3-9-2012

Shadow Price Guided Genetic Algorithms

Gang Shen
Georgia State University

Follow this and additional works at: http://scholarworks.gsu.edu/cs_diss

Recommended Citation
http://scholarworks.gsu.edu/cs_diss/64

This Dissertation is brought to you for free and open access by the Department of Computer Science at ScholarWorks @ Georgia State University. It has been accepted for inclusion in Computer Science Dissertations by an authorized administrator of ScholarWorks @ Georgia State University. For more information, please contact scholarworks@gsu.edu.
SHADOW PRICE GUIDED GENETIC ALGORITHMS

by

GANG SHEN

Under the Direction of Yan-Qing Zhang

ABSTRACT

The Genetic Algorithm (GA) is a popular global search algorithm. Although it has been used successfully in many fields, there are still performance challenges that prevent GA’s further success. The performance challenges include: difficult to reach optimal solutions for complex problems and take a very long time to solve difficult problems. This dissertation is to research new ways to improve GA’s performance on solution quality and convergence speed. The main focus is to present the concept of shadow price and propose a two-measurement GA. The new algorithm uses the fitness value to measure solutions and shadow price to evaluate components. New shadow price Guided operators are used to achieve good measurable evolutions. Simulation results have shown that the new shadow price Guided genetic algorithm (SGA) is effective in terms of performance and efficient in terms of speed.

INDEX WORDS: Genetic algorithm, Shadow price, Optimization, Performance, Hybrid Algorithm, Linear programming, Heuristic algorithm, k-opt, Traveling salesman problem, Cutting stock problem, Stock reduction problem, Cloud computing, Green computing
SHADOW PRICE GUIDED GENETIC ALGORITHMS

by

GANG SHEN

A Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

in the College of Arts and Sciences

Georgia State University

2012
SHADOW PRICE GUIDED GENETIC ALGORITHMS

by

GANG SHEN

Committee Chair: Yan-Qing Zhang
Committee: Raj Sunderraman
YingShu Li
Yichuan Zhao

Electronic Version Approved:

Office of Graduate Studies
College of Arts and Sciences
Georgia State University
May 2012
I thank my advisor, Dr. Yan-Qing Zhang, for his guidance and help for my Ph.D. study. I truly appreciate the time and patience he spend helping me completing the program in research, publishing, and dissertation work. I also thank Dr. Rajshekhar Sunderraman for advising throughout my study and being a member of dissertation committee. I am grateful for Dr. Yichuan Zhao and Dr. YingShu Li’s review and suggestion of my research work.
# TABLE OF CONTENTS

ACKNOWLEDGMENTS iv

TABLE OF CONTENTS v

LIST OF TABLES viii

LIST OF FIGURES x

LIST OF ABBREVIATIONS xi

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 IMPORTANCE OF THE RESEARCH 4

CHAPTER 3 GENETIC ALGORITHM 6

3.1 Principles of Genetic Algorithm 6

3.2 Opportunities 11

CHAPTER 4 RELATED WORK 13

4.1 Transforming Problem 13

4.2 Improving GA Operators 14

4.3 Adding Local Search 15

4.4 Hybriding with Other Algorithms 17

4.5 Using Parallel Processing 19

4.6 Miscellaneous Approaches 25

CHAPTER 5 DUALITY AND SHADOW PRICE in LINEAR PROGRAMMING 27

5.1 Definition 27

5.2 Shadow Prices in Linear Programming 29

CHAPTER 6 SHADOW PRICE GUIDED GENETIC ALGORITHM 32
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>The Concept</td>
<td>32</td>
</tr>
<tr>
<td>6.2</td>
<td>A Simple Example</td>
<td>34</td>
</tr>
<tr>
<td>6.3</td>
<td>Define Shadow Price</td>
<td>38</td>
</tr>
<tr>
<td>6.4</td>
<td>The Complete Algorithm</td>
<td>40</td>
</tr>
<tr>
<td>7</td>
<td>Introduction</td>
<td>42</td>
</tr>
<tr>
<td>7.2</td>
<td>Problem Definition</td>
<td>42</td>
</tr>
<tr>
<td>7.3</td>
<td>Shadow Price Definition</td>
<td>43</td>
</tr>
<tr>
<td>7.4</td>
<td>Shadow Price Guided Mutation Operator</td>
<td>45</td>
</tr>
<tr>
<td>7.5</td>
<td>Shadow Price Guided Crossover Operator</td>
<td>46</td>
</tr>
<tr>
<td>7.6</td>
<td>Solution Validation</td>
<td>46</td>
</tr>
<tr>
<td>7.7</td>
<td>Other Techniques</td>
<td>48</td>
</tr>
<tr>
<td>7.8</td>
<td>Experiments</td>
<td>49</td>
</tr>
<tr>
<td>7.9</td>
<td>Summary</td>
<td>50</td>
</tr>
<tr>
<td>8</td>
<td>Introduction</td>
<td>51</td>
</tr>
<tr>
<td>8.2</td>
<td>Problem Definition</td>
<td>52</td>
</tr>
<tr>
<td>8.3</td>
<td>Basic Terminologies</td>
<td>54</td>
</tr>
<tr>
<td>8.4</td>
<td>Shadow Price Definition</td>
<td>55</td>
</tr>
<tr>
<td>8.5</td>
<td>Shadow Price Guided Mutation Operator</td>
<td>56</td>
</tr>
<tr>
<td>8.6</td>
<td>Shadow Price Guided Crossover Operator</td>
<td>58</td>
</tr>
<tr>
<td>8.7</td>
<td>Experiments</td>
<td>59</td>
</tr>
<tr>
<td>8.8</td>
<td>Results Analysis</td>
<td>69</td>
</tr>
<tr>
<td>8.9</td>
<td>Production Consideration</td>
<td>71</td>
</tr>
<tr>
<td>8.10</td>
<td>Summary</td>
<td>73</td>
</tr>
</tbody>
</table>
LIST OF TABLES

Table 6.1 Simulation results 37
Table 6.2 Distribution of the Number of Generations 38
Table 6.3 Distribution of the Fitness Values 39
Table 7.1 Distance Matrix for gr17.tsp 44
Table 7.2 Comparison with the Ray, Bandyopadhyay, and Pal (2004) 49
Table 7.3 Comparison with the Zhong, Zhang, and Chen 50
Table 7.4 Comparison with the Wong, Low, and Chong (2008) 50
Table 8.1 Test results for the CSP with multiple stock lengths 53
Table 8.2 Test results for the CSP with single stock length 53
Table 8.3 Sample problem, the stock length is 14 54
Table 8.4 Test case summary 63
Table 8.5 Mean Fitness Value Comparison 63
Table 8.6 Total Waste Comparison 65
Table 8.7 Number of Stocks with Waste Comparison 66
Table 8.8 Speed Comparison 68
Table 8.9 Mean fitness value and number of stocks used 72
Table 8.10 Total waste, number of stocks with waste, and distinct pattern count 72
Table 9.1 A Sample Task Schedule 77
Table 9.2 Published Processor Specification 88
Table 9.3 Energy Consumption Comparison 89
Table 9.4 Speed Comparison 91
Table 9.5 SPGA Time Improvement over GA for 10 Processors 93
Table 9.6 SPGA Time Improvement over GA for 20 Processors 93
Table 9.7 SPGA Time Improvement over GA for 30 Processors 93
Table 9.8 SPGA Time Improvement over GA for 40 Processors 93
Table 9.9 SPGA Time Improvement over GA for 50 Processors 94
Table 9.10 SPGA Search Speed Improvement in Time(s) 94
Table 9.11 SPGA Search Speed Improvement in Generations 94
Table 10.1 Sample CSP 101
Table 10.2 GA Result of Sample CSP 101
Table 10.3 Result from Using the Gilmore and Gomory LP Algorithm 102
Table 10.4 Convert LP Solutions to Integer Using Stock 1376 103
Table 10.5 Convert LP Solutions to Integer Using Stock 1392 103
Table 10.6 Comparison Study on Item Variations 106
Table 10.7 Comparison Study on Stock Count Variations 106
Table 10.8 Production Problem Run Result 107
LIST OF FIGURES

Figure 3.1 Genetic Algorithm 10
Figure 5.1 Gilmore and Gomory LP Algorithm 31
Figure 6.1 New GA Framework with Shadow Price Guided Operators 40
Figure 7.1 A Sample Tour 47
Figure 7.2 Result from Mutation 48
Figure 8.1 Algorithm B’s mutation operator 60
Figure 8.2 Algorithm C’s mutation operator 61
Figure 8.3 Algorithm D’s mutation operator 62
Figure 8.4 Average Mean Fitness Value Comparison 64
Figure 8.5 Maximum Mean Fitness Value Comparison 64
Figure 8.6 Average Total Waste Comparison 65
Figure 8.7 Minimum Total Waste Comparison 66
Figure 8.8 Average Number of Stocks with Waste Comparison 67
Figure 8.9 Minimum Number of Stocks with Waste Comparison 67
Figure 8.10 Best Solution Found Generation Comparison 68
Figure 8.11 Time(s) Comparison 69
LIST OF ABBREVIATIONS

Adaptive Hill-Climbing Crossover Local Search  AHCXLS
Ant Colony Optimization  ACO
Bee Colony Optimization  BCO
Cutting Stock Problem  CSP
Discrete Particle Swarm Optimization  DPSO
Evolutionary Algorithm  EA
Field Programmable Gate Array  FPGA
Genetic Algorithm  GA
Group Crossover  BPCX
Infeasibility Driven Evolutionary Algorithm  IDEA
Integer Linear Programming  ILP
KiloWatt-Hours  kWh
Linear Programming  LP
Million Instructions Per Second  MIPS
Minimizing Stock Mix Problem  MSMP
Mixed Integer Linear Programming  MILP
Mixed Integer Programming  MIP
Neural Network  NN
Parallel Genetic Algorithm  PGA
Particle Swarm Optimization  PSO
Shadow Price Guided GA  SGA
Simulated Annealing  SA
<table>
<thead>
<tr>
<th>Term</th>
<th>Acronym</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stock Reduction Problem</td>
<td>SRP</td>
</tr>
<tr>
<td>System on a Programmable Chip</td>
<td>SOPC</td>
</tr>
<tr>
<td>Traveling Salesman Problem</td>
<td>TSP</td>
</tr>
<tr>
<td>Uniform Grouping Crossover</td>
<td>UGCX</td>
</tr>
</tbody>
</table>
CHAPTER 1 INTRODUCTION

Optimization is to search for the best solution from a domain of feasible solutions. In the simplest form, it is to find the minimal or maximal value of a function while satisfying a set of constraints. It is a process of searching for the best solutions using certain algorithms and techniques. One most cited example of optimization is to find the best way to achieve maximum profits utilizing limited resources.

Integer optimization is a special branch of general optimization that requires integer solutions for the problem. This constraint only limits the final result in integer and does not pose integer requirement to intermediate solutions. Thus, the intermediate solution can be in integer or real. This constraint is often modeled from real life problems. For example, job scheduling is an integer optimization problem; product can only be produced in integer units.

Other complicated constraints in optimizations include, complex objective functions, multiple objectives optimization, etc. Objective functions can be linear, polynomial, table lookup, etc. There can be multiple objective functions to be optimized in the same time.

Linear programming (LP) is the classic optimization algorithm. It is very efficient and widely used in production especially for large complex linear optimization problems. But it is limited to linear objective functions and constrains. The general LP results are in fractions. Integer linear programming (ILP) and Mixed Integer Linear Programming (MIP) are special cases of LP that provide integer solutions. Although they can solve many practical problems, ILP and MIP are less efficient than LP and difficult to solve. Both ILP and MIP are extensions of classic LP. They typically follow classic LP technique and add additional steps, algorithms (such as branch and bound, cutting plane method, etc.) to produce integer solutions. Fractions are
commonly used in the algorithms’ intermediate solutions and these fractional intermediate solutions are not valid solutions.

Genetic Algorithm (GA) (John Holland, 1975, 1992) is a bio-inspired global search algorithm that mimics nature’s evolution process. It is a multi-point, reward-based search algorithm. In the search process, there are multiple valid solutions evolving forward together. The reward-based search refers to the fact that only elite solutions participating next generation’s evolution. It’s an integer intrinsic search process that fits integer optimization problem very well. Unlike invalid fractional intermediate solutions in the LP search process, every solution in GA’s search process are valid integer solutions although they may not be the optimal solutions. The reward-based approach also suits for multi-objective optimizations since the elitism only requires comparing the objective function regardless the function is linear or non-linear.

GA has been used successfully in many fields. Recent survey suggests that at least thirty-six human-competitive results were produced by genetic programming (Koza et al. 2005). It is a very straightforward algorithm and can be implemented rather quickly.

The challenges for GA’s performances are solution quality and search time. These two concerns impede the practical applications of the algorithm. GA is a population based search algorithm and there are many solutions in each generation. Solutions in the generation need to be involved in one or more evolution operations in each generation to move forward. Based on the size of the population, huge amount of calculation may be needed for each generation. Compound with necessary randomness in the search process, GA can take very long time to find optimal solutions.

Furthermore, GA may not always provide the optimal solutions. GA generally depends on generations of evolution to move the solution forward. The most common stopping criterion
is to limit the maximum number of generations, maximum allowed searching time, or solution reaches acceptable quality. GA cannot prove the final solution is optimal or not. So, there is certain randomness in the quality of the final solutions.

My research focuses on improving GA’s performance in both solution quality and search speed. GA only measures the solution fitness value. The evolution operators are mostly randomly applied since there is no measurement on the components. I propose using the “Shadow Price” concept to measure the components of the solution in the GA search process. I can improve GA operators using the shadow price. Thus, I establish a two-measurement GA. The fitness value is used to measure solution and the shadow price is used to measure component within a solution. I will propose the theory and use it to solve several classic NP hard problems.
CHAPTER 2 IMPORTANCE OF THE RESEARCH

There are tremendous social and economic values in finding optimal solutions. The value of best utilizing limited resources to maximize social benefit can be seen in daily life or in the event of disaster. For example, it is very important to most efficiently use limited transportation equipment and crew to move stranded passengers in the event of large-scale flight interruption such as that caused by volcano eruptions, terrorist attacks, etc.

Significant economic value of optimization is everywhere. For example, trimming rolls for paper machine is a typical optimization problem and referred as the cutting stock problem (CSP). The goal is to improve trim efficiency. A 300 inch wide paper machine can produce half million tons of medium weight paper a year. If the price is 600 dollars per ton, the total value of the paper is 300 million dollars. A one percent trim efficiency improvement is equivalent to 3 million dollars a year for this machine. In a paper box plant, trimming corrugator is another CSP and the trim efficiency improvement worth even more since it trims multiple layers of paper. For a medium sized paper product company that operates multiple paper machines and paper box plants, a minor trim efficiency improvement has hug economic impact.

GA is a new global optimization search method that has been used successfully in many fields (Koza, Keane, Streeter, Mydlowec, Yu, & Lanza, 2005). Comparing to other complex optimization algorithms such as LP, GA can be used quickly to model the problem and solve it with excellent results. It does not add many constraints to the problem.

However, the performance that includes both the solution quality and convergence speed limits GA’s further success in many fields. To reach optimal or near optimal solutions, GA needs many generations of evolution and takes much more time than other algorithms such as LP based algorithms. GA’s performance is acceptable in many situations, such as static job scheduling,
airline flight and crew scheduling, pre-production forecasting, post-production analysis, etc. In other areas where real time or near real time optimization is need, such as real time job scheduling, flight position control, production adjustment, etc., GA’s performance may not be acceptable.

With the guidance from my advisors, I search for ways to improve GA’s performance. I mainly focus on establish a secondary measurement that applies to components of the solution. The secondary measurement acts as a complement to the solution’s fitness value measurement. This new component measurement can improve GA operators and greatly improve GA’s performance.
CHAPTER 3 GENETIC ALGORITHM

3.1 Principles of Genetic Algorithm

GA (Figure 3.1) is a reward based multi solution search algorithm. It is a branch of bio inspired evolutionary algorithm (EA). Comparing to other single solution search algorithms such as LP, k-opt algorithm, etc., there are multiple feasible solutions concurrently evolve toward the best solution in the GA search process. The multiple generation search process ensures GA a global search algorithm.

There are generally four major phases in the GA search process, initialization, evolution, selection, and termination.

In the initialization phase, a startup solution population is created. Random generating initial solutions are commonly used. All solutions in the population have to be feasible. The population varies based on the problem to be solved and computing power available. It can be range from 10s to hundreds or thousands. The initial solution shall spread out in the search space. The more diverse the initial solutions, the better performance GA can achieve since it ensures global search.

The evolution phase evolves current generation forward. The goal is to generate new solutions based on current available solutions and hopefully the newly generated solutions are better than current ones. There are two major methods to generate new solutions, binary operator crossover and unary operator mutation.

The crossover operator mimics parents producing child in nature. Two solutions are selected from the current generation’s solution pool and function as the “parents” to breed. Based on problem domain, a breeding method is used to create the “child” solution. The child solution inherits certain attributes from both parents. Typical, a certain sub population is selected to
participate the crossover operation. There are multiple ways to selection parents. The general goal is to create a child solution that poses good characteristics of both parents and better than both parents.

To generate a new solution, the unary mutation operator modifies the state(s) of one or a small number of components of an existing solution. Most time, the newly generated solution is much different than the original solution and may not even be a valid solution. Based on the problem, the mutation operator may or may not generate a better solution. But it is a very important operator that functions as an insurance of a global search. That is, it can bring search to an area of search space that has not been visited before. It is especially important when GA search stuck to a local optimal solution. In this case, mutation operator can lead search to another area and effectively breaks the local trap. There are many methods to select which solution to mutate and which component(s) to mutate.

Aside from mutation and crossover operators, several new solutions are randomly generated in the evolution process in general as well. This is to further broaden the search space and serves as an extra insurance of a global search.

After evolution phase generates enough new solutions, selection phase evaluates each solution and select good solutions to create the next generation to continue evolution. It is also called elitism. A fitness function is typically used to evaluate and compare solutions. Based on different problem, the fitness function can be a simple linear function, a polynomial function, a table look up, or a very complex optimization problem itself. As one of the stopping criteria in general, this fitness function is also used to measure whether solutions meet predefined threshold or not. There are many different approaches to select candidate solutions to participate next
generation. Selecting good solutions can ensure search towards optimal solutions. Selecting random solution ensures global search and avoid local optimal trap.

The termination phase evaluates the “goodness” of current solutions and decides whether continue to evolve or stop. Since the optimal solution(s) is unknown for most problems, predefined acceptable solution (defined by fitness function) can be used as one terminating criterion. Maximum number of generations or maximum allowed time is also commonly used as stopping criteria. Search progress is another barometer to evaluate GA’s searching process. It can be measured by x progress in y generations. Combination of criteria or single criterion can be used as the termination condition for search. After search stops, the best solution represents the current search result. It can be optimal or near optimal based on the stopping criteria.

Random selection is used throughout the GA algorithm. It is used to select solution participating mutation operation, crossover operation, or to participating next generation’s evolution. There are two classic random selection method, roulette wheel and tournament.

In the roulette wheel selection, each candidate is assigned a probability of getting selected. The sum of all candidates’ probabilities is equal to one. The probability of a solution is related to its attribute(s). The fitness value can be a good choice. Obviously, solution with a large probability has a better chance to be selected. The solution with small probability has a less chance to be selected but still can be selected.

The tournament selection conducts one stage or multi stage tournament. It starts with randomly organize candidates into groups. Within each group, a winning candidate is selected based on probabilities assigned to the candidates. One way (Tournament Selection, 2010) is to assign the best candidate a probability of p, the second best is assigned to p(1-p), the third best is assigned to p(1-p)^2, etc. Roulette wheel selection can also be used here. Winners from each
group are random grouped again for next stage tournament. The process repeats until desired number of candidates are selected.

In summary, there are three GA operators that produce new solutions in the evolution phase. They are mutation, crossover, and randomize. The mutation operator changes the state of a component of a solution to move it closer to the optimal solution. The crossover operator tries to create a better new solution from two existing solutions. Randomize operator introduces new solutions. The initialization phase builds up the initial feasible solution pool to start off the search process. The selection phase creates new generation of solutions to evolve forward from current all available solutions. The termination phase ends the search process when predefined criteria are met.
Initialization
Populate solution set with random feasible solutions

Evolution
Pick a solution to mutate
Mutate the solution
Adjust solution to a feasible solution
Add the new solution to the population
Repeat for n times

Crossover
Pick two solutions to crossover
Create a new solution from parent solutions
Adjust solution to a feasible solution
Add the new solution to the population
Repeat for m times

Randomize
Create a random new solution
Add the new solution to the population
Repeat for k times

Selection
Select solutions to create next generation

Termination
Is stopping criteria met?
Yes
No

Start
Stop
Yes
No

Figure 3.1 Genetic Algorithm
3.2 Opportunities

The main challenge that prevents GA’s further success is its performance issue. This includes solution quality and search speed.

Randomness is used throughout the search process, such as building up the initial solutions, choosing candidates to apply mutation or crossover operations, selecting solutions to form next generations. It is also used in the GA operators. Mutation operator randomly selects a component to mutate and mutate to a random state. Crossover operator randomly selects one or many crossover point(s) to create new solution. All these randomness guides GA to randomly select one or more solutions to evolve and move them to random state. The GA does not have a uniformed search direction. It searches multiple directions in the same time. The selection ensures GA search moving towards optimal solutions since better solutions are added into generations to further evolve. It moves solution population closer to optimal solutions from generation to generations in general.

Randomness is absolutely necessary to GA. It ensures GA a global search algorithm and avoid local optimal trap. But it also slows down the search process since randomness can lead search to all directions and cause many unnecessary searches. In the worst case, the randomness can stall the search process and leads to sub optimal solutions, or visits all viable solutions.

There is a large amount of calculation in the GA search process. Within each generation of search, each individual solution has to go through the process of inspection, evolution operation, fitness value evaluation, and selection. It really takes much more time to process all solutions in a generation than other single solution search algorithms such as heuristic, LP, etc. Multiplying by many generations of evolution (synchronized or desynchronized), the total calculation amount is very large. Parallel computing techniques can certainly help. But for large
complex GA search problems, where there are thousands of solutions in each generation and search for thousands of generations, modern parallel computing techniques still cannot make decisive impacts.

The other time consuming effort in the GA search process is the fitness function calculation. For a simple problem, the fitness function can be a polynomial function which calculation is rather straightforward and quick. However, the fitness function can be quite complex in certain cases. For example, the fitness function can be a complicated matrix operation or an optimization problem itself. Although GA poses little constraint on the optimization problem, complex fitness function can add significant search time for complex problem since the fitness function has to be calculated for all solutions.

Because GA takes long time to search, time constraint and/or generation constraint are typically used as the stopping criteria. The idea is to get the best answer, which may not be the optimal solution, within an acceptable time frame. This is the consequence from the GA’s slow search speed. GA can stop searching prematurely and provide inferior result. The solution quality is suffered due to the search speed issue.
CHAPTER 4 RELATED WORK

Since its introduction, much work has been dedicated to study GA’s performance. Ishibuchi, Nojima, and Tsutomu (2006) studied the performance between single-objective GA and multi-objective GA. Using multi-objective knapsack problem, they demonstrated that multi-objective GA outperformed single-objective GA for low count of objectives problem. This is because multi-objective GA can easily move away from local optimal. But when the objective count increases, the multi-objective GA became less efficient. Simoncini, Collard, Verel, and Clergue (2007) studied the impact of selection pressure to the performance of GA. They confirmed that the selection pressure influence the GA performance using the anisotropic selection and the stochastic tournament selection. More accurately compare and measure GA’s performance has also been studied (Ang, Chong, & Li, 2002; Deng, Huang, & Tang, 2007).

Various innovations have been applied to GA to improve its performance. These approaches can be roughly categorized as 1) transforming problem, 2) improving GA operators, 3) adding local search, 4) hybriding with other algorithms, 5) using parallel processing, and 6) miscellaneous approaches.

4.1 Transforming Problem

Divide and conquer has long been used to solve complex problems. The idea is to divide a large complex problem into smaller simpler problems. After solving each individual smaller problem, results are combined to get the final solution. Zhang and Li (2007) applied the divide and conquer theory into the EA. They decomposed the multi-objective optimization problem into related scalar optimization sub problems. The scalar simpler sub problems are optimized simultaneously and results from them are combined as the final solution. By decomposing, the
computation complexity is reduced greatly. Their experiments proved the new algorithm is very efficient for 0-1 knapsack problems and continuous multi objective optimization problems.

Approximating is useful when certain tolerance is allowed in the value. This has important practical values in many fields where tolerance is allowed or near optimal solution is accepted. Paenke, Branke, and Jin (2006) and Regis and Shoemaker (2004) addressed the fitness function’s computation complexity problem by substituting it with an approximate modal. Much time can be saved by calculating simpler approximate fitness function. Their experiments proved that the approximating is efficient and result qualities are acceptable.

