3-9-2012

Shadow Price Guided Genetic Algorithms

Gang Shen
Georgia State University

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

ACKNOWLEDGMENTS

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

LIST OF TABLES

LIST OF FIGURES

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

Stock Reduction Problem	SRP
System on a Programmable Chip	SOPC
Traveling Salesman Problem	TSP
Uniform Grouping Crossover	UGCX

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.

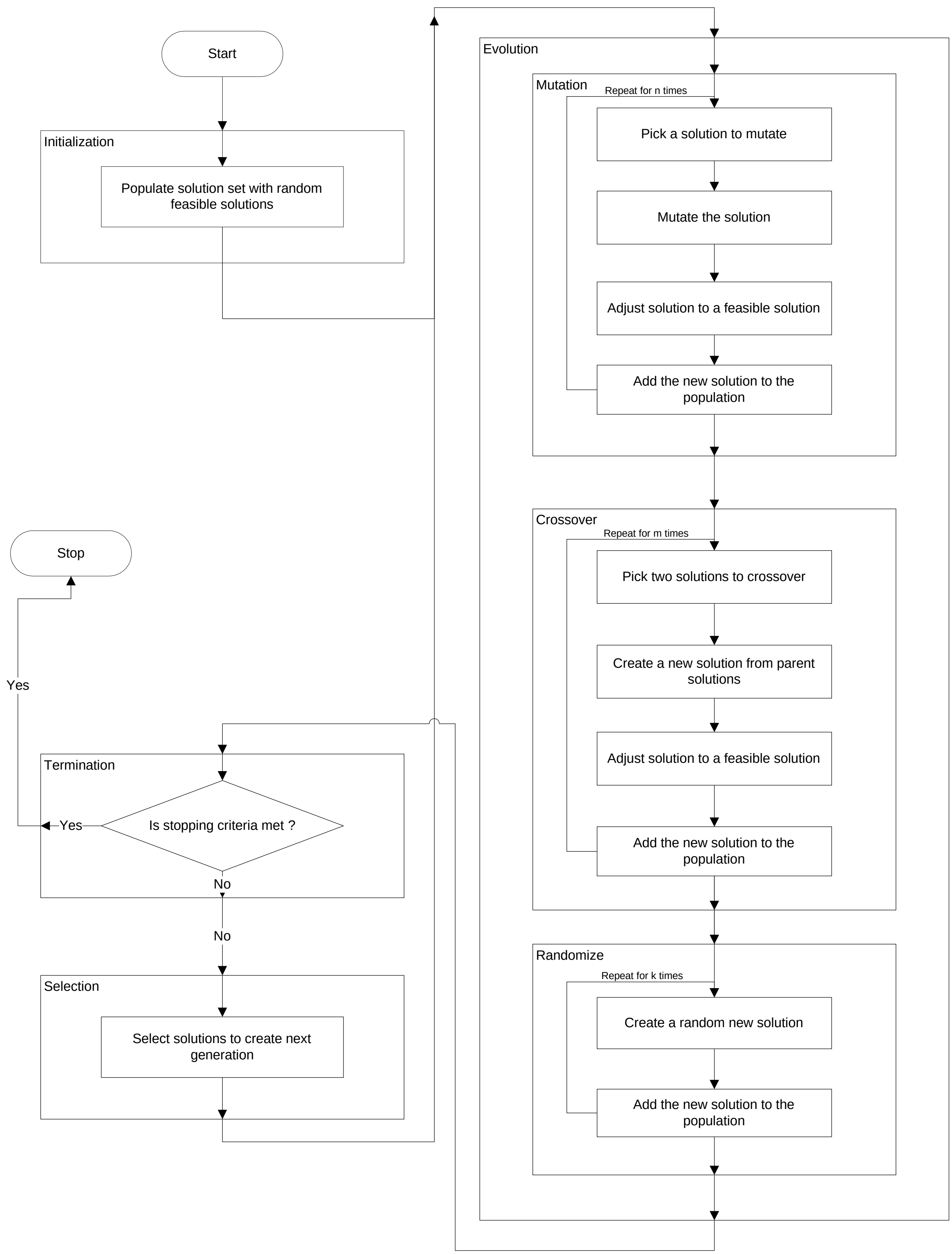

Figure 3.1 Genetic Algorithm

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 Ronnqvist (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 difference 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 neighing 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 scatted 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 Xibilia (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 $x_1, x_2, \cdots, x_n$ which will maximize

$$z = c_1x_1 + c_2x_2 + \cdots + c_nx_n \qquad (5.1)$$

Subject to the constraints

$$\begin{aligned} &a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \le b_1 \\ &a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \le b_2 \\ &\vdots \\ &a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{m1n}x_n \le b_m \\ &x_j \ge 0,\ j = 1,2,\cdots n \end{aligned} \qquad (5.2)$$

More conveniently, we can use a matrix notation. Let

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix},\ b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix},\ x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix},\ c = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} \qquad (5.3)$$

A LP standard form can be rewritten as

$$\text{Maximize} \quad z = c^T x \qquad (5.4)$$

$$\text{Subject to} \quad Ax \le b$$

$$x \ge 0$$

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 $y_1, y_2, \cdots, y_m$ which will minimