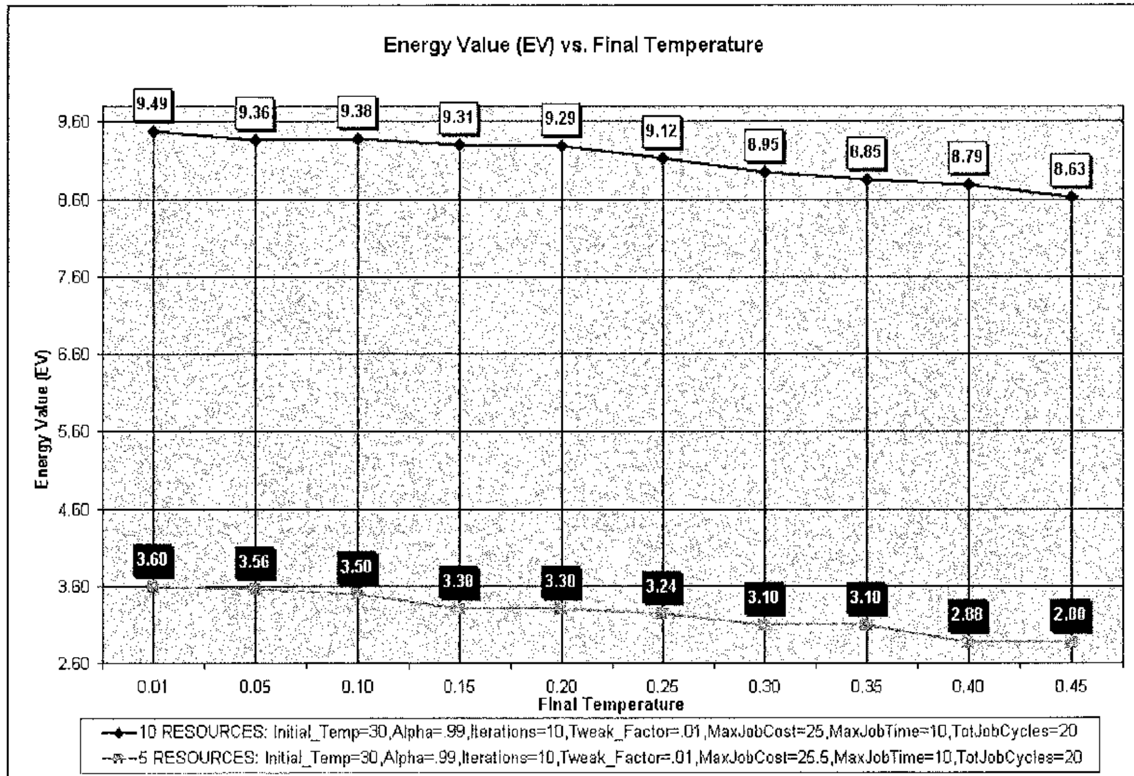


Figure 31: Sample Energy Value vs. Final Temperature SA Test Run

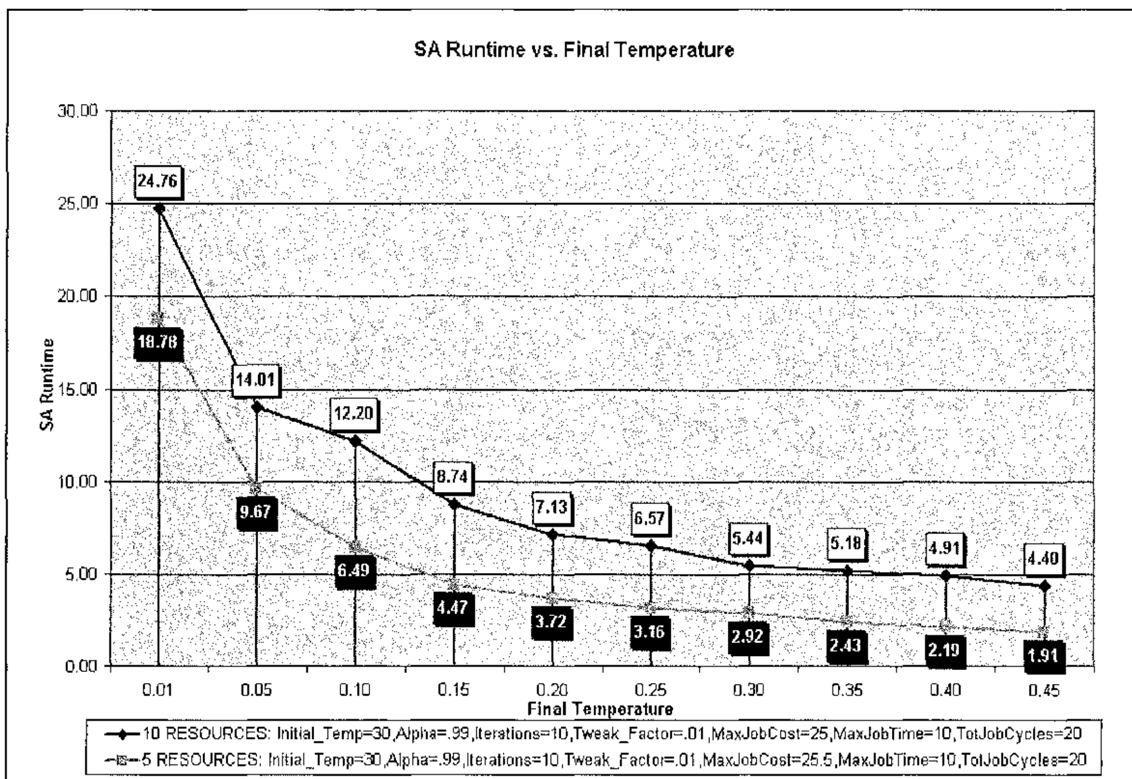


Figure 32: Sample Simulated Annealing vs. Final Temperature Test Run

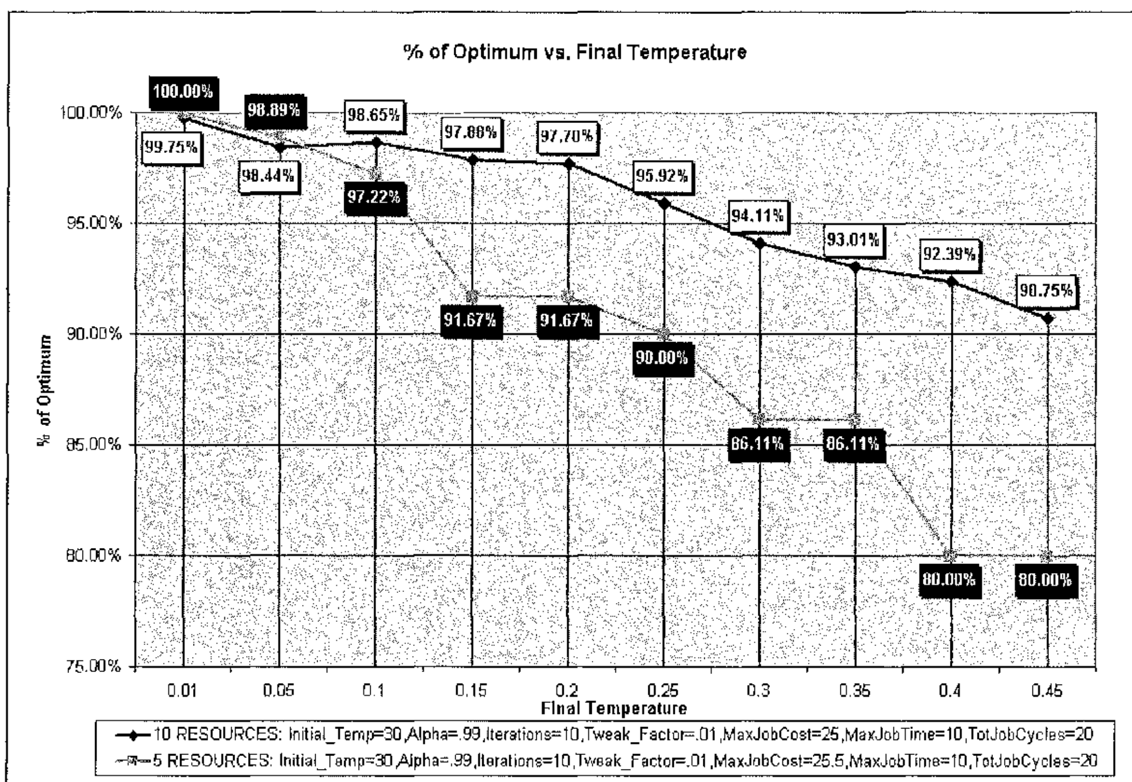


Figure 33: Sample Percentage of Optimum vs. Final Temperature SA Test Run

We found the value ranges listed below to work best with our solution domain and problem characteristics for the SA algorithm:

- Initial Temperature: 5–30
- Tweak Factor: 0.01 (1%)
- Alpha: 0.94–0.99 (94–99%)
- Iterations: 10

Chapter 6

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

The SA algorithm outperforms the GA in all of our test scenarios. When the solution domain was small, the SA algorithm achieved our near-optimal threshold much faster than the GA. We made the solution domain small by limiting the number of available resources to five. As we increased the size of the solution domain, the GA performed slightly better but was still outperformed by the SA algorithm, as Figure 34 shows.

<table>
<tr><td></td><td>GA</td><td>Average of all runs (100 each)</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>5 Resources</td><td>Max Cost</td><td>Speed/Cost</td><td>Avg Fitness Value</td><td>Avg Exec Time</td><td>Avg # of Generations</td><td>Avg Time to Find Solution</td><td>Avg % of Optimum</td><td>Optimum Solution</td><td>Optimum Fitness Value</td></tr>
<tr><td></td><td>$ 25.00</td><td>12345/13567</td><td>2.31</td><td>7.69</td><td>32.67</td><td>34.30</td><td>96.25%</td><td>38|0|0|0|62</td><td>2.4</td></tr>
<tr><td></td><td></td><td>13579/13567</td><td>9.19</td><td>0.81</td><td>44.7</td><td>8.90</td><td>99.89%</td><td>4|12|20|28|36</td><td>9.2</td></tr>
<tr><td></td><td></td><td>13579/12345</td><td>9.18</td><td>0.82</td><td>34.54</td><td>6.91</td><td>99.01%</td><td>4|12|20|28|36</td><td>9.2</td></tr>
<tr><td></td><td>$ 25.50</td><td>12345/13567</td><td>3.42</td><td>6.58</td><td>30.30</td><td>9.24</td><td>95.00%</td><td>32|0|0|0|68</td><td>3.6</td></tr>
<tr><td></td><td></td><td>13579/13567</td><td>9.19</td><td>0.81</td><td>37.34</td><td>7.27</td><td>99.90%</td><td>4|12|20|28|36</td><td>9.2</td></tr>
<tr><td></td><td></td><td>13579/12345</td><td>9.19</td><td>0.81</td><td>34.03</td><td>6.56</td><td>99.85%</td><td>4|12|20|28|36</td><td>9.2</td></tr>
<tr><td>10 Resources</td><td>Max Cost</td><td>Speed/Cost</td><td>Avg Fitness Value</td><td>Avg Exec Time</td><td>Avg # of Generations</td><td>Avg Time to Find Solution</td><td>Avg % of Optimum</td><td>Optimum Solution</td><td>Optimum Fitness Value</td></tr>
<tr><td></td><td>$ 25.00</td><td>12345678910 / 1356789101112</td><td>9.40</td><td>0.60</td><td>42.70</td><td>4.16</td><td>98.81%</td><td>2|0|0|0|2|14|17|19|22|24</td><td>9.51</td></tr>
<tr><td></td><td>SA</td><td>Average of all runs (100 each)</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>5 Resources</td><td>Max Cost</td><td>Speed/Cost</td><td>Avg Energy Value</td><td>Avg Exec Time</td><td># of Iterations</td><td>Avg Time to Find Solution</td><td>Avg % of Optimum</td><td>Optimum Solution</td><td>Optimum Energy Value</td></tr>
<tr><td></td><td>$ 25.00</td><td>12345/13567</td><td>2.32</td><td>7.68</td><td>10</td><td>12.23</td><td>96.75%</td><td>38|0|0|0|62</td><td>2.4</td></tr>
<tr><td></td><td></td><td>13579/13567</td><td>9.13</td><td>0.87</td><td>10</td><td>0.32</td><td>99.25%</td><td>4|12|20|28|36</td><td>9.2</td></tr>
<tr><td></td><td></td><td>13579/12345</td><td>9.13</td><td>0.87</td><td>10</td><td>0.31</td><td>99.19%</td><td>4|12|20|28|36</td><td>9.2</td></tr>
<tr><td></td><td>$ 25.50</td><td>12345/13567</td><td>3.46</td><td>6.54</td><td>10</td><td>2.58</td><td>96.00%</td><td>32|0|0|0|68</td><td>3.6</td></tr>
<tr><td></td><td></td><td>13579/13567</td><td>9.14</td><td>0.86</td><td>10</td><td>0.31</td><td>99.30%</td><td>4|12|20|28|36</td><td>9.2</td></tr>
<tr><td></td><td></td><td>13579/12345</td><td>9.12</td><td>0.88</td><td>10</td><td>0.31</td><td>99.10%</td><td>4|12|20|28|36</td><td>9.2</td></tr>
<tr><td>10 Resources</td><td>Max Cost</td><td>Speed/Cost</td><td>Avg Energy Value</td><td>Avg Exec Time</td><td># of Iterations</td><td>Avg Time to Find Solution</td><td>Avg % of Optimum</td><td>Optimum Solution</td><td>Optimum Energy Value</td></tr>
<tr><td></td><td>$ 25.00</td><td>12345678910 / 1356789101112</td><td>9.37</td><td>0.63</td><td>10.00</td><td>2.31</td><td>98.56%</td><td>2|0|0|0|2|14|17|19|22|24</td><td>9.51</td></tr>
</table>

Figure 34: Results and Comparison of the GA and SA Test Runs.

Although both algorithms could be used to find a near-optimal solution, the SA algorithm is the better choice for a Grid Resource Broker and any dual constraint optimization problem. The SA algorithm reached our near-optimal threshold much quicker than our GA based solution. The GA had slightly better performance as the number of available resources increased, but it still lagged behind the SA algorithm. With the SA algorithm's variables set to their optimal values, the SA algorithm performed very well. To identify the optimal values for the variables requires trial and error, or even possibly the use of some other kind of optimization algorithm or program. To implement a stochastic algorithm-based GRB effectively requires *a priori* knowledge of the size of the solution domain, the attributes of the job to run on the grid, and the characteristics of the available resources.

6.2 Future Work

The current resource allocation solutions do not account for the duties of a GRB communicating with a Utility Grid, which would implement a "pay-per-use" model. This model would offer many desirable features not available in current grid offerings, but it would also introduce a host of problems to overcome before implementation.

A real world implementation of a GRB would have to take into account many more variables than have been addressed in this thesis. Resource failure during execution would have to be addressed, along with how to recognize new resources during job execution. Another issue would he error handling associated with the inability to find a valid solution. This issue could be handled by providing an alternative solution, or

several best effort scenarios. Also to be resolved is the possibility of intra task communication within a single job and dynamically collocating these tasks to minimize latency.

Further evaluation is needed on other stochastic and non-stochastic algorithms to determine which ones performed better under which scenarios. These algorithms could then be hybridized and have their performance compared to the "pure" versions of each algorithm. Also, with more testing resource availability, these solutions could be tested on larger solution domains, which would give a better picture of their overall performance potential.

REFERENCES

[Abraham00]
Abraham, A., R. Buyya, and B. Nath, "Nature's Heuristics for Scheduling Jobs on Computation Grids," Proceedings of the 8th IEEE International Conference on Advanced Computing and Communications (2000), pp. 45-52.

[Braun01]
Braun, T.D., et al., "A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems," Journal of Parallel and Distributed Computing 61 (2001), pp. 810-837.

[Buyya02]
Buyya, R., "Economic-based Distributed Resource Management and Scheduling for Grid Computing," Ph.D. dissertation, Department of Computer Science and Software Engineering, Monash University, Melbourne, Australia, April 12, 2002.

[Buyya05]
Buyya, R., "Grid Computing: Making the Global Cyberinfrastructure for eScience and eBusiness a Reality", International MultiConference in Computer Science and Computer Engineering, Las Vegas, June 28, 2005.

[Eiben03]
Eiben, A.E. and J.E. Smith, Introduction to Evolutionary Computing, Springer-Verlag Berlin Heidelberg, Germany, 2003.

[Eglese90]
Eglese, R.W., "Simulated Annealing: A Tool for Operational Research," European Journal of Operational Research 46, 3 (1990), pp. 271-281.

[Ferriera03]
Fierriera, L., et al., "Introduction to Grid Computing With Globus," IBM Redbooks, (2003).

[Goldberg89]
Goldberg, D.E., Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wesley Publishing Company, Inc., 1989.

[Haupt04]
Haupt, R.L. and S.E. Haupt, Practical Genetic Algorithms, John Wiley & Sons, Inc., New Jersey, 2004.

[Jones03]
Jones, T., AI Application Programming, Hingham, Massachusetts, Charles River Media, 2003.

[Kim04]
Kim, S. and J.B. Weissman, "A GA-based Approach for Scheduling Decomposable Data Grid Applications," Proceedings of the International Conference on Parallel Processing (ICPP'04) (2004), pp. 406-413.

[Kirkpatrick83]
Kirkpatrick, S., C.D. Gelatt, and M.P. Vecchi, "Optimization by Simulated Annealing," Science 220, 4598 (1983), pp. 671-680.

[Krishnakumar89]
Krishnakumar, K., "Micro-genetic Algorithms for Stationary and Non-Stationary Function Optimization," SPIE 1196 (1989), pp. 289-296.

[Man99]
Man, K.F., K.S. Tang and S. Kwong, Genetic Algorithms, Springer-Verlag London Limited, London, 1999.

[Menascé04]
Menascé, D.A. and E. Casalicchio, "QoS in Grid Computing," IEEE Internet Computing 8, 4 (2004), pp. 85-87.

[Menascé04A]
Menascé, D.A. and E. Casalicchio, "A Framework for Resource Allocation in Grid Computing," Proceedings of The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'04), (October, 2004).

[Menascé04B]
Menascé, D.A. and E. Casalicchio, "Quality of Service Aspects and Metrics in Grid Computing," Proceedings of The Computer Measurement Group Conference (December, 2004).

[Menascé04C]
Menascé, D.A., "Mapping Service-Level Agreements in Distributed Applications," IEEE Internet Computing 8, 5 (2004), pp. 100-102.

[Pham00]
Pham, D.T. and D. Karaboga, Intelligent Optimisation Techniques: Genetic Algorithms, Tabu Search, Simulated Annealing, and Neural Networks, Springer-Verlag London Limited, London, 2000.

[Wright32]
Wright, S., "The Roles of Mutation, Inbreeding, Crossbreeding, and Selection in Evolution," Proceedings of 6th International Congress on Genetics 1, 1 (1932), Ithaca, NY, pp. 356-366.

[YarKhan02]
YarKhan, A. and J.J. Dongarra, "Experiments with Scheduling Using Simulated Annealing in a Grid Environment," Lecture Notes in Computer Science: Proceedings of the Third International Workshop on Grid Computing 2563 (2002), pp. 232-242.

[Ye01]
Ye, J. and S. Papavassiliou, "Dynamic Market-driven Allocation of Network Resources Using Genetic Algorithms in a Competitive Electronic Commerce Marketplace," International Journal of Network Management 11 (2001), pp. 375-385.

[Yu06]
Yu, J. and R. Buyya, "A Budget Constrained Scheduling of Workflow Applications on Utility Grids using Genetic Algorithms," Proceedings of 15th IEEE International Symposium on High Performance Distributed Computing (HPDC 2006) (June 19-23, 2006).

APPENDIX A

Appendix A: Optimal Solution Code Listings

```
import java.io.*;
import java.util.*;
import java.lang.Math;
import java.math.*;

/**
* @author James Sweeney
* March, 2007
*
* Main ExhaustiveSearch class
* This is used to perform an exhaustive optimal search.
* The components are the same as the GA, except the removal of
* uneeded functions, and the addition of the exhaustiveSearch routine.
*/
public class ExhaustiveOptimalSearch{
        @SuppressWarnings({"unchecked"})

        // Probability of Crossover
        int crossoverProbability;

        // Probability of Mutation
        int mutationProbability;

        // Initial population in the pool
        int initialPopulation;

        // Number of generations
        int numberOfGenerations;

        // Dimension of the Chromosomes
        int chromosomeDimension;

        // Dicimal percision to use for each gene
        int decimalPrecision;

         // Total number of cycles needed to run the job
        int totalJobCycles;

        // Maximum time allowed for job execution
        int maxJobExecutionTime;

        // Maximum cost allowed for job execution
        double maxJobCost;

        // Current Chromosome pool
        Chromosome currChromosomes;
        Vector <Chromosome>currGenChromosomes;

        // Next generation Chromosome pool
        Chromosome nextChromosomes;
        Vector <Chromosome>nextGenChromosomes;

        static ExhaustiveOptimalSearch myExhaustiveOptimalSearch;

        /*******************************************************************************
***
        * Constructor for ExhaustiveOptimalSearch class.
        *
        * @param initialPopulation, int value for number of chromosomes in each
population
```

```
        * @param numberOfGenerations, int value for number of generations to run (if
used)
        * @param chromosomeDimension, int value for number of genes in each chromosome
        *
        ******************************************************************************
**
        */
        public ExhaustiveOptimalSearch(int initialPopulation,
                    int numberOfGenerations,
                    int chromosomeDimension) {

                this.initialPopulation     = initialPopulation;
                this.numberOfGenerations   = numberOfGenerations;
                this.chromosomeDimension   = chromosomeDimension;
                this.decimalPrecision      = 2;
                this.currChromosomes       = new Chromosome(chromosomeDimension);
                this.currGenChromosomes    = new Vector <Chromosome>();

                this.nextChromosomes       = new Chromosome(chromosomeDimension);
                this.nextGenChromosomes    = new Vector <Chromosome>();


                 for (int i=0;i < this.initialPopulation;i++) {
                   this.currGenChromosomes.add(new Chromosome(chromosomeDimension));


                   for (int x=0;x < this.chromosomeDimension;x++) {
                    if (x==0) {
                                                                          //
Resource#,Speed,Cost,Job Share
                                                                          //                    \  |  / --
---|
                                                                          //                    v v v v
                      ((Chromosome)this.currGenChromosomes.get(i)).setThisGene(x,1,1,0);
                    } else if (x==1) {
                      ((Chromosome)this.currGenChromosomes.get(i)).setThisGene(x,3,2,0);
                    } else if (x==2) {
                      ((Chromosome)this.currGenChromosomes.get(i)).setThisGene(x,5,3,0);
                    } else if (x==3) {
                      ((Chromosome)this.currGenChromosomes.get(i)).setThisGene(x,7,4,0);
                     } else if (x==4) {
                      ((Chromosome)this.currGenChromosomes.get(i)).setThisGene(x,9,5,0);
                    }
                   }

                }

                //Probability of Crossover (ex: 60 => 60%)
                this.crossoverProbability = 65;

                //Probability of Mutation (ex: 5 => 5%)
                this.mutationProbability  = 2;

                // Total number of cycles needed to run the job
                this.totalJobCycles       = 20;

                // Maximum time allowed for job execution
                this.maxJobExecutionTime  = 10;

                // Maximum cost allowed for job execution
                this.maxJobCost           = 25.50;

        } // End of CLASS CONSTRUCTOR: ExhaustiveOptimalSearch

        /**********************************************************************
        *                .:: getScaled ::.
        * Scale decimal number via the rounding mode BigDecimal.ROUND_HALF_UP.
        * Method is used to make sure that the correct number of decimal
        * places are used for each gene.
        *
        * @param value, double value to be scaled
```

```
        * @param scale, int value for number of decimals to scale to
        * @return double representing the newly scaled number
        *********************************************************************
        */
        static public double getScaled(double value, int scale) {
          double result = value; //default: unscaled

          //use BigDecimal String constructor as this is the only exact way for double
values
          result = new BigDecimal(value).setScale(scale,
BigDecimal.ROUND_HALF_UP).doubleValue();

          // Could also use:
          // result = Math.round(value * 100.0) / 100.0;
          return result;
        } // End of METHOD: getScaled

        /**********************************************************
        *          .:: exhaustiveSearch ::.
        * Method used to perform an exhaustive search for the
        * optimal solution using the global chromosome
        * parameters: totalJobCycles, maxJobCost, maxJobExecutionTime
        *
        **********************************************************
        */
       public static void exhaustiveSearch() {
          double oldFitness = 0.00;
          double newFitness = 0.00;
          double oldCost = 0.00;
          double newCost = 0.00;
          Chromosome tempChrom;
          int chromTotal = 0;
          int totalShare = 0;
          int intR0 = 0;
          int intR1 = 0;
          int intR2 = 0;
          int intR3 = 0;
          int intR4 = 0;

          System.out.println("\n*** Exhaustive Search started: " + new Date().toString()
+ " ***");

          tempChrom = new Chromosome(myExhaustiveOptimalSearch.chromosomeDimension);

          try {
            BufferedWriter out = new BufferedWriter(new FileWriter("Optimal_result
S13579 C13567.txt"));
            out.write("*** Exhaustive Search started: " + new Date().toString() + "
***");
            out.newLine();
            out.flush();

            for (int R0 = 0;R0 <= 100; R0++) {
              for (int R1 = 0;R1 <= 100; R1++) {
                for (int R2 = 0;R2 <= 100; R2++) {
                  for (int R3 = 0;R3 <= 100; R3++) {
                    for (int R4 = 0;R4 <= 100; R4++) {

                      //           Resource#,Speed,Cost,Job Share
                      //                         \  |  /       :
                      //                          v v v        v
                      tempChrom.setThisGene(0,1,1,(R0*0.01));
                      tempChrom.setThisGene(1,3,2,(R1*0.01));
                      tempChrom.setThisGene(2,5,3,(R2*0.01));
                      tempChrom.setThisGene(3,7,4,(R3*0.01));
                      tempChrom.setThisGene(4,9,5,(R4*0.01));

                     newCost =
tempChrom.getChromosomeCost(myExhaustiveOptimalSearch.totalJobCycles);
                      newFitness = tempChrom.getFitnessValue(myExhaustiveOptimalSearch);
                      totalShare = R0 + R1 + R2 + R3 + R4;
```

```
                    chromTotal++;

                    if ( (totalShare == 100) && (newFitness >= 0) && (newFitness >
oldFitness) &&

(tempChrom.getChromosomeCost(myExhaustiveOptimalSearch.totalJobCycles) <=
myExhaustiveOptimalSearch.maxJobCost) ) {
                        oldFitness = newFitness;
                        oldCost = newCost;
                        intR0 = R0;
                        intR1 = R1;
                        intR2 = R2;
                        intR3 = R3;
                        intR4 = R4;
                        System.out.println("Chromosome(" + chromTotal + ") ::" + R0 +
"::" + R1 + "::" + R2 + "::" + R3 + "::" + R4 + ":: TotShare=" +
                                   totalShare  + "  FV=" + newFitness + " Cost=" +
newCost + "\n");
                        out.write("Chromosome(" + chromTotal + ") ::" + R0 + "::" + R1 +
"::" + R2 + "::" + R3 + "::" + R4 + ":: TotShare=" +
                                   totalShare  + "  FV=" + newFitness + " Cost=" +
newCost + "\n");
                        out.newLine();
                        out.flush();

                    } else if ( (totalShare == 100) && (newFitness >= 0) &&
(newFitness == oldFitness) &&

(tempChrom.getChromosomeCost(myExhaustiveOptimalSearch.totalJobCycles) <=
myExhaustiveOptimalSearch.maxJobCost) &&

(tempChrom.getChromosomeCost(myExhaustiveOptimalSearch.totalJobCycles) < oldCost) ) {
                        oldFitness = newFitness;
                        oldCost = newCost;
                        intR0 = R0;
                        intR1 = R1;
                        intR2 = R2;
                        intR3 = R3;
                        intR4 = R4;
                        System.out.println("Chromosome(" + chromTotal + ") ::" + R0 +
"::" + R1 + "::" + R2 + "::" + R3 + "::" + R4 + ":: TotShare=" +
                                   totalShare  + "  FV=" + newFitness + " Cost=" +
newCost + "\n");
                        out.write("Chromosome(" + chromTotal + ") ::" + R0 + "::" + R1 +
"::" + R2 + "::" + R3 + "::" + R4 + ":: TotShare=" +
                                   totalShare  + "  FV=" + newFitness + " Cost=" +
newCost + "\n");
                        out.newLine();
                        out.flush();
                    }
                  }
                }
              }
            }
          }
          out.write("Fittest Chromosome...after searching through:" + chromTotal);
            out.newLine();
          out.write("::" + intR0 + "::" + intR1 + "::" + intR2 + "::" + intR3 + "::" +
intR4 + "::");
          out.newLine();
          out.write("FV=" + oldFitness);
          out.newLine();
          out.write("Cost=" + oldCost);
          out.newLine();
          out.write("\n*** Exhaustive Search ended: " + new Date().toString() + "
***");
          out.flush();
          out.close();

          System.out.println("Fittest Chromosome...after searching through:" +
chromTotal);
```

```
            System.out.println("::" + intR0 + "::" + intR1 + "::" + intR2 + "::" + intR3
+ "::" + intR4 + "::");
            System.out.println("FV=" + oldFitness);
            System.out.println("Cost=" + oldCost);
            System.out.println("\n*** Exhaustive Search ended: " + new Date().toString()
+ " ***");

          } catch (IOException e) {}

        }


/*************************************************************************************
*********
        * Main method for ExhaustiveOptimalSearch class
        *
        * @param args, Passed in values for: InitialPopulation, NumberOfGenerations,
ChromosomeDimension
        *

**************************************************************************************
********
        */
        public static void main(String[] args) {
          int population=0, generations=0, dimension=0;
          int gen=0;
          long timeBefore=0, timeAfter=0, timeDiff=0;
          double totalDisplay = 0.00;
          String popDescription = "";
          Vector  <Chromosome> tempVector=null;
          boolean done = false;

          try {
            if (args.length < 3)
              throw new ExhaustiveOptimalSearchException("Requires all 3 parameters:");

            population  = Integer.parseInt(args[0]);
            generations = Integer.parseInt(args[1]);
            dimension   = Integer.parseInt(args[2]);

            if (population < 1) {
              throw new ExhaustiveOptimalSearchException("Initial_Population must be
greater than 0");
            } else if (generations < 1) {
              throw new ExhaustiveOptimalSearchException("Chromosome_Dimension must be
greater than 0");
            }
            if ((population % 2) != 0) {
              throw new ExhaustiveOptimalSearchException("Initial_Population must be an
even number");
            }
          }
          catch (ExhaustiveOptimalSearchException e) {
            System.out.println("\n" + e);
            System.out.println("USE: java ExhaustiveOptimalSearch 'Initial_Population
Number_of_Generations Chromosome_Dimension'");
            System.out.println("EXAMPLE: java ExhaustiveOptimalSearch 6 5 5");
            System.out.println("...exiting, goodbye.");
            System.exit(1);
          }

         myExhaustiveOptimalSearch = new
ExhaustiveOptimalSearch(population,generations,dimension);

          /* Used when executing the exhaustive search funtion */
          myExhaustiveOptimalSearch.exhaustiveSearch();

        } // End of METHOD: main

} // End of CLASS: ExhaustiveOptimalSearch
```

```
import java.io.*;

/**
 * @author James Sweeney
 * March, 2007
 *
 * Class that stores the gene information
 */
public class Gene implements Serializable {

        private static final long serialVersionUID = 1;
        protected int geneSpeed;
        protected double geneCost;
        protected double geneJobShare;

        /*
         * Constructor for the Gene class
         *
         * @param speed, representing speed of this resource (gene)
         * @param cost, representing cost of this resource (gene)
         * @param jobShare, representing percentage of the job that this resource (gene)
will process
         */
        public Gene (int speed, double cost, double jobShare) {
                this.geneSpeed    = speed;
                this.geneCost     = cost;
                this.geneJobShare = jobShare;
        }

        public int getGeneSpeed() {
          return (this.geneSpeed);
        }

        public void setGeneSpeed(int s) {
          this.geneSpeed = s;
        }

        public double getGeneCost() {
          return (this.geneCost);
        }

        public void setGeneCost(double c) {
          this.geneCost = c;
        }

        public double getGeneJobShare() {
          return (this.geneJobShare);
        }

        public void setGeneJobShare(double js) {
          this.geneJobShare = js;
        }

}
```

```
import java.io.*;

/**
 * @author James Sweeney
 * March, 2007
 *
 * Class that stores the chromosome information
 */
public class Chromosome implements Serializable {

	private static final long serialVersionUID = 1;

	// Gene storage for each Chromosome
	protected Gene [] genes;

	/*
	 * Constructor for the Chromosome class
	 * @param numGenes
	 */
	public Chromosome (int numGenes) {
		genes = new Gene [numGenes];
	}

	/********************************************
	 *		.:: getGenes ::.
	 * @return Gene [] representing the chromosome
	 ********************************************
	 */
	public Gene [] getGenes() {
		return (genes);
	}

	/*********************************************************************
	 *		.:: setThisGene ::.
	 * @param int representing the gene to set
	 * @param int representing the speed of the gene
	 * @param double representing the cost of the gene
	 * @param double representing the percentage of job assigned to that gene
	 *********************************************************************
	 */
	public void setThisGene(int thisGene, int geneSpeed, double geneCost, double
geneJobShare) {
		genes[thisGene] = new Gene(geneSpeed,geneCost,geneJobShare);
	}

	/*********************************************
	 *		.:: getGeneJobShareTotal ::.
	 * @return double representing % of job allocated
	 *********************************************
	 */
	public double getGeneJobShareTotal() {
		double geneTotal = 0;
		for (int i=0;i < this.genes.length;i++) {
			geneTotal += genes[i].getGeneJobShare();
		}
		return (ExhaustiveOptimalSearch.getScaled(geneTotal, 2));
	}

	/****************************************************************************
***********
	 *		.:: getFitnessValue ::.
	 * @param thisExhaustiveOptimalSearch, ExhaustiveOptimalSearch which contains
the chromosome about which you wold like to find the FV
	 * @return double representing the Fitness Value

*****************************************************************************************
**
	 */
	public double getFitnessValue(ExhaustiveOptimalSearch
thisExhaustiveOptimalSearch) {
		double completionTime = 0;
```

```
          double tempCompletionTime = 0;

          for (int i=0;i < this.genes.length;i++) {
            tempCompletionTime = ( thisExhaustiveOptimalSearch.totalJobCycles *
genes[i].getGeneJobShare() ) / genes[i].getGeneSpeed();
            if (tempCompletionTime > completionTime) {
              completionTime = tempCompletionTime;
            }
          }
          return (
thisExhaustiveOptimalSearch.getScaled((thisExhaustiveOptimalSearch.maxJobExecutionTime -
completionTime), 2) );
        }

        /*****************************************************************************
***********************
         *          .:: getChromosomeCompletionTime ::.
         * @param thisExhaustiveOptimalSearch, ExhaustiveOptimalSearch which contains
the chromosome about which you wold like to find the completion time
         * @return double representing the time in ms, that it will take for the
chromosome to run

*****************************************************************************************
**************
         */
        public double getChromosomeCompletionTime(ExhaustiveOptimalSearch
thisExhaustiveOptimalSearch) {
          double completionTime = 0;
          double tempCompletionTime = 0;

          for (int i=0;i < this.genes.length;i++) {
            tempCompletionTime = ( thisExhaustiveOptimalSearch.totalJobCycles *
genes[i].getGeneJobShare() ) / genes[i].getGeneSpeed();
            if (tempCompletionTime > completionTime) {
              completionTime = tempCompletionTime;
            }
          }
          return ( thisExhaustiveOptimalSearch.getScaled((completionTime), 2) );
        }

        /***************************************************************
         *          .:: getChromosomeCost ::.
         * @param totalJobCycles, Total job cycles required to run the job
         * @return double representing the cost
         ***************************************************************
         */
        public double getChromosomeCost(int totalJobCycles) {
          double chromCost = 0;
          for (int i=0;i < this.genes.length;i++) {
            chromCost += ( ( (genes[i].getGeneJobShare() * totalJobCycles) /
genes[i].getGeneSpeed() ) * genes[i].getGeneCost() );
          }
          return (ExhaustiveOptimalSearch.getScaled(chromCost, 2));
        }

}
```

```
/**
 * @author James Sweeney
 * March, 2007
 *
 * ExhaustiveOptimalSearch exception class
 */
public class ExhaustiveOptimalSearchException extends Exception
{
    private static final long serialVersionUID = 1;
    /**
     * ExhaustiveOptimalSearchException constructor
     * @param msg, Error message
     */

    ExhaustiveOptimalSearchException(String msg)
    {
        super(msg);
    }
}
```

APPENDIX B

Appendix B: Genetic Algorithm Code Listings

```
import java.io.*;
import java.util.*;
import java.lang.Math;
import java.math.*;

/**
 * @author James Sweeney
 * March, 2007
 *
 * Main GA class
 * This the main class for the Genetic Algorithm.
 */
public class GA{

        // Probability of Crossover
        int crossoverProbability;

        // Probability of Mutation
        int mutationProbability;

        // Initial population in the pool
        int initialPopulation;

        // Number of generations
        int numberOfGenerations;

        // Dimension of the Chromosomes
        int chromosomeDimension;

        // Dicimal percision to use for each gene
        int decimalPrecision;

         // Total number of cycles needed to run the job
        int totalJobCycles;

        // Maximum time allowed for job execution
        int maxJobExecutionTime;

        // Maximum cost allowed for job execution
        double maxJobCost;

        // Current Chromosome pool
        Chromosome currChromosomes;
        Vector <Chromosome>currGenChromosomes;

        // Next generation Chromosome pool
        Chromosome nextChromosomes;
        Vector <Chromosome>nextGenChromosomes;

        static GA myGA;

        /*******************************************************************************
***
         * Constructor for GA class.
         *
         * @param initialPopulation, int value for number of chromosomes in each
population
         * @param numberOfGenerations, int value for number of generations to run (if
used)
         * @param chromosomeDimension, int value for number of genes in each chromosome
```

```
        *
**********************************************************************************
        */
        public GA(int initialPopulation,
                  int numberOfGenerations,
                  int chromosomeDimension) {

              this.initialPopulation     = initialPopulation;
              this.numberOfGenerations   = numberOfGenerations;
              this.chromosomeDimension   = chromosomeDimension;
              this.decimalPrecision      = 2;
              this.currChromosomes       = new Chromosome(chromosomeDimension);
              this.currGenChromosomes    = new Vector <Chromosome>();

              this.nextChromosomes       = new Chromosome(chromosomeDimension);
              this.nextGenChromosomes    = new Vector <Chromosome>();


               for (int i=0;i < this.initialPopulation;i++) {
                 this.currGenChromosomes.add(new Chromosome(chromosomeDimension));

               // ALSO MUST BE CHANGED IN CREATECHROMOSOME PROCEDURE
               // If changing resource Speed/Cost values

                 // Use for 5 available resource testing
                  for (int x=0;x < this.chromosomeDimension;x++) {
                  if (x==0) {
                                        /* Gene-Resource #, Speed, Cost, Job Share*/
                    ((Chromosome)this.currGenChromosomes.get(i)).setThisGene(x,1,1,0);
                  } else if (x==1) {
                    ((Chromosome)this.currGenChromosomes.get(i)).setThisGene(x,3,3,0);
                  } else if (x==2) {
                    ((Chromosome)this.currGenChromosomes.get(i)).setThisGene(x,5,5,0);
                  } else if (x==3) {
                    ((Chromosome)this.currGenChromosomes.get(i)).setThisGene(x,7,6,0);
                   } else if (x==4) {
                    ((Chromosome)this.currGenChromosomes.get(i)).setThisGene(x,9,7,0);
                  }
                 }

                 /* Use for 10 available resource testing
                  for (int x=0;x < this.chromosomeDimension;x++) {
                  if (x==0) {
                                        /* Gene-Resource #, Speed, Cost, Job
Share*//*
                    ((Chromosome)this.currGenChromosomes.get(i)).setThisGene(x,1,1,0);
                  } else if (x==1) {
                    ((Chromosome)this.currGenChromosomes.get(i)).setThisGene(x,2,3,0);
                  } else if (x==2) {
                    ((Chromosome)this.currGenChromosomes.get(i)).setThisGene(x,3,5,0);
                  } else if (x==3) {
                    ((Chromosome)this.currGenChromosomes.get(i)).setThisGene(x,4,6,0);
                   } else if (x==4) {
                    ((Chromosome)this.currGenChromosomes.get(i)).setThisGene(x,5,7,0);
                  } else if (x==5) {
                    ((Chromosome)this.currGenChromosomes.get(i)).setThisGene(x,6,8,0);
                  } else if (x==6) {
                    ((Chromosome)this.currGenChromosomes.get(i)).setThisGene(x,7,9,0);
                  } else if (x==7) {
                    ((Chromosome)this.currGenChromosomes.get(i)).setThisGene(x,8,10,0);
                  } else if (x==8) {
                    ((Chromosome)this.currGenChromosomes.get(i)).setThisGene(x,9,11,0);
                  } else {
((Chromosome)this.currGenChromosomes.get(i)).setThisGene(x,10,12,0);
                  }
                 }
                 */

              }
```

```
            //Probability of Crossover (ex: 60 => 60%)
            this.crossoverProbability = 80;

            //Probability of Mutation (ex: 5 => 5%)
            this.mutationProbability  = 5;

            // Total number of cycles needed to run the job
            this.totalJobCycles        = 20;

            // Maximum time allowed for job execution
            this.maxJobExecutionTime  = 10;

            // Maximum cost allowed for job execution
            this.maxJobCost            = 25.50;

        } // End of CLASS CONSTRUCTOR: GA

    /*******************************************************************
     *                 .:: getScaled ::.
     * Scale decimal number via the rounding mode BigDecimal.ROUND_HALF_UP.
     * Method is used to make sure that the correct number of decimal
     * places are used for each gene.
     *
     * @param value, double value to be scaled
     * @param scale, int value for number of decimals to scale to
     * @return double representing the newly scaled number
     *******************************************************************
     */
    static public double getScaled(double value, int scale) {
        double result = value; //default: unscaled

        //use BigDecimal String constructor as this is the only exact way for double
values
        result = new BigDecimal(value).setScale(scale,
BigDecimal.ROUND_HALF_UP).doubleValue();

        // Could also use:
        // result = Math.round(value * 100.0) / 100.0;
        return result;
    } // End of METHOD: getScaled

    /*****************************************************
     *                 .:: getRandom (int) ::.
     * return a integer random number between 0 and upperBound
     * @param upperBound of the range for randomization
     * @return int, randomly generated number
     *****************************************************
     */
    int getRandom(int upperBound) {
        int iRandom = (int) (Math.random() * upperBound);
        return (iRandom);
    } // End of METHOD: getRandom (int)

    /****************************************************
     *                 .:: getRandom (double) ::.
     * return a double random number between 0 and upperBound
     * @param upperBound of the range for randomization
     * @return double, randomly generated number
     ****************************************************
     */
    double getRandom(double upperBound) {
        // Gives a random number that is:
        // 0.00 <= dRandom < upperBound
        // This does exclude returning the maximum value
        double dRandom = (Math.random() * upperBound);
        return (dRandom);
    } // End of METHOD: getRandom (double)

    /*******************************************************************
     *                 .:: propogateThisChromosome ::.
```

```
   * @param chrom, Chromosome that will be propogated to the next generation
   *
   ***********************************************************************
   */
  public void propagateThisChromosome(Chromosome chrom) {
    nextGenChromosomes.add(chrom);
  } // End of METHOD: propagateThisChromosome


  /*******************************************************
   *                 .:: propogateFittestChromosome ::.
   * @param parent1, Chromsome representing parent 1
   * @param parent2, Chromsome representing parent 2
   * @param child1, Chromsome representing child 1
   * @param child2, Chromsome representing child 2
   *
   *******************************************************
   */
  public void propagateFittestChromosome(Chromosome parent1, Chromosome parent2,
Chromosome child1, Chromosome child2) {
    int propRandNumber=99;
    Chromosome propChromosome = null;
    Chromosome bestChild  = null;
    Chromosome bestParent = null;

    // Find the best child...
    if (child1.getFitnessValue(myGA) == child2.getFitnessValue(myGA) ) {
      if (child1.getChromosomeCost(myGA.totalJobCycles) <
child2.getChromosomeCost(myGA.totalJobCycles) )
        bestChild = child1;
      else
        bestChild = child2;
    } else if (child1.getFitnessValue(myGA) > child2.getFitnessValue(myGA) ) {
        bestChild = child1;
    } else {
        bestChild = child2;
    }

    // ...now find the best parent...
    if (parent1.getFitnessValue(myGA) == parent2.getFitnessValue(myGA) ) {
      if (parent1.getChromosomeCost(myGA.totalJobCycles) <
parent2.getChromosomeCost(myGA.totalJobCycles) )
        bestParent = parent1;
      else
        bestParent = parent2;
    } else if (parent1.getFitnessValue(myGA) > parent2.getFitnessValue(myGA) ) {
        bestParent = parent1;
    } else {
        bestParent = parent2;
    }

    // ...evaluate bestChild and the bestParent...
    if ( (bestChild.getFitnessValue(myGA) == bestParent.getFitnessValue(myGA)) ) {
      if (bestChild.getChromosomeCost(myGA.totalJobCycles) <
bestParent.getChromosomeCost(myGA.totalJobCycles) ) {
        // bestChild and bestParent have equal FV, but bestChild has lower
cost...propagate bestChild
        propChromosome = bestChild;
      } else {
          // bestChild and bestParent have equal FV, but bestParent has lower
cost...choose randomly to propagate
          propRandNumber = getRandom(2);
          if (propRandNumber == 1)
            propChromosome = bestChild;
          else
            propChromosome = bestParent;
      }
    } else if (bestChild.getFitnessValue(myGA) > bestParent.getFitnessValue(myGA) ) {
        // bestChild has a better FV than bestParent, so propagate bestChild
        propChromosome = bestChild;
    } else {
```

```
            // bestParent has a better FV than bestParent...choose randomly to propagate
            propRandNumber = getRandom(2);
            if (propRandNumber == 1)
              propChromosome = bestChild;
            else
              propChromosome = bestParent;
      }

      propagateThisChromosome(propChromosome);

    }  // End METHOD: propagateFittestChromosome

    /*****************************************************
     *             .:: propogateFittestParentChromosome ::.
     * @param poolSize, number of chromosomes in this generation
     *
     *****************************************************
     */
    public void propagateFittestParentChromosome(int poolSize) {
      int chromosome1 = getRandom(poolSize);
      int chromosome2 = getRandom(poolSize);
      Chromosome propChromosome = null;
      Chromosome temp = null;

      /* To ensure that this parent hasn't already been chosen */
      if ( currGenChromosomes.get(chromosome1) == currGenChromosomes.get(chromosome2) )
{
        do {
          chromosome2 = getRandom(poolSize);
        } while ( currGenChromosomes.get(chromosome1) ==
currGenChromosomes.get(chromosome2) );
      }

      if ( ((Chromosome)currGenChromosomes.get(chromosome1)).getFitnessValue(myGA) ==
           ((Chromosome)currGenChromosomes.get(chromosome2)).getFitnessValue(myGA) ) {
        if (
((Chromosome)currGenChromosomes.get(chromosome1)).getChromosomeCost(myGA.totalJobCycles)
<

((Chromosome)currGenChromosomes.get(chromosome2)).getChromosomeCost(myGA.totalJobCycles)
) {
           try {
             propChromosome =
(Chromosome)(ObjectCloner.deepCopy(currGenChromosomes.get(chromosome1)));
           } catch (Exception e) {
             System.out.println("ObjectCloner exception: " + e);
           }
        } else {
            try {
              propChromosome =
(Chromosome)(ObjectCloner.deepCopy(currGenChromosomes.get(chromosome2)));
            } catch (Exception e) {
              System.out.println("ObjectCloner exception: " + e);
            }
        }
      } else if (
((Chromosome)currGenChromosomes.get(chromosome1)).getFitnessValue(myGA) >

((Chromosome)currGenChromosomes.get(chromosome2)).getFitnessValue(myGA) ) {
          try {
            propChromosome =
(Chromosome)(ObjectCloner.deepCopy(currGenChromosomes.get(chromosome1)));
          } catch (Exception e) {
            System.out.println("ObjectCloner exception: " + e);
          }
      } else {
          try {
            propChromosome =
(Chromosome)(ObjectCloner.deepCopy(currGenChromosomes.get(chromosome2)));
          } catch (Exception e) {
            System.out.println("ObjectCloner exception: " + e);
```

```
        }
      }

      propagateThisChromosome(propChromosome);

    }  // End of method propagateFittestParentChromosome


    /**************************************************************************
     *              .:: checkForCrossover ::.
     * @return boolean, True to perform crossover, False to not perform crossover
     *
     **************************************************************************
     */
    public boolean checkForCrossover () {
      int crossoverCheckValue  = getRandom(101);

      /* If the random number chosen 'n', from 0 to 100, is less than or
         equal to the crossoverProbability, then crossover */
      if (crossoverCheckValue <= this.crossoverProbability)
        return true;
      else
        return false;
    } // End of METHOD: checkForCrossover


    /************************************************************************
     *              .:: checkForMutation ::.
     * @return boolean, True to perform mutation, False to not perform mutation
     *
     ************************************************************************
     */
    public boolean checkForMutation () {
      int mutationCheckValue  = getRandom(101);

      /* If the random number chosen 'n', from 0 to 100, is less than or
         equal to the mutationProbability, then mutate */
      if (mutationCheckValue <= this.mutationProbability)
        return true;
      else
        return false;
    } // End of METHOD: checkForMutation


    /********************************************************
     *              .:: getSumFitness ::.
     * @param chromPopulation, Vector of chromosome population of which to sum the
fitness
     * @return double representing the sum of all the FV's in this population
     *
     ********************************************************
     */
    public double getSumFitness(Vector chromPopulation) {
      double fvTotal = 0.00;

      for (int i=0;i < chromPopulation.size();i++) {
        fvTotal += ((Chromosome)chromPopulation.get(i)).getFitnessValue(myGA);
      }
      return  getScaled(fvTotal, this.decimalPrecision);
    } // End of METHOD: getSumFitness


    /******************************************************************
     *              .:: rouletteSelectChromosome ::.
     *
     * Method that takes a Vector of chromosomes, which represents
     * a generation, and selects one using a biased roullete wheel
     * method.
     *
     * @param chromPopulation, Vector representing the total population
     * @return int representing the selected chromosome
```

```
     *************************************************************
     */
    public int rouletteSelectChromosome(Vector chromPopulation) {
      int i = -1;
      double random    = 0;
      double partsum   = 0;

      random = getRandom(1.0) * getSumFitness(chromPopulation);

      if (random != 0) {
        do {
          i++;
          partsum += ((Chromosome)chromPopulation.get(i)).getFitnessValue(myGA);
        } while ( (i < (this.initialPopulation - 1)) && (partsum < random));

      }
      else {
        i = getRandom(chromPopulation.size());
      }
      return i;
    } // End of METHOD: rouletteSelectChromosome


    /*******************************************************
     *                 .:: doMutation ::.
     *
     * Method that takes a given chromosome, and mutates it
     * by subtracting 5% from a randomly selected genes job
     * share, and then adding that 5% to another randomly
     * selected gene's job share.
     *
     * @param mutationChromosome, Chromosome to mutate
     *******************************************************
     */
    protected void doMutation(Chromosome mutationChromosome) {
      int mutationGeneMinus = 0;
      int mutationGeneAdd   = 0;
      Chromosome tempChromosome = null;
      double mutationAmount = 0.05;

      do {

        try {
          tempChromosome = (Chromosome)(ObjectCloner.deepCopy(mutationChromosome));
        } catch (Exception e) {
          System.out.println("ObjectCloner exception: " + e);
        }

        // Find a gene in the given chromosome, from which the mutationAmount (ex: .05 -
> 5%) can be subtracted without
        // leaving a negative number.
        do {
          mutationGeneMinus = getRandom(chromosomeDimension);
        } while ( (tempChromosome.genes[mutationGeneMinus].getGeneJobShare() -
mutationAmount) < 0 );

        // Subtract the mutationAmount
        tempChromosome.genes[mutationGeneMinus].setGeneJobShare(
                  getScaled((tempChromosome.genes[mutationGeneMinus].getGeneJobShare() -
mutationAmount), this.decimalPrecision));

        // Find a gene to add the mutationAmount, but not the one just reduced by the
mutationAmount
        do {
          mutationGeneAdd = getRandom(chromosomeDimension);
        } while (mutationGeneAdd == mutationGeneMinus);

        // Add the mutationAmount
        tempChromosome.genes[mutationGeneAdd].setGeneJobShare(
                  getScaled((tempChromosome.genes[mutationGeneAdd].getGeneJobShare() -
mutationAmount), this.decimalPrecision));
```

```
        // Adjust the new chromosome, to make sure the entire job is allocated
        myGA.adjustGeneDistribution(tempChromosome);

      // Repeat process if any of the following are true:
      //    - New Chromosome doesn't meet the Maximum Cost requiremets
      //    - New Chromosome doesn't account for 100% of the job
      //    - New Chromosome doesn't meet the Fitness Value requirments (within time
limits)
      } while ( (tempChromosome.getChromosomeCost(myGA.totalJobCycles) >
myGA.maxJobCost)          ||