The goal of problem transformation is to optimize one or more smaller simpler problem(s) instead directly working on the more complex larger problems. Combining smaller problems’ result, the final solution can be provided for the original problem. By optimizing less computation intensive smaller simpler problems and reducing search space, the algorithm can find optimal or near optimal solutions quicker.

4.2 Improving GA Operators

Syswerda (1991) introduced a new order crossover operation to preserver some order information from both parents. It starts with randomly selecting n components from a parent. Other non-selected components are passed to the child solution directly from the other parent. They shall maintain their position like their parent. The selected n components are inserted into the child solution based on their order from the first parent to complete the solution. For example, there are two solutions S1= (A, B, C, D, E, F), S2= (B, F, E, D, C, A). If (B, D, E) is randomly selected to preserve order from S1, the initial child solution from S2 using non-selected components is C= (_, F, _, _, C, A). Adding selected components back in, the final child solution from the crossover is C= (B, F, D, E, C, A).
Nagata and Kobayashi (1999) introduced an edge assembly crossover operator to preserve the edge information from both parents. They started with building AB circles (parents are named A, B) by selecting connecting edges from each parent alternately. The result is a set of AB circles. A heuristic algorithm was used to connecting all AB circles into a final solution. They applied the edge assembly crossover operator to the Traveling Salesman Problem (TSP) and achieved good results.

Zhao, Dong, Li, and Yang (2008) added the pheromone concept from the Ant Colony Optimization Algorithm (ACO) to enhance the crossover operation. They also used heuristic method to solve the multiple-traveling salesman problem (mTSP). In their crossover operator, the heuristic method use edge length and next city information. To decide which city to visit, the child will look at both parents’ next visiting cities. If both cities from parents have already been visited in the current solution, pheromone trail is used to select next visiting city.

The objective of improving GA operators is to pass some information from parent(s) to the newly generated the child. There is no evaluation of whether the information passed actually will move the search to the optimal solutions or not. It relies on the selection mechanism to control the evolution towards the optimal since the selection will filter out inferior solutions. This approach works in general at the cost of more calculations.

4.3 Adding Local Search

Noman and Iba (2008) designed a strategy adaptive hill-climbing crossover local search (AHCXLS) in their EA. It used a simple hill-climbing algorithm to determine the search length adaptively. It took feedback from search result to determine the search length. In their algorithm, crossover is repeated until no better solution can be generated. They noticed, “there is no straightforward method of selecting the most promising individuals for XLS”. So, they opted to
crossover with one good candidate based on the fitness value and one randomly selected solution.

Yang and Liu (2008) applied the local search to the solutions are have gone through evolution operation. They searched the neighbor of the solution and replaced it with the best is can find. Experiments shown the performance were much improved.

Tsai, Yang, and Kao (2002) added neighbor-join to the edge assembly crossover operation. The neighbor-join operator will generate new solutions by using edges from other solutions or generate new edges based on some heuristic information. The goal is to improve solution quality.

Zhao, Dong, Li, and Yang (2008) used local search function to replace the mutation operation. They used three types of local search to solve the mTSP problem. 1) Relocation moves one city to a different location in the solution. 2) Exchange swaps positions of two cities. 3) 2-opt swaps end portions of two routes. They rotated these three local search operators. These were used in addition to their improvement on the crossover operator described in the above section.

Tseng and Chen (2009) used a two-phase genetic local search algorithm. The genetic algorithm was used to search for promising areas in the first phase. The local search was used to find the best solutions for the problem. Kaur and Murugappan (2008) used the nearest neighbor as the local search algorithm to help populate initial solution pool for the GA. This way, the algorithm starts from some better positions. Xuan and Li (2005) used local optimizer, 2-opt, to optimize every solution after evolution. Zhang and Koduru (2005) used steepest ascent hill climbing as the local search algorithm and also used blend crossover to improve GA’s performance.
In this category, GA is improved by adding local search capability. The local search can be used to enhance crossover operator, mutation operator, initial population build up, and optimize resulting solutions from the evolution. Strictly speaking, adding local search to GA results a hybrid algorithm. Since local search is used more often, I give it its own separate category.

4.4 Hybriding with Other Algorithms

There are many hybrid algorithms that combine GA with many other search algorithms such as Dantzig(1963) Simplex method, Nelder- Mead simplex method (Koduru, Dong, Das, Welch, Roe, & Charbit, 2008; Nelder & Mead, 1965), etc. Most time, these additional search algorithms perform local search while GA conducts global search. They are either used to optimize solutions that have been applied GA operators (Koduru, Das, Welch, Roe, & Lopez- Dee, 2005; Robin, Orzati, Moreno, Otte, & Bachtold, 2003) or used in conjunction with the GA operators to improve its performance (Bersini, 2002; Tsutsui, Goldberg, & Sastry, 2001). Although these are very important approaches, GA is the main algorithm and other algorithms are simply assisting GA.

LP, on the other hand, has many ways to work with GA to create efficient hybrid algorithms. Bredstrom, Carlsson, and Ronqvist (2005) developed models and methods that address the combined supply chain and production-planning problem. They developed a mixed-integer-programming (MIP) model and solved the model using a heuristic solution based on branch and bound. The model typically takes hours to solve. So, they created a GA algorithm to solve the model. Each solution in the GA is a schedule and they used LP to make other decisions for the schedule such as deciding shipping quantity in this case. To further speed up the LP computation, they created a performance LP model to approximate the solution. Similar
approaches had also been used by El-Araby, Yorino, and Zoka, (2005), El-Araby, Yorino, and Sasaki (2002), and Leou (2008) where GAs were used to derive solution and successive linear programming (SLP) and Simplex method were used to obtain the fitness values. In these approaches, GA is the main driver of the program to conduct global search. LP is the help algorithm that optimizes each solution and calculates fitness value.

LP has also been used to lead the search in the LP and GA hybrid algorithms. To design the optimal fuel-cell-based energy network, Hayashi, Takeuchi, and Nozaki (2008) designed a hybrid algorithm to account for the differences of equipment. Some energy equipment’s CO$_2$ emission can be express in linear format and some cannot. LP cannot be used to precisely optimize the overall modal. The hybrid algorithm used LP to design the optimal configuration and evaluate the fitness function for equipment. GA takes the best LP configuration and optimizes the overall installation while take in consideration of each equipment different CO$_2$ emission characteristics. To design an optimal open magnetic resonance imaging magnet, Wang, Xu, Dai, Zhao, Yan, and Kim (2009) first used LP to design the source current distribution and used GA to optimize the section size of the cross-section of the coil. Pandey, Dong, Agrawal, and Sivalingam (2007), Garg, Konugurthi, and Buyya (2009) designed similar hybrid algorithms that use LP to generate initial solutions and have GA to fine-tune the solution. Although this kind of LP/GA hybrid algorithm is straightforward conceptually, LP is used to create initial solutions and GA searches for the final best solutions, it is a very efficient approach. By using LP optimized solutions, GA is really starting the search from near optimal solutions. Thus, GA’s search time is reduced significantly and can quickly reach optimal solutions. In certain cases, GA can simply fine tune the LP optimized solutions.
Mantovani, Modesto, and Garcia (2001) combined GA and LP in a more efficient way. They divided the reactive planning optimization problem into operating and planning sub problems. The operating sub problem, a nonlinear and no convex problem, was solved by GA. The planning sub problem, using real variables and linear problem, was solved by LP. Similar approach was also used by Feng, Wang, and Li (2009).

LP and GA have different strengths. LP is very efficient in solving linear, non-integer problems. GA has very little constraints on the objective function. LP can typically reach optimal solution in a very short period of time. GA is slower. Integer LP is less efficient. Combining LP and GA can typically reach optimal solutions for integer optimization problems quickly.

4.5 Using Parallel Processing

Parallel implementations of genetic algorithm (Alba & Tomassini, 2002; Liang, Chung, Wong, & Duan, 2007; Massa et al., 2005; Ortiz-Garcia et al. 2009) have also been proposed and experimented. There are a number of experiments, published papers with good results. With the decreasing cost of computing resource, parallel algorithm became more and more appealing as one of the methods to improve algorithm efficiency. There are many different ways to implement parallel GA (PGA).

Hardware implementation of PGA refers to one kind of implementation in which partial or complete algorithm (binary code) is encoded into the computer chips. The computer chips become specialized for PGA purpose only. The code in the computer chips runs based on computer clock cycles without software control. The common benefit of this implementation is speed since there is no software involved. Jelodar, Kamal, Fakhraie, and Ahmadabadi (2006) experimented a hardware based PGA using System-on-a-Programmable-Chip (SOPC). They implemented three genetic algorithms on SOPC using three different architectures: a) Standard
single processor genetic algorithm. b) Parallel GA using Master/Slave architecture c) Coarse-grained PGA. To overcome the inflexibility of hardware based algorithm implementation, the authors designed a mixed implementation approach: fitness evaluation in software and all other GA/PGA elements in hardware. This approach allows complex fitness functions required by different category of problems. The experiments result showed the hardware based PGA is 50 times faster than software based PGA.

Scott, Samal, and Seth (1995) presented another working hardware based GA using FPGA (field programmable gate array). There are two phases in the process. In phase I, user enters the parameters of GA and the fitness function, system translate them into hardware image and programs the FPGA. In phase II, upon front-end give a “go” signal, programmed FPGA run the algorithms without any software interruption. When it’s finished, “done” signal was send to the front-end. Finally, Front-end read the result. The authors’ experiment showed speedup factor about 15.

Software implementation refers to PGA implementations where the algorithms run on common computing resources without modify any underline hardware. Typically, there are a group of general-purpose computers working together to implement PGA. There are four models, 1) Global (master/slave) Model, 2) Fine-Grained Model, 3) Coarse-Grained Model, and 4) Hybrid Model.

Cantu-Paz (1997) published one of the frequent cited papers on the global model of PGA. Based on the principle of divide and conquer, the classic global model uses one global population and divides the task of evaluating fitness values of chromosomes among multiple processors. In the model, there is a master processor that controls the whole process. The PGA algorithm is very similar with serial GA. The master processor starts the PGA process, it
initializes the population, and send chromosomes to multiple processors (slaves) to evaluate fitness value. After receive result from slave processors, master process performance all other GA operators, such as selection, mutation, crossover, etc. With newly created population, master processor sends chromosomes to slave processors to evaluate again. The process repeats until the goal is satisfied.

Benkhider, Baba-Ali, and Drias (2007) proposed a generation less concept on GA and two variation of general PGA model. The new GA mimic human population where there is general concept of generation, no distinct clear-cut separation of generation and multiple generations coexist in the same time. The new GA assigns each chromosome an effective start and end time, i.e. a life span. Each chromosome would be replaced after it past its assigned end time. In the meanwhile, new chromosomes were “born” and added to the population. They proposed two new variations of global PGA. In the semi-asynchronous parallel approach, there are two separate processes on the master processor. One is responsible for assigning chromosomes to slave processors to evaluation and receiving results from them. The other one is responsible of creating new chromosomes. The two processes works concurrently. Main algorithm suspends when these two processes start to work and only resumes until both processes complete their work. All GA operators are blocked when these two processes are active. So, it is a semi asynchronous method. In the asynchronous master/slave approach, the two processes do not block any other process. The other process is the main process. It’s the main process that responsible for all GA operations (selection, mutation, crossover, etc.). It’s also responsible for creating new chromosomes. Both processes work independent of each other and only exchange chromosomes when necessary. Thus, this is complete asynchronous approach.
The fine-grained architecture targets massive parallel computers. In this architecture, there is only one population in the algorithm just like the global PGA architecture. There is no master processor. There are a lot of inter-connected processors. They are connected in multiple ways and most common is the grid structure. Each processor is responsible for a very small population of chromosomes. Each processor executes a serial GA on its own population and exchange result with neighbor processors. The ideal case is to have only one individual for every processing element available. The efficient communication among interconnected node makes the PGA very fast.

Lee, Park, and Kim (2000) proposed a binary tree structure to connect processors. Each processor forwards its best individual to two next level processors and receives one from the top processor. This is one-direction propagation. This slows down the chromosome migration rate. And the tree structure is dynamic generated based on the position of the best chromosome. They tested their proposal on CrayT3E with 64 processors and showed better performance. Li and Kirley (2002) introduced a new concept “Percolation” into fine-grained PGA architecture. The goal is to ease the selection pressure. They introduced a “seeding” method to the PGA in the fine-grained architecture. When algorithm starts, a large number of random chosen processors start with a chromosome and neighboring processors forms demes. With the process evolving, new processors become active and assigned with chromosomes. New processors join neighboring demes to form larger demes. Eventually, all processors are active and forms one deme. This process forms demes slowly and dynamically. There is no predefined size of deme. This approach controls the rate of migration. Population diversity is maintained and high quality solutions shall spread to all processors gradually.
Coarse-grained parallel genetic algorithm model uses multiple populations that evolve separately and exchange individuals occasionally. It is also referred as multi-deme or distributed PGAs. The basic idea of coarse-grained model is to divide the search space into several sub-populations and assign each participating processor a sub-population. Each processor evolves its population forward till goals are met. In the process, processors may exchange some good chromosomes for speed up purpose. Although one processor may responsible of divide the initial population to start the process and collect results at the end, there is no master processor that controls each processor. Matsumura, Nakamura, Miyazato, Onaga, and Okech (1997) experimented on ring, torus, and hypercube topologies. They concluded that Ring topology and emigrant method provide the best result.

In an attempt to use cycle-steal method to harvest the computing power that scattered over the Internet, Berntsson and Tang (2003) studied the coarse-grained architecture of PGA. They conducted multiple experiments with different topologies, different migration rate, different migration intervals and different failure scenarios. They used 4 faster processors and 4 slow processors to build a heterogeneous computing network. To work with Internet's latency and bandwidth problems, they concluded that a small migration rate with long migration intervals and a fully connected topology would be the best choice.

The hybrid model, a combination of different model of PGA, is a new model that results in algorithms that have the benefits of different PGA models. The new model may show better performance than any of the models alone. The combined model is more complex and difficult to program. But they do not introduce new analytic problems, and it can be useful when working with complex applications. The combination can varies, such as coarse-grained with global model, coarse-grained model with coarse-grained model, coarse-grained model with fine grained
model, etc. The combination does not limit to within the PGA models. New models can include other optimization algorithms, such as LP, nearest neighbor algorithm, etc.

Lee, Park, & Kim (2001) proposed a hybrid PGA architecture to address two issues, to connect large amount of processors in the PGA calculation and to control the migration speed to achieve better result (alleviating super chromosome dominating solution space issue). High-level processors used coarse-grained model to connect to each other. Chromosome migration rate is low. Lower level processors using fine-grained PGA model and the migration rate is high. The fine-grained PGA used binary tree model to organize. The tree is built dynamically based on the location of the best solution and communication is one directional, from top to bottom only. The tree structure decides the processor to receive chromosome from or processors to send to. To further minimize the dominating solution issue, limits are put on migration policy.

Zhao, Man, Wan, & Bi (2008) introduced a multi-agent hybrid parallel genetic algorithm. They combined global PGA model with coarse-grained PGA model. In the new model, there are master agents and slave agents. Each master agent (M-agent) is in charge of several slave agents (A-agent) to form a global master slave PGA model. The M-agent responsible for the evolution process and A-agent helps with the parallel calculation. Several M-agents connect to each other to form a coarse grained PGA model.

Genetic algorithm is a good candidate to be parallelized. The simple algorithm made it easy to be implemented and tested. It’s a fault tolerant algorithm since its population can be large. PGA can make GA fast and efficient. A good design of PGA shall have following attributes. It fully utilizes available computing resources. Communication is efficient and simple. Migration policy ensures a diverse sub populations and fast to converge to the global optimal solution.
4.6 Miscellaneous Approaches

Yuen, S.Y., & Chow (2009) used a binary space partitioning tree to archive the solutions that GA has visited. Based on the binary tree, they designed a novel adaptive mutation operator. The mutation operation is replaced by searching the tree. They start with locating the solution to be mutated in the tree. Then, they find the nearest neighbor-unvisited subspace of the solution and random select one as the mutation result. If all nearest neighbor solution has been visited, backtrack to the parent and repeat the process. In the meanwhile, fully visited sub tree can be trimmed from the tree. The algorithm visits a nearest unvisited neighbor subspace and randomly finds an unvisited solution in it. They named the algorithm as “A Genetic Algorithm That Adaptively Mutates and Never Revisits”.

Throughout GA’s search process, random number is used frequently. A random number generator is typically used. It is an algorithm that generates long sequences of random numbers based on the initial value. These random numbers are not true random since they are predictable and repeatable. The same sequence of numbers can be reproduced by the same algorithm using the same initial value. They are pseudo random numbers. Caponetto, Fortuna, Fazzino, and Xibilila (2003) replaced random number with chaotic time series sequences in the algorithm. Simulation results and their statistical analysis using the t-test method showed distinct improvement from using chaotic sequences for the tested problems.

Singh, Isaacs, Nguyen, Ray, Yao (2008) and Singh, Isaacs, Ray, Smith (2008) proposed an Infeasibility Driven Evolutionary Algorithm (IDEA). The algorithm ranks solutions based on the original objectives (fitness function) along with additional objectives that reflects constraint violation measurement instead of solely rely on the fitness function. It explicitly maintains
several infeasible solutions in the generation to maintain the diversity of solution pool. The experiments result showed a fast convergence to optimal solutions.

There are many other development that enhancing the GA’s performance such as cooperative co-evolution (Adra, Dodd, Griffin, & Fleming, 2009), convergence accelerator (Tan, Teo, & Lau, 2007), etc. Due to the fact that GA is a straight forward global search algorithm and has demonstrated its effectiveness in many applications, more and more researchers are spending more time enhancing it with many other algorithms or methods. In the meanwhile, GA is enjoying more and more applications in many fields.
CHAPTER 5 DUALITY AND SHADOW PRICE in LINEAR PROGRAMMING

5.1 Definition

Dantzig (1963) stated, “The linear programming model needs an approach to finding a solution to a group of simultaneous linear equations and linear inequalities that minimize a linear form.” LP is the algorithm to search for an optimal value for a linear objective function that satisfies linear equations and linear inequalities.

Kolman and Beck (1980) defined the standard form for LP as,

For values of \( x_1, x_2, \ldots, x_n \) which will maximize

\[
z = c_1 x_1 + c_2 x_2 + \ldots + c_n x_n \tag{5.1}\]

Subject to the constraints

\[
a_{11} x_1 + a_{12} x_2 + \ldots + a_{1n} x_n \leq b_1 \\
a_{21} x_1 + a_{22} x_2 + \ldots + a_{2n} x_n \leq b_2 \\
\vdots \\
a_{m1} x_1 + a_{m2} x_2 + \ldots + a_{mn} x_n \leq b_m \\
x_j \geq 0, j = 1, 2, \ldots n \tag{5.2}\]

More conveniently, we can use a matrix notation. Let

\[
A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\
a_{21} & a_{22} & \cdots & a_{2n} \\
\vdots & \vdots & \ddots & \vdots \\
a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}, \quad
b = \begin{bmatrix} b_1 \\
b_2 \\
\vdots \\
b_m \end{bmatrix}, \quad
x = \begin{bmatrix} x_1 \\
x_2 \\
\vdots \\
x_n \end{bmatrix}, \quad
\begin{bmatrix} c_1 \\
c_2 \\
\vdots \\
c_n \end{bmatrix} \tag{5.3}\]

A LP standard form can be rewritten as

\[
\text{Maximize} \quad z = c^T x \tag{5.4}\]

\[
\text{Subject to} \quad Ax \leq b \\
\quad \quad x \geq 0
\]
The Duality Theorem states that there is an equivalent LP problem for every LP problem. One is called the primal problem and the other is called the dual problem. Dantzig (1963) proved the duality theorem. The dual problem for the above standard form is given below.

For values of \( y_1, y_2, \ldots, y_m \) which will minimize

\[
z' = b_1 y_1 + b_2 y_2 + \cdots + b_m y_m \tag{5.5}
\]

Subject to the constraints

\[
a_{11} y_1 + a_{21} y_2 + \cdots + a_{m1} y_m \geq c_1 \\
a_{12} y_1 + a_{22} y_2 + \cdots + a_{m2} y_m \geq c_2 \\
\vdots \\
a_{1n} y_1 + a_{2n} y_2 + \cdots + a_{mn} y_m \geq c_n \\
y_j \geq 0, j = 1, 2, \ldots, m
\tag{5.6}
\]

The matrix representation is

\[
\begin{bmatrix}
  y_1 \\
y_2 \\
\vdots \\
y_m
\end{bmatrix}
\]

Minimize

\[
z' = b^T y
\tag{5.7}
\]

Subject to

\[
A^T y \geq c
\]

\[
y \geq 0
\]

where

The Duality Theorem also states that if the primal problem has an optimal solution \((x_0)\) and the dual problem has an optimal solution \((y_0)\), then

\[
z = c^T x_0 = z' = b^T y_0 \tag{5.8}
\]

Solving one LP problem is equivalent to solving its dual problem. Kolman and Beck (1980) described the shadow prices as,

The \( j \)th constraint of the dual problem is
The coefficient $a_{ij}$ represents the amount of input $i$ per unit of output $j$, and the right-hand side is the value per unit of output $j$. This means that the units of the dual variable $y_i$ are the “value per unit of input $i$”; the dual variables act as prices, costs, or values of one unit of each of the inputs. They are referred as dual prices, fictitious prices, shadow prices, etc.

In general term, shadow price is the contribution to the objective function that can be made by relaxing a constraint by one unit. Different constraints have different shadow prices, and every constraint has a shadow price. Each constraint’s shadow price changes along with the algorithm searching progress.

5.2 Shadow Prices in Linear Programming

LP has been used widely in various industrial fields. With a concrete mathematical model, it provides direct relationships among profit and constraints, output and constraints, other goals and constraints, etc. The linear models can be solved efficiently. Dantzig’s (1963) Simplex method is one of them.

LP requires all constraints and all possible activities that meet the constraints listed in the tabular format. This is not a problem when the number of possible activities is small, such as maximizing profit for a small manufacturer. Constraints are material or labor and the objective function is defined as profit. It is rather straightforward to define the linear constraints, construct the linear objective function and search for optimal solutions for this category of problems.

It gets complicated where the number of possible activities is very large, such as the typical scheduling problems and the cutting stock problems. For these problems, there are a very large number of possible activities and make it very challenging to list them in the linear
constraints. For a good-sized airline, there are complex flight schedules, a large number of routes, and many flight crews. Various goals can be optimized, such as finding the minimal number of crews needed to cover all flights while satisfying airline regulations, creating the crew schedules while balancing flight hours among crews, creating crew schedules to minimize cost, etc. There are many possible combination of assigning crews to flights. This is an activity number explosion problem. For each activity, a separate variable need to be defined for the objective function and a separate column in the constraint matrix needs to be created in LP. This creates a very large number of variables and constraint columns. It is almost impossible to create a LP model with all possible activity combinations listed and constraints defined for this kind of problems. Solving these huge problems will be very time consuming and inefficient.

Gilmore and Gomory (1961, 1963, 1965, & 1966) developed a dynamic column generation algorithm to deal with this kind of combination explosion LP problem. They demonstrated their algorithm using the complex cutting stock problem. Figure 5.2.1 is the high level flow chart of their algorithm.

The Gilmore and Gomory’s breakthrough is separating the large problem into two smaller problems. The objective for the main LP problem (Figure 5.1 Main LP Problem) is to find the best solution using current available activities. The sub problem (Figure 5.1 Sub Knapsack Problem) is a knapsack problem. The solution from the main problem provides the coefficients for the sub problem’s constraints. The solution from the sub problem is a newer and better activity that can be utilized by the main algorithm. The process alternates between solving the main and the sub problem until there is no better solution that can be generated by the sub algorithm.
The coefficients supplied by the main algorithm to the sub algorithm are the shadow prices (dual prices). The knapsack sub problem is constructed using these shadow prices. For different iterations, the main algorithm provides the sub algorithm with different shadow prices based on the current best solution. That is, the shadow prices change along with the algorithm’s searching process.

![Flowchart of Gilmore and Gomory LP Algorithm](image)

Figure 5.1 Gilmore and Gomory LP Algorithm
CHAPTER 6 SHADOW PRICE GUIDED GENETIC ALGORITHMS

6.1 The Concept

We have developed a secondary measurement (Shen & Zhang, 2011-1) for solutions in the GA using the shadow price concept. We use the shadow prices to measure components in a solution as a complement measurement to the fitness function. Thus, we establish a two-measurement system: fitness values are used to evaluate overall solutions and shadow prices are used to evaluate components.