                (getScaled( tempChromosome.getGeneJobShareTotal(),this.decimalPrecision)
!= 1.00) ||
                (tempChromosome.getFitnessValue(myGA) < 0.00) );

      // Copy tempChromosome information over to mutationChromosome to complete the
mutation
      for (int x=0;x < chromosomeDimension;x++) {

mutationChromosome.genes[x].setGeneJobShare(tempChromosome.genes[x].getGeneJobShare());
      }

    } // End of METHOD: doMutation


    /*********************************************************
     *                  .:: doCrossover ::.
     * Method to choose 2 random chromosomes, and then
     * perform a crossover on them.
     *
     * @param int poolSize, int representing the chromosome pool
     *********************************************************
     */
    protected void doCrossover(int poolSize) {
      int crossoverChromosome1 = rouletteSelectChromosome(myGA.currGenChromosomes);
      int crossoverChromosome2 = rouletteSelectChromosome(myGA.currGenChromosomes);
      Chromosome child1 = null;
      Chromosome child2 = null;

      /* To ensure that this parent doesn't mate with itself (I8-P) */
      if ( (Chromosome)myGA.currGenChromosomes.get(crossoverChromosome1) ==
           (Chromosome)myGA.currGenChromosomes.get(crossoverChromosome2) ) {
        do {
          crossoverChromosome2 = rouletteSelectChromosome(myGA.currGenChromosomes);
        } while ( (Chromosome)myGA.currGenChromosomes.get(crossoverChromosome1) ==
                  (Chromosome)myGA.currGenChromosomes.get(crossoverChromosome2) );
      }

      try {
        child1 =
(Chromosome)(ObjectCloner.deepCopy(myGA.currGenChromosomes.get(crossoverChromosome1)));
        child2 =
(Chromosome)(ObjectCloner.deepCopy(myGA.currGenChromosomes.get(crossoverChromosome2)));

        crossoverTheseChromosomes(child1, child2);

        adjustGeneDistribution(child1);

        // Does the chromosome meet the Maximum Cost requirements ?
        if (child1.getChromosomeCost(myGA.totalJobCycles) > myGA.maxJobCost) {

          // The Maximum Cost requirements were NOT met, so...
          do {
            // Create a new chromosome to replace the rejected one
            child1 = (Chromosome)(ObjectCloner.deepCopy(myGA.createChromosome()));

            //Adjust the new chromosome, to make sure the entire job is allocated
            myGA.adjustGeneDistribution(child1);

            //Repeat process if any of the following are true:
            //     - New Chromosome doesn't meet the Maximum Cost requiremets
```

```
            //    - New Chromosome doesn't account for 100% of the job
            //    - New Chromosome doesn't meet the Fitness Value requirments (within
time limits)
          } while ( (child1.getChromosomeCost(myGA.totalJobCycles) > myGA.maxJobCost)
||
                    (getScaled( child1.getGeneJobShareTotal(),this.decimalPrecision) !=
1.00) ||
                    (child1.getFitnessValue(myGA) < 0.00) );
        }

        adjustGeneDistribution(child2);

        // Does the chromosome meet the Maximum Cost requirements ?
        if (child2.getChromosomeCost(myGA.totalJobCycles) > myGA.maxJobCost) {

          // The Maximum Cost requirements were NOT met, so...
          do {
            // Create a new chromosome to replace the rejected one
            child2 = (Chromosome)(ObjectCloner.deepCopy(myGA.createChromosome()));

            // Adjust the new chromosome, to make sure the entire job is allocated
            myGA.adjustGeneDistribution(child2);

            // Repeat process if any of the following are true:
            //    - New Chromosome doesn't meet the Maximum Cost requiremets
            //    - New Chromosome doesn't account for 100% of the job
            //    - New Chromosome doesn't meet the Fitness Value requirments (within
time limits)
          } while ( (child2.getChromosomeCost(myGA.totalJobCycles) > myGA.maxJobCost)
||
                    (getScaled( child2.getGeneJobShareTotal(),this.decimalPrecision) !=
1.00) ||
                    (child2.getFitnessValue(myGA) < 0.00) );
        }

      } catch (Exception e) {
        System.out.println("ObjectCloner exception: " + e);
      }

      propagateFittestChromosome(
((Chromosome)myGA.currGenChromosomes.get(crossoverChromosome1)),((Chromosome)myGA.currGe
nChromosomes.get(crossoverChromosome2)),child1, child2);

    } // End of METHOD: doCrossover

    /**********************************************
     *       .:: crossoverTheseChromosomes ::.
     * Crossover 2 chromosomes at a random point
     *
     * @param Chromosome c1, Chromosome to crossover
     * @param Chromosome c2, Chromosome to crossover
     **********************************************
     */
    protected void crossoverTheseChromosomes(Chromosome c1, Chromosome c2) {
      int crossoverPoint = getRandom(this.chromosomeDimension+1);
      Chromosome temp = null;

      /* Creates a new separate copy of c1, which will be
         use later to populate c2.  Uses Java serialization
         to do the "deep copy" */
      try {
        temp = (Chromosome)(ObjectCloner.deepCopy(c1));
      } catch (Exception e) {
        System.out.println("ObjectCloner exception: " + e);
      }

      for (int i=0;i < crossoverPoint;i++) {
        c1.genes[i].setGeneJobShare(c2.genes[i].getGeneJobShare());
      }

      for (int i=0;i < crossoverPoint;i++) {
```

```
        c2.genes[i].setGeneJobShare(temp.genes[i].getGeneJobShare());
      }

    } // End of METHOD: crossoverTheseChromosomes

    /****************************************************
     *          .:: adjustGeneDistribution ::.
     * Used to adjust a chromosome, so that the sum of the
     * genes job shares equal 1 (100%, entire job allocated)
     *
     * @param c1, Chromosome to adjust
     ****************************************************
     */
    protected void adjustGeneDistribution(Chromosome c1) {
      boolean adjusting = false;

        if ( getScaled(c1.getGeneJobShareTotal(),this.decimalPrecision) != 1) {
          if ( getScaled(c1.getGeneJobShareTotal(),this.decimalPrecision) < 1) {
            do {
              int adjustGene = getRandom(this.chromosomeDimension);
              if ( (c1.getGenes()[adjustGene].getGeneJobShare() +
                   (1 - getScaled(c1.getGeneJobShareTotal(),this.decimalPrecision))) <=
1) {
                   c1.getGenes()[adjustGene].setGeneJobShare(
c1.getGenes()[adjustGene].getGeneJobShare() + (1 -
getScaled(c1.getGeneJobShareTotal(),this.decimalPrecision)) );
                   c1.getGenes()[adjustGene].setGeneJobShare(
getScaled(c1.getGenes()[adjustGene].getGeneJobShare(), this.decimalPrecision) );
                   adjusting = true;
              }
            } while (!adjusting);

          } else {
              if ( (getScaled(c1.getGeneJobShareTotal(),this.decimalPrecision) - 1) >=
0) {
                do {
                  int adjustGene = getRandom(this.chromosomeDimension);
                  if ( (c1.getGenes()[adjustGene].getGeneJobShare() -
                       (getScaled(c1.getGeneJobShareTotal(),this.decimalPrecision) - 1))
>= 0 ) {
                       c1.getGenes()[adjustGene].setGeneJobShare(
c1.getGenes()[adjustGene].getGeneJobShare() -
(getScaled(c1.getGeneJobShareTotal(),this.decimalPrecision) - 1) ) ;
                       c1.getGenes()[adjustGene].setGeneJobShare(
getScaled(c1.getGenes()[adjustGene].getGeneJobShare(), this.decimalPrecision) );
                    adjusting = true;
                  } else {
                    c1.getGenes()[adjustGene].setGeneJobShare(0.00);
                  }
                } while (!adjusting);
              }
          }
        }

    } // End of METHOD: adjustGeneDistribution


    /***************************************************************************
     *          .:: createChromosome ::.
     * Used to create a new chromosome, and then insert it into a specific place in
     * a Vector of chromosomes
     *
     * @param thisChromosome, Vector of chromosomes
     * @param chromosomeIndex, int representing the index of the chromosome to replace
     *
     ***************************************************************************
     */
    public void createChromosome(Vector thisChromosome,int chromosomeIndex) {
      double geneSum = 0.00;

        do {
```

```
          do {
            for (int x=0;x < this.chromosomeDimension;x++) {

((Chromosome)thisChromosome.get(chromosomeIndex)).genes[x].setGeneJobShare(getRandom(100
));
              geneSum +=
((Chromosome)thisChromosome.get(chromosomeIndex)).genes[x].getGeneJobShare();
            }
          } while (geneSum == 0);    /* In the rare case the sum is 0 */

          for (int z=0;z < this.chromosomeDimension;z++) {
            ((Chromosome)thisChromosome.get(chromosomeIndex)).genes[z].setGeneJobShare(
((Chromosome)thisChromosome.get(chromosomeIndex)).genes[z].getGeneJobShare() / geneSum
);
            ((Chromosome)thisChromosome.get(chromosomeIndex)).genes[z].setGeneJobShare(
getScaled((((Chromosome)thisChromosome.get(chromosomeIndex)).genes[z].getGeneJobShare())
, this.decimalPrecision) );
          }
        } while (
((Chromosome)thisChromosome.get(chromosomeIndex)).getFitnessValue(myGA) < 0.00 );
        geneSum = 0.00;

    } // End of METHOD: createChromosome


    /*****************************************************
     *          .:: createChromosome ::.
     * Used to create a new chromosome
     *
     * @return Chromosome, that was just created
     *
     *****************************************************
     */
    public Chromosome createChromosome() {
      double geneSum = 0.00;
      Chromosome newChrom = new Chromosome(myGA.chromosomeDimension);


        // Use for 5 available resources testing
        for (int x=0;x < myGA.chromosomeDimension;x++) {
          if (x==0) {
              /* Gene-Resource #, Speed, Cost, Job Share*/
              newChrom.setThisGene(x,1,1,0);
          } else if (x==1) {
              newChrom.setThisGene(x,3,3,0);
          } else if (x==2) {
              newChrom.setThisGene(x,5,5,0);
          } else if (x==3) {
              newChrom.setThisGene(x,7,6,0);
          } else {
              newChrom.setThisGene(x,9,7,0);
          }

                  /* Use for 10 available resources testing
                  for (int x=0;x < this.chromosomeDimension;x++) {
                    if (x==0) {
                                            /* Gene-Resource #, Speed, Cost, Job
Share*//*
                      ((Chromosome)this.currGenChromosomes.get(i)).setThisGene(x,1,1,0);
                    } else if (x==1) {
                      ((Chromosome)this.currGenChromosomes.get(i)).setThisGene(x,2,3,0);
                    } else if (x==2) {
                      ((Chromosome)this.currGenChromosomes.get(i)).setThisGene(x,3,5,0);
                    } else if (x==3) {
                      ((Chromosome)this.currGenChromosomes.get(i)).setThisGene(x,4,6,0);
                     } else if (x==4) {
                      ((Chromosome)this.currGenChromosomes.get(i)).setThisGene(x,5,7,0);
                    } else if (x==5) {
                      ((Chromosome)this.currGenChromosomes.get(i)).setThisGene(x,6,8,0);
                    } else if (x==6) {
                      ((Chromosome)this.currGenChromosomes.get(i)).setThisGene(x,7,9,0);
```

```
                } else if (x==7) {
                  ((Chromosome)this.currGenChromosomes.get(i)).setThisGene(x,8,10,0);
                } else if (x==8) {
                  ((Chromosome)this.currGenChromosomes.get(i)).setThisGene(x,9,11,0);
                } else {
((Chromosome)this.currGenChromosomes.get(i)).setThisGene(x,10,12,0);
                }
               }
               */
        }

        do {
          do {
            for (int x=0;x < myGA.chromosomeDimension;x++) {
              newChrom.genes[x].setGeneJobShare(getRandom(100));
              geneSum += newChrom.genes[x].getGeneJobShare();

            }
          } while (geneSum == 0);    /* In the rare case the sum is 0 */

          for (int z=0;z < myGA.chromosomeDimension;z++) {
            newChrom.genes[z].setGeneJobShare( newChrom.genes[z].getGeneJobShare() /
geneSum );
            newChrom.genes[z].setGeneJobShare(
getScaled((newChrom.genes[z].getGeneJobShare()), myGA.decimalPrecision) );
          }
        } while ( newChrom.getFitnessValue(myGA) < 0.00 );
        geneSum = 0.00;

        return newChrom;

    } // End of METHOD: createChromosome


    /***************************************************
     *          .:: initPopulationPool ::.
     * Used to create the initial pool of chromosomes
     *
     ***************************************************
     */
    protected void initPopulationPool() {
      double geneSum      = 0.00;
      double total        = 0.00;
      double totalDisplay = 0.00;
      double adjustment   = 0.00;
      int    adjustGene   = 0;

      /* Create the initial population.  Randomly select the percentage
         of the job, that each gene(resource) will be assigned. */
      for (int i=0;i < this.initialPopulation;i++) {
        myGA.createChromosome(this.currGenChromosomes,i);
      }

      /* Find the total percentage of the job being distributed amongst
         the genes in a given chromosome, to make sure it equals 1 (100%).
         If this is not true, then adjust accordingly. */
      for (int i=0;i < this.initialPopulation;i++) {
        do {
          total = 0.00;
          for (int x=0;x < this.chromosomeDimension;x++) {
            total +=
((Chromosome)myGA.currGenChromosomes.get(i)).genes[x].getGeneJobShare();
          }

          /* Adjust one of the genes in the chromosome, so that the total equals 1 (or
100%) */
          if ( getScaled((1 - total),this.decimalPrecision) != 0) {

            /* In case the gene selected is 0.00, and the adjustment amount
               is less than 0, we don't want to subtract from it,
```

```
               which would realize a negative number. */
             if ( getScaled((1 - total),this.decimalPrecision) < 0) {
               do {
                 adjustGene = getRandom(this.chromosomeDimension);
               } while (
((Chromosome)myGA.currGenChromosomes.get(i)).genes[adjustGene].getGeneJobShare() <
                                                              getScaled((total -
1),this.decimalPrecision) );
             }

             adjustment = getScaled((1 - total), this.decimalPrecision);

             /* Does an adjustment need to be made? */
             if ( (getScaled(adjustment, this.decimalPrecision)) != 0.00) {
               /* Add the "adjustment" to the randomly selected gene */

((Chromosome)myGA.currGenChromosomes.get(i)).genes[adjustGene].setGeneJobShare(

((Chromosome)myGA.currGenChromosomes.get(i)).genes[adjustGene].getGeneJobShare() +
adjustment);
               /* Scale value to the decimal precision */

((Chromosome)myGA.currGenChromosomes.get(i)).genes[adjustGene].setGeneJobShare(

getScaled(((Chromosome)myGA.currGenChromosomes.get(i)).genes[adjustGene].getGeneJobShare
(), this.decimalPrecision) );
             }
           }

           /* Does the adjusted chromosome meet the Fitness Value requirements (within
time limits)? */
           if ( ((Chromosome)myGA.currGenChromosomes.get(i)).getFitnessValue(myGA) <
0.00) {
             /* Create a new chromosome to replace the rejected one */
             myGA.createChromosome(myGA.currGenChromosomes,i);
           }

           /* Does the chromosome meet the Maximum Cost requirements ? */
           if (
((Chromosome)myGA.currGenChromosomes.get(i)).getChromosomeCost(myGA.totalJobCycles) >
myGA.maxJobCost) {
             /* The Maximum Cost requirements were NOT met, so... */
             do {
               /* Create a new chromosome to replace the rejected one */
               myGA.createChromosome(myGA.currGenChromosomes,i);

               /* Adjust the new chromosome, to make sure the entire job is allocated */
               myGA.adjustGeneDistribution( (Chromosome)myGA.currGenChromosomes.get(i) );

               /* Repeat process if any of the following are true:
                  - New Chromosome doesn't meet the Maximum Cost requiremets
                  - New Chromosome doesn't account for 100% of the job
                  - New Chromosome doesn't meet the Fitness Value requirments (within
time limits) */
             } while (
(((Chromosome)myGA.currGenChromosomes.get(i)).getChromosomeCost(myGA.totalJobCycles) >
myGA.maxJobCost)          ||
                      (getScaled(
((Chromosome)myGA.currGenChromosomes.get(i)).getGeneJobShareTotal(),this.decimalPrecisio
n) != 1.00) ||

(((Chromosome)myGA.currGenChromosomes.get(i)).getFitnessValue(myGA) < 0.00) );
           }

         } while ( ((Chromosome)myGA.currGenChromosomes.get(i)).getFitnessValue(myGA) <
0.00 ||
                    (getScaled(total,this.decimalPrecision) != 1.00) ||

(((Chromosome)myGA.currGenChromosomes.get(i)).getChromosomeCost(myGA.totalJobCycles) >
myGA.maxJobCost) );
      }
```

```
    } // End of METHOD: initPopulationPool


/*************************************************************************************
     *          .:: displayThisPopulation ::.
     * Used to print the details of each chromosome population
     * to the screen
     *
     * @param thisGA, GA instance to be displayed
     * @param thisPopulation, Vector population to be displayed
     * @param popDescription, String description to display that identifies the
population
     *

*************************************************************************************
     */
    public void displayThisPopulation (GA thisGA, Vector thisPopulation, String
popDescription) {
            double totalDisplay = 0.00;

            System.out.print("*****************************");
            for (int i=0;i < thisGA.chromosomeDimension + 1;i++)
              System.out.print("********");
            System.out.println("");
            System.out.println(popDescription);
            System.out.print("*****************************");
            for (int i=0;i < thisGA.chromosomeDimension + 1;i++)
              System.out.print("********");
            System.out.println("");
            System.out.println("* C    JA      FV     CT     JC        GENES");
            System.out.print("*");
            for (int i=0;i < 37;i++)
              System.out.print(" ");
            for (int i=0;i < thisGA.chromosomeDimension;i++) {
              System.out.print(i);
              if (i < thisGA.chromosomeDimension - 1)
                System.out.print("        ");
            }
            System.out.println("");
            System.out.print("*****************************");
            for (int i=0;i < thisGA.chromosomeDimension + 1;i++)
              System.out.print("********");
            System.out.println("");
            for (int i=0;i < thisGA.initialPopulation;i++) {
              for (int x=0;x < thisGA.chromosomeDimension;x++)
                totalDisplay +=
((Chromosome)thisPopulation.get(i)).genes[x].getGeneJobShare();
              System.out.print("  " + i +
                                "  " + (getScaled(totalDisplay, this.decimalPrecision) *
100) + "%" +
                                "  " +
((Chromosome)thisPopulation.get(i)).getFitnessValue(thisGA) +
                                "  " +
((Chromosome)thisPopulation.get(i)).getChromosomeCompletionTime(thisGA) +
                                "  " +
((Chromosome)thisPopulation.get(i)).getChromosomeCost(thisGA.totalJobCycles) + "    ");
              for (int x=0;x < thisGA.chromosomeDimension;x++)
                System.out.print("|| " +
((Chromosome)thisPopulation.get(i)).genes[x].getGeneJobShare() + " ");
              System.out.println("");

System.out.println("=========================================================================
=======");
              totalDisplay = 0.00;
            }

    } // End of METHOD: displayThisPopulation
```

```
    /*******************************************************
     *          .:: checkConvergence ::.
     * Used check to see if all the chromosomes in the
     * current generation are identical.  IE: All resources/genes
     * are allocated identically to the other chromosomes in the
     * current generation.
     *
     * @return boolean, true if all converge, false if different
     *******************************************************
     */
       public boolean checkConvergence() {

         for (int i=0;i < myGA.initialPopulation;i++) {
           for (int x=0;x < myGA.chromosomeDimension;x++) {
             if (
((Chromosome)myGA.currGenChromosomes.get(i)).genes[x].getGeneJobShare() !=

((Chromosome)myGA.currGenChromosomes.get(0)).genes[x].getGeneJobShare() ) {
                  return false;
             }
           }
         }
         return true;
       } // End of METHOD: checkConvergence


/**************************************************************************************
*********
     * Main method for GA class
     *
     * @param args, Passed in values for: InitialPopulation, NumberOfGenerations,
ChromosomeDimension
     *

**************************************************************************************
********
     */
     public static void main(String[] args) {
          int population=0, generations=0, dimension=0;
          int gen=0;
          long timeBefore=0, timeAfter=0, timeDiff=0;
          double totalDisplay = 0.00;
          String popDescription = "";
          Vector  <Chromosome> tempVector=null;
          boolean done = false;

          System.out.println("\n*** GA started: " + new Date().toString() + " ***");

          try {
            if (args.length < 3)
              throw new GAException("Requires all 3 parameters:");

            population  = Integer.parseInt(args[0]);
            generations = Integer.parseInt(args[1]);
            dimension   = Integer.parseInt(args[2]);

            if (population < 1) {
              throw new GAException("Initial_Population must be greater than 0");
            } else if (generations < 1) {
              throw new GAException("Chromosome_Dimension must be greater than 0");
            }
            if ((population % 2) != 0) {
              throw new GAException("Initial_Population must be an even number");
            }
          }
          catch (GAException e) {
            System.out.println("\n" + e);
            System.out.println("USE: java GA 'Initial_Population Number_of_Generations
Chromosome_Dimension'");
            System.out.println("EXAMPLE: java GA 6 5 5");
```

```
          System.out.println("...exiting, goodbye.");
          System.exit(1);
        }

        /* Open a file to store population information */
        ReportWriter rw = new ReportWriter();

      /* Complete 10 times for averaging purposes */
      for (int zz=0;zz < 10;zz++) {
        myGA = new GA(population,generations,dimension);

        /* Begin timings */
        timeBefore = System.currentTimeMillis();
        System.out.println("Begin time in milliseconds:" + timeBefore);

        /* Initialize the beginning population pool */
        myGA.initPopulationPool();

        /* Initial population is created, now cycle through multiple generations */
        //for (gen=0;gen < myGA.numberOfGenerations;gen++) {
        /* do loop used to perform loop until convergence */
        do {

          /* Check to see if there will be a crossover or mutation */
          for (int i=0;i < myGA.initialPopulation;i++) {
            /* Check to see if there will be a crossover performed */

            if (myGA.checkForCrossover()) {
              myGA.doCrossover(myGA.initialPopulation);
            } else {
              /* No crossover, so check for a mutation */
              if (myGA.checkForMutation()) {
               if (
((Chromosome)myGA.currGenChromosomes.get(i)).getChromosomeCost(myGA.totalJobCycles) >
myGA.maxJobCost ) {
                 System.exit(1);
                }
              }
              myGA.propagateFittestParentChromosome(myGA.initialPopulation);
            }
          }

          /* Make new generation the current generation */
          try {
            tempVector = (Vector
<Chromosome>)(ObjectCloner.deepCopy(myGA.nextGenChromosomes));
          } catch (Exception e) {
            System.out.println("ObjectCloner exception: " + e);
          }
          myGA.currGenChromosomes.removeAllElements();
          myGA.currGenChromosomes = tempVector;
          myGA.nextGenChromosomes.removeAllElements();

          /* Check to make sure that the next generation is empty */
          if (myGA.nextGenChromosomes.isEmpty()) {
            /* All is well, proceed to next generation */
          } else {
            /* Error, next generation should be current generation, and empty...make
some noise */
            popDescription = "* Generation:" + gen + ", nextGenChromosomes <== 0" ;
            myGA.displayThisPopulation(myGA,myGA.nextGenChromosomes,popDescription);
            System.out.println("ERROR!!!! (NextGen) population fitness average: " +
getScaled(myGA.getSumFitness(myGA.currGenChromosomes)/myGA.initialPopulation,
myGA.decimalPrecision)  + "\n\n");
          }

          gen++;
          done = myGA.checkConvergence();

        } while (!done);
```

```
          /* End timings */
          timeAfter = System.currentTimeMillis();
          timeDiff = timeAfter - timeBefore;

          rw.writeThisInfo(
((Chromosome)myGA.currGenChromosomes.get(0)).getChromosomeCost(myGA.totalJobCycles) +
"," +

((Chromosome)myGA.currGenChromosomes.get(0)).getChromosomeCompletionTime(myGA) + "," +

((Chromosome)myGA.currGenChromosomes.get(0)).getFitnessValue(myGA) + "," +
                          getScaled((timeDiff * .001),myGA.decimalPrecision) + "," +
                          gen
                        );


System.out.println("**********************************************************************
*******");
          System.out.println("  TOTAL # OF    AVG CHROM    AVG CHROM        AVG CHROM
TOTAL GA");
          System.out.println("  GENERATIONS   JOB COST   COMPLETION TIME        FV
RUNTIME");
          System.out.println("       " + gen +
                          "          " +
((Chromosome)myGA.currGenChromosomes.get(0)).getChromosomeCost(myGA.totalJobCycles) +
                          "          " +
((Chromosome)myGA.currGenChromosomes.get(0)).getChromosomeCompletionTime(myGA) +
                          "             " +
((Chromosome)myGA.currGenChromosomes.get(0)).getFitnessValue(myGA) +
                          "        " + timeDiff + "ms" + " ==> " + getScaled((timeDiff
* .001),myGA.decimalPrecision) + "s" );

System.out.println("**********************************************************************
*******");

          gen = 0;

        } // end of for loop for averaging purposes

        rw.closeReportFile();

        popDescription = "Convergence" ;
        myGA.displayThisPopulation(myGA,myGA.currGenChromosomes,popDescription);

        System.out.println("\n*** GA ended: " + new Date().toString() + " ***");

      } // End of METHOD: main

} // End of CLASS: GA
```

```java
import java.io.*;

/**
 * @author James Sweeney
 * March, 2007
 *
 * Class that stores the gene information
 */
public class Gene implements Serializable {

	private static final long serialVersionUID = 1;
	protected int geneSpeed;
	protected double geneCost;
	protected double geneJobShare;

	/*
	 * Constructor for the Gene class
	 *
	 * @param speed, representing speed of this resource (gene)
	 * @param cost, representing cost of this resource (gene)
	 * @param jobShare, representing percentage of the job that this resource (gene)
will process
	 */
	public Gene (int speed, double cost, double jobShare) {
		this.geneSpeed    = speed;
		this.geneCost     = cost;
		this.geneJobShare = jobShare;
	}

	public int getGeneSpeed() {
	  return (this.geneSpeed);
	}

	public void setGeneSpeed(int s) {
	  this.geneSpeed = s;
	}

	public double getGeneCost() {
	  return (this.geneCost);
	}

	public void setGeneCost(double c) {
	  this.geneCost = c;
	}

	public double getGeneJobShare() {
	  return (this.geneJobShare);
	}

	public void setGeneJobShare(double js) {
	  this.geneJobShare = js;
	}

}
```

```
import java.io.*;

/**
 * @author James Sweeney
 * March, 2007
 *
 * Class that stores the chromosome information
 */
public class Chromosome implements Serializable {

	 private static final long serialVersionUID = 1;

	// Gene storage for each Chromosome
	protected Gene [] genes;

	/*
	 * Constructor for the Chromosome class
	 * @param numGenes
	 */
	public Chromosome (int numGenes) {
		genes = new Gene [numGenes];
	}

	/*******************************************
	 *          .:: getGenes ::.
	 * @return Gene [] representing the chromosome
	 *******************************************
	 */
	public Gene [] getGenes() {
		return (genes);
	}

	/*********************************************************************
	 *          .:: setThisGene ::.
	 * @param int representing the gene to set
	 * @param int representing the speed of the gene
	 * @param double representing the cost of the gene
	 * @param double representing the percentage of job assigned to that gene
	 *********************************************************************
	 */
	public void setThisGene(int thisGene, int geneSpeed, double geneCost, double
geneJobShare) {
		genes[thisGene] = new Gene(geneSpeed,geneCost,geneJobShare);
	}

	/*********************************************
	 *          .:: getGeneJobShareTotal ::.
	 * @return double representing % of job allocated
	 *********************************************
	 */
	public double getGeneJobShareTotal() {
		double geneTotal = 0;
		for (int i=0;i < this.genes.length;i++) {
			geneTotal += genes[i].getGeneJobShare();
		}
		return (GA.getScaled(geneTotal, 2));
	}

	/***************************************************************************
**********
	 *          .:: getFitnessValue ::.
	 * @param thisGA, GA which contains the chromosome about which you wold like to
find the FV
	 * @return double representing the Fitness Value
*****************************************************************************************
**
	 */
	public double getFitnessValue(GA thisGA) {
	  double completionTime = 0;
	   double tempCompletionTime = 0;
```

```
          for (int i=0;i < this.genes.length;i++) {
            tempCompletionTime = ( thisGA.totalJobCycles * genes[i].getGeneJobShare() )
/ genes[i].getGeneSpeed();
            if (tempCompletionTime > completionTime) {
              completionTime = tempCompletionTime;
            }
          }
         return ( thisGA.getScaled((thisGA.maxJobExecutionTime - completionTime), 2) );
       }

       /*****************************************************************************
************************
         *          .:: getChromosomeCompletionTime ::.
         * @param thisGA, GA which contains the chromosome about which you wold like to
find the completion time
         * @return double representing the time in ms, that it will take for the
chromosome to run

***************************************************************************************
**************
         */
       public double getChromosomeCompletionTime(GA thisGA) {
        double completionTime = 0;
          double tempCompletionTime = 0;

          for (int i=0;i < this.genes.length;i++) {
            tempCompletionTime = ( thisGA.totalJobCycles * genes[i].getGeneJobShare() )
/ genes[i].getGeneSpeed();
            if (tempCompletionTime > completionTime) {
              completionTime = tempCompletionTime;
            }
          }
         return ( thisGA.getScaled((completionTime), 2) );
       }

       /***************************************************************
        *          .:: getChromosomeCost ::.
        * @param totalJobCycles, Total job cycles required to run the job
        * @return double representing the cost
        ***************************************************************
        */
       public double getChromosomeCost(int totalJobCycles) {
         double chromCost = 0;
         for (int i=0;i < this.genes.length;i++) {
           chromCost += ( ( (genes[i].getGeneJobShare() * totalJobCycles) /
genes[i].getGeneSpeed() ) * genes[i].getGeneCost() );
         }
         return (GA.getScaled(chromCost, 2));
       }

}
```

```
/**
 * @author James Sweeney
 * March, 2007
 *
 * GA exception class
 *
 */
public class GAException extends Exception
{

    private static final long serialVersionUID = 1;

    /**
     * GAException constructor
     * @param msg, Error message
     */

    GAException(String msg)
    {
        super(msg);
    }
}
```

```
import java.io.*;
import java.util.*;
import java.awt.*;

 /**
 * @author James Sweeney (Dave Miller code)
 * March, 2007
 *
 * Class used to make a deep copy of an object
 */
public class ObjectCloner
{
   // so that nobody can accidentally create an ObjectCloner object
   private ObjectCloner(){}
   // returns a deep copy of an object
   static public Object deepCopy(Object oldObj) throws Exception
   {
      ObjectOutputStream oos = null;
      ObjectInputStream ois = null;
      try
      {
         ByteArrayOutputStream bos =
               new ByteArrayOutputStream(); // A
         oos = new ObjectOutputStream(bos); // B
         // serialize and pass the object
         oos.writeObject(oldObj);   // C
         oos.flush();               // D
         ByteArrayInputStream bin =
               new ByteArrayInputStream(bos.toByteArray()); // E
         ois = new ObjectInputStream(bin);                  // F
         // return the new object
         return ois.readObject(); // G
      }
      catch(Exception e)
      {
         System.out.println("Exception in ObjectCloner = " + e);
         throw(e);
      }
      finally
      {
         oos.close();
         ois.close();
      }
   }

}
```

```
import java.io.*;
import java.util.*;

/**
 * @author James Sweeney
 * March, 2007
 *
 * Class that is used to create a file to be used to write information
 */
public class ReportWriter {

  String fileName = "report.txt";
  File f;
  FileWriter fw;
  PrintWriter pw;

  /*
   * Constructor for the ReportWriter class
   */
  public ReportWriter () {
    try {
      f  = new File(fileName);
      fw = new FileWriter(f);
      pw = new PrintWriter(fw);
      //pw.println("\n*** GA started: " + new Date().toString() + " ***");
    }
    catch(IOException e) {
      System.out.println("Exception writing file:" + e);
    }
  }

    /**********************************************************************
     *          .:: writeThisPopulation ::.
     * @param thisGA, GA that contains the population of interest
     * @param thisPopulation, Vector containing the cromosomes in the population
     * @param popDescription, String describing the population
     *
     **********************************************************************
     */
    public void writeThisPopulation (GA thisGA, Vector thisPopulation, String
popDescription) {
            double totalDisplay = 0.00;

            pw.print("\n\n*****************************");
            for (int i=0;i < thisGA.chromosomeDimension + 1;i++)
              pw.print("********");
            pw.println("");
            pw.println(popDescription);
            pw.print("*****************************");
            for (int i=0;i < thisGA.chromosomeDimension + 1;i++)
              pw.print("********");
            pw.println("");
            pw.println("* C    JA       FV     CT     JC       GENES");
            pw.print("*");
            for (int i=0;i < 37;i++)
              pw.print(" ");
            for (int i=0;i < thisGA.chromosomeDimension;i++) {
              pw.print(i);
              if (i < thisGA.chromosomeDimension - 1)
                pw.print("       ");
            }
            pw.println("");
            pw.print("*****************************");
            for (int i=0;i < thisGA.chromosomeDimension + 1;i++)
              pw.print("********");
            pw.println("");
            for (int i=0;i < thisGA.initialPopulation;i++) {
              for (int x=0;x < thisGA.chromosomeDimension;x++)
                totalDisplay +=
((Chromosome)thisPopulation.get(i)).genes[x].getGeneJobShare();
              pw.print("  " + i +
```

```
                                    "  " + (thisGA.getScaled(totalDisplay,
thisGA.decimalPrecision) * 100) + "%" +
                                    "  " +
((Chromosome)thisPopulation.get(i)).getFitnessValue(thisGA) +
                                    "  " +
((Chromosome)thisPopulation.get(i)).getChromosomeCompletionTime(thisGA) +
                                    "  " +
((Chromosome)thisPopulation.get(i)).getChromosomeCost(thisGA.totalJobCycles) + "    ");
                for (int x=0;x < thisGA.chromosomeDimension;x++)
                  pw.print("|| " +
((Chromosome)thisPopulation.get(i)).genes[x].getGeneJobShare() + " ");
                pw.println("");

pw.println("=========================================================================="
);
                totalDisplay = 0.00;
              }

    } // End of METHOD: writeThisPopulation

    /*************************************
     *         .:: writeThisInfo ::.
     * @param info, String to write to file
     *
     *************************************
     */
    public void writeThisInfo(String info) {
      pw.println(info);
    }

    /*************************************
     *         .:: closeReportFile ::.
     * Method to record the time that the
     * GA ended, and close the file
     *
     *************************************
     */
    public void closeReportFile() {
      pw.close();
    }

}
```