Using GA to solve a problem P, there is a current solution population R that has n solutions and each solution has m components. The jth solution is defined as

\[ S_j = (a_{1j}, a_{2j}, \ldots, a_{mj}) \]

where \( a_{ij} \) represents the ith component in jth solution. Then, the current solution space is

\[ R = (S_1^T, S_2^T, \ldots, S_n^T) \]

and we can define a correspondent LP problem as:

\[
A = \begin{bmatrix}
  a_{11} & a_{12} & \cdots & a_{1n} \\
  a_{21} & a_{22} & \cdots & a_{2n} \\
  \vdots & \vdots & \ddots & \vdots \\
  a_{m1} & a_{m2} & \cdots & a_{mn}
\end{bmatrix}
, \quad
b = \begin{bmatrix} b_1 \\
  b_2 \\
  \vdots \\
  b_m 
\end{bmatrix}
, \quad
x = \begin{bmatrix} x_1 \\
  x_2 \\
  \vdots \\
  x_n
\end{bmatrix}
, \quad
c = \begin{bmatrix} c_1 \\
  c_2 \\
  \vdots \\
  c_n
\end{bmatrix}
\]

(6.1)

Optimize \( z = c^T x \)

(6.2)

Subject to \( Ax \leq (\leq) (\geq) b \)

- \( x \) is binary variable 0 or 1
- and \( \sum_{i=1}^{n} x_i = 1 \)

\( c_i \) is the fitness value of each solution. The objective is to find the solution with the best fitness value. There shall be only one \( x=1 \) and the rest shall be 0.
This approach cannot deal with the combination explosion situation. We cannot possibly enumerate all feasible combinations in the $A$ matrix. For example, there are over 3 million possible combinations for a merely 10 cities’ traveling salesman problem. Secondly, we cannot always define the $b$ vector. We probably can create the $b$ vector for the value-combination problems. But for the position-combination problems, such as the traveling salesman problem, it is very difficult to find the meaning of the $b$ vector or define the relationship between $Ax$ and $b$.

The key of our approach is to use shadow price to compare components to further improve EA. In EA, we define the shadow price as the relative potential improvement to the solution’s (chromosome) fitness value with a change of a component (gene). It’s a relative potential improvement since the concept is defined on a single component and a component change may force other components’ change to maintain solution feasibility. The improvement may or may not be realizable. A change of component states the fact that component change can be a value change or a position change.

Shadow prices can take on different meanings or values for different problems. In the traveling salesman problem, it can simply be the possible distance reduction from changing the next visiting city. But the definition has to be clear and comparable among components.

The fitness value represents the current solution’s position in the search space. The shadow prices represent potential improvements and directions to evolve. The shadow prices are only meaningful in the process of evolution. They shall be used for selecting components to evolve and for setting directions for evolution operators. While choosing candidate solutions that are close to the optimal to further evolve, we shall also include solutions with bigger potential improvements. The potential improvement of a solution can be defined as the sum of all components’ potential improvements, which is the sum of all components’ shadow prices.
6.2 A Simple Example

Let’s illustrate our proposal with a simple example. Suppose a problem is defined as

Maximize \[ w = 40 \times (x + 1)^3 + 30 \times (y + 1)^3 + 10 \times (z + 1)^3 \] (6.3)

Subject to \[ x + 15 \times z \leq 45 \] (6.4)

\[ y + 10 \times z \leq 45 \] (6.5)

\[ x^2 + y^2 + z^2 \leq 3000 \] (6.6)

\[ x \geq 0; y \geq 0; z \geq 0; \] (6.7)

It is not a LP problem since the objective function (6.3) and the constraint (6.6) are not linear. The optimal solution is \( w = 4896905 \) when \((x, y, z) = (45, 31.22, 0)\). Using GA to solve this problem, we define the fitness function as

\[ f(x, y, z) = 40 \times (x + 1)^3 + 30 \times (y + 1)^3 + 10 \times (z + 1)^3 \] (6.8)

We can see from the fitness function that increasing \( x \), \( y \) or \( z \) value increases the fitness value, which fits the objective. There also exist some relationships among \( x \), \( y \), \( z \)’s contributions to the fitness value. That is, when \((x + 1)^3\) is increased by 1, the fitness function can be improved by 40. When \((y + 1)^3\) is increased by 1, the fitness function is improved by 30. The fitness function is only improved by 10 when \((z + 1)^3\) is increased by 1. From another perspective, increasing \((x + 1)^3\) by 1 can produce 3 times more contribution towards fitness value compared to \((z + 1)^3\). And \((y + 1)^3\) is 2 times more efficient than \((z + 1)^3\). So, we have relationships about contributions among \((x + 1)^3\), \((y + 1)^3\), and \((z + 1)^3\). But we still cannot derive direct relationships among \( x \), \( y \), and \( z \) since their cube functions is used in the fitness function instead of their linear format. Same change on \( x \), \( y \), and \( z \) will produce different impact on their cube functions when \( x \neq y \neq z \).
Although the direct contribution relationships among $x$, $y$, and $z$ are unknown, it is clear that, in general, increasing $x$ yields bigger improvement on fitness value than increasing $y$ does, and $y$ is more efficient than $z$. From constraints (6.4) and (6.5), we can derive $x \leq 45, y \leq 45, z \leq 3$.

There for, we define shadow prices $S$ as

$$S(x) = \begin{cases} 
40 \times (79 - x), & x \in [0,45) \\
0, & x = 45 
\end{cases} \quad (6.9)$$

$$S(y) = \begin{cases} 
30 \times (46 - y), & y \in [0,45) \\
0, & y = 45 
\end{cases} \quad (6.10)$$

$$S(z) = 10 \times (3 - z), z \in [0,45] \quad (6.11)$$

The shadow price definition points out the fact that increasing $x$ is more efficient than $y$ and increasing $y$ is more efficient than $z$. The fitness value can potentially be increased by 40 when $x$ is increased by 1. It’s a relative potential improvement since $x$’s cube function is used in the fitness function and $y$ or $z$ may need to be adjusted due to constraints. Although we can simply use coefficient (40, 30, 10) as the shadow prices, these will only represent the potential improvements and give no directions for GA to search. With the above definitions, we can clearly figure out which component has the priority and the direction to evolve. That is increasing $x$ first whenever possible, then $y, z$. So, we define the shadow price as the relative potential improvement to the solution’s (chromosome) fitness value with a change of a component (gene).

Suppose we have the following three solutions in a generation of evolution.

$$p_1 = (15,20,2); f(x, y, z) = 441940; S(x, y, z) = (2560,780,10);$$

$$p_2 = (15,10,2); f(x, y, z) = 204040; S(x, y, z) = (2560,1080,10);$$
Let’s mutate $p_1$. The fitness value gives no hint about how to evolve. The shadow prices for $p_1$ indicate that $x$ has the most potential to improve fitness value since it has the biggest shadow price. We select $x$ to mutate and try to mutate $x$ into a lower shadow price state, which is to realize its potential. Since $x \in [0, 45]$ and increasing $x$ will reduce shadow price from the definition of $S(x)$, we shall increase $x$ and select a number between 15 and 45. We choose 22. But $(22, 20, 2)$ violates constraint (1). We adjust $z$ and get feasible solution $p_4$

$$p_4 = (22, 20, 1); f(x, y, z) = 764590; S(x, y, z) = (2280, 780, 20);$$

From the above mutation operation, we improve the fitness value by 322650 and reduce $x$’s shadow price. Classic operator mutates a random component to a random direction. The impact to the fitness value is random as well. Applying shadow prices to mutation operator is better.

To apply a crossover operator on $p_2$ and $p_3$, fitness values again give us no directions. But from their shadow prices, $z$ in $p_2$ and $y$ in $p_3$ have the smallest shadow prices. So, the crossover operation shall use them to create the new solution as $(x, 15, 2)$. Since both 10 and 15 satisfy all constraints and 15’s shadow price is smaller, we select 15 for $x$. So, the new solution from the crossover operation is

$$p_5 = (15, 15, 2); f(x, y, z) = 286990; S(x, y, z) = (2560, 930, 10);$$

The new solution’s fitness value is better than both parents. With several components’ shadow price reduced, we materialize some potential. Comparing to classic randomized crossover operator, this solution is much better.

We solved this sample problem using classic genetic algorithm and our proposed algorithm for a comparison study. To ensure the comparison is valid, we did not introduce any
other techniques. All steps of both algorithms were the same except mutation and crossover operators. To set the same start up basis, we used the same initial population. Algorithms were terminated when there was no improvement for continuous 100 generations. We ran both algorithms 10 times. Results from table 6.1 show our new algorithm not only reached better solutions than classic algorithm but also used fewer generations. It demonstrates the effectiveness of our proposed shadow price guided genetic algorithm.

Table 6.1
Simulation results

<table>
<thead>
<tr>
<th>Testing</th>
<th>Proposed GA</th>
<th>Classic GA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Generations</td>
<td>Generations</td>
</tr>
<tr>
<td>1</td>
<td>171</td>
<td>181</td>
</tr>
<tr>
<td>2</td>
<td>173</td>
<td>206</td>
</tr>
<tr>
<td>3</td>
<td>201</td>
<td>128</td>
</tr>
<tr>
<td>4</td>
<td>218</td>
<td>218</td>
</tr>
<tr>
<td>5</td>
<td>108</td>
<td>145</td>
</tr>
<tr>
<td>6</td>
<td>112</td>
<td>305</td>
</tr>
<tr>
<td>7</td>
<td>173</td>
<td>210</td>
</tr>
<tr>
<td>8</td>
<td>228</td>
<td>157</td>
</tr>
<tr>
<td>9</td>
<td>115</td>
<td>161</td>
</tr>
<tr>
<td>10</td>
<td>270</td>
<td>384</td>
</tr>
<tr>
<td>Average</td>
<td>176.9</td>
<td>209.5</td>
</tr>
</tbody>
</table>

To conduct a statistical analysis and formal performance comparison between our proposed algorithm and the classic algorithm, we have conducted a simulation study with 100 runs of each algorithm. Table 6.2 presents the mean, standard deviation, medium, and inter-quartile range for the number of generations from both algorithms. Results indicate that the proposed algorithm uses a significantly smaller number of generations compared to the classic algorithm (Wilcoxon Two-Sample Test p<0.0001). Table 6.3 lists the mean, standard deviation, median, and inter-quartile range for the fitness values of the two algorithms. Results indicate that the proposed algorithm produces significantly larger fitness value than the classic algorithm.
(Wilcoxon Two-Sample Test p<0.0001). In summary, our proposed GA performs much better than the classic GA.

### Table 6.2
**Distribution of the Number of Generations**

<table>
<thead>
<tr>
<th></th>
<th>N</th>
<th>Mean</th>
<th>Standard Deviation</th>
<th>Medium</th>
<th>Inter-quartile Range</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proposed GA Algorithm</td>
<td>100</td>
<td>165.3</td>
<td>55.1</td>
<td>163.5</td>
<td>87</td>
<td>104</td>
<td>352</td>
</tr>
<tr>
<td>Classic GA Algorithm</td>
<td>100</td>
<td>210.7</td>
<td>79.4</td>
<td>198</td>
<td>118</td>
<td>107</td>
<td>464</td>
</tr>
</tbody>
</table>

### Table 6.3
**Distribution of the Fitness Values**

<table>
<thead>
<tr>
<th></th>
<th>N</th>
<th>Mean</th>
<th>Standard Deviation</th>
<th>Medium</th>
<th>Inter-quartile Range</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proposed GA Algorithm</td>
<td>100</td>
<td>4895755</td>
<td>990</td>
<td>4895971</td>
<td>2271</td>
<td>4894634</td>
<td>4896905</td>
</tr>
<tr>
<td>Classic GA Algorithm</td>
<td>100</td>
<td>4881970</td>
<td>22295</td>
<td>4893435</td>
<td>22164</td>
<td>4780972</td>
<td>4896905</td>
</tr>
</tbody>
</table>

For the above example, we defined the shadow price as the components’ relative potential improvement to the fitness value. We used shadow prices to select component(s) to operate on and evolve to directions based on future shadow prices. We demonstrated that the shadow price guided operators are better than classic GA operators. We illustrated that our proposed two-measurement system, fitness value and shadow price, is better than the one fitness value measurement system.

6.3 **Define Shadow Price**

Based on different problems, shadow prices can take on different meanings or values. In the traveling salesman problem, it can simply be the possible distance reduction from changing the next visiting city from the current one (Shen & Zhang, 2011-1). In manufacture, shadow price can be the cost of material, time, etc. (Shen & Zhang, 2010-1, 2010-2, 2012-1). In green computing, it can be defined as average energy consumption per instruction (Shen & Zhang,
2011-2) or embedded in the procedure (Shen & Zhang, 2012-3). But the definition has to be clear and comparable among components. Here are a few guidelines on how to select shadow price.

1) The shadow price shall enable comparison among components since this is its main function in the search. A concrete value is preferred over fuzzy values. The minimum requirement is that the shadow price shall allow components comparison within a solution. This makes it usable for the mutation operation. If the shadow price definition enables components comparison across solutions, crossover operations can benefit from it.

2) The shadow price shall reflect the attribute of a component such as price, cost, material, etc. The attribute shall directly or indirectly impact the solution quality (fitness value). This requirement is to relate shadow price directly to the problem. Solution’s change can change shadow price and vice versa.

3) The shadow price for the solution (sum of shadow prices from all components) shall change with the quality of the solution (fitness value). There is no need to define a math function to associate them. The only requirement is to ensure that the shadow price is consistent with the search process. Since it reflects the potential improvement in the solution from components’ perspective, solution’s shadow price shall reduce while search process finds better solutions. In other words, better solution’s shadow price shall be smaller than worse solution’s shadow price. This has to holds true for all feasible solutions in the search space. This is to define evolution direction.

4) The shadow price calculation shall be simple and fast. The shadow price concept and algorithm introduces more calculations, such as calculating components’ shadow prices, comparisons, etc. A quick, straightforward shadow price calculation is necessary.
6.4 The Complete Algorithm

Figure 6.1 New GA Framework with Shadow Price Guided Operators
The principle of our algorithm (Figure 6.1) is to use the shadow prices as the guide to direct the search for the optimal solution. For each current feasible solution, we use shadow prices to select components and to set the evolution direction. In detail, for the mutation operator, we shall pick a component with a higher shadow price to mutate and shall mutate to a lower shadow priced state. The goal of the crossover operator is to generate a new solution that inherits good components, which have low shadow prices, from both parents.
CHAPTER 7 OPTIMIZING THE TRAVELING SALESMAN PROBLEM WITH SGA

7.1 Introduction

The Traveling Salesman Problem (TSP) is a classic NP hard combinatorial problem. It has been routinely used as a benchmark to verify new algorithms. There are two major categories of algorithms used to solve the problem, exact or approximate algorithms. Exact algorithms, such as testing all permutations or branch and bound, typically either take very long time to compute or reach unsatisfied results.

There are a lot approximate algorithms that achieve good results. Genetic Algorithm (Choi, Kim, & Kim, 2003; Kaur & Murugappan, 2008; Ray, Bandyopadhyay, & Pal 2004), Ant Colony Optimization (ACO) (Bianchi, Gambardella, & Dorigo, 2002; Hung, Su, & Lee, 2007), Neural Network (NN) (Hasegawa, Ikeguchi, & Aihara, 2002; Vishwanathan & Wunsch, 2001), Discrete Particle Swarm Optimization (DPSO) (Wang, Huang, Zhou, & Pang, 2003; Wang, Zhang, Yang, Hu, & Liu, 2005; Zhi et al., 2004; Zhong, Zhang, & Chen, 2007), Bee Colony Optimization (BCO) (Wong, Low, and Chong 2008), Simulated Annealing (SA) (Kirkpatrick, Gelatt, & Vecchi, 1983), Collective Intelligence (Kulkarni & Tai, 2009), and hybrid algorithms (Lee, Lee, & Su, 2002; Yang & Zhuang, 2010) have been used to solve the TSP. They all have achieved good results. We also use the TSP to validate our proposed algorithm and compare results with several of above-mentioned algorithms.

7.2 Problem Definition

The Traveling Salesman Problem (symmetric) can be simply stated as: for a given number of cities and defined travel distances between any city pairs, find the shortest path (or
cost) for a salesman to visit all cities once and only once, and finally return to the departure city. Obviously, the fitness function is the distance of the complete path (or cost).

The TSP is a classic NP hard problem. It is a well-documented and widely studied combinational optimization problem. There are a good number of research documents, published reference problems, and solutions.

### 7.3 Shadow Price Definition

In the TSP, any city is connected to all other cities by a distance. For a given solution, any city is connected to two and only two other cities. Let’s define the TSP as having n cities, $C_1$, $C_2$, $C_3$, ..., $C_n$, and the distance is $D_{ij}$ for distance from $C_i$ to $C_j$. We define a city $j$’s shadow price $S_j$ in a given tour, $C_1$, $C_2$, ..., $C_i$, $C_j$, $C_k$, ..., $C_n$, as

$$
S_j = \sum_{q=1}^{n} (D_{ij} - D_{aq}) + \sum_{r=1}^{n} (D_{jk} - D_{rj})
$$

(7.1)

where $D_{ij} > D_{aq}$ and $D_{jk} > D_{rj}$ and $q \neq j$, $r \neq j$

The shadow price for a city is defined as the sum of all possible distance savings by changing the connected cities. This is a relative number that represents the shadow price concept. Simply connecting to one or two closer cities may not shorten the tour distance since the disconnected cities have to be rejoined into the tour again. The new connections may increase or decrease the total tour distance.

Table 7.1 is a sample TSP distance table from the gr17.tsp from TSPLIB (2009).
We number the cities from 1 to 17. Let’s assume we have a tour as

\[
C_1 \rightarrow C_2 \rightarrow C_3 \rightarrow C_4 \rightarrow C_5 \rightarrow C_7 \rightarrow C_8 \rightarrow \ldots \rightarrow C_{17} \rightarrow C_1
\]

Let’s compute shadow prices for city 4, 5 and 6

\[
S_4 = \sum_{q=1}^{17} (D_{4q} - D_{q4}) + \sum_{r=1}^{17} (D_{4r} - D_{r4}) = \sum_{q=1}^{17} (228 - D_{q4}) + \sum_{r=1}^{17} (383 - D_{r4}) = 3813 \quad (7.2)
\]

\[
S_5 = \sum_{q=1}^{17} (D_{5q} - D_{q5}) + \sum_{r=1}^{17} (D_{5r} - D_{r5}) = \sum_{q=1}^{17} (383 - D_{q5}) + \sum_{r=1}^{17} (267 - D_{r5}) = 2100 \quad (7.3)
\]

\[
S_6 = \sum_{q=1}^{17} (D_{6q} - D_{q6}) + \sum_{r=1}^{17} (D_{6r} - D_{r6}) = \sum_{q=1}^{17} (267 - D_{q6}) + \sum_{r=1}^{17} (63 - D_{r6}) = 1697 \quad (7.4)
\]

Above definition and calculation provide us the method to compare components (cities) in a solution. From the above shadow prices, we can derive that \(C_4\) can produce potentially more improvement to the solution than \(C_5\). These are possible improvements since they may not be realizable. This is the concept of shadow price we proposed earlier.
We also define a tour’s shadow price as the summation of all cities’ shadow prices. Obviously, tours with higher shadow prices have bigger room for improvement. The optimal tour’s shadow price is not guaranteed to be zero nor the smallest by our definition. But, a zero shadow priced tour is the optimal tour. For an edge in the tour, a connection from one city to another city, the shadow price is defined as the total shadow prices from both cities. This is to keep consistent with TSP tour’s shadow price definition.

7.4 Shadow Price Guided Mutation Operator

There are two methods to select a subset of solutions for mutation, routes with higher shadow prices or routes with low fitness values. It makes sense to choose routes with low fitness values since they are potentially better or closer to the optimal solutions. But the solutions that are closer to the optimal may not always evolve to the optimal. On the other hand, higher shadow priced routes have the best chances of making big improvements. Since GA encourages diversity in its population, we use a mixed subset for mutation.

We select a mutation component (city) based on components’ shadow prices. We prefer components with high shadow prices since they promise better improvements. To avoid a local optimal trap, we randomly select a component from a pool of high shadow priced components. In the above example, \( C_4 \) has a better chance of being selected to mutate than \( C_5 \) or \( C_6 \).

Mutate to the shortest connection promises the biggest improvement but increases the risk of being trapped into a local optimal solution. Using the smallest connection improvement may lose opportunities for quick improvements and slow down the search process. Again, we create a pool of shorter connections and select one randomly as the new connection. The pool size is adjusted dynamically to better reflect the current search progress. In above example, we may choose one city from \( (C_1, C_6, C_7, C_8, C_9, C_{13}, C_{14}, C_{17}) \) if we were to mutate \( C_4 \).
7.5 Shadow Price Guided Crossover Operator

The goal of crossover is to pass good connections (genes) from the parents to the child. High shadow priced routes are relatively far from the optimal solutions compared to others. But they may have good connections that the child can still benefit from and vice versa. The same argument applies to the fitness value as well. It seems that randomly selecting two routes to crossover is fair and simple. In order to inherit a good portion of better connections in the crossover operation, we choose to select at least one parent route that with a good fitness value. The other parent is randomly selected in the current population.

We use a simple edge insertion algorithm for the crossover operation. The route with a good fitness value (smaller) is cloned as the start of the new child route. A number (a dynamic parameter) of good connections from the other route are inserted into the child route. These good connections are randomly selected from a pool of low shadow priced connections. In this case, low shadow priced connections are good connections that have less room for improvements. After the crossover operation, we verify the feasibility of the child route and make adjustments if necessary. In the above gr17 solution example, edge \((C_5, C_6)\) has a better chance to be passed to the child than edge \((C_4, C_5)\) since \(S_5 + S_6 \leq S_4 + S_5\). In semantics, \((C_5, C_6)\) is a relatively better connection than \((C_4, C_5)\).

7.6 Solution Validation

The resulting solution from a GA operator need to be validated to ensure its feasibility and adjusted if necessary. The mutation operation creates a new connection between two cities and creates two disconnected graphs. Let’s assume we have a tour from table 7.1’s sample problem (Figure 7.1) as

\[ C_1 \rightarrow C_2 \rightarrow C_3 \rightarrow C_4 \rightarrow C_5 \rightarrow C_6 \rightarrow C_7 \rightarrow C_8 \rightarrow \ldots \rightarrow C_{17} \rightarrow C_1 \]
If GA select $C_4$ to mutated and reconnect it to $C_8$, two disconnected graphs are created (Figure 7.2). This is an invalidate solution.

There are two methods to adjust the solution. One is inserting the disconnected segment into the other side of the mutated city. In the example, we disconnect $C_3$ and $C_4$; connect $C_3$ to $C_5$ and $C_7$ to $C_4$. The other method is inspecting every connection to find the best location to insert the disconnected segment. The first method maintains the stability of the rest tour and fast. The second method seeks the local optimal and less efficient. One of the two methods is randomly selected to adjust solution in our algorithm. Similar methods are used to validate results from crossover operation.

Figure 7.1 A Sample Tour
7.7 Other Techniques

A shadow price modified 2-opt operator is also used in our algorithm. “In optimization, 2-opt is a simple local search algorithm first proposed by Croes in 1958 for solving the traveling salesman problem. The main idea behind it is to take a route that crosses over itself and reorder it so that it does not.” (Watson et al., 1998). Combining the 2-opt operator with other operations in the genetic algorithm produced good results for the TSP (Wikipedia 2-opt, 2009). It is a very simple heuristic local search algorithm and hampered by performance. The operation time is $O(n^2)$.

Armed with the shadow price information, we use 2-opt operation to speed up the algorithm by eliminating obviously very bad connections in the route. Instead of applying to all connections, we only use 2-opt operations for certain high shadow priced connections. The time used is $O(n)$. 
We use a simple coding schema. For the route start from city 1, \( C_1 \rightarrow C_2 \rightarrow C_3 \rightarrow \ldots \rightarrow C_n \rightarrow C_1 \), we encode it as \((C_1, C_2, C_3, \ldots, C_n, C_1)\).

7.8 Experiments

We coded our proposed algorithm in C# and executed it on a Pentium 4 2.8GHz machine with 2 GB of RAM. While comparing speed with other published results, we only need to consider CPU specification and programming language since the memory footprint is rather small for the TSP.

We chose TSPLIB (2009) as the test cases and the data source for our experiment. It is one of the mostly used test case sources to verify algorithm’s efficiency. It provides many TSP cases with proven optimal routes. Each test case was run ten times.

To gauge the effectiveness of our algorithm, we compared our results with other published Bio inspired researches that used the same test cases from TSPLIB. Table 7.2 is the results of our algorithm compared with an innovative genetic algorithm. Table 7.3 is the results of our algorithm against an improved Particle Swam Optimization algorithm. Table 7.4 shows how our proposed algorithm stacks up against an improved Bee Colony Optimization algorithm.