APPENDIX C

Appendix C: Simulated Annealing Code Listings

```
import java.io.*;
import java.util.*;
import java.lang.Math;
import java.math.*;

/**
 * @author James Sweeney
 * March, 2007
 *
 * Main Simulated Annealing class
 *
 */
public class SimulatedAnnealing{

        /* Annealing Schedule Variables */

        // Variables for the bounds of the cooling schedule
        double initialTemperature; // starting point for algorithm
        double finalTemperature;   // stopping point for algorithm
        double tweakFactor;        // used to perturb the solution

        // Constant used for geometric cooling
        double alpha;              // used as a multiplier to decrement the temperature

        // Number of iterations performed at each temperature change (plateau)
        int stepsPerChange;        // number of iterations at each temperature level

        // Number of available resources for each solution
        int solutionDimension;

        // Decimal precision to use for each resource
        int decimalPrecision;

        // Total number of cycles needed to run the job
        int totalJobCycles;

        // Maximum time allowed for job execution
        int maxJobExecutionTime;

        // Maximum cost allowed for job execution
        double maxJobCost;

        Solution current, working, best;

        static SimulatedAnnealing mySA;

        /*****************************************************************************
****
         * Constructor for SimulatedAnnealing class.
         *
         * @param initialTemperature, double value for starting temperature of the
algorithm
         * @param finalTemperature, double value for the ending temperature of the
algorithm
         * @param iterations, int value for the number of cycles at each temperature
level
         * @param alpha, double value used as a multiplier to decrement the temperature
         * @param tweakFactor, double value used perturb the solution.
         *

*********************************************************************************
```

```
 *
 */
public SimulatedAnnealing(double initialTemperature,
                          double finalTemperature,
                          int    iterations,
                          double alpha,
                          double tweakFactor) {

  this.initialTemperature = initialTemperature;
  this.finalTemperature   = finalTemperature;
  this.stepsPerChange     = iterations;
  this.alpha              = alpha;
  this.tweakFactor        = tweakFactor;

  this.solutionDimension   = 5;

  current = new Solution(solutionDimension);
  working = new Solution(solutionDimension);
  best    = new Solution(solutionDimension);

  for (int x=0;x < solutionDimension;x++) {
    if (x==0) {
      /* Gene-Resource #, Speed, Cost, Job Share*/
      current.setThisResource(x,1,1,0);
      working.setThisResource(x,1,1,0);
      best.setThisResource(x,1,1,0);
    } else if (x==1) {
      current.setThisResource(x,3,3,0);
      working.setThisResource(x,3,3,0);
      best.setThisResource(x,3,3,0);
    } else if (x==2) {
      current.setThisResource(x,5,5,0);
      working.setThisResource(x,5,5,0);
      best.setThisResource(x,5,5,0);
    } else if (x==3) {
      current.setThisResource(x,7,6,0);
      working.setThisResource(x,7,6,0);
      best.setThisResource(x,7,6,0);
    } else if (x==4) {
      current.setThisResource(x,9,7,0);
      working.setThisResource(x,9,7,0);
      best.setThisResource(x,9,7,0);
    }
      //To be used with 10 available resources
      /*else if (x==5) {
      current.setThisResource(x,6,8,0);
      working.setThisResource(x,6,8,0);
      best.setThisResource(x,6,8,0);
    } else if (x==6) {
      current.setThisResource(x,7,9,0);
      working.setThisResource(x,7,9,0);
      best.setThisResource(x,7,9,0);
    } else if (x==7) {
      current.setThisResource(x,8,10,0);
      working.setThisResource(x,8,10,0);
      best.setThisResource(x,8,10,0);
    } else if (x==8) {
      current.setThisResource(x,9,11,0);
      working.setThisResource(x,9,11,0);
      best.setThisResource(x,9,11,0);
    } else {
      current.setThisResource(x,10,12,0);
      working.setThisResource(x,10,12,0);
      best.setThisResource(x,10,12,0);
    } */
  }

  this.decimalPrecision    = 2;

 // Total number of cycles needed to run the job
 this.totalJobCycles      = 20;
```

```
        // Maximum time allowed for job execution
        this.maxJobExecutionTime = 10;

        // Maximum cost allowed for job execution
        this.maxJobCost          = 25.50;

      }


    /*********************************************************************
     *                  .:: getScaled ::.
     * Scale decimal number via the rounding mode BigDecimal.ROUND_HALF_UP.
     * Method is used to make sure that the correct number of decimal
     * places are used for each resource.
     *
     * @param value, double value to be scaled
     * @param scale, int value for number of decimals to scale to
     * @return double representing the newly scaled number
     *********************************************************************
     */
    static public double getScaled(double value, int scale) {
        double result = value; //default: unscaled

        //use BigDecimal String constructor as this is the only exact way for double
values
        result = new BigDecimal(value).setScale(scale,
BigDecimal.ROUND_HALF_UP).doubleValue();

        // Could also use:
         //result = Math.round(value * 100.0) / 100.0;
        return result;
    } // End of METHOD: getScaled


    /*******************************************************
     *                  .:: getRandom (int) ::.
     * return a integer random number between 0 and upperBound
     * @param upperBound of the range for randomization
     * @return int, randomly generated number
     *******************************************************
     */
    int getRandom(int upperBound) {
        int iRandom = (int) (Math.random() * upperBound);
        return (iRandom);
    } // End of METHOD: getRandom (int)

    /*****************************************************
     *                  .:: getRandom (double) ::.
     * return a double random number between 0 and upperBound
     * @param upperBound of the range for randomization
     * @return double, randomly generated number
     *****************************************************
     */
    double getRandom(double upperBound) {
        // Gives a random number that is:
        // 0.00 <= dRandom < upperBound
        // This does exclude returning the maximum value
        double dRandom = (Math.random() * upperBound);
        return (dRandom);
    } // End of METHOD: getRandom (double)

    /*******************************************************
     *                  .:: tweakSolution ::.
     *
     * Method that takes a given Solution, and tweaks it
     * by subtracting "tweakFactor" (ex: .05 ==> 5%) from a
     * randomly selected resources job share,
     * and then adding that "tweakFactor" to another randomly
     * selected resources's job share. The "tweakFactor" variable
     * is passed in at program startup.
```

```
     *
     * @param thisSolution, Solution to tweak
     *****************************************************
     */
    protected Solution tweakSolution(Solution thisSolution) {
      int resourceMinus = 0;
      int resourceAdd   = 0;

      /* Find a resource in the given solution, from which tweakFactor can be subtracted
without
         leaving a negative number. */
      do {
        resourceMinus = getRandom(solutionDimension);
      } while ( (thisSolution.resources[resourceMinus].getResourceJobShare() -
this.tweakFactor) < 0 );

      /* Subtract the tweakFactor */
      thisSolution.resources[resourceMinus].setResourceJobShare(
              getScaled((thisSolution.resources[resourceMinus].getResourceJobShare()
- this.tweakFactor), this.decimalPrecision));

      /* Find a resource to add tweakFactor, but not the one just reduced by tweakFactor
*/
      do {
        resourceAdd = getRandom(solutionDimension);
      } while (resourceAdd == resourceMinus);

      /* Add the tweakFactor */
      thisSolution.resources[resourceAdd].setResourceJobShare(
              getScaled((thisSolution.resources[resourceAdd].getResourceJobShare() +
this.tweakFactor), this.decimalPrecision));

      try {

          /* Does the solution meet the Maximum Cost requirements ? */
          if (thisSolution.getSolutionCost(mySA.totalJobCycles) > mySA.maxJobCost) {
            /* The Maximum Cost requirements were NOT met, so... */
            do {
              do {
                /* Create a new solution to replace the rejected one */
                thisSolution = mySA.createSolution();

                /* Adjust the new solution, to make sure the entire job is allocated */
                mySA.adjustResourceDistribution(thisSolution);

                /* Repeat process if any of the following are true:
                   - New solution doesn't meet the Maximum Cost requiremets
                   - New solution doesn't account for 100% of the job
                   - New solution doesn't meet the Energy requirments (within time
limits) */
              } while ( (thisSolution.getSolutionCost(mySA.totalJobCycles) >
mySA.maxJobCost)          ||
                        (getScaled(
thisSolution.getResourceJobShareTotal(),this.decimalPrecision) != 1.00) ||
                        (thisSolution.getEnergy(mySA) < 0.00) );
            } while (! thisSolution.isValid());
          }
       } catch (Exception e) {
        System.out.println("ObjectCloner exception: " + e);
       }

    return thisSolution;

    } // End of METHOD: tweakSolution


    /*****************************************************
     *          .:: adjustResourceDistribution ::.
     * Used to adjust a Resource, so that the sum of the
     * Resource job shares equal 1 (100%, entire job allocated)
     *
```

```
     * @param s1, Resource to adjust
     ****************************************************
     */
    protected void adjustResourceDistribution(Solution s1) {
      boolean adjusting = false;

        if ( getScaled(s1.getResourceJobShareTotal(),mySA.decimalPrecision) != 1) {
          if ( getScaled(s1.getResourceJobShareTotal(),mySA.decimalPrecision) < 1) {
            do {
              int adjustResource = getRandom(mySA.solutionDimension);
              if ( (s1.getResources()[adjustResource].getResourceJobShare() +
                   (1 - getScaled(s1.getResourceJobShareTotal(),mySA.decimalPrecision)))
<= 1) {
                   s1.getResources()[adjustResource].setResourceJobShare(
s1.getResources()[adjustResource].getResourceJobShare() + (1 -
getScaled(s1.getResourceJobShareTotal(),mySA.decimalPrecision)) );
                   s1.getResources()[adjustResource].setResourceJobShare(
getScaled(s1.getResources()[adjustResource].getResourceJobShare(),
mySA.decimalPrecision) );
                   adjusting = true;
              }
            } while (!adjusting);

          } else {
              if ( (getScaled(s1.getResourceJobShareTotal(),mySA.decimalPrecision) - 1)
>= 0) {
                do {
                  int adjustResource = getRandom(mySA.solutionDimension);
                  if ( (s1.getResources()[adjustResource].getResourceJobShare() -
                       (getScaled(s1.getResourceJobShareTotal(),mySA.decimalPrecision) -
1)) >= 0 ) {
                       s1.getResources()[adjustResource].setResourceJobShare(
s1.getResources()[adjustResource].getResourceJobShare() -
(getScaled(s1.getResourceJobShareTotal(),mySA.decimalPrecision) - 1) ) ;
                       s1.getResources()[adjustResource].setResourceJobShare(
getScaled(s1.getResources()[adjustResource].getResourceJobShare(),
mySA.decimalPrecision) );
                    adjusting = true;
                  } else {
                    s1.getResources()[adjustResource].setResourceJobShare(0.00);
                  }
                } while (!adjusting);
              }
          }
        }

    } // End of METHOD: adjustGeneDistribution


    /****************************************************
     *           .:: createSolution ::.
     * Used to create a new solution
     *
     * @return Solution, that was just created
     *
     ****************************************************
     */
    public Solution createSolution() {
      double resourceSum = 0.00;
      Solution newSolution = new Solution(mySA.solutionDimension);

        for (int x=0;x < mySA.solutionDimension;x++) {
          if (x==0) {
              /* Resource #, Speed, Cost, Job Share*/
              newSolution.setThisResource(x,1,1,0);
          } else if (x==1) {
              newSolution.setThisResource(x,3,3,0);
          } else if (x==2) {
              newSolution.setThisResource(x,5,5,0);
          } else if (x==3) {
              newSolution.setThisResource(x,7,6,0);
```

```
          } else if (x==4) {
             newSolution.setThisResource(x,9,7,0);
          }
            // To be used with 10 available resources
            /*else if (x==5) {
             newSolution.setThisResource(x,6,8,0);
          } else if (x==6) {
             newSolution.setThisResource(x,7,9,0);
          } else if (x==7) {
             newSolution.setThisResource(x,8,10,0);
          } else if (x==8) {
             newSolution.setThisResource(x,9,11,0);
          } else {
             newSolution.setThisResource(x,10,12,0);
          }
          */
        }

        do {
          do {
            for (int x=0;x < mySA.solutionDimension;x++) {
              newSolution.resources[x].setResourceJobShare(getRandom(100));
              resourceSum += newSolution.resources[x].getResourceJobShare();
            }
          } while (resourceSum == 0);    /* In the rare case the sum is 0 */

          for (int z=0;z < mySA.solutionDimension;z++) {
            newSolution.resources[z].setResourceJobShare(
newSolution.resources[z].getResourceJobShare() / resourceSum );
            newSolution.resources[z].setResourceJobShare(
getScaled((newSolution.resources[z].getResourceJobShare()), mySA.decimalPrecision) );
          }

          mySA.adjustResourceDistribution(newSolution);

        } while ( newSolution.getEnergy(mySA) < 0.00 );

        try {
          /* Does the Solution meet the Maximum Cost requirements ? */
          if (newSolution.getSolutionCost(mySA.totalJobCycles) > mySA.maxJobCost) {

            /* The Maximum Cost requirements were NOT met, so set valid flag to false */
              newSolution.setValid(false);
              return newSolution;
          }
        } catch (Exception e) {
          System.out.println("ObjectCloner exception: " + e);
        }

       resourceSum = 0.00;

       if (newSolution.getSolutionCost(mySA.totalJobCycles) > mySA.maxJobCost) {
          mySA.printSolution(newSolution, "...OVER COST - NEW SOLUTION...");
       }

        return newSolution;

    } // End of METHOD: createSolution


/**********************************************************************************
     *          .:: displayThisPopulation ::.
     * Used to print the details of each chromosome population to the screen
     *
     * @param thisSolution, Solution to be displayed
     * @param description, String description to display that identifies the solution
     *
*********************************************************************************
     */
```

```
    public void printSolution(Solution thisSolution, String description) {
       double totalDisplay = 0.0;

            System.out.print("***************************");
            for (int i=0;i < mySA.solutionDimension + 1;i++)
              System.out.print("********");
            System.out.println("");
            System.out.println(description);
            System.out.print("***************************");
            for (int i=0;i < mySA.solutionDimension + 1;i++)
              System.out.print("********");
            System.out.println("");
            System.out.println("*  JA    ENERGY    CT     JC       RESOURCES");
            System.out.print("*");
            for (int i=0;i < 37;i++)
              System.out.print(" ");
            for (int i=0;i < mySA.solutionDimension;i++) {
              System.out.print(i);
              if (i < mySA.solutionDimension - 1)
                System.out.print("       ");
            }
            System.out.println("");
            System.out.print("***************************");
            for (int i=0;i < mySA.solutionDimension + 1;i++)
              System.out.print("********");
            System.out.println("");

              for (int x=0;x < mySA.solutionDimension;x++)
                totalDisplay += thisSolution.getResources()[x].getResourceJobShare();
              System.out.print(
                                "  " + (getScaled(totalDisplay, mySA.decimalPrecision) *
100) + "%" +
                                "  " + thisSolution.getEnergy(mySA) +
                                "  " + thisSolution.getSolutionCompletionTime(mySA) +
                                "  " + thisSolution.getSolutionCost(mySA.totalJobCycles)
+ "   ");
              for (int x=0;x < mySA.solutionDimension;x++)
                System.out.print("|| " +
thisSolution.getResources()[x].getResourceJobShare() + " ");
              System.out.println("");

System.out.println("====================================================================
=======");

    } // End of METHOD: printSolution


/*****************************************************************************************
**************
     * Main method for SA class
     *
     * @param args, Passed in values for:
initialTemperature,finalTemperature,iterations,alpha,tweakFactor
     *

*****************************************************************************************
*************
     */
     public static void main(String[] args) {

          double temperature = 0.00;
          long timeBefore=0, timeAfter=0, timeDiff=0;
          int step;
          double initialTemperature=0.00,
finalTemperature=0.00,alpha=0.00,tweakFactor=0.00;
          int iterations=0;
          boolean solution = false;

          System.out.println("\n*** SA started: " + new Date().toString() + " ***");