Overall, our proposed new algorithm did better in the solution quality and speed than any of the others (Shen & Zhang 2011-1).

<table>
<thead>
<tr>
<th></th>
<th>Optimal</th>
<th>Ray, et al. 2004</th>
<th>Our result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best</td>
<td>Best</td>
<td>Average</td>
</tr>
<tr>
<td>GR24</td>
<td>1272</td>
<td>1272</td>
<td>1272</td>
</tr>
<tr>
<td>Bayg29</td>
<td>1610</td>
<td>1610</td>
<td>1610</td>
</tr>
<tr>
<td>GR48</td>
<td>5046</td>
<td>5046</td>
<td>5046</td>
</tr>
<tr>
<td>ST70</td>
<td>675</td>
<td>685</td>
<td>675</td>
</tr>
<tr>
<td>KroA100</td>
<td>21282</td>
<td>21504</td>
<td>21282</td>
</tr>
</tbody>
</table>
Table 7.3
Comparison with Zhong, Zhang, and Chen (2007)

<table>
<thead>
<tr>
<th></th>
<th>Optimal</th>
<th>Zhong, et al 2007</th>
<th>Our result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Best</td>
<td>Average</td>
<td>Avg Time(s)</td>
</tr>
<tr>
<td>Eil51</td>
<td>426</td>
<td>427</td>
<td>433.64</td>
</tr>
<tr>
<td>Berlin52</td>
<td>7542</td>
<td>7542</td>
<td>7598.76</td>
</tr>
<tr>
<td>Eil76</td>
<td>538</td>
<td>540</td>
<td>551.72</td>
</tr>
<tr>
<td>KroA100</td>
<td>21282</td>
<td>21296</td>
<td>21689.30</td>
</tr>
<tr>
<td>KroA200</td>
<td>29368</td>
<td>29563</td>
<td>30374.30</td>
</tr>
</tbody>
</table>

Table 7.4
Comparison with Wong, Low, and Chong (2008)

<table>
<thead>
<tr>
<th></th>
<th>Optimal</th>
<th>Wong, et al. 2008</th>
<th>Our Result</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>% from optimal</td>
<td>Distance</td>
<td>% from optimal</td>
</tr>
<tr>
<td></td>
<td>Best</td>
<td>Average</td>
<td></td>
</tr>
<tr>
<td>ATT48</td>
<td>10628</td>
<td>0.31</td>
<td>0.83</td>
</tr>
<tr>
<td>EIL51</td>
<td>426</td>
<td>0.47</td>
<td>0.85</td>
</tr>
<tr>
<td>EIL76</td>
<td>538</td>
<td>0.19</td>
<td>2.01</td>
</tr>
<tr>
<td>EIL101</td>
<td>629</td>
<td>0.95</td>
<td>2.29</td>
</tr>
<tr>
<td>KROA100</td>
<td>21282</td>
<td>2.26</td>
<td>3.43</td>
</tr>
<tr>
<td>KROB100</td>
<td>22141</td>
<td>2.24</td>
<td>3.1</td>
</tr>
<tr>
<td>KROC100</td>
<td>20749</td>
<td>0.5</td>
<td>1.5</td>
</tr>
<tr>
<td>KROD100</td>
<td>21294</td>
<td>1.64</td>
<td>3.25</td>
</tr>
<tr>
<td>KROE100</td>
<td>22068</td>
<td>1.73</td>
<td>2.2</td>
</tr>
<tr>
<td>KROA150</td>
<td>26524</td>
<td>5.03</td>
<td>6.39</td>
</tr>
<tr>
<td>KROB150</td>
<td>26130</td>
<td>1.55</td>
<td>3.68</td>
</tr>
<tr>
<td>KROA200</td>
<td>29368</td>
<td>2.02</td>
<td>4.26</td>
</tr>
<tr>
<td>KROB200</td>
<td>29437</td>
<td>3.1</td>
<td>6.36</td>
</tr>
<tr>
<td>LIN105</td>
<td>14379</td>
<td>0.32</td>
<td>1.24</td>
</tr>
<tr>
<td>LIN318</td>
<td>42029</td>
<td>6.32</td>
<td>7.55</td>
</tr>
</tbody>
</table>

7.9 Summary

For the TSP, we define shadow price for a city as the sum of all possible distance savings by changing the connected cities. It was used to evaluate components and to direct evolutionary progress mainly towards the optimal solution. We used it as a secondary solution measurement in our proposed two-measurement EA. The simulation results have shown that our new SGA was effective and efficient.
CHAPTER 8 OPTIMIZING THE CUTTING STOCK PROBLEM WITH SGA

8.1 Introduction

The Cutting Stock Problem (CSP) is a very important problem in many industries with great economic values. It’s a difficult integer optimization problem. The classic Linear Programming algorithm was first used to solve the CSP (Gilmore & Gomory, 1961, 1963, 1965, 1966). The dynamic column generation technique used a fix-sized matrix to solve the problem. But the solution was in fraction. An integer rounding routine had to be applied to the result to generate a meaningful solution. Producing infeasible or lower efficiency solutions were expected from the rounding process.

Many other CSP algorithms were developed in the operations research field. For instance, the LP based branch-and-cut-and-price algorithms (Alves & Carvalho, 2008; Belov & Scheithauer, 2006) are combinations of LP based branch-and-bound, column generation technique and cutting plane algorithms. These are integer LP algorithms that can provide optimal solutions. Their deficiencies are the degeneracy problem, the single linear objective function limitation and less efficient than traditional non-integer LP algorithms. The heuristic algorithms (Cherri, Arenales, & Yanasse, 2009; Cui & Lu, 2009; Liu, Chu, & Wang, 2008; Poldi & Marcos, 2009; Song, Chu, Nie, & Bennell, 2006) use a set of rules, patterns, and steps to generate feasible solution. They are very quick and can provide acceptable near optimal results for small CSPs. They are not effective in solving large complex problems since they may degenerate to only providing feasible solutions. The hybrid algorithms (Aktin & Özdemir, 2009; Cui & Yang, 2010; Yanasse & Lamosa, 2007; Yanasse & Limeira, 2006) combine LP, heuristic algorithms, and other algorithms. They can provide very good solutions for targeted fields and their performance various.
Hinterding and Khan (1994) successfully solved the CSP using GA. The solution was in integer and the process was very efficient. Other bio-inspired algorithms such as the Ant Colony Algorithm (Levine & Ducatelle, 2004; Lu, Wang, & Chen, 2008; Yang, Li, Huang, Tan, & Zhou, 2009), the Evolutionary Algorithm (Chiong, Chang, Chai, & Wong, 2008; Yao, Newton, & Hoffman, 2002), and the Annealing Algorithm (Yue & Gao, 2009) were also used to solve the CSP. These algorithms provided good integer solutions.

8.2 Problem Definition

The CSP is to find the best arrangement of orders to cut from stocks such that minimal number of stocks is used. The objective is to use the least amount of stocks to satisfy various item requirements. The CSP is formulated as (Hinterding & Khan, 1994):

Minimize $W = \sum_{j \in J} w_j x_j$, \hspace{1cm} (8.1)

Subject to $\sum_{j \in J} a_{ij} x_j = N_i$ for $i=1,2,\ldots,n$. \hspace{1cm} (8.2)

$x_j \geq 0$, integer for $j \in J$.

Where, $n =$ number of orders.

$w_j =$ waste per run of pattern $j$.

$a_{ij} =$ number of pieces of item $i$ in pattern $j$.

$x_j =$ number of runs of pattern $j$.

$N_i =$ number of pieces of item $i$.

If there is only one stock length $L$ in the problem, and $l_i$ is the length of order $i$, then

$L = \sum_{i=1}^{n} a_{ij} l_i + w_j$ for $j \in J$. \hspace{1cm} (8.3)
Adding more stock lengths to the CSP increases the size of the problem and requires more search time. But it does not increase the complexity of the problem. Compared to the CSP with single stock length, the CSP with multiple stock lengths can have more item combinations to potentially improve the trim efficiency. Tables 8.1 and 8.2 present two experimental results for the CSP with multiple stock lengths and the CSP with single stock length (Hinterding & Khan, 1994). Both tables include total evaluations, the mean fitness values, the standard deviations, and the evaluation number when the optimal solution was found. The fitness value represents the efficiency of the solution. A high fitness value means high efficiency and low waste. Std. Dev. is the standard deviation to show the distribution of the solutions.

Table 8.1
Test results for the CSP with multiple stock lengths

<table>
<thead>
<tr>
<th>Case</th>
<th>Evaluations</th>
<th>Mean fitness</th>
<th>Std. Dev.</th>
<th>Found at</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1184</td>
<td>1</td>
<td>0</td>
<td>407</td>
</tr>
<tr>
<td>2</td>
<td>1184</td>
<td>1</td>
<td>0</td>
<td>740</td>
</tr>
<tr>
<td>3</td>
<td>1184</td>
<td>1</td>
<td>0</td>
<td>407</td>
</tr>
<tr>
<td>4</td>
<td>2294</td>
<td>0.9995</td>
<td>0.0022</td>
<td>2294</td>
</tr>
<tr>
<td>5</td>
<td>2294</td>
<td>0.9998</td>
<td>0.0007</td>
<td>2294</td>
</tr>
</tbody>
</table>

Table 8.2
Test results for the CSP with single stock length

<table>
<thead>
<tr>
<th>Case</th>
<th>Evaluations</th>
<th>Mean fitness</th>
<th>Std. Dev.</th>
<th>Found at</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a</td>
<td>1184</td>
<td>0.9133</td>
<td>0</td>
<td>296</td>
</tr>
<tr>
<td>2a</td>
<td>1184</td>
<td>0.9227</td>
<td>0.0018</td>
<td>1184</td>
</tr>
<tr>
<td>3a</td>
<td>1184</td>
<td>1</td>
<td>0</td>
<td>407</td>
</tr>
<tr>
<td>4a</td>
<td>1184</td>
<td>0.9642</td>
<td>0</td>
<td>851</td>
</tr>
<tr>
<td>5a</td>
<td>2294</td>
<td>0.8479</td>
<td>0.007</td>
<td>2294</td>
</tr>
</tbody>
</table>

The data from Tables 8.1 and 8.2 suggest that the solutions for the CSP with multiple stock lengths have better fitness values than the ones for the CSP with single stock length, and the total evaluations are almost the same for both type CSPs. They exhibit the fact that the CSP
with single stock length is at least as complex as the CSP with multiple stock lengths. We used the CSP with single stock length to demonstrate our new algorithm.

### 8.3 Basic Terminologies

In the CSP, a pattern is one possible combination of items that can be cut from one single stock. The total length of all items in a pattern shall be less or equal to the stock length. A trim, a solution of the CSP, is a set of patterns satisfying the order requirements. When using GA or EA to solve the CSP, a pattern corresponds to a gene and a trim corresponds to a chromosome. In the group based coding schema, a group is a set of items that represents a pattern. The group based coding schema is much better than the order based coding schema (Hinterding & Khan, 1994). We use group based coding schema.

<table>
<thead>
<tr>
<th>Item Length</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. Required</td>
<td>5</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

We use a sample problem (Table 8.3) from Hinterding and Khan (1994) to introduce our new algorithm. In Table 8.3, the data in the first row are the lengths of different order items and the data in the second row are their quantities to be produced. The objective is to use the least number of stocks to produce these items.

We use the length of the item to represent the item. In the sample problem, (3,4,5) represents a pattern that contains one length 3 item, one length 4 item, and one length 5 item. The waste of this pattern is 2 since the total item length is 12 and the stock length is 14. The set of patterns \{(3,3,8), (5,9), (4,10), (7,7), (3,3,8), (7,7), (4,10), (6,6), (3,10)\} represents a trim that satisfies the item requirements. This trim’s waste is 3, which is generated by the last two patterns.
8.4 Shadow Price Definition

In the CSP, pattern selection links to the trim efficiency directly since the trim waste is the summation of waste from all its patterns. The patterns in a trim are evaluated by the waste they produce. In the above sample problem, the pattern (3, 4, 5)’s total length is 12 and it yields a waste of 2. The total length of pattern (3, 3, 8) is 14 and it produces no waste. Obviously, pattern (3, 3, 8) is better than pattern (3, 4, 5), and pattern (3, 3, 8) shall be used more often in the trim. There are limitations on whether a good pattern can be used or how many times it can be used in a given CSP. Since the requirement for the length 8 item is 2 in the sample problem, pattern (3, 3, 8) can only be used twice to produce 2 length 8 items and 4 length 3 items. This leaves one length 3 item to be produced since the original requirement is 5. This makes pattern (3, 4, 5) a candidate for the trim even though it produces a waste of 2. Pattern selection is the key for the CSP algorithm.

From another perspective, we can analyze the price with the stock length. There is no waste in pattern (3, 3, 8) since both the stock length and the total length of all items are 14. The price for the length 3 item is 3 and the length 8 item is 8. There is a waste of 2 in pattern (3, 4, 5) since the total length for all items is 12 and the stock length is 14. The price of 14 is selected to fulfill the total item length requirement of 12. Proportionally, the price for the length 3 item is 3*14/12=3.5, the length 4 item is 4*14/12=4.67, and the length 5 item is 5*14/12=5.83. In comparison, we pay more to produce the length 3 item in pattern (3, 4, 5) than in pattern (3, 3, 8). We use a stock length of 3.5 to produce one length 3 item and waste 0.5 in pattern (3, 4, 5) in contrast to using a stock length of 3 to produce the item and yield no waste in pattern (3, 3, 8).

The shadow price concept represents the price of an item paid in a trim. It is the average cost of an item in a trim. We use $S_i$ to denote the shadow price of item $i$ and $SP_{ij}$ to denote the shadow
price of item $i$ in pattern $j$. All other notations in formulas conform to the previously used symbols.

$$SP_{ij} = l_i \times \frac{L}{\sum_{k} l_k}$$ for $k$ is the number of items in pattern $j$, \hspace{1cm} (8.4)

$$S_i = \sum_{j} \frac{SP_{ij}}{N_i}$$ for $j$ is the number of patterns in the trim. \hspace{1cm} (8.5)

An item’s shadow price is equal to or greater than its length. When it is greater than its length, more stock is used in the trim to produce this item than needed. Waste is generated to produce this item. If it is equal to the item’s length, there is no waste in the trim to produce this item. The shadow price of a pattern is the sum of the shadow prices from all items contained in the pattern. It represents the total price of these items in the current trim using this pattern. If the shadow prices are used in a new pattern, the new pattern’s shadow price represents the items’ total price from the previous trim and the stock length represents their current price.

### 8.5 Shadow Price Guided Mutation Operator

The goal of the mutation operator is to introduce new patterns to the trim when using the GA to solve the CSP. Adding a new pattern to the trim is a complicated process since existing patterns may be dropped and additional new patterns may be added to complete the trim. The fitness value of the trim can only be improved by adding better patterns. It is very challenging to create better patterns.

Randomly generated new patterns and the group mutation operator (Falkenauer & Delchambre, 1992) were used in Hinterding and Khan (1994)’s experiments. Poor patterns were replaced by randomly generated new patterns.

Instead of generating random patterns, we use a different approach to create new patterns. We intentionally introduce good patterns to the trim to improve its fitness value. For an existing
trim, we first calculate the shadow prices for all items based on all patterns in the trim. Then, we search for a pattern with the biggest shadow price such that

\[
\text{Maximize } S = \sum_{i=1}^{n} a_i SP_i , \quad \text{(8.6)}
\]

\[
L \geq \sum_{i=1}^{n} a_i l_i . \quad \text{(8.7)}
\]

If a new pattern is found by maximizing the above function and its shadow price is greater than the stock length, the new pattern uses less stock to produce the items in the pattern than the existing trim. The existing trim needs the shadow price to produce these items in the new pattern. The new pattern only needs one stock. The new pattern produces less waste since the stock length is less than the shadow price. If we create a new trim by inserting this new pattern into the existing trim, the new shadow prices for the items in the pattern shall be smaller than their previous values. These items are cheaper in the new trim. The new trim’s fitness value shall be better than the previous trim as well.

Our new mutation operator starts with calculating the shadow prices for all items. Then, it searches for a new pattern with a shadow price that is greater than the stock length. If a new pattern is found, it inserts the pattern into the trim at a random location. Finally, it validates the trim. The operation stops if it cannot find a pattern with a shadow price greater than the stock length.

In the trim \{(3, 3, 8), (5, 9), (4, 10), (7, 7), (3, 3, 8), (7, 7), (4, 10), (6, 6), (3, 10)\} for the sample problem, the length 6 item’s shadow price is 7 and the length 8 item’s shadow price is 8. Pattern (6, 8) is a potential good pattern since its shadow price of 15 is greater than the stock length of 14. That is, it needs a total stock length of 15 to produce one length 6 item and one length 8 item in the previous trim. Now, it only needs a total stock length of 14.
8.6 Shadow Price Guided Crossover Operator

The group crossover (BPCX) is a very straightforward operator (Falkenauer & Delchambre, 1992). It mainly consists of the following steps: (1) randomly split a parent trim into two sections, (2) copy the first section to the child trim, (3) append all patterns from the second parent trim to the child trim, and (4) finally append the second section from the first parent trim to the child trim. The child trim is validated while patterns are added. Uniform Grouping Crossover (UGCX) (Hinterding & Khan, 1994) adds pattern order to the group crossover operator. Both BPCX and UGCX randomly merge two parent trims into one child trim. There is no intention to improve the child trim in the process.

In the CSP, two trims can have different patterns and efficiencies. The same items in these two trims may consume different amount of stocks since they may belong to different patterns. Patterns with less waste are always better. If we quantify an item and its stock consumption with the shadow price, we can create a better child trim using the crossover operator that selects better patterns from both parents.

We propose a new crossover operator using the shadow price. The novel crossover operator has the following major steps: (1) copy a parent trim to the child trim, (2) calculate shadow wastes (shadow price – item length) for all items in the child trim, (3) rank the items by their shadow wastes, (4) select an item with a big shadow waste, (5) select all patterns containing this item from the other parent and insert them into the child trim, and (6) finally, validate the child trim.

In the sample problem, we have two trims {{3, 3, 8}, (5, 9), (4, 10), (7, 7), (3, 3, 8), (7, 7), (4, 10), (6, 6), (3, 10}} and {{6, 8}, (5, 9), (4, 10), (7, 7), (6, 8), (7, 7), (4, 10), (3, 3, 3, 3), (3, 10)}. The novel crossover operator copies the first trim to the child trim and calculates the
shadow waste for each item. The shadow waste for the length 3 item is 0.046, for the length 6 item is 1, for the length 10 item is 0.26, and 0 for all other items. Since length 6 item’s shadow waste is the biggest, all patterns containing this item from the second parent are copied into the child trim. The patterns are (6, 8) and (6, 8). By adding good patterns from the second parent into the child trim, we increase the chance of creating a better child trim.

8.7 Experiments

To compare our algorithm with others, we adopted the widely used fitness function that defined in Liang et al. (2002) as follows:

\[
\text{Maximize } f = 1 - \frac{1}{m+1} \left( \sum_{j=1}^{m} \frac{w_j}{L} + \sum_{j=1}^{m} \frac{v_j}{m} \right). \tag{8.8}
\]

In the fitness function, \( m \) stands for the number of patterns in the trim. The first term within the parenthesis is used to minimize the total waste. The second term is used to minimize the number of patterns with waste, where \( v_j = 1 \) when the \( j \)th pattern has a waste, and 0 if no waste. The objectives of the fitness function (8.8) are (1) minimizing the trim waste and (2) reducing the number of patterns with waste.

We implemented Hinterding and Khan (1994)’s algorithm as Algorithm A. We created three new algorithms B, C and D with different mutation operators. The new shadow price Guided crossover operator was used for all three versions. All four algorithms and the algorithms we compared with used the same fitness function defined above.

In Algorithm B (Figure 8.1), a few patterns that generated waste were removed from the trim before the new shadow price based pattern was inserted, and simple sequential patterns were created for any untrimmed items from the deleted patterns. In Algorithm C (Figure 8.2), the new shadow price based pattern was added to the trim without removing any patterns. In Algorithm D
(Figure 8.3), a few patterns that generated waste were removed from the trim before the new shadow price based pattern was inserted, and several more shadow price based patterns were created for any untrimmed items from the deleted patterns.

Figure 8.1 Algorithm B’s mutation operator
Mutation

Select a sub population to mutate

Have all solutions mutated?

No

Select a solution to mutate

Calculate shadow prices for all items

Find a pattern with the biggest shadow price

Is the new pattern’s shadow price greater than the stock length?

Yes

Copy the solution to a new solution

Insert the new pattern into the new solution at a random position

Validate the new solution

No

Yes

Figure 8.2 Algorithm C’s mutation operator
We implemented all algorithms in C#. Each test case was run 10 times and results were averaged for comparison. To compare with other published algorithms, we selected the commonly used test cases (Liang et al., 2002). There are 10 single length CSPs ranging from 20
items to 600 items. Table 8.4 lists the test case name, the number of different item sizes and the total items required.

<table>
<thead>
<tr>
<th>Case</th>
<th>Size Count</th>
<th>Total Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a</td>
<td>8</td>
<td>20</td>
</tr>
<tr>
<td>2a</td>
<td>8</td>
<td>50</td>
</tr>
<tr>
<td>3a</td>
<td>8</td>
<td>60</td>
</tr>
<tr>
<td>4a</td>
<td>8</td>
<td>60</td>
</tr>
<tr>
<td>5a</td>
<td>18</td>
<td>126</td>
</tr>
<tr>
<td>6a</td>
<td>18</td>
<td>200</td>
</tr>
<tr>
<td>7a</td>
<td>24</td>
<td>200</td>
</tr>
<tr>
<td>8a</td>
<td>24</td>
<td>400</td>
</tr>
<tr>
<td>9a</td>
<td>36</td>
<td>400</td>
</tr>
<tr>
<td>10a</td>
<td>36</td>
<td>600</td>
</tr>
</tbody>
</table>

Table 8.5 compares mean fitness values from our four algorithms and other algorithms (Hinterding & Khan, 1994; Liang et al., 2002; Lu, Wang, & Chen, 2008). The average and the maximum fitness values are calculated for other algorithms and our shadow price based algorithms (Algorithm B, C, and D). A higher fitness value means less waste, higher trim efficiency and a fewer number of stocks with waste. Figures 8.4 and 8.5 chart the average and the maximum fitness values.

<table>
<thead>
<tr>
<th>Case</th>
<th>Other Algorithms</th>
<th>Our New Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a</td>
<td>0.8056</td>
<td>0.9133</td>
</tr>
<tr>
<td>2a</td>
<td>0.8912</td>
<td>0.9231</td>
</tr>
<tr>
<td>3a</td>
<td>0.9921</td>
<td>1</td>
</tr>
<tr>
<td>4a</td>
<td>0.9113</td>
<td>0.9638</td>
</tr>
<tr>
<td>5a</td>
<td>0.8312</td>
<td>0.8481</td>
</tr>
<tr>
<td>6a</td>
<td>0.889</td>
<td>0.9389</td>
</tr>
<tr>
<td>7a</td>
<td>0.9529</td>
<td>0.9796</td>
</tr>
<tr>
<td>8a</td>
<td>0.884</td>
<td>0.9567</td>
</tr>
<tr>
<td>9a</td>
<td>0.9003</td>
<td>0.9701</td>
</tr>
<tr>
<td>10a</td>
<td>0.899</td>
<td>0.9735</td>
</tr>
</tbody>
</table>
Figure 8.4 Average Mean Fitness Value Comparison

Figure 8.5 Maximum Mean Fitness Value Comparisons

Table 8.6 compares total waste from our four algorithms and other algorithms (Chiong, Chang, Chai, & Wong, 2008; Liang et al., 2002). The average and the minimum total wastes are calculated and charted (figures 8.6 and 8.7) for other algorithms and our shadow price based
algorithms. Table 8.7 compares the number of stocks with waste among our four algorithms and other algorithms. The average and minimum values are calculated in table 8.7 and charted in figures 8.8 and 8.9. In both comparisons, solutions with less total waste and less number of stocks with waste are better.

Table 8.6
Total Waste Comparison

*With 53 stocks, the minimum total waste is 11450. 11370 is a typo by the authors.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1a</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2a</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>3a</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4a</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>5a</td>
<td>11370*</td>
<td>11966</td>
<td>11450</td>
<td>11622</td>
<td>11450</td>
<td>11450</td>
<td>11450</td>
<td>11450</td>
</tr>
<tr>
<td>6a</td>
<td>240.6</td>
<td>309.4</td>
<td>120.2</td>
<td>223.4</td>
<td>120.2</td>
<td>103</td>
<td>103</td>
<td>103</td>
</tr>
<tr>
<td>7a</td>
<td>84</td>
<td>189.6</td>
<td>84</td>
<td>119.2</td>
<td>84</td>
<td>84</td>
<td>84</td>
<td>84</td>
</tr>
<tr>
<td>8a</td>
<td>308</td>
<td>788</td>
<td>200</td>
<td>432</td>
<td>200</td>
<td>104</td>
<td>92</td>
<td>92</td>
</tr>
<tr>
<td>9a</td>
<td>250</td>
<td>730</td>
<td>142</td>
<td>374</td>
<td>142</td>
<td>94</td>
<td>106</td>
<td>22</td>
</tr>
<tr>
<td>10a</td>
<td>190</td>
<td>1037.2</td>
<td>166</td>
<td>464.4</td>
<td>166</td>
<td>118</td>
<td>130</td>
<td>10</td>
</tr>
</tbody>
</table>

Figure 8.6 Average Total Waste Comparisons
Figure 8.7 Minimum Total Waste Comparisons

Table 8.7
Number of Stocks with Waste Comparison

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1a</td>
<td>2.8</td>
<td>2</td>
<td>2</td>
<td>2.3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2a</td>
<td>4.7</td>
<td>4</td>
<td>4</td>
<td>4.2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3a</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4a</td>
<td>3.2</td>
<td>1.02</td>
<td>1</td>
<td>1.7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5a</td>
<td>27.1</td>
<td>22.8</td>
<td>22.2</td>
<td>24.0</td>
<td>22.2</td>
<td>22.4</td>
<td>22</td>
<td>22</td>
<td>22.1</td>
<td>22</td>
</tr>
<tr>
<td>6a</td>
<td>26.5</td>
<td>29.96</td>
<td>23.5</td>
<td>26.7</td>
<td>23.5</td>
<td>21.1</td>
<td>21.1</td>
<td>21</td>
<td>21.1</td>
<td>21</td>
</tr>
<tr>
<td>7a</td>
<td>6.6</td>
<td>7.48</td>
<td>4</td>
<td>6</td>
<td>4</td>
<td>2.5</td>
<td>2.7</td>
<td>1.8</td>
<td>2.3</td>
<td>1.8</td>
</tr>
<tr>
<td>8a</td>
<td>27.4</td>
<td>56.24</td>
<td>30.3</td>
<td>38.0</td>
<td>27.4</td>
<td>19.9</td>
<td>21.2</td>
<td>16.6</td>
<td>19.2</td>
<td>16.6</td>
</tr>
<tr>
<td>9a</td>
<td>17.6</td>
<td>48.54</td>
<td>23.7</td>
<td>29.9</td>
<td>17.6</td>
<td>14.1</td>
<td>15.6</td>
<td>5.3</td>
<td>11.7</td>
<td>5.3</td>
</tr>
<tr>
<td>10a</td>
<td>11.4</td>
<td>73.06</td>
<td>31.7</td>
<td>38.7</td>
<td>11.4</td>
<td>13</td>
<td>12.2</td>
<td>1</td>
<td>8.7</td>
<td>1</td>
</tr>
</tbody>
</table>
For the algorithms speed evaluation, comparing with other published algorithms is difficult since the differences from experimental hardware and implementation software can skew the result badly. So, we compare among our implementation of Hinterding’s algorithm (Algorithm A) and our new algorithms (Algorithm B, C, and D) since they all coded in the same language and tested on the same hardware platform. Table 8.8 lists the average generation
number when the best solution was found and the average time spent for these algorithms.

Figures 8.10 and 8.11 present them in chart.

Table 8.8
Speed Comparison

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1a</td>
<td></td>
<td>3.5</td>
<td>2.3</td>
<td>7.6</td>
<td>2.5</td>
<td>0.48</td>
<td>0.76</td>
<td>0.69</td>
<td>1.48</td>
<td></td>
</tr>
<tr>
<td>2a</td>
<td></td>
<td>21.1</td>
<td>10.1</td>
<td>28.6</td>
<td>9.8</td>
<td>1.06</td>
<td>1.36</td>
<td>1.48</td>
<td>2.16</td>
<td></td>
</tr>
<tr>
<td>3a</td>
<td></td>
<td>11.3</td>
<td>7.9</td>
<td>18.5</td>
<td>4.5</td>
<td>0.84</td>
<td>1.22</td>
<td>1.18</td>
<td>1.82</td>
<td></td>
</tr>
<tr>
<td>4a</td>
<td></td>
<td>44.3</td>
<td>24.1</td>
<td>22.2</td>
<td>7.1</td>
<td>1.02</td>
<td>1.43</td>
<td>1.36</td>
<td>2.10</td>
<td></td>
</tr>
<tr>
<td>5a</td>
<td></td>
<td>226.7</td>
<td>74.2</td>
<td>129.9</td>
<td>125.3</td>
<td>6.93</td>
<td>13.39</td>
<td>19.53</td>
<td>68.85</td>
<td></td>
</tr>
<tr>
<td>6a</td>
<td></td>
<td>522.8</td>
<td>208.5</td>
<td>253.6</td>
<td>133.9</td>
<td>19.48</td>
<td>11.44</td>
<td>13.78</td>
<td>13.32</td>
<td></td>
</tr>
<tr>
<td>7a</td>
<td></td>
<td>650.8</td>
<td>352.7</td>
<td>225.2</td>
<td>90.2</td>
<td>17.43</td>
<td>12.29</td>
<td>11.89</td>
<td>9.59</td>
<td></td>
</tr>
<tr>
<td>8a</td>
<td></td>
<td>890.5</td>
<td>377.8</td>
<td>402.1</td>
<td>243.6</td>
<td>56.10</td>
<td>27.66</td>
<td>36.86</td>
<td>33.79</td>
<td></td>
</tr>
<tr>
<td>9a</td>
<td></td>
<td>849.4</td>
<td>564.9</td>
<td>529.8</td>
<td>450.3</td>
<td>60.75</td>
<td>41.36</td>
<td>50.48</td>
<td>62.10</td>
<td></td>
</tr>
<tr>
<td>10a</td>
<td></td>
<td>986</td>
<td>621.7</td>
<td>686.2</td>
<td>411.2</td>
<td>683.21</td>
<td>58.89</td>
<td>88.13</td>
<td>68.41</td>
<td></td>
</tr>
</tbody>
</table>

Figure 8.10 Best Solution Found Generation Comparisons
8.8 Results Analysis

All experimental results indicated that our proposed shadow price based genetic algorithms B, C and D performed much better than other current algorithms and Algorithm A. Comparing both the average and the best solutions, our new algorithms achieved better quality results than other algorithms. Algorithm D had the best results in all cases. Solution quality was evaluated by the fitness value, the total waste, and the number of stocks with waste. Measured by the generation count when the best solution was found and the total search time, our new algorithms spent about same amount of time as the other algorithm for small cases. But our new algorithms were much faster when the complexity or the size of the case increased (from case 6a to 10a).

Introducing the shadow price concept into GA had two effects. In traditional GA, random search was employed since the GA operators added random patterns into the solution. In our new algorithm, shadow price enabled operators always inserted good patterns into the solution. Inserting good patterns is the only way to improve the quality of the solution. With good
patterns, our new operators guided the search toward the optimal solution with good speed. Since GA is a multi-solution search algorithm, the local optimal traps were avoided by adding new random solutions and some randomness in the new operators. Adding good patterns improved solution’s quality and shortened search time.

The other effect was that the new shadow price enabled operators enforced reusing of good patterns. The random pattern generator in traditional GA did not prompt good pattern reuse since it did not know the quality of the pattern and consecutively generated patterns were different. Always searching for good patterns, the new algorithm enforced good pattern reuse since the same good pattern were generated repeatedly as long as it could be used in the solution. Reusing good patterns improved solution’s quality and algorithm’s search speed.

From 1a to 10a, test cases’ sizes count and total items count increased. This increased their complexity, search space, and search time. Experimental results showed that our algorithms were a little better than other algorithms in solving small cases. This was expected since these cases’ search spaces were small and the opportunity for pattern reuse was limited. In complex cases, our algorithms outperformed others significantly on result quality and speed. In large and complex search spaces, guided searching and pattern reusing enabled our new algorithms to get quality results with speed.

In our algorithms (Shen & Zhang 2012-2), algorithm D achieved better results than algorithm B and C. It also spent more time than the other two. This was because algorithm D employed local search algorithm in two places and others used it only in one place. More local searches enabled algorithm D to get better results but more computations were required for each generation. Table 8 shows algorithm D reached best solutions with fewer generations but spent more time overall since each generation took longer to complete.
In sum, the shadow price based GA operators added guidance to the search process and enabled reusing of good patterns. They empowered our new algorithms to achieve better results with less time than other algorithms. Our experimental results validated our theory and design.

8.9 Production Consideration

In production, there are other important CSP related problems such as the order continuity problem, and the knife changing problem, etc. For example, the order continuity problem was defined to minimize the order open time in a trim (Hinterding & Khan, 1994).

Trim efficiency is very important in production since it is directly related to the production cost and the material waste. Knife changing is an important factor that keeps continuous production. Frequent knife changes may slow down the production process and automatic slitters can cost up to a million dollars. An order’s open time is defined as the time span between its first and the last item produced. A vehicle’s open time is the duration between its first and the last item loaded. As for the continuity problem, the time period that an order is open in a trim is not very important since an order can be shipped using multiple vehicles. The real important issue is how long a vehicle is open since this is constrained by production facilities such as the loading dock space, the warehouse space, etc. It is a production disaster if the produced items cannot be loaded into a vehicle for shipping and there is no warehouse space for storage.

Knife changing and continuity are conflict objectives. Since items for a vehicle may come from different patterns, frequent knife changes facilitate quick vehicle loading and infrequent knife changes prolong the vehicle open time. But both of them are related to the number of different patterns in the trim. Fewer different patterns require less knife changes and faster vehicle loading.
We modified the fitness function to reduce the number of different patterns.

\[ f = 1 - \frac{1}{m+1} \left( \sum_{j=1}^{m} \sqrt{\frac{w_j}{L}} + \sum_{j=1}^{m} \frac{v_j}{m} + \frac{(P_m)^2}{m} \right). \quad (8.9) \]

In the fitness function, \( p \) is the count of different patterns in the trim. We reran our algorithms with the new fitness function and tested cases 6a to 10a since test cases 1a to 5a were too small to produce meaningful results. Table 8.9 presents the mean fitness values and the number of stocks used. Table 8.10 presents the total waste, the number of stocks with waste, and the distinct pattern count.

<table>
<thead>
<tr>
<th>Case</th>
<th>Items</th>
<th>Mean Fitness</th>
<th>Stocks Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>6a</td>
<td>200</td>
<td>0.9710</td>
<td>0.9820</td>
</tr>
<tr>
<td>7a</td>
<td>200</td>
<td>0.9511</td>
<td>0.9665</td>
</tr>
<tr>
<td>8a</td>
<td>400</td>
<td>0.9730</td>
<td>0.9730</td>
</tr>
<tr>
<td>9a</td>
<td>400</td>
<td>0.9699</td>
<td>0.9813</td>
</tr>
<tr>
<td>10a</td>
<td>600</td>
<td>0.9716</td>
<td>0.9895</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Case</th>
<th>Total Waste</th>
<th>Stocks with Waste</th>
<th>Distinct Pattern Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>6a</td>
<td>128.8</td>
<td>103</td>
<td>103</td>
</tr>
<tr>
<td>7a</td>
<td>84</td>
<td>84</td>
<td>84</td>
</tr>
<tr>
<td>8a</td>
<td>200</td>
<td>104</td>
<td>104</td>
</tr>
<tr>
<td>9a</td>
<td>142</td>
<td>118</td>
<td>118</td>
</tr>
<tr>
<td>10a</td>
<td>154</td>
<td>106</td>
<td>118</td>
</tr>
</tbody>
</table>

The test results showed that all three shadow price based algorithms (B, C, D) performed better than the traditional Algorithm A on all measurements of the fitness value, the total stock used, the total waste, the number of stocks with waste, and the distinct pattern count. Algorithms B, C and D showed strength in different measurements. Algorithm D performed the best in the fitness value, the total stock used, the total waste, and the number of stocks with waste.
Algorithm C used the least number of distinct patterns with a little sacrifice of efficiency. Algorithm B’s performance is between Algorithm C and Algorithm D.

**8.10 Summary**

The key to quickly reach optimal or near-optimal solutions for the CSP is to continuously add and reuse good patterns in the trim. Using the shadow price to analyze the current trim, we can easily identify which items need to be improved and which items produce less waste. Instead of using random patterns, our algorithms select patterns with big shadow prices to reduce the waste and improve the trim efficiency. In our new algorithm, shadow price was used directly to generate new patterns.

Our experiments proved that our proposed shadow price based SGA outperformed current bio-inspired algorithms. The experiment of minimizing patterns also demonstrated the versatility of our new algorithm.

**CHAPTER 9 OPTIMIZING THE GREEN COMPUTING PROBLEMS WITH SGA**

**9.1 Introduction**

Green computing is to use computers in environmental friendly ways. Computers consume energy in two common ways, direct and indirect computing related consumption. Energy consumed by supporting devices, such as air conditioning in the data center, is the indirect energy consumption. Energy used by computers is the direct energy consumption. Together, computing related energy consumption is roughly equivalent to the aviation industry’s energy consumption. It accounts for 2% of anthropogenic CO2 from its share of energy consumption (Consortium for School Networking Initiative 2010).
A computer center can host 10,000 or 150,000 servers (Church, Greenberg, & Hamilton 2008). These mega data centers can support many large companies’ daily operations, conduct many e-commerce transactions, perform large scale scientific researches, and provide services to many other clients. These data centers use large amount of energy (Laszewski, Wang, Younge, & He 2009; Wang, Laszewski, Dayal, He, & Furlani 2009). The energy used by the US servers and data centers is significant. It is estimated that they consumed about 61 billion kilowatt-hours (kWh) in 2006 (1.5 percent of total U.S. electricity consumption) for a total electricity cost of about $4.5 billion. If the trend continues, this demand would rise to 12 gigaWatts (GW) by 2011. It would require an additional 10 power plants (US Environmental Protection Agency 2007).

Green energy is electricity generated from renewable sources such as solar, wind, geothermal, biomass, and small hydro. They are renewable sources and more environmentally friendly than traditional electricity generation. They emit little or no air pollution and leave behind no radioactive waste like nuclear. Most importantly, they are naturally replenished by the earth and sun (Yahoo Green, 2010).

Brown energy is power generated from environmentally hostile technology. The vast majority of electricity in the United States comes from coal, nuclear, large hydro, and natural gas plants. They are the single greatest source of air pollution in the United States, contributing to both smog and acid rain. They are the greatest single contributor of global climate change gases including carbon dioxide and nitrogen oxide (Yahoo Green, 2010).

Majority of the power we consumed today is non-renewable environmental hostile energy. In 2006, green energy only accounts for 7% of total US energy supply. Petroleum, coal and natural gas burning generate 86% of the total energy supply (U.S. Energy Information Administration, 2010)
Many research projects have conducted to improve data centers’ energy efficiency, such as improving the design of the data center (Hamann, López, & Stepanchuk, 2010), improving equipment (Cabusao et al., 2010), and improving air conditioning (Iyengar, Schmidt, & Caricari, 2010). They focused on reducing energy consumption and improving supporting devices’ efficiency.

Efficient task scheduling in data center is another approach to save energy. With optimized task scheduling, computers can complete tasks using less energy. It also reduces energy consumptions from supporting devices. Combined energy savings from efficient task scheduling in a large data center can be significant.

Intelligent task scheduling can be categorized as heuristic algorithms (Li, Liu, & Qian, 2009; Miao, Qi, Hou, & Dai, 2007; Wang, Laszewski, Dayal, He, & Furlani, 2009; Wang, Laszewski, Dayal, & Wang, 2010; Xie, Wang, & Wei, 2005; Zhang, Li, & Zhang, 2010), bio-inspired search algorithms (Chang, Wu, Shann, & Chung, 2008; Tian, & Arslan, 2003), and hybrid algorithms derived from them (Liu, Yang, Luo, & Wang, 2006; Miao, Qi, Hou, Dai, & Shi, 2008; Page & Naughton, 2005). Heuristic algorithms can find good solutions among all possible ones, but they do not guarantee that the best will be found. These algorithms usually find a solution close to the optimal and they find it very fast.

Bio-inspired search algorithms find best solutions by simulating nature. The typical algorithms are Genetic Algorithm (GA), Particle Swarm Optimization (PSO), Ant Colony Optimization (ACO), etc. They can find optimal or near optimal solutions. They are less efficient than heuristic algorithms. We used SGA to solve the green computing scheduling problems and achieved very good results.
9.2 Problem Definition

In general, the amount of power an electrical device uses is the product of supplying voltage and the current it draws. The energy consumed is the product of power and time. In addition, computer processor’s speed varies based on the voltage supplied. Within limits, a processor runs faster with higher voltage. Thus, the power consumption of a processor is directly linked to its running speed. Over-clocking is one such technique that speeds up the processor by raising the voltage. The cost of this speed increase is more energy consumption. Tasks can be completed faster with higher speed. It’s an optimization problem to achieve a balance between energy and time.

From green computing perspective, efficient task scheduling can be defined as either minimizing energy consumption with schedule length constraint or minimizing schedule length with energy consumption constraint (Li 2008). The objective of the first problem is to use the least amount of energy to complete all tasks within a given time frame. It is used mainly in real time processing environments. The second problem is to complete tasks as fast as possible under given energy limitation. Its objective is to use energy efficiently and has great usage in mobile computing, sensor network, etc.

The first problem (P1) can be defined as (Zhang, Li, & Zhang, 2010): $n$ computers in a cloud computing system are used to finish $m$ tasks by the deadline time $T$. Assume that $m_i$ tasks $P_k^i$ for $k=1, 2, \ldots, m_i$ are executed on computer $i$ for $m = \sum_{i=1}^{n} m_i$. A changeable speed for task $P_k^i$ is denoted as $S_k^i$ for $i=1, 2, \ldots, n$, and $k=1, 2, \ldots, m_i$. The speed is defined as the number of instructions per second. The number of instructions of task $P_k^i$ is denoted as $R_k^i$. The execution
time for $P^i_k$ on computer $i$ is $\frac{R^i_k}{S^i_k}$. The total execution time for $m_i$ tasks $P^i_k$ on computer $i$ is defined as $T_i = \sum_{k=1}^{m_i} \frac{R^i_k}{S^i_k}$. For example (Table 7.3.1), $m_i$ tasks $P^i_k$ for $m_1 = 4$, $m_2 = 4$, and $m_3 = 3$ on three processors,

<table>
<thead>
<tr>
<th>Processor 1</th>
<th>$P^1_1$</th>
<th>$P^1_2$</th>
<th>$P^1_3$</th>
<th>$P^1_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor 2</td>
<td>$P^2_1$</td>
<td>$P^2_2$</td>
<td>$P^2_3$</td>
<td>$P^2_4$</td>
</tr>
<tr>
<td>Processor 3</td>
<td>$P^3_1$</td>
<td>$P^3_2$</td>
<td>$P^3_3$</td>
<td></td>
</tr>
</tbody>
</table>

The energy for $P^i_k$ on computer $i$ is $E^i_k = C_i R^i_k [S^i_k]^{\alpha_i-1}$ (9.1)

where $C_i$ is a constant, $\alpha_i = 1 + \frac{2}{\phi_i} \geq 3$ for $0 < \phi_i \leq 1$, $i=1, 2, \ldots, n$, and $k=1, 2, \ldots, m_i$.

The total energy is $E = \sum_{i=1}^{n} \sum_{k=1}^{m_i} C_i R^i_k [S^i_k]^{\alpha_i-1}$ (9.2)

The optimization problem for P1 is

Minimize $E = \sum_{i=1}^{n} \sum_{k=1}^{m_i} C_i R^i_k [S^i_k]^{\alpha_i-1}$ (9.3)

Constraints: $1 \leq m_i \leq m - n + 1$, $m = \sum_{i=1}^{n} m_i$, $m > n$, $\sum_{k=1}^{m_i} \frac{R^i_k}{S^i_k} \leq T$ and $a_i \leq S^i_k \leq b_i$ where $a_i$ is the minimum speed and $b_i$ is the maximum speed of computer $i$, respectively, for $i=1, 2, \ldots, n$, and $k=1, 2, \ldots, m_i$. 
The goal of P1 (9.3) is to design a new energy aware task scheduling algorithm that can find an optimal or near optimal schedule to compete all \( m \) tasks on \( n \) computers with minimum or near minimum energy \( E \) by the deadline time \( T \).

The second problem (P2) to be optimized can be defined as, using shortest time to finish \( m \) tasks on \( n \) heterogeneous computers and the total energy can be consumed is less than or equal to \( E \).

\[
\text{Minimize } T = \max_{i \in \mathbb{N}}(T_i) \quad (9.4)
\]

\[
T_i = \sum_{j=1}^{k} \left( \frac{c_i(R_j^i)}{\beta_j^i} \right)^{\frac{1}{\alpha_i-1}} \quad (9.5)
\]

\[
E \geq \sum_{i=1}^{n} E_i \quad (9.6)
\]

In the objective function (9.4), \( T_i \) is the execution time of processor \( i \). Equation (9.5) is the execution time of \( k \) tasks assigned to processor \( i \). Since speed is commonly used in the specification of processor, equation (9.5) can be simplified into

\[
T_i = \sum_{j=1}^{k} \frac{R_j^i}{S_j^i} \quad (9.7)
\]

\[
E_i = c_i \sum_{j=1}^{k} \left( R_j^i \left( S_j^i \right)^{\alpha_i-1} \right) \quad (9.8)
\]

The objective is to find a schedule such that \( m \) tasks are completed in the shortest time and energy consumed is within the constraint \( E \). There are multiple sub optimization problems in the definition, energy, task, and speed. The first is to optimal distributing energy limitation \( E \) to each processor \( E_i \). The second is to optimal assigning tasks \( \{R\} \) to each processor. And the last one is to determine optimal running speed for each task assigned to a processor. All three sub problems are connected. Assigning higher energy to a processor enables it to process more tasks in short period of time. Higher running speed demands more energy. Since the objective is to
minimize the longest running time of a processor, all processors have to cooperate in energy and task assignment. It's a very complicated combinational optimization problem.

It is proven that the schedule length is minimized when all tasks assigned to a processor execute with the same power (Li 2008). To achieve the best result, tasks assigned to the same processor shall be executed at the same speed since power determines speed.

Thus, equation (9.5, 9.7, and 9.8) can be simplified to:

\[ T_i = \left( \frac{c_i (\sum_{j=1}^{k} R_i^j)^{a_i}}{E_j} \right)^{\frac{1}{a_i-1}} \]  
\[ (9.5') \]

\[ T_i = \frac{\sum_{j=1}^{k} R_i^j}{S_i} \]  
\[ (9.7') \]

\[ E_i = c_i (\sum_{j=1}^{k} R_i^j) (S_i)^{a_i-1} \]  
\[ (9.8') \]

Since all tasks running with the same speed on the same processor and the objective is to complete the tasks as fast as possible with assigned energy for the processor, we can use formula (9.5') to calculate the executing time, or formula (9.9) to calculate optimal speed. This solves the third sub optimization problem. What we need to solve now are the sub problems of distributing total allowed energy consumption to each processors and assigning tasks to them such that the executing time is minimal.

\[ S_i = \left( \frac{E_i}{c_i (\sum_{j=1}^{k} R_i^j)} \right)^{\frac{1}{a_i-1}} \]  
\[ (9.9) \]

There can be two objective functions when optimizing execution time, minimizing either the concurrent running time on all available processors (the max of all processors’ running time) or the accumulated execution time from all processors (summation of all processors’ running time). Since tasks assigned to a processor shall be executing in the same speed, the later optimal problem becomes quite simple. Optimal can be achieved by selecting the most efficient
processor and assign all tasks to it. We choose to optimize the difficult problem of optimizing concurrent running time (9.4).

Instead of a minimal function, standard deviation on execution time can also be used as the objective function to optimize. It measures the distances from each processors’ execution time to the average. The idea is to make all processors sharing the work load and enforce their execution time closing to the medium. This is a good objective function in general but may not work in a heterogeneous processors environment. In a very diverse setup, processors’ energy and execution efficiency can different significantly from one to the other. There can be optimal solutions that no work is assigned to less efficient processors. A simple minimal function is both efficient in calculation and flexible to cover most scenarios.

Both problem P1 and P2 are integer combinatorial optimization problems. The time and energy consumption calculations are complicated.

**9.3 Shadow Price Guided GA Operator for P1**

Encoding is straightforward for this problem. The solution consist a list of all processors. Each processor has a list of tasks assigned to it. Each task is associated with a few attributes, such as total instruction count, execution speed and time, etc.