```

```
        try {
          if (args.length < 5)
            throw new SAException("Requires all 5 parameters:");

          initialTemperature = Double.parseDouble(args[0]);
          finalTemperature   = Double.parseDouble(args[1]);
          iterations         = Integer.parseInt(args[2]);
          alpha              = Double.parseDouble(args[3]);
          tweakFactor        = Double.parseDouble(args[4]);

          if (initialTemperature < 1) {
            throw new SAException("Initial_Temperature must be greater than or equal
to 1.0");
          }
          if (finalTemperature < 0.01) {
            throw new SAException("Final_Temperature must be greater than 0.01");
          }
          if (iterations < 1) {
            throw new SAException("iterations must be greater than or equal to 1");
          }
          if (alpha >= 1) {
            throw new SAException("alpha must be less than 1");
          }
          if (tweakFactor >= 1) {
            throw new SAException("tweakFactor must be less than 1");
          }

        }
        catch (SAException e) {
          System.out.println("\n" + e);
          System.out.println("USE: java SimulatedAnnealing 'initialTemperature
finalTemperature iterations alpha tweakFactor'");
          System.out.println("EXAMPLE: java SimulatedAnnealing 50 .05 10 .99 .05");
          System.out.println("...exiting, goodbye.");
          System.exit(1);
        }

        /* Open a file to store population information */
        ReportWriter rw = new ReportWriter();

    /* Complete 10 times for averaging purposes */
    for (int zz=0;zz < 10;zz++) {
        mySA = new
SimulatedAnnealing(initialTemperature,finalTemperature,iterations,alpha,tweakFactor);

        temperature = mySA.initialTemperature;

        /* Begin timings */
        timeBefore = System.currentTimeMillis();
        System.out.println("Begin time in milliseconds:" + timeBefore);


       do {
         do {
           /* Create a new Solution to replace the rejected one */
           mySA.current = mySA.createSolution();

           /* Adjust the new Solution, to make sure the entire job is allocated */
           mySA.adjustResourceDistribution(mySA.current);

           /* Repeat process if any of the following are true:
              - New Solution doesn't meet the Maximum Cost requiremets
              - New Solution doesn't account for 100% of the job
              - New Solution doesn't meet the Energy requirments (within time limits)
*/
         } while ( (mySA.current.getSolutionCost(mySA.totalJobCycles) >
mySA.maxJobCost)          ||
                   (getScaled(
mySA.current.getResourceJobShareTotal(),mySA.decimalPrecision) != 1.00) ||
                   (mySA.current.getEnergy(mySA) < 0.00) );
        } while (! mySA.current.isValid());
```

```
          try {
            mySA.working = (Solution)(ObjectCloner.deepCopy(mySA.current));
            mySA.best = (Solution)(ObjectCloner.deepCopy(mySA.current));
          } catch (Exception e) {
            System.out.println("ObjectCloner exception: " + e);
          }

          do {

            for (step = 0;step < mySA.stepsPerChange;step++) {

              try {
                mySA.working = (Solution)(ObjectCloner.deepCopy(mySA.current));
              } catch (Exception e) {
                System.out.println("ObjectCloner exception: " + e);
              }

              mySA.working = mySA.tweakSolution(mySA.working);

              double test  = mySA.getRandom(1.0);
              double delta = mySA.working.getEnergy(mySA) -
mySA.current.getEnergy(mySA);
              double calc  = 0.00;

              if (delta > 0) {

                try {
                  mySA.current = (Solution)(ObjectCloner.deepCopy(mySA.working));
                } catch (Exception e) {
                  System.out.println("ObjectCloner exception: " + e);
                }

                if (mySA.working.getEnergy(mySA) > mySA.best.getEnergy(mySA)) {
                  try {
                    mySA.best = (Solution)(ObjectCloner.deepCopy(mySA.working));
                  } catch (Exception e) {
                    System.out.println("ObjectCloner exception: " + e);
                  }
                } else if ( (mySA.working.getEnergy(mySA) == mySA.best.getEnergy(mySA))
&&
                            (mySA.working.getSolutionCost(mySA.totalJobCycles) <=
mySA.best.getSolutionCost(mySA.totalJobCycles)) ) {
                  try {
                    mySA.best = (Solution)(ObjectCloner.deepCopy(mySA.working));
                  } catch (Exception e) {
                    System.out.println("ObjectCloner exception: " + e);
                  }
                }

              } else if (test < Math.exp(delta/temperature) ) {
                try {
                  mySA.current = (Solution)(ObjectCloner.deepCopy(mySA.working));
                } catch (Exception e) {
                  System.out.println("ObjectCloner exception: " + e);
                }
              }

            }

            temperature *= mySA.alpha;

          } while (temperature > mySA.finalTemperature);

          /* End timings */
          timeAfter = System.currentTimeMillis();
          timeDiff = timeAfter - timeBefore;

          rw.writeThisInfo( mySA.best.getSolutionCost(mySA.totalJobCycles) + "," +
                            mySA.best.getSolutionCompletionTime(mySA) + "," +
                            mySA.best.getEnergy(mySA) + "," +
```

```
                            getScaled((timeDiff * .001),mySA.decimalPrecision) + "," +
                            mySA.stepsPerChange
                           );
          mySA.printSolution(mySA.best, zz + ": BEST SOLUTION FOUND");
          System.out.println("      " + timeDiff + "ms" + " --> " + getScaled((timeDiff *
.001),mySA.decimalPrecision) );

       } // end of for loop for averaging purposes

       rw.closeReportFile();

     } // END of METHOD: Main


} // End of CLASS: Simulated Annealing
```

```java
import java.io.*;

/**
 * @author James Sweeney
 * March, 2007
 *
 * Class that stores the Resource information
 */
public class Resource implements Serializable {

	// Speed of this resource
	protected int resourceSpeed;

	// Cost of this resource
	protected double resourceCost;

	// Percentage of the job assigned to this resource
	protected double resourceJobShare;

	/*
	 * Constructor for the Resource class
	 *
	 * @param speed, representing speed of this resource
	 * @param cost, representing cost of this resource
	 * @param jobShare, representing percentage of the job assigned to this Resource
	 */
	public Resource (int speed, double cost, double jobShare) {
		this.resourceSpeed    = speed;
		this.resourceCost     = cost;
		this.resourceJobShare = jobShare;
	}

	/*********************************************************
	 *                    .:: getResourceSpeed ::.
	 * @return int representing the speed of this Resource
	 *
	 *********************************************************
	 */
	public int getResourceSpeed() {
	  return (this.resourceSpeed);
	}

	/*********************************************************
	 *                    .:: setResourceSpeed ::.
	 * @param s representing the speed of this Resource
	 *
	 *********************************************************
	 */
	public void setResourceSpeed(int s) {
	  this.resourceSpeed = s;
	}

	/*********************************************************
	 *                    .:: getResourceCost ::.
	 * @return int representing the cost of this Resource
	 *
	 *********************************************************
	 */
	public double getResourceCost() {
	  return (this.resourceCost);
	}

	/*********************************************************
	 *                    .:: setResourceCost ::.
	 * @param c representing the cost of this Resource
	 *
	 *********************************************************
	 */
	public void setResourceCost(double c) {
	  this.resourceCost = c;
	}
```

```
        /***************************************************************************
**
         *                   .:: getResourceJobShare ::.
         * @return double representing the percentage of the job assigned to this
Resource
         *

*****************************************************************************
         */
        public double getResourceJobShare() {
          return (this.resourceJobShare);
        }

        /**************************************************************************
         *                   .:: setResourceJobShare ::.
         * @param js representing the percentage of the job assigned to this Resource
         *
         **************************************************************************
         */
        public void setResourceJobShare(double js) {
          this.resourceJobShare = js;
        }

}  // End of CLASS: Resource
```

```
import java.io.*;

/**
 * @author James Sweeney
 * March, 2007
 *
 * Class that stores the Solution information
 */
public class Solution implements Serializable {
        boolean validSolution;

        // Resource storage for each Solution
        protected Resource [] resources;

        /*
         * Constructor for the Resource class
         * @param numResources
         */
        public Solution (int numResources) {
                resources = new Resource [numResources];
                validSolution = true;
        }

        /************************************************************
         *                      .:: getResources ::.
         * @return Resource [], an array of Resources for this Solution
         *
         ************************************************************
         */
        public Resource [] getResources() {
                return (resources);
        }

        /************************************************************
         *                      .:: setValid ::.
         * @param v, boolean value depicting the solution validity
         *
         ************************************************************
         */
        public void setValid(boolean v) {
                validSolution = v;
        }

        /************************************************************
         *                      .:: isValid ::.
         * @return boolean, true if solution is valid, else false
         *
         ************************************************************
         */
        public boolean isValid() {
                return (validSolution);
        }

        /*****************************************************************************
**************
         *                      .:: setThisResource ::.
         * @param thisResource, integer index to the resource to set
         * @param resourceSpeed, integer value representing the speed of this resource
         * @param resourceCost, integer value reprenting the cost of this resource
         * @param resourceJobShare, integer value representin the percentage of job this
resource owns
         *
****************************************************************************************
*****
         */
        public void setThisResource(int thisResource, int resourceSpeed, double
resourceCost, double resourceJobShare) {
                resources[thisResource] = new
Resource(resourceSpeed,resourceCost,resourceJobShare);
        }
```

```
        /*******************************************************************************
***
         *                          .:: getResourceJobShareTotal ::.
         * @return double representing the percentage of the job allocated to all
resources
         *

*******************************************************************************
         */
        public double getResourceJobShareTotal() {
                double resourceTotal = 0;
                for (int i=0;i < this.resources.length;i++) {
                        resourceTotal += resources[i].getResourceJobShare();
                }

                return (SimulatedAnnealing.getScaled(resourceTotal, 2));
        }

        /*************************************************************
         *                          .:: getEnergy ::.
         * @return double representing the Energy Value for this Solution
         *
         *************************************************************
         */
        public double getEnergy(SimulatedAnnealing thisSA) {
          double completionTime = 0;
           double tempCompletionTime = 0;

           for (int i=0;i < this.resources.length;i++) {
             tempCompletionTime = ( thisSA.totalJobCycles *
resources[i].getResourceJobShare() ) / resources[i].getResourceSpeed();
             if (tempCompletionTime > completionTime) {
               completionTime = tempCompletionTime;
             }
           }

          return ( thisSA.getScaled((thisSA.maxJobExecutionTime - completionTime), 2) );
        }

        /***************************************************************************
         *                          .:: getSolutionCompletionTime ::.
         * @return double representing the time it will take for this Solution to run
         *
         ***************************************************************************
         */
        public double getSolutionCompletionTime(SimulatedAnnealing thisSA) {
          double completionTime = 0;
           double tempCompletionTime = 0;

           for (int i=0;i < this.resources.length;i++) {
             tempCompletionTime = ( thisSA.totalJobCycles *
resources[i].getResourceJobShare() ) / resources[i].getResourceSpeed();
             if (tempCompletionTime > completionTime) {
               completionTime = tempCompletionTime;
             }
           }

          return ( thisSA.getScaled((completionTime), 2) );
        }

        /**********************************************************
         *                          .:: getSolutionCost ::.
         * @return double representing the cost to run this Solution
         *
         **********************************************************
         */
        public double getSolutionCost(int totalJobCycles) {
          double solCost = 0;
          for (int i=0;i < this.resources.length;i++) {
```

```
            solCost += ( ( (resources[i].getResourceJobShare() * totalJobCycles) /
resources[i].getResourceSpeed() ) * resources[i].getResourceCost() );
          }
          return (SimulatedAnnealing.getScaled(solCost, 2));
        }

} // End of CLASS: Solution
```

```
/**
 * @author James Sweeney
 * March, 2007
 *
 * SA exception class
 */
public class SAException extends Exception
{
    /**
     * SAException constructor
     * @param msg
     */
    SAException(String msg)
    {
        super(msg);
    }
}
```

```
import java.io.*;
import java.util.*;
import java.awt.*;

 /**
 * @author James Sweeney (Dave Miller code)
 * March, 2007
 *
 *
 */
public class ObjectCloner
{
   // so that nobody can accidentally create an ObjectCloner object
   private ObjectCloner(){}
   // returns a deep copy of an object
   static public Object deepCopy(Object oldObj) throws Exception
   {
      ObjectOutputStream oos = null;
      ObjectInputStream ois = null;
      try
      {
         ByteArrayOutputStream bos =
               new ByteArrayOutputStream(); // A
         oos = new ObjectOutputStream(bos); // B
         // serialize and pass the object
         oos.writeObject(oldObj);   // C
         oos.flush();               // D
         ByteArrayInputStream bin =
               new ByteArrayInputStream(bos.toByteArray()); // E
         ois = new ObjectInputStream(bin);                  // F
         // return the new object
         return ois.readObject(); // G
      }
      catch(Exception e)
      {
         System.out.println("Exception in ObjectCloner = " + e);
         throw(e);
      }
      finally
      {
         oos.close();
         ois.close();
      }
   }

}
```

```
import java.io.*;
import java.util.*;

/**
 * @author James Sweeney
 * March, 2007
 *
 * Class that is used to create a file to be used to write information
 */
public class ReportWriter {

  String fileName = "report.txt";
  File f;
  FileWriter fw;
  PrintWriter pw;

  /*
   * Constructor for the ReportWriter class
   */
  public ReportWriter () {
    try {
      f  = new File(fileName);
      fw = new FileWriter(f);
      pw = new PrintWriter(fw);
    }
    catch(IOException e) {
      System.out.println("Exception writing file:" + e);
    }
  }

    /*************************************
     *        .:: writeThisInfo ::.
     * @param info, String to write to file
     *
     *************************************
     */
    public void writeThisInfo(String info) {
      pw.println(info);
    }

    /*************************************
     *        .:: closeReportFile ::.
     * Method to record the time that the
     * GA ended, and close the file
     *
     *************************************
     */
    public void closeReportFile() {
      pw.close();
    }

}
```

VITA

James P. Sweeney has a Bachelor of Science degree from Davis & Elkins College in Computer Science with a minor in mathematics, 1994, and expects to receive a Master of Science in Computer and Information Sciences from the University of North Florida, August 2007. Dr. Sanjay Ahuja of the University of North Florida is serving as James' thesis advisor. James is currently employed as a senior programmer analyst at Mayo Clinic Jacksonville and has been with the clinic four years. Prior to that, James was a programmer and consultant with a variety of companies including; Alltel Information Services, Merrill Lynch, Prudential, IBM Global Services, JMFE/Southeast Toyota, Convergys, NASA, and the National Radio Astronomy Observatory.

James has on-going interests in grid and distributed computing, stochastic algorithms, and natural language parsing. James has programming experience in Visual Basic, Java, COBOL, Perl, and SQL to name a few. James' academic work has included the use of Pascal, Fortran 90, and Ada, as well. James is a competitive cyclist who enjoys the outdoors as well as all things electronic. Married for the last five years, James has a four-year-old Boxer.