Shadow price definition shall reflect the cost of execution each individual task after assigned to a processor. In this problem, it’s the energy consumption of the task. Due to the fact that different tasks have different number of instructions, task energy consumption can’t be used to compare the efficiency of assignments since large task will consume more energy. We can use average energy consumption per instruction as the shadow price. Although this helps comparing assignments efficiency, the evolution direction is still not clear. The goal is to reduce shadow
price, i.e., reduce energy consumption per instruction for a task. There are two methods to achieve this, reducing the task execution speed, assigning task to a more efficient processor.

The minimal power consumption is achieved when all tasks assigned to the processor are running at the same speed (Li 2008). This greatly simplified the calculation. To minimize the processor’s energy consumption, we sum up all instructions from assigned tasks and calculate the minimal energy consumption with the max time allowed. This solves the second optimization sub problem.

Since the optimal speeds for all tasks assigned for a processor are the same, we define the shadow price as the average energy consumption per instruction for P1. Furthermore, we move the shadow price definition to the processor since there is only one value per processor. This also defines the evolution direction and method. That is to reduce processor’s average per instruction energy consumption by moving tasks among processors.

We define two mutation operations (Shen & Zhang, 2011-2), move one task from one processor to another and exchange a task between two processors. We further categorize the operations as original and shadow price guided mutation operations. Here is the complete algorithm.

Begin
1. Validate there is at least one feasible solution.
2. Build initial population.
3. While stop criteria has not met
   3.1 Select a sub population to randomly apply one of the following operations
      • Classic mutation operation (Move). Randomly select two processors and move one randomly selected task from one processor the other.
      • Classic mutation operation (Exchange). Exchange two randomly selected tasks between two randomly selected processors.
      • Shadow priced guided mutation operation (Move).
         (a) Calculate shadow prices for all processors.
         (b) Establish a pool of high shadow priced processors and random select one processor (Pa).
         (c) Establish a pool of low shadow priced processors and random select one processor (Pb).
         (d) Random select one task from Pa and move it to Pb.
      • Shadow priced guided mutation operation (Exchange).
         (a) Calculate shadow prices for all processors.
         (b) Establish a pool of high shadow priced processors and random select one processor (Pa).
(c) Establish a pool of low shadow priced processors and random select one processor (Pb).
(d) Sort Pa and Pb’s tasks based on their instruction count.
(e) Establish a task pool from Pa’s tasks whose instruction counts are more than average and random select one task.
(f) Establish a task pool from Pb’s tasks whose instruction counts are less than average and random select one task.
(g) Exchange the selected tasks between Pa and Pb.

3.2 Add random solutions
3.3 Filter and build next generation
End While

The mutation operation randomly applies one of the four algorithms for each candidate solution. When GA search starts, all four operations have equal opportunities to be used for a given solution. The odds of applying each operation changes with the search algorithm progressing. Especially when search is trapped in a local optimal or getting close to finish, classic mutation operations have better possibilities to be chosen.

9.4 Shadow Price Guided GA Operator for P2

The goal (9.4) of this problem is to schedule $m$ tasks on $n$ computers such that the concurrent execution time is minimal and the total energy consumption is less than or equals to $E$. Since it is most efficient to schedule tasks on the same processor at the same speed (Li 2008), the optimization problem breaks down to two sub problems, optimal distribute energy constraint $E$ to $n$ computers and optimal assign $m$ tasks to $n$ computers. The original third sub optimization problem, minimizing execution time for a processor $i$ with $m_i$ tasks and energy cap of $E_i$, can be solved directly using equation (9.5’). and speed can be calculated using formula (9.9).

There are two steps to construct a solution, distribute energy constraint and assign tasks. It does not impact the solution which task completes first. But both tasks have to be completed before the fitness value can be calculated for a solution.

To solve the scheduling problem, we can either treat it as a nested two optimization problems or an optimization problem with two sub tasks. In the nested optimization problem
scenario, one sub problem will be selected as the parent problem and used to drive the other child problem. For example, if we select energy constraint distribution as the parent problem, the search process starts with creating various combinations of energy assignments to processors. Each energy constraint assignment will be treated as a separate optimization problem and solved individually. Various task assignments are evaluated and the assignment that with the least concurrent execution time is the fitness value for the energy assignment. The search process evolves the parent energy assignments and searches for best task combinations for each new assignment. The process repeats until the optimal solution is found.

We can also treat the two optimization tasks as two separate parameters in the same optimization problem and create a flat model. In the nested model, parent searches for the optimal energy assignment and the child searches for the best task assignments within the parent energy assignment. In the flat model, both parameters work together to optimize the same objective of minimizing solution execution time for all processors. Thus, we can ignore the relationship between these two parameters and only focus on the relationships from the two parameters to the solution. We can tune one parameter at a time and rotate. The process can be repeated until the optimal solution is found.

Nested optimization problems are difficult to solve and takes more time to converge (Shen & Zhang 2012-1). In comparison, flat models are easier to solve since there is only one objective function. The complexity is that there are more parameters in the GA operations. Optimizing nested models use tree search and optimizing flat models use linear search with rotating parameters.

To work with flat model, we define two mutation operations, energy mutation and task mutation. There are two sub energy mutations, exchange energy between two processors and
move some energy from one processor to the other. There are also two sub task mutations, exchange a pair of tasks between two processors and move one task from one processor to the other. The processors and tasks are randomly selected in the operations. There is no preference or direction to move the search process.

Our enhanced mutation operation only moves some energy from one processor to the other. Since the objective is to minimize concurrent execution time and more energy can improve speed, we want to move some energy from a short run time processor to a long run time processor. The long run time processor will benefit from added energy and shorten the run time. But the short run time processor may not have extra energy to give. There may be multiple reasons that cause processor use less time, such as the processor is very efficient and can run very fast with little energy, the processor is assigned with large amount of energy, or the processor is assigned with small tasks. So short run time cannot be used to select energy donor processor. A combination of higher energy and less run time makes a good selection criterion.

In our definition, shadow price represents a component’s potential. Here, shadow price is the combination of a processor’s run time and energy assigned. Run time takes precedence over energy since we are selecting the energy donor processor. A processor’s shadow price is high when it has a short run time and large energy. A processor’s shadow price is low when it has a long run time or a short run time and smaller energy. We want to mutate a processor from high shadow priced state to a low shadow priced state. The mutation direction set by the shadow prices is to mutate a processor with below average run time to a longer run time or less energy state.

The shadow price definition for P1, average energy consumption per instruction or average time spent per instruction, does not work for the task mutation here since each processor
can be assigned with different energy and tasks. The average energy or time per instruction cannot be used to compare among processors. High average energy consumption per instruction can exist for processors with various energy or task assignments. Same fact holds true for time spent per instruction.

The goal of task mutation is to move task from a long running processor to a short time running processor. The task donor processor is easy to pick. It can simply be one of the long run time processors. The receiving processor shall be one of the short run time processors. We need to be very careful about selecting receiving processor since it can dramatically increase its run time. Since we are not rearranging energy in this task mutation, the ideal receiving processor is the one that its energy or execution time is not very sensitive to task increase. That is, task increase is not the most influential factor in a processor’s executing time or energy calculation. Formula (9.5’) shows execution time calculation with fixed energy and (9.8’) shows energy calculation with known speed. Both are exponential functions. In an exponential function, exponent has far bigger impact to the result than the base. In both (9.5’) and (9.8’), task instruction count is in the base and $\alpha$ is in the exponent. Since $\alpha$ is a positive number and greater or equal to 3, $\alpha$ can generate bigger impact to the execution time and energy consumption. While comparing 2 processors with same tasks, the one with bigger $\alpha$ consumes more energy if speeds are the same or takes more execution time if energies are the same. So, it is preferred to add a task to a processor with smaller $\alpha$ since it may cause much small increase to the execution time. We define the shadow prices as the combination of execution time and $\alpha$. We want to mutate the task from a long execution time processor to a processor with short execution time and a smaller $\alpha$. 
Our shadow price enhanced algorithm (Shen & Zhang 2012-3) follows standard GA algorithm framework. To avoid local optimal traps, we combine enhanced mutation with standard mutation operations.

Begin
1. Validate there is at least one feasible solution.
2. Build initial population.
3. While stop criteria has not met Repeat
   3.1 Select a sub population to randomly apply one of the following operations
      a. Energy move mutation operation
         a) Randomly select two processors
         b) Move some energy from one processor to the other processor
         c) Validate the new solution
      b. Energy exchange mutation operation
         a) Randomly select two processors
         b) Exchange energy assignments between them
         c) Validate the new solution
      c. Task move mutation operation
         a) Randomly select two processors
         b) Randomly select a task from one processor
         c) Move the randomly selected task from one processor to the other processor
         d) Validate the new solution
      d. Task exchange mutation operation
         a) Randomly select two processors
         b) Randomly select a task from each processor
         c) Exchange the selected tasks between the two processors
         d) Validate the new solution
      e. Shadow price enhanced energy mutation operation
         a) Sort all processors based on execution time
         b) Split processors into 2 sets, long run time processors and short run time processors
         c) Create a subset from long run time processors to establish an energy receiving processor pool $S_r$
         d) Random select one processor from $S_r$ as the receiving processor $P_r$
         e) Re-short the short run time processor set based on energy assignment
         f) Create a subset from short run time processors to establish an high energy donating processor pool $S_d$
         g) Random select one processor from $S_d$ as the energy donating processor $P_d$
         h) Move some energy from $P_d$ to $P_r$
         i) Validate the new solution
      f. Shadow price enhanced task mutation operation
         a) Sort all processors based on execution time
         b) Split processors into 2 sets, long run time processors and short run time processors
         c) Create a subset from long run time processors to establish an task donating processor pool $S_d$
         d) Random select one processor from $S_d$ as the donating processor $P_d$
         e) Re-short the short run time processor set based on processor’s $\alpha$ value
         f) Create a subset from short run time processors to establish a small $\alpha$ value task receiving processor pool $S_r$
         g) Random select one processor from $S_r$ as the task receiving processor $P_d$
         h) Randomly select one task from $P_d$ and move to $P_r$
         i) Validate the new solution
   3.2 Add random solutions
Shadow price represents a state of a component relative to the current solution. It can take on many different forms. In this green scheduling problem, the shadow price is embedded in the mutation operations due to its complexity. It’s a procedure. It measures the processor execution time, energy consumption, and processor’s attribute $\alpha$. It can greatly improve the search speed and solution quality.

9.5 Experiments for P1

To evaluate our new algorithm, we conducted a comparative study between GA and our new shadow price guided GA. Both algorithms followed the standard GA framework and were identical except the mutation operations used. All four mutation operations were used in the shadow price guided GA and only two classic mutation operations were used in the classic GA. Both algorithms used the same calculation to optimize the power consumption for a processor after tasks have been assigned.

We coded and tested both algorithms in Microsoft C#. All experiments were run on a Lenovo Thinkpad laptop T410 that equipped with Intel Core i5-M520 2.4 GHz CPU and 4 GB of memory running Windows 7. Each test case was run at least 10 times. Results were averaged and reported.

We first located published specifications for commercial released CPUs (Wikipedia, 2010) and selected 20 latest ones for our experiment (Table 9.2). Million instructions per second (MIPS) was used to measure the speed of the processors.
Table 9.2
Published Processor Specification

<table>
<thead>
<tr>
<th>ID</th>
<th>Processor</th>
<th>Inst. / Second (MIPS/MHZ)</th>
<th>Inst. /clock cycle</th>
<th>Year</th>
<th>Min Speed (MIPS)</th>
<th>OverClocking Improvement (%)</th>
<th>Max Speed (MIPS)</th>
<th>C</th>
<th>Φ</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DEC Alpha 21064 EV4</td>
<td>300 / 150</td>
<td>2.7</td>
<td>1992</td>
<td>300</td>
<td>0.09</td>
<td>327</td>
<td>84</td>
<td>0.65</td>
</tr>
<tr>
<td>2</td>
<td>Intel Pentium III</td>
<td>1,354 / 500</td>
<td>2.7</td>
<td>1999</td>
<td>1354</td>
<td>0.15</td>
<td>1557</td>
<td>7</td>
<td>0.75</td>
</tr>
<tr>
<td>3</td>
<td>AMD Athlon</td>
<td>3,561 / 1.2</td>
<td>3</td>
<td>2000</td>
<td>3561</td>
<td>0.23</td>
<td>4380</td>
<td>8</td>
<td>0.57</td>
</tr>
<tr>
<td>4</td>
<td>AMD Athlon XP 2400+</td>
<td>5,935 / 2.0</td>
<td>3</td>
<td>2002</td>
<td>5935</td>
<td>0.14</td>
<td>6766</td>
<td>86</td>
<td>0.68</td>
</tr>
<tr>
<td>5</td>
<td>Pentium 4 Extreme Edition</td>
<td>9,726 / 3.2</td>
<td>3</td>
<td>2003</td>
<td>9726</td>
<td>0.07</td>
<td>10407</td>
<td>74</td>
<td>0.8</td>
</tr>
<tr>
<td>6</td>
<td>AMD Athlon FX-57</td>
<td>12,000 / 2.8</td>
<td>4.3</td>
<td>2005</td>
<td>12000</td>
<td>0.09</td>
<td>13080</td>
<td>50</td>
<td>0.83</td>
</tr>
<tr>
<td>7</td>
<td>AMD Athlon 64 3800+ X2 (Dual Core)</td>
<td>14,564 / 2.0</td>
<td>7.3</td>
<td>2005</td>
<td>14564</td>
<td>0.13</td>
<td>16457</td>
<td>60</td>
<td>0.73</td>
</tr>
<tr>
<td>8</td>
<td>ARM Cortex A8</td>
<td>2,000 / 1.0</td>
<td>2</td>
<td>2005</td>
<td>2000</td>
<td>0.18</td>
<td>2360</td>
<td>52</td>
<td>0.56</td>
</tr>
<tr>
<td>9</td>
<td>Xbox360 IBM &quot;Xenon&quot; Triple Core</td>
<td>19,200 / 3.2</td>
<td>6</td>
<td>2005</td>
<td>19200</td>
<td>0.23</td>
<td>23040</td>
<td>55</td>
<td>0.51</td>
</tr>
<tr>
<td>10</td>
<td>AMD Athlon FX-60 (Dual Core)</td>
<td>18,938 / 2.6</td>
<td>7.3</td>
<td>2006</td>
<td>18938</td>
<td>0.17</td>
<td>22157</td>
<td>62</td>
<td>0.63</td>
</tr>
<tr>
<td>11</td>
<td>Intel Core 2 Extreme X6800</td>
<td>27,079 / 2.93</td>
<td>9.2</td>
<td>2006</td>
<td>27079</td>
<td>0.21</td>
<td>32766</td>
<td>19</td>
<td>0.66</td>
</tr>
<tr>
<td>12</td>
<td>Intel Core 2 Extreme QX6700</td>
<td>49,161 / 2.66</td>
<td>18.5</td>
<td>2006</td>
<td>49161</td>
<td>0.16</td>
<td>57027</td>
<td>22</td>
<td>0.6</td>
</tr>
<tr>
<td>13</td>
<td>PS3 Cell BE (PPE only)</td>
<td>10,240 / 3.2</td>
<td>3.2</td>
<td>2006</td>
<td>10240</td>
<td>0.23</td>
<td>12595</td>
<td>45</td>
<td>0.94</td>
</tr>
<tr>
<td>14</td>
<td>P.A. Semi PA6T-1682M</td>
<td>8,800 / 2.0</td>
<td>4.4</td>
<td>2007</td>
<td>8800</td>
<td>0.25</td>
<td>11000</td>
<td>11</td>
<td>0.75</td>
</tr>
<tr>
<td>15</td>
<td>Intel Core 2 Extreme QX9770</td>
<td>59,455 / 3.2</td>
<td>18.6</td>
<td>2008</td>
<td>59455</td>
<td>0.17</td>
<td>69562</td>
<td>92</td>
<td>0.55</td>
</tr>
<tr>
<td>16</td>
<td>Intel Core i7 Extreme 965EE</td>
<td>76,383 / 3.2</td>
<td>23.9</td>
<td>2008</td>
<td>76383</td>
<td>0.18</td>
<td>90132</td>
<td>11</td>
<td>0.65</td>
</tr>
<tr>
<td>17</td>
<td>AMD Phenom II X4 940 Black Edition</td>
<td>42,820 / 3.0</td>
<td>14.3</td>
<td>2009</td>
<td>42820</td>
<td>0.15</td>
<td>49243</td>
<td>42</td>
<td>0.56</td>
</tr>
<tr>
<td>18</td>
<td>AMD Phenom II X6 1090T</td>
<td>68,200 / 3.2</td>
<td>21.3</td>
<td>2010</td>
<td>68200</td>
<td>0.12</td>
<td>76384</td>
<td>77</td>
<td>0.91</td>
</tr>
<tr>
<td>19</td>
<td>Intel Core i7 Extreme Edition i980EE</td>
<td>147,600 / 3.3</td>
<td>44.7</td>
<td>2010</td>
<td>147600</td>
<td>0.14</td>
<td>168264</td>
<td>29</td>
<td>0.71</td>
</tr>
<tr>
<td>20</td>
<td>IBM 5.2-GHz x196</td>
<td>52,286 / 5.2</td>
<td>10.05</td>
<td>2010</td>
<td>52286</td>
<td>0.15</td>
<td>60129</td>
<td>66</td>
<td>0.69</td>
</tr>
</tbody>
</table>

The energy consumption for task \( k \) on computer \( i \), \( P_k^i \), is \( E_k^i = C_i R_k^i [S_k^i]^\alpha_i^{-1} \) and \( \alpha_i = 1 + \frac{2}{\phi_i} \geq 3 \) for \( 0 < \phi_i \leq 1 \). To calculate processor’s energy, we need to define constants \( C \) and \( \phi \) for each processor. Since \( a_i \leq S_k^i \leq b_i \) and speed \( S_k^i \), varies based on task, processor assigned and time constraint, we also need to define the minimum and maximum speed for each processor.
The published CPU specifications define speed certified by manufacture. The CPU is most stable at this speed. A lot experiments have been done to improve their speed by overclocking. Overclocking consumes more energy. For our experiments, we use published speed as the processor’s minimum speed. We use a random number between 5% and 25% as the overclocking speed improvement to define the maximum speed for each processor (Table 7.3.2).

We randomly generated constants $C$ and $\phi$ for each processor. To improve the quality of random number, we used public available true random number generating services (Random,

<table>
<thead>
<tr>
<th>$C_p$</th>
<th>$C_t$</th>
<th>$E_{ga}$</th>
<th>$E_{ga}$</th>
<th>$E_{ga-Ega}$</th>
<th>$E_{ga}$</th>
<th>$E_{ga-Ega}$</th>
<th>$E_{ga}$</th>
<th>$E_{ga-Ega}$</th>
<th>$E_{ga}$</th>
<th>$E_{ga-Ega}$</th>
<th>$E_{ga}$</th>
<th>$E_{ga-Ega}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>700</td>
<td>171421</td>
<td>979031</td>
<td>269031</td>
<td>269031</td>
<td>979031</td>
<td>269031</td>
<td>979031</td>
<td>269031</td>
<td>979031</td>
<td>269031</td>
<td>979031</td>
</tr>
<tr>
<td>100</td>
<td>1000</td>
<td>171421</td>
<td>979031</td>
<td>269031</td>
<td>269031</td>
<td>979031</td>
<td>269031</td>
<td>979031</td>
<td>269031</td>
<td>979031</td>
<td>269031</td>
<td>979031</td>
</tr>
<tr>
<td>150</td>
<td>500</td>
<td>410421</td>
<td>309021</td>
<td>139021</td>
<td>139021</td>
<td>309021</td>
<td>139021</td>
<td>309021</td>
<td>139021</td>
<td>309021</td>
<td>139021</td>
<td>309021</td>
</tr>
<tr>
<td>150</td>
<td>1000</td>
<td>410421</td>
<td>309021</td>
<td>139021</td>
<td>139021</td>
<td>309021</td>
<td>139021</td>
<td>309021</td>
<td>139021</td>
<td>309021</td>
<td>139021</td>
<td>309021</td>
</tr>
<tr>
<td>200</td>
<td>1000</td>
<td>410421</td>
<td>309021</td>
<td>139021</td>
<td>139021</td>
<td>309021</td>
<td>139021</td>
<td>309021</td>
<td>139021</td>
<td>309021</td>
<td>139021</td>
<td>309021</td>
</tr>
<tr>
<td>200</td>
<td>3000</td>
<td>410421</td>
<td>309021</td>
<td>139021</td>
<td>139021</td>
<td>309021</td>
<td>139021</td>
<td>309021</td>
<td>139021</td>
<td>309021</td>
<td>139021</td>
<td>309021</td>
</tr>
<tr>
<td>500</td>
<td>1000</td>
<td>103821</td>
<td>123821</td>
<td>403821</td>
<td>403821</td>
<td>123821</td>
<td>403821</td>
<td>123821</td>
<td>403821</td>
<td>123821</td>
<td>403821</td>
<td>123821</td>
</tr>
<tr>
<td>500</td>
<td>500</td>
<td>103821</td>
<td>123821</td>
<td>403821</td>
<td>403821</td>
<td>123821</td>
<td>403821</td>
<td>123821</td>
<td>403821</td>
<td>123821</td>
<td>403821</td>
<td>123821</td>
</tr>
</tbody>
</table>

Table 9.3

Energy Consumption Comparison
2010) instead of using C# library to generate pseudo random numbers. Table 3 list the minimum speed, overclocking improvement, maximum speed, constants $C$ and $\phi$ for each processor used in our experiments.

We also used random number generate service (Random, 2010) to generate tasks’ instruction count for our experiments. We set the range of instruction count between 500 and 100,000.

Experiment cases were created using different combination of processor count and task count. Time constraint for each experiment case was randomly created first. It was validated to ensure that there are feasible solutions. Then, it was shortened to ensure not many processors can be idle in the optimal solutions. This was to avoid the situation that all tasks were assigned to a few high efficient processors.

The first test compared final solution quality between two algorithms. Table 9.3 compares average energy consumptions under different maximum generation limits ($G_{max}$). For each combination of CPU count ($C_p$) and task count ($C_t$), it lists GA energy consumption ($E_{ga}$), SGA energy consumption ($E_{sga}$), and there difference ($E_{ga} - E_{sga}$). Since the objective is to minimize energy usage, $E_{ga} - E_{sga}$ greater than 0 states SGA is better than GA and vice versa. Table 9.3 shows for all the test cases and maximum generation limits, SGA used less energy than GA to complete the tasks. SGA achieved better solution than GA.

Next, we conducted speed test between the two algorithms. For each test case, we used average energy consumption from above test as the stopping criteria. There is no generation limit. Algorithm only stops when solution is equal or better than the target energy usage. Table 9.4 lists the testing result. It lists generations used ($G_{ga}$, $G_{sga}$) and time used ($T_{ga}$, $T_{sga}$). It also computes the difference ($G_{ga} - G_{sga}$, $T_{ga} - T_{sga}$). In this test, less generation and time used is
better. If $Gga$-$Gsga$ or $Tga$-$Tsga$ is greater than 0, SGA use less generation or time than GA. SGA is faster than GA to achieve the same result quality and vice versa.

Table 9.4
Speed Comparison

<table>
<thead>
<tr>
<th>Cp</th>
<th>Ct</th>
<th>Gga</th>
<th>Gsga</th>
<th>Gga-Gsga</th>
<th>Tga</th>
<th>Tsga</th>
<th>Tga-Tsga</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>500</td>
<td>633</td>
<td>403</td>
<td>230</td>
<td>1.297</td>
<td>0.834</td>
<td>0.463</td>
</tr>
<tr>
<td>10</td>
<td>1000</td>
<td>814</td>
<td>600</td>
<td>214</td>
<td>4.124</td>
<td>3.058</td>
<td>1.066</td>
</tr>
<tr>
<td>10</td>
<td>1500</td>
<td>936</td>
<td>715</td>
<td>221</td>
<td>8.935</td>
<td>6.622</td>
<td>2.313</td>
</tr>
<tr>
<td>10</td>
<td>3000</td>
<td>1113</td>
<td>808</td>
<td>305</td>
<td>29.792</td>
<td>21.956</td>
<td>7.836</td>
</tr>
<tr>
<td>10</td>
<td>5000</td>
<td>1022</td>
<td>805</td>
<td>217</td>
<td>68.884</td>
<td>54.008</td>
<td>14.876</td>
</tr>
<tr>
<td>20</td>
<td>500</td>
<td>941</td>
<td>650</td>
<td>291</td>
<td>1.871</td>
<td>1.31</td>
<td>0.561</td>
</tr>
<tr>
<td>20</td>
<td>1000</td>
<td>820</td>
<td>675</td>
<td>145</td>
<td>3.342</td>
<td>2.716</td>
<td>0.626</td>
</tr>
<tr>
<td>20</td>
<td>1500</td>
<td>707</td>
<td>587</td>
<td>120</td>
<td>4.826</td>
<td>4.014</td>
<td>0.812</td>
</tr>
<tr>
<td>20</td>
<td>2000</td>
<td>832</td>
<td>691</td>
<td>141</td>
<td>8.211</td>
<td>6.919</td>
<td>1.292</td>
</tr>
<tr>
<td>20</td>
<td>5000</td>
<td>1007</td>
<td>817</td>
<td>190</td>
<td>42.62</td>
<td>34.748</td>
<td>7.872</td>
</tr>
<tr>
<td>30</td>
<td>500</td>
<td>967</td>
<td>634</td>
<td>333</td>
<td>1.755</td>
<td>1.182</td>
<td>0.573</td>
</tr>
<tr>
<td>30</td>
<td>1000</td>
<td>943</td>
<td>685</td>
<td>258</td>
<td>3.308</td>
<td>2.53</td>
<td>0.778</td>
</tr>
<tr>
<td>30</td>
<td>1500</td>
<td>962</td>
<td>755</td>
<td>207</td>
<td>5.337</td>
<td>4.171</td>
<td>1.166</td>
</tr>
<tr>
<td>30</td>
<td>2000</td>
<td>1003</td>
<td>762</td>
<td>241</td>
<td>7.899</td>
<td>6.1</td>
<td>1.799</td>
</tr>
<tr>
<td>30</td>
<td>3000</td>
<td>1084</td>
<td>832</td>
<td>252</td>
<td>13.893</td>
<td>10.435</td>
<td>3.458</td>
</tr>
<tr>
<td>30</td>
<td>5000</td>
<td>1141</td>
<td>930</td>
<td>211</td>
<td>29.71</td>
<td>23.837</td>
<td>5.873</td>
</tr>
<tr>
<td>40</td>
<td>500</td>
<td>965</td>
<td>652</td>
<td>313</td>
<td>1.876</td>
<td>1.292</td>
<td>0.584</td>
</tr>
<tr>
<td>40</td>
<td>1000</td>
<td>961</td>
<td>733</td>
<td>228</td>
<td>3.195</td>
<td>2.396</td>
<td>0.799</td>
</tr>
<tr>
<td>40</td>
<td>1500</td>
<td>965</td>
<td>742</td>
<td>223</td>
<td>4.673</td>
<td>3.791</td>
<td>0.882</td>
</tr>
<tr>
<td>40</td>
<td>2000</td>
<td>1027</td>
<td>793</td>
<td>234</td>
<td>6.848</td>
<td>5.138</td>
<td>1.71</td>
</tr>
<tr>
<td>40</td>
<td>3000</td>
<td>1131</td>
<td>849</td>
<td>282</td>
<td>12.402</td>
<td>9.776</td>
<td>2.626</td>
</tr>
<tr>
<td>40</td>
<td>5000</td>
<td>1222</td>
<td>916</td>
<td>306</td>
<td>26.614</td>
<td>20.318</td>
<td>6.296</td>
</tr>
<tr>
<td>50</td>
<td>500</td>
<td>964</td>
<td>613</td>
<td>351</td>
<td>1.973</td>
<td>1.292</td>
<td>0.681</td>
</tr>
<tr>
<td>50</td>
<td>1000</td>
<td>948</td>
<td>688</td>
<td>260</td>
<td>2.962</td>
<td>2.158</td>
<td>0.804</td>
</tr>
<tr>
<td>50</td>
<td>1500</td>
<td>1056</td>
<td>761</td>
<td>295</td>
<td>4.674</td>
<td>3.372</td>
<td>1.302</td>
</tr>
<tr>
<td>50</td>
<td>2000</td>
<td>1009</td>
<td>807</td>
<td>202</td>
<td>5.829</td>
<td>4.62</td>
<td>1.209</td>
</tr>
<tr>
<td>50</td>
<td>3000</td>
<td>1087</td>
<td>856</td>
<td>231</td>
<td>10.858</td>
<td>8.578</td>
<td>2.28</td>
</tr>
<tr>
<td>50</td>
<td>5000</td>
<td>1179</td>
<td>919</td>
<td>260</td>
<td>21.367</td>
<td>16.899</td>
<td>4.468</td>
</tr>
</tbody>
</table>

Since table 9.4 shows all values of $Gga$-$Gsga$ and $Tga$-$Tsga$ are greater than 0, GA used more time than SGA to find equivalent results. SGA is faster than GA to find targeted result.

9.6 Experiments for P2

Similar to P1, we also conducted a comprehensive comparative study between GA and our new shadow price enhanced GA. Same set of testing data and environment was used.
Step 1 of the algorithm checks for the existence of a valid solution. Formula (9.8) and (9.8’) show that energy consumption is at the lowest level when the speed is minimized for a given processor. To check if a processor can complete the tasks with limited energy, we only need to test it at its lowest speed. To check existence of at least one valid solution, we test all tasks for each processor at its lowest speed and compare energy consumptions. If there is one processor consumes less than or equal to energy constraint, there is at least one valid solution exist for the problem. Otherwise, there is no feasible solution for the problem and algorithm aborts.

We studied algorithms’ performance in two aspects, result quality and convergence speed. For result quality, we test algorithm with various test cases under fixed energy constraint and fixed generation of evolutions.

Tables 9.5-9.9 show comparison test results between GA and SPGA. To make it easy to read, only the integer portion of data is displayed. The processor count ranges from 10 to 50. The task count \( R \) ranges from 500 to 5000. The max generation limits \( (G_{max}) \) are 500, 1000, 1500, 2000, 3000, and 5000. All combinations of task count \( R \) and generation limit \( G_{max} \) are tested for each processor count setup. Each test case was run at least 10 times. Results were averaged and reported. The improvement percentages from SPGA optimal solution \( (T_{spga}) \) over GA optimal solutions \( (T_{ga}) \) are reported in the tables. Since the objective is to minimize concurrent execution time, a positive number shows GA’s best solution takes long time than SPGA and SPGA’s result is better than GA. Tables 9.5-9.9 show all positive results. SPGA best solutions used less execution time than GA best solutions in all test cases. SPGA reached better solutions than GA.
### Table 9.5 SPGA Time Improvement over GA ($T_{ga} - T_{spga}$)$ \times 100/T_{ga}$ for 10 Processors

<table>
<thead>
<tr>
<th>$G_{\text{max}}$</th>
<th>$R = 500$</th>
<th>$R = 1000$</th>
<th>$R = 1500$</th>
<th>$R = 2000$</th>
<th>$R = 3000$</th>
<th>$R = 5000$</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>44</td>
<td>25</td>
<td>15</td>
<td>8</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>1000</td>
<td>85</td>
<td>56</td>
<td>30</td>
<td>22</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>1500</td>
<td>84</td>
<td>76</td>
<td>51</td>
<td>37</td>
<td>19</td>
<td>8</td>
</tr>
<tr>
<td>2000</td>
<td>74</td>
<td>78</td>
<td>71</td>
<td>50</td>
<td>27</td>
<td>13</td>
</tr>
<tr>
<td>3000</td>
<td>57</td>
<td>70</td>
<td>77</td>
<td>75</td>
<td>45</td>
<td>24</td>
</tr>
<tr>
<td>5000</td>
<td>23</td>
<td>58</td>
<td>68</td>
<td>72</td>
<td>76</td>
<td>50</td>
</tr>
</tbody>
</table>

### Table 9.6 SPGA Time Improvement over GA ($T_{ga} - T_{spga}$)$ \times 100/T_{ga}$ for 20 Processors

<table>
<thead>
<tr>
<th>$G_{\text{max}}$</th>
<th>$R = 500$</th>
<th>$R = 1000$</th>
<th>$R = 1500$</th>
<th>$R = 2000$</th>
<th>$R = 3000$</th>
<th>$R = 5000$</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>76</td>
<td>58</td>
<td>49</td>
<td>39</td>
<td>28</td>
<td>26</td>
</tr>
<tr>
<td>1000</td>
<td>69</td>
<td>79</td>
<td>70</td>
<td>60</td>
<td>55</td>
<td>41</td>
</tr>
<tr>
<td>1500</td>
<td>57</td>
<td>79</td>
<td>79</td>
<td>74</td>
<td>62</td>
<td>49</td>
</tr>
<tr>
<td>2000</td>
<td>49</td>
<td>78</td>
<td>81</td>
<td>80</td>
<td>73</td>
<td>56</td>
</tr>
<tr>
<td>3000</td>
<td>42</td>
<td>70</td>
<td>76</td>
<td>80</td>
<td>79</td>
<td>67</td>
</tr>
<tr>
<td>5000</td>
<td>31</td>
<td>56</td>
<td>69</td>
<td>75</td>
<td>78</td>
<td>77</td>
</tr>
</tbody>
</table>

### Table 9.7 SPGA Time Improvement over GA ($T_{ga} - T_{spga}$)$ \times 100/T_{ga}$ for 30 Processors

<table>
<thead>
<tr>
<th>$G_{\text{max}}$</th>
<th>$R = 500$</th>
<th>$R = 1000$</th>
<th>$R = 1500$</th>
<th>$R = 2000$</th>
<th>$R = 3000$</th>
<th>$R = 5000$</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>88</td>
<td>75</td>
<td>61</td>
<td>49</td>
<td>25</td>
<td>18</td>
</tr>
<tr>
<td>1000</td>
<td>90</td>
<td>86</td>
<td>79</td>
<td>72</td>
<td>54</td>
<td>36</td>
</tr>
<tr>
<td>1500</td>
<td>85</td>
<td>90</td>
<td>86</td>
<td>80</td>
<td>68</td>
<td>53</td>
</tr>
<tr>
<td>2000</td>
<td>81</td>
<td>87</td>
<td>90</td>
<td>86</td>
<td>82</td>
<td>68</td>
</tr>
<tr>
<td>3000</td>
<td>69</td>
<td>82</td>
<td>88</td>
<td>89</td>
<td>88</td>
<td>79</td>
</tr>
<tr>
<td>5000</td>
<td>46</td>
<td>76</td>
<td>82</td>
<td>86</td>
<td>90</td>
<td>86</td>
</tr>
</tbody>
</table>

### Table 9.8 SPGA Time Improvement over GA ($T_{ga} - T_{spga}$)$ \times 100/T_{ga}$ for 40 Processors

<table>
<thead>
<tr>
<th>$G_{\text{max}}$</th>
<th>$R = 500$</th>
<th>$R = 1000$</th>
<th>$R = 1500$</th>
<th>$R = 2000$</th>
<th>$R = 3000$</th>
<th>$R = 5000$</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>81</td>
<td>67</td>
<td>56</td>
<td>45</td>
<td>31</td>
<td>39</td>
</tr>
<tr>
<td>1000</td>
<td>81</td>
<td>83</td>
<td>74</td>
<td>68</td>
<td>60</td>
<td>53</td>
</tr>
<tr>
<td>1500</td>
<td>76</td>
<td>85</td>
<td>84</td>
<td>79</td>
<td>73</td>
<td>61</td>
</tr>
<tr>
<td>2000</td>
<td>70</td>
<td>83</td>
<td>87</td>
<td>85</td>
<td>80</td>
<td>66</td>
</tr>
<tr>
<td>3000</td>
<td>58</td>
<td>79</td>
<td>84</td>
<td>86</td>
<td>86</td>
<td>74</td>
</tr>
<tr>
<td>5000</td>
<td>43</td>
<td>71</td>
<td>79</td>
<td>82</td>
<td>85</td>
<td>85</td>
</tr>
</tbody>
</table>
Table 9.9 SPGA Time Improvement over GA ($T_{ga} - T_{spga}$) x 100/$T_{ga}$ for 50 Processors

<table>
<thead>
<tr>
<th>$G_{max}$</th>
<th>$R=500$</th>
<th>$R=1000$</th>
<th>$R=1500$</th>
<th>$R=2000$</th>
<th>$R=3000$</th>
<th>$R=5000$</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>88</td>
<td>74</td>
<td>69</td>
<td>63</td>
<td>52</td>
<td>37</td>
</tr>
<tr>
<td>1000</td>
<td>90</td>
<td>85</td>
<td>81</td>
<td>76</td>
<td>72</td>
<td>61</td>
</tr>
<tr>
<td>1500</td>
<td>88</td>
<td>91</td>
<td>89</td>
<td>84</td>
<td>80</td>
<td>72</td>
</tr>
<tr>
<td>2000</td>
<td>85</td>
<td>89</td>
<td>91</td>
<td>88</td>
<td>84</td>
<td>76</td>
</tr>
<tr>
<td>3000</td>
<td>80</td>
<td>85</td>
<td>89</td>
<td>91</td>
<td>90</td>
<td>83</td>
</tr>
<tr>
<td>5000</td>
<td>61</td>
<td>80</td>
<td>85</td>
<td>88</td>
<td>91</td>
<td>89</td>
</tr>
</tbody>
</table>

To study algorithms’ convergence speed, we reran all test cases with same energy constraints. Instead of limiting max generations, we set a target fitness value for the algorithms. The search only stops when the best solution’s execution time meets the target value. The algorithm can take as much time or generations as needed to reach the target. Average execution times from above test cases were used as the target value.

Table 9.10 SPGA Search Speed Improvement in Time(s), $ST_{ga}-ST_{spga}$

<table>
<thead>
<tr>
<th>$P_c$</th>
<th>$R=500$</th>
<th>$R=1000$</th>
<th>$R=1500$</th>
<th>$R=2000$</th>
<th>$R=3000$</th>
<th>$R=5000$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3</td>
<td>3</td>
<td>10</td>
<td>18</td>
<td>33</td>
<td>60</td>
</tr>
<tr>
<td>20</td>
<td>3</td>
<td>8</td>
<td>12</td>
<td>15</td>
<td>36</td>
<td>52</td>
</tr>
<tr>
<td>30</td>
<td>5</td>
<td>7</td>
<td>10</td>
<td>13</td>
<td>21</td>
<td>42</td>
</tr>
<tr>
<td>40</td>
<td>6</td>
<td>8</td>
<td>12</td>
<td>17</td>
<td>31</td>
<td>43</td>
</tr>
<tr>
<td>50</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>38</td>
</tr>
</tbody>
</table>

Tests were run for each combination of processor count ($P_c$) and task count $R$. Each test was run at least ten times. Results were averaged and reported. Table 9.10 shows SPGA’s search time ($ST_{spga}$) savings over GA’s search time ($ST_{ga}$), $ST_{ga}-ST_{spga}$. A positive value shows GA takes longer time than SPGA to reach equivalent results. SPGA is faster with a positive value. Table 9.11 compares evolution generations used from GA ($G_{ga}$) over SPGA ($G_{spga}$), $G_{ga}-G_{spga}$. A positive value states that GA took more generations of evolution than the SPGA to reach targeted solutions. Both tables show SPGA is faster than GA. Table 9.10 measures the speed in search time and Table 9.11 measures speed in evolution generations.
Table 9.11 SPGA Search Speed Improvement in Generations, $G_{ga}-G_{spga}$

<table>
<thead>
<tr>
<th>$P_c$</th>
<th>$R=500$</th>
<th>$R=1000$</th>
<th>$R=1500$</th>
<th>$R=2000$</th>
<th>$R=3000$</th>
<th>$R=5000$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>791</td>
<td>538</td>
<td>783</td>
<td>845</td>
<td>782</td>
<td>407</td>
</tr>
<tr>
<td>20</td>
<td>945</td>
<td>1471</td>
<td>1424</td>
<td>1157</td>
<td>1456</td>
<td>1012</td>
</tr>
<tr>
<td>30</td>
<td>1154</td>
<td>1172</td>
<td>1273</td>
<td>1145</td>
<td>1115</td>
<td>1132</td>
</tr>
<tr>
<td>40</td>
<td>1327</td>
<td>1256</td>
<td>1454</td>
<td>1489</td>
<td>1714</td>
<td>1331</td>
</tr>
<tr>
<td>50</td>
<td>1479</td>
<td>1435</td>
<td>1365</td>
<td>1387</td>
<td>1193</td>
<td>1239</td>
</tr>
</tbody>
</table>

All test data and studies showed that final schedules from SPGA used less time to complete all tasks than final schedules from GA. SPGA achieved better solutions than GA. SPGA also used less time and fewer generations of evolution than GA to reach optimal solutions. Overall, SPGA find better results than GA and faster than GA.

9.7 Summary

Green energy aware computing is one of the most active research fields. There are many complex and challenging topics. Energy aware task scheduling in a multiple heterogeneous processors environment is a typical problem.

We applied our new shadow price guided GA to solve the energy aware task scheduling problems and achieved very good results. Experiments showed our new algorithm achieved better results than the standard GA and used less time.

CHAPTER 10 OPTIMIZING THE STOCK REDUCTION PROBLEM WITH SGA

10.1 Introduction

In production, the CSP is directly linked to the stock assortment in the inventory. Increasing the number of different length stocks can reduce the waste from stock cutting. On the other hand, inventory incurs all kinds of expenses, such as stock cost, warehouse management, air conditioning, stock aging, etc. Efficient inventory management calls for simple stock
assortment and minimal stock on hand while still meeting production requirements with the least waste.

It is an NP hard problem to choose the minimal stock mix and still maintains high trim efficiency with low waste. The parent is a Minimizing Stock Mix Problem (MSMP), and the children are CSPs. This general problem is commonly referred to as the Stock Reduction Problem (SRP). It is an integer combinatorial optimization problem and GA is a good choice to solve it. It can solve the parent’s integer combinatorial MSMP and the children CSPs with integer results. However, GA takes a long time to solve complex CSPs. Furthermore, it can be very time consuming to use an algorithm that nesting GA within another GA to solve the SRP.

LP algorithms are efficient but limited to linear objective functions and best at non-integer problems. GA has little restriction on the objective functions but may take a long time to converge. A hybrid algorithm merging GA and LP may combine their technical merits to generate satisfactory solutions.

In most LP/GA hybrid algorithms, a divide and conquer strategy is used to separate the problem into sub problems. LP and GA solve sub problems separately based on their strengths. LP solves non-integer problems, and GA solves integer problems. These hybrid algorithms may not very efficient at solving the SRP since both the MSMP and the CSP are integer optimization problems.

We propose a new hybrid algorithm that uses GA to solve the parent problem (MSMP), and combines LP and GA to solve the sub problems (CSPs). We use LP to improve GA’s performance and GA to improve LP’s integer results. Our test results have shown that our algorithm can solve the SRP effectively.
10.2 Problem Definition

The goal of the SRP is to reduce inventory by minimizing the number of different stocks needed, i.e., simplifying stock assortment. To satisfy daily production requirements and lead-time variability, a certain level of inventory for each stock-keeping unit (SKU) need to be maintained. It is called Safety Stock. It is very expensive to keep a large number of SKUs. Reducing SKUs is a method to lower inventory and cost. There are also other tangible benefits, such as easy management, easy inventory replenishment, etc.

However, it is difficult to define the inventory cost or to measure the cost savings from the stock reduction. There are a lot of different costs in production, and not all of them are in the form of polynomial functions. For example, the space in owned warehouse is free and it is not free if the storage space is rented, warehouse temperature control indirectly links to the inventory level, en-route stock may or may not be included in the cost based on contract, etc. Most times, inventory cost is simply a part of the overall production costs. But there is one kind of cost that is concrete and directly linked to the stock reduction – the cost from trim loss. Reducing stock variety can lower trim efficiency (stock cutting efficiency) and produce more waste. Waste in production is directly linked to cost. Thus, we chose trim efficiency as the objective for the SRP.

The SRP can be defined in two ways. One is to minimize the number of different stocks needed to satisfy demand while maximizing the trim efficiency. The other is to minimize the number of different stocks needed to satisfy demand while meeting a trim efficiency requirement - a threshold.

The two definitions are different but an algorithm that solves the second problem can easily be used to solve the first. We can start with solving the CSP using all stocks and get the best trim efficiency. Then, the first problem can be transformed to the second problem by using
the previous result as the trim efficiency requirement. Solving this problem also provides the correct answer for the first problem. So, we use the second problem definition and define the fitness function as,

\[
\text{Minimize } f = (P + S) \cdot C + L \tag{10.1}
\]

\[
P = \begin{cases} 
1 - (E - E_t) & \text{when } E \geq E_t \\
S & \text{else} 
\end{cases} \tag{10.2}
\]

Where, 
\( S = \) number of different stocks used in solution, 
\( L = \) total length of different stocks used in solution, 
\( E = \) trim efficiency, 
\( E_t = \) trim efficiency threshold, 
\( C = \) constant.

In the fitness function (10.1), \( P \) represents the trim efficiency status. It is less than 1 when the current solution meets the trim efficiency requirement. If there are multiple solutions that meet the requirement, \( P \) also states the preference for high trim efficiency (10.2). If the current solution does not meet the trim efficiency requirement, a big penalty of \( S \) (the number of different stocks used in this solution) is used. \( L \) is to signal the preference for shorter stocks when possible. Constant \( C \) is used to adjust the precision of the trim efficiency.

### 10.3 LP/GA Hybrid Algorithm

In our new hybrid algorithm (Shen & Zhang 2012-1), there are three sub algorithms: (1) GA based stock mix minimizing algorithm, (2) the rule-based chromosome preprocess algorithm, and (3) LP/GA combined cutting stock algorithm.

The stock mix minimizing algorithm responsible for selecting subsets of minimal stocks to create sub CSPs and controlling the overall algorithm. It is based on the traditional GA. The first step is to ensure that there are feasible solutions. It solves the CSP with all available stocks and compares the trim efficiency with the threshold. The algorithm stops if there is no feasible solution (i.e., with all stocks available, the trim efficiency is still worse than the requirement).
Otherwise, it builds up the initial solution pool with random chromosomes. Then, it loops through generations of GA operations until the stopping criterion is met. The stopping criterion is that either the algorithm stops progressing or the max number of generations is reached.

There are three mutation operators in our algorithm: removing one stock from the mix, adding one to the stock mix, and swapping one stock in the mix with an unused stock. The algorithm uses one of the three operators randomly.

---

Begin

1. Build a CSP with all available stocks and solve it. If the solution's trim efficiency is worse than the threshold, the algorithm stops with no solution.

2. Build the initial solution population.
   2.1 Select a random subset of stocks.
   2.2 Build a CSP using the stock subset.
   2.3 Solve the CSP.
   2.4 Store the CSP and result in the solution repository.
   2.5 Repeat steps 2.1 through 2.4 to fill the population.

3. Select a subset from current population and mutate.
   3.1 Select a solution from the subset.
   3.2 Extract the stock list from the solution.
   3.3 Randomly apply one of the following mutation operators to the stock list.
      - Add one stock to the list.
      - Remove one stock from the list.
      - Switch one stock from the list with an unused stock.
   3.4 Create a new CSP using the new stock list.
   3.5 If the new CSP exists in the repository, goto 3.1.
   3.6 Apply the preprocess algorithm to the new CSP. If it can derive the result, goto 3.1.
   3.7 Solve the new CSP.
   3.8 Store the CSP and result in the solution repository.
   3.9 Repeat steps 3.1 through 3.8 for all solutions in the subset.

4. Generate random solutions.
   4.1 Select a random subset of stocks.
   4.2 Build a CSP using the stock subset.
   4.3 If the new CSP exists in the repository, goto step 4.1.
   4.4 Apply the preprocess algorithm to the new CSP. If it can derive the result, goto 4.1.
   4.5 Solve the CSP.
   4.6 Store the CSP and result in the solution repository.
   4.7 Repeat steps 4.1 through 4.6 to generate random solutions.

5. Add new solutions from steps 3 and 4 to the current population and sort the population based on solutions’ fitness values.

6. Select top solutions from the current population to create a new population for the next generation.

7. Repeat steps 3 through 6 till either algorithm stops progressing or max generation is met.

8. Select the best solution from the current population as the final solution.

End
The rule-based chromosome preprocess algorithm trims the workload for the cutting stock algorithm. Based on previously solved problems, we can draw conclusions for certain new stock mixes quickly without actually solving the corresponding CSPs. For example, a stock mix will not meet the requirement if it is a subset of the stock mix from a previously solved CSP whose solution does not meet the trim efficiency requirement. The new CSP does not need to be solved. On the flip side, if a stock mix is a superset of the stock mix from a previously solved CSP whose solution meets the trim efficiency requirement, we can be sure that the new stock mix will meet the requirement and the new CSP does not need to be solved either. Both rules state that solving these new problems will not improve the fitness value and the solution. We can safely skip them to reduce the workload and speed up the algorithm.

Let’s assume that there are two solved problems with stock mix of (10, 20, 30) and (20, 40). The first one does not meet the trim efficiency requirement and the second one does. If there is a new CSP with a stock mix of (10, 20), we do no need to solve it since (10,20) is a subset of (10, 20, 30). Indeed, if the CSP with (10, 20, 30) cannot satisfy the requirement, the new CSP with (10, 20) cannot either. If there is another new CSP with a stock mix of (20, 30, 40) which is a super set of (20, 40), we can just declare that it meets the requirement without actually solving it. Since the objective is to reduce the stock mix and the new CSP with (20, 30, 40) cannot improve the fitness value, we can safely discard it.

---

Begin
1. Extract the stock list from the new CSP.
2. Search the solution repository for a CSP whose stock list contains the current stock list.
3. If a historical CSP is found and its result does not meet the threshold, return the historical CSP’s result and stop.
4. Search the solution repository for a CSP whose stock list is contained by the current stock list.
5. If a historical CSP is found and its result meets the threshold, return the historical CSP’s result and stop.
6. If no historical CSP can be found, return null result.
End
The last one is the cutting stock algorithm. It is used to calculate the fitness function for each sub problem created by the above two algorithms. This algorithm is the key to the performance and the usability of our stock reduction algorithm. We use a sample problem (Table 10.1) to illustrate our new algorithm. It is a real problem from paper industry. Table 10.2 lists the results of our ten runs using SGA to solve the problem.

Table 10.1
Sample CSP

<table>
<thead>
<tr>
<th>Available Stock Lengths</th>
<th>816</th>
<th>832</th>
<th>848</th>
<th>864</th>
<th>880</th>
<th>896</th>
<th>912</th>
<th>928</th>
<th>944</th>
<th>960</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>976</td>
<td>992</td>
<td>1008</td>
<td>1024</td>
<td>1040</td>
<td>1056</td>
<td>1072</td>
<td>1088</td>
<td>1104</td>
<td>1120</td>
</tr>
<tr>
<td></td>
<td>1136</td>
<td>1152</td>
<td>1168</td>
<td>1184</td>
<td>1200</td>
<td>1216</td>
<td>1232</td>
<td>1248</td>
<td>1264</td>
<td>1280</td>
</tr>
<tr>
<td></td>
<td>1296</td>
<td>1312</td>
<td>1328</td>
<td>1344</td>
<td>1360</td>
<td>1376</td>
<td>1392</td>
<td>1408</td>
<td>1424</td>
<td>1440</td>
</tr>
<tr>
<td></td>
<td>1456</td>
<td>1472</td>
<td>1488</td>
<td>1504</td>
<td>1520</td>
<td>1536</td>
<td>1552</td>
<td>1568</td>
<td>1584</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Target Efficiency</th>
<th>0.99</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Item Length</th>
<th>404</th>
<th>408</th>
<th>473</th>
<th>527</th>
<th>545</th>
<th>576</th>
<th>584</th>
<th>585</th>
<th>597</th>
<th>604</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. Required</td>
<td>58</td>
<td>159</td>
<td>105</td>
<td>7</td>
<td>76</td>
<td>1</td>
<td>226</td>
<td>7</td>
<td>42</td>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Item Length</th>
<th>606</th>
<th>636</th>
<th>690</th>
<th>780</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. Required</td>
<td>62</td>
<td>20</td>
<td>9</td>
<td>284</td>
</tr>
</tbody>
</table>

Table 10.2
GA Result of Sample CSP

<table>
<thead>
<tr>
<th>Run</th>
<th>Time (s)</th>
<th>Waste</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3839</td>
<td>6387</td>
<td>0.9900</td>
</tr>
<tr>
<td>2</td>
<td>1345</td>
<td>6323</td>
<td>0.9901</td>
</tr>
<tr>
<td>3</td>
<td>1967</td>
<td>6339</td>
<td>0.9901</td>
</tr>
<tr>
<td>4</td>
<td>2154</td>
<td>6387</td>
<td>0.9900</td>
</tr>
<tr>
<td>5</td>
<td>1323</td>
<td>6355</td>
<td>0.9901</td>
</tr>
<tr>
<td>6</td>
<td>2629</td>
<td>6371</td>
<td>0.9901</td>
</tr>
<tr>
<td>7</td>
<td>1737</td>
<td>6307</td>
<td>0.9902</td>
</tr>
<tr>
<td>8</td>
<td>1534</td>
<td>6291</td>
<td>0.9902</td>
</tr>
<tr>
<td>9</td>
<td>1405</td>
<td>6387</td>
<td>0.9900</td>
</tr>
<tr>
<td>10</td>
<td>2075</td>
<td>6067</td>
<td>0.9905</td>
</tr>
<tr>
<td>Average</td>
<td>2001</td>
<td>6321</td>
<td>0.9901</td>
</tr>
</tbody>
</table>
Since the target trim efficiency for the sample problem is 0.99, GA stops when it reaches the target. It does not mean that 0.9905 is the best trim efficiency for this problem. If we remove the target or raise it, the average trim efficiency shall be better than 0.9901. But the real challenge is that GA took an average of 2001 seconds to solve the problem. That’s about 33 minutes. The best time was 22 minutes and the worst time was 64 minutes. The algorithm was run on a powerful Apple Mac Pro Dual Xeon 2.66GHz Dual Core desktop with 6 GB Memory.

There are mainly two reasons causing GA’s poor performance problem. There is a big quantity variance among the items, the smallest is 1 and the largest is 284. This kind of distribution prevents a good pattern from being reused multiple times and a lot more patterns have to be generated. The second reason is that the large number of different stocks greatly expands the number of possible patterns to evaluate. This is the intrinsic performance issue when applying GA to complex production problems.

Table 10.3

Result From Using the Gilmore and Gomory LP Algorithm

<table>
<thead>
<tr>
<th>Index</th>
<th>Pattern</th>
<th>Pattern Length</th>
<th>Stock Length</th>
<th>Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>545,780</td>
<td>1325</td>
<td>1328</td>
<td>76</td>
</tr>
<tr>
<td>2</td>
<td>408,584</td>
<td>992</td>
<td>992</td>
<td>89</td>
</tr>
<tr>
<td>3</td>
<td>473,597</td>
<td>1070</td>
<td>1072</td>
<td>35</td>
</tr>
<tr>
<td>4</td>
<td>606,780</td>
<td>1386</td>
<td>1392</td>
<td>19</td>
</tr>
<tr>
<td>5</td>
<td>408,473,606</td>
<td>1487</td>
<td>1488</td>
<td>43</td>
</tr>
<tr>
<td>6</td>
<td>408,473,604</td>
<td>1485</td>
<td>1488</td>
<td>20</td>
</tr>
<tr>
<td>7</td>
<td>404,780</td>
<td>1184</td>
<td>1184</td>
<td>58</td>
</tr>
<tr>
<td>8</td>
<td>780,780</td>
<td>1560</td>
<td>1568</td>
<td>50.5</td>
</tr>
<tr>
<td>9</td>
<td>690,780</td>
<td>1470</td>
<td>1472</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>585,597</td>
<td>1182</td>
<td>1184</td>
<td>7</td>
</tr>
<tr>
<td>11</td>
<td>408,473,527</td>
<td>1408</td>
<td>1408</td>
<td>7</td>
</tr>
<tr>
<td>12</td>
<td>576,780</td>
<td>1356</td>
<td>1360</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>584,584</td>
<td>1168</td>
<td>1168</td>
<td>68.5</td>
</tr>
<tr>
<td>14</td>
<td>636,780</td>
<td>1416</td>
<td>1424</td>
<td>20</td>
</tr>
<tr>
<td>Efficiency</td>
<td>0.998247</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Apparently, using GA to solve the sub CSPs within the stock reduction algorithm is not feasible. With a large number of possible stock combinations and the sub CSPs created from
them, it will take days to solve a complex SRP. Let’s turn to LP. Table 10.3 shows the LP solution for the above problem.

The Gilmore and Gomory’s LP algorithm achieved an efficiency of 0.998247 within 0.421 seconds. However, pattern 8 and 13 have fractional sets of 50.5 and 68.5. As we mentioned above, the cutting stock is an integer problem and a half set cannot be produced. We can either round them down to 50 and 68 sets with shortages of one 780 and one 584, or round them both up to 51 and 69 sets with extras of one 780 and one 584. Neither solution meets the demand exactly. To satisfy the demand, we can round both sets down and add another new pattern that creates one 780 and one 584 to the solution. The closest stock length for this is 1376 with a waste of 12. With the new pattern, the solution’s efficiency is 0.998234 (Table 10.4). Instead of adding a new stock to the solution, the stock with a length of 1392 from pattern 4 can also be used. The efficiency is 0.998209 using 1392 (Table 10.5). Both rounding methods introduce very little loss on the trim efficiency.

Table 10.4
Convert LP Solutions to Integer Using Stock 1376

<table>
<thead>
<tr>
<th>Index</th>
<th>Pattern</th>
<th>Pattern Length</th>
<th>Stock Length</th>
<th>Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>545,780</td>
<td>1325</td>
<td>1328</td>
<td>76</td>
</tr>
<tr>
<td>2</td>
<td>408,584</td>
<td>992</td>
<td>992</td>
<td>89</td>
</tr>
<tr>
<td>3</td>
<td>473,597</td>
<td>1070</td>
<td>1072</td>
<td>35</td>
</tr>
<tr>
<td>4</td>
<td>606,780</td>
<td>1386</td>
<td>1392</td>
<td>19</td>
</tr>
<tr>
<td>5</td>
<td>408,473,606</td>
<td>1487</td>
<td>1488</td>
<td>43</td>
</tr>
<tr>
<td>6</td>
<td>408,473,604</td>
<td>1485</td>
<td>1488</td>
<td>20</td>
</tr>
<tr>
<td>7</td>
<td>404,780</td>
<td>1184</td>
<td>1184</td>
<td>58</td>
</tr>
<tr>
<td>8</td>
<td>780,780</td>
<td>1560</td>
<td>1568</td>
<td>50</td>
</tr>
<tr>
<td>9</td>
<td>690,780</td>
<td>1470</td>
<td>1472</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>585,597</td>
<td>1182</td>
<td>1184</td>
<td>7</td>
</tr>
<tr>
<td>11</td>
<td>408,473,527</td>
<td>1408</td>
<td>1408</td>
<td>7</td>
</tr>
<tr>
<td>12</td>
<td>576,780</td>
<td>1356</td>
<td>1360</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>584,584</td>
<td>1168</td>
<td>1168</td>
<td>68</td>
</tr>
<tr>
<td>14</td>
<td>636,780</td>
<td>1416</td>
<td>1424</td>
<td>20</td>
</tr>
<tr>
<td>15</td>
<td>584,780</td>
<td>1364</td>
<td>1376</td>
<td>1</td>
</tr>
</tbody>
</table>

Efficiency 0.998234
Table 10.5
Convert LP Solutions to Integer Using Stock 1392

<table>
<thead>
<tr>
<th>Index</th>
<th>Pattern Length</th>
<th>Stock Length</th>
<th>Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>545,780</td>
<td>1325</td>
<td>1328</td>
</tr>
<tr>
<td>2</td>
<td>408,584</td>
<td>992</td>
<td>992</td>
</tr>
<tr>
<td>3</td>
<td>473,597</td>
<td>1070</td>
<td>1072</td>
</tr>
<tr>
<td>4</td>
<td>606,780</td>
<td>1386</td>
<td>1392</td>
</tr>
<tr>
<td>5</td>
<td>408,473,606</td>
<td>1487</td>
<td>1488</td>
</tr>
<tr>
<td>6</td>
<td>408,473,604</td>
<td>1485</td>
<td>1488</td>
</tr>
<tr>
<td>7</td>
<td>404,780</td>
<td>1184</td>
<td>1184</td>
</tr>
<tr>
<td>8</td>
<td>780,780</td>
<td>1560</td>
<td>1568</td>
</tr>
<tr>
<td>9</td>
<td>690,780</td>
<td>1470</td>
<td>1472</td>
</tr>
<tr>
<td>10</td>
<td>585,597</td>
<td>1182</td>
<td>1184</td>
</tr>
<tr>
<td>11</td>
<td>408,473,527</td>
<td>1408</td>
<td>1408</td>
</tr>
<tr>
<td>12</td>
<td>576,780</td>
<td>1356</td>
<td>1360</td>
</tr>
<tr>
<td>13</td>
<td>584,584</td>
<td>1168</td>
<td>1168</td>
</tr>
<tr>
<td>14</td>
<td>636,780</td>
<td>1416</td>
<td>1424</td>
</tr>
<tr>
<td>15</td>
<td>584,780</td>
<td>1364</td>
<td>1392</td>
</tr>
</tbody>
</table>

| Efficiency | 0.998209 |

In the above process, we first use LP algorithm to solve the CSP. Then, we round the fractional LP result to an integer solution and still maintain excellent trim efficiency. The rounded integer solution may not be the best solution, but it meets our trim efficiency requirement of 0.99 as well.

The efficiency loss from the above process varies by problems and tends to be very small when there are a lot of sets. The Gilmore and Gomory’s LP algorithm uses a fix-sized matrix and the number of total patterns is limited by the number of different items in the problem. The maximum pattern count is 14 in the above example. If all patterns require fractional sets, we need 7 new patterns of 1 set each to meet the demand using the above approach. Since the available stocks space at 16, the most waste from each set is 16. The waste from these 7 new sets is 16x7=112. We also add in one half-length of the smallest stock if a new pattern only contains one item. The final total waste is 112+816/2=520. Dividing the total waste by the current total stock length of 634656, we have 0.0008. That is, our simple rounding routine only cost us about 0.0008 on efficiency loss in the worst case. This is acceptable in most cases in production since
there are many other factors that can cause more trim loss. For large problems, the overall trim efficiency is dominated by the large integer sets and the impact from the fractional sets is very small.

Our hybrid LP/GA cutting stock algorithm (Algorithm 10.3) is based on the above approach. LP is used first to solve the CSP with the stock mix defined from the previous two algorithms.

Begin
1. Solve the CSP using LP algorithm.
2. If the solution does not meet the threshold, return the solution and stop.
3. If the solution meets the threshold, round the solution into integer.
4. If the integer solution meets the threshold, return the solution and stop.
5. If the integer solution does not meet the threshold, solve the CSP using SGA.
6. Return the result from SGA.
End

Algorithm 10.3

If the LP result does not meet the targeted trim efficiency requirement, the sub problem is declared unsolvable with the current stock mix. A large value is assigned to the fitness function as a penalty. If the LP result meets the trim efficiency requirement, we use the above-mentioned rounding process to get an integer solution. We round down the solution to integer sets and use a local optimizer to find the best patterns to complete the solution. If the converted integer solution meets the efficiency requirement, we declare the problem is solved with success and the current stock mix can satisfy the required trim efficiency. Otherwise, we start SGA to solve the problem and seed it with the integer solution converted from the LP solution. The result from SGA is the final answer for the current problem.

In summary, we use GA as the main algorithm to drive the hybrid LP/GA algorithm. GA creates a series of sub CSPs with different stock mixes, the rule based preprocessor trims down the search space, and finally the hybrid LP/GA algorithm solves the CSPs.
10.4 Experiments

To evaluate our proposed new algorithm, we first conducted a comparison study. We coded our algorithm and a pure GA based algorithm in Microsoft C#. The pure GA based algorithm used preprocesses algorithm and SGA (from section 7.2) to solve the CSPs. Both algorithms were run on an Apple Mac Pro Dual Xeon 2.66GHz Dual Core desktop with 6 GB Memory and Windows XP on VMWare. The test problems were created based on an expanded version of Liang et al. (2002)’s problem 9. Each problem was run 10 times by each algorithm. Results were averaged and reported.

Table 10.6
Comparison Study on Item Variations

<table>
<thead>
<tr>
<th>Problem Name</th>
<th>Total Stock Count</th>
<th>Width Count</th>
<th>Item Count</th>
<th>Time (s)</th>
<th>Efficiency</th>
<th>Waste</th>
<th>Time (s)</th>
<th>Efficiency</th>
<th>Waste</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>10</td>
<td>36</td>
<td>400</td>
<td>1030</td>
<td>0.9954</td>
<td>164</td>
<td>53</td>
<td>0.9954</td>
<td>164</td>
</tr>
<tr>
<td>b12i</td>
<td>10</td>
<td>43</td>
<td>480</td>
<td>950</td>
<td>0.9951</td>
<td>199</td>
<td>50</td>
<td>0.9959</td>
<td>164</td>
</tr>
<tr>
<td>b14i</td>
<td>10</td>
<td>50</td>
<td>560</td>
<td>764</td>
<td>0.9946</td>
<td>250</td>
<td>2</td>
<td>0.9967</td>
<td>150</td>
</tr>
<tr>
<td>b16i</td>
<td>10</td>
<td>57</td>
<td>640</td>
<td>1419</td>
<td>0.9940</td>
<td>320</td>
<td>22</td>
<td>0.9977</td>
<td>120</td>
</tr>
<tr>
<td>b18i</td>
<td>10</td>
<td>64</td>
<td>717</td>
<td>1567</td>
<td>0.9926</td>
<td>461</td>
<td>85</td>
<td>0.9947</td>
<td>331</td>
</tr>
<tr>
<td>b20i</td>
<td>10</td>
<td>72</td>
<td>800</td>
<td>3073</td>
<td>0.9915</td>
<td>618</td>
<td>115</td>
<td>0.9946</td>
<td>391</td>
</tr>
</tbody>
</table>

Table 10.7
Comparison Study on Stock Count Variations

<table>
<thead>
<tr>
<th>Problem Name</th>
<th>Total Stock Count</th>
<th>Width Count</th>
<th>Item Count</th>
<th>Time (s)</th>
<th>Efficiency</th>
<th>Waste</th>
<th>Time (s)</th>
<th>Efficiency</th>
<th>Waste</th>
</tr>
</thead>
<tbody>
<tr>
<td>base</td>
<td>10</td>
<td>36</td>
<td>400</td>
<td>1030</td>
<td>0.9954</td>
<td>164</td>
<td>53</td>
<td>0.9954</td>
<td>164</td>
</tr>
<tr>
<td>b12s</td>
<td>12</td>
<td>36</td>
<td>400</td>
<td>1681</td>
<td>0.9954</td>
<td>164</td>
<td>53</td>
<td>0.9955</td>
<td>162</td>
</tr>
<tr>
<td>b14s</td>
<td>14</td>
<td>36</td>
<td>400</td>
<td>2562</td>
<td>0.9954</td>
<td>164</td>
<td>104</td>
<td>0.9951</td>
<td>176</td>
</tr>
<tr>
<td>b16s</td>
<td>16</td>
<td>36</td>
<td>400</td>
<td>3935</td>
<td>0.9951</td>
<td>176</td>
<td>136</td>
<td>0.9951</td>
<td>176</td>
</tr>
<tr>
<td>b18s</td>
<td>18</td>
<td>36</td>
<td>400</td>
<td>4366</td>
<td>0.9954</td>
<td>164</td>
<td>98</td>
<td>0.9947</td>
<td>189.5</td>
</tr>
<tr>
<td>b20s</td>
<td>20</td>
<td>36</td>
<td>400</td>
<td>4516</td>
<td>0.9951</td>
<td>176</td>
<td>91</td>
<td>0.9958</td>
<td>149</td>
</tr>
</tbody>
</table>

Table 10.6 shows the performance comparison between the two algorithms when the problem item count was changed. From Liang et al.’s base problem, we created subsequent problems by increasing the width count and the item count by a factor of 20% to upsize the
problem. Table 10.7 shows the performance comparison when we add more stock lengths to the problem. Both comparisons concluded that our hybrid algorithm was much faster than the pure GA approach while still maintains good trim efficiency. They also showed that our hybrid algorithm was more effective and efficient for bigger and more complex problems.

We further tested our new algorithm on 12 real production problems (Table 10.8). It took our algorithm from a few minutes up to 45 minutes to solve a problem with good trim efficiency. The pure GA approach would have taken a very long time to solve these problems and it may not be acceptable in industry.

<table>
<thead>
<tr>
<th>Name</th>
<th>Total Stock Count</th>
<th>Total Width Count</th>
<th>Total Item Count</th>
<th>Time (s)</th>
<th>Efficiency</th>
<th>Waste</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>49</td>
<td>14</td>
<td>1076</td>
<td>792.8</td>
<td>0.9916</td>
<td>5353.4</td>
</tr>
<tr>
<td>s2</td>
<td>49</td>
<td>27</td>
<td>4502</td>
<td>367.1</td>
<td>0.9942</td>
<td>11058.4</td>
</tr>
<tr>
<td>s3</td>
<td>49</td>
<td>38</td>
<td>24184</td>
<td>198.1</td>
<td>0.9994</td>
<td>6226.6</td>
</tr>
<tr>
<td>s4</td>
<td>49</td>
<td>119</td>
<td>29438</td>
<td>975.3</td>
<td>0.9999</td>
<td>953.6</td>
</tr>
<tr>
<td>s5</td>
<td>49</td>
<td>44</td>
<td>17441</td>
<td>180.2</td>
<td>0.9971</td>
<td>20737.6</td>
</tr>
<tr>
<td>s6</td>
<td>49</td>
<td>59</td>
<td>8948</td>
<td>254</td>
<td>0.9982</td>
<td>7057</td>
</tr>
<tr>
<td>s7</td>
<td>49</td>
<td>94</td>
<td>41598</td>
<td>566.8</td>
<td>0.9993</td>
<td>13163.8</td>
</tr>
<tr>
<td>s8</td>
<td>49</td>
<td>49</td>
<td>32958</td>
<td>171.8</td>
<td>0.9949</td>
<td>72739</td>
</tr>
<tr>
<td>s9</td>
<td>49</td>
<td>71</td>
<td>19307</td>
<td>561.1</td>
<td>0.9998</td>
<td>1593.4</td>
</tr>
<tr>
<td>s10</td>
<td>49</td>
<td>142</td>
<td>49869</td>
<td>2656.3</td>
<td>0.9998</td>
<td>5311.6</td>
</tr>
<tr>
<td>s11</td>
<td>49</td>
<td>46</td>
<td>17638</td>
<td>528.2</td>
<td>0.9979</td>
<td>17631.8</td>
</tr>
<tr>
<td>s12</td>
<td>49</td>
<td>51</td>
<td>21083</td>
<td>1532.2</td>
<td>0.9966</td>
<td>33655</td>
</tr>
</tbody>
</table>

**10.5 Summary**

In this study, we created a hybrid algorithm to solve very complex nested optimization problem. We used SGA and improved the fitness function calculation performance.

To solve the SRP, we use GA to solve the stock mix selection and the minimizing problem. We design a rule-based preprocessor to trim the search space, and then apply the hybrid
SGA/LP to solve the CSPs. Our experiments have shown that the new hybrid algorithm is efficient and practicable for solving real complex industrial problems effectively.

Traditional hybrid methods use GA and LP to solve different sub problems separately. Our new hybrid algorithm uses both GA and LP to solve the same problem. We Guided GA with shadow price information. SGA provides good optimization results, and LP ensures fast convergence. Our hybrid algorithm can solve the complex SRPs effectively.
CHAPTER 11 CONCLUSION AND FUTURE WORK

11.1 Conclusion

In this dissertation, a shadow price Guided two-measurement enabled genetic algorithm is proposed. It targets the GA’s performance challenge. The new algorithm’s improvements in both solution quality and search speed were proven in the experiments.

The proposed shadow price concept complements the fitness evaluation in the GA’s search process. There are two entities in the GA search process, solution (chromosome) and components (genes). Fitness values are used to compare and filter solutions. Shadow prices are used to compare and select components in the search process. Together, they constitute the proposed two-measurement GA.

The key of our approach is to use shadow price to compare components to further improve GA. We define the shadow price as the relative potential improvement to the solution’s fitness value with a change of a component. The fitness value represents the current solution’s position in the search space. The shadow prices represent potential improvements and directions to evolve.

In the proposed shadow price guided GA, many better solutions are generated under the guidance of shadow price. This reduces the amount of unnecessary calculation and speed up the search process. It also enabled SGA to produce better result.

In the traveling salesman problem experiment, shadow price defines potential improvement from a component’s change. In the cutting stock problem experiment, shadow price is the cost of the material and directly used to generate better patterns. Procedure embedded shadow price in green computing clearly defines the search direction. Stock reduction problem experiment blends new SGA with LP to improve the fitness evaluation performance.
Theory analysis and all experiments proved the effectiveness of our proposed concept of the shadow price guided two-measurement enabled genetic algorithm.

11.2 Future Work

Our proposed shadow price guided GA has speed up the search process and improved the search result. Due to the fact that GA is a population based search technique, there are a lot of calculations in the search algorithm. It needs continuous improvement.

In the CSP experiments, we used shadow price to directly generate next better solutions. We find this is much superior than simply give the directions to search. We shall investment more effort to further research using shadow price to generate better solutions directly.

The other area that we like to further study is the nested optimization problems where the objective function is an optimization problem itself. This kind of objective function put extra stress on the search engine’s calculation workload. Our research is the continuation of the hybrid approach used in the stock reduction problem. We shall find more methods to further improve the convergence speed.
REFERENCES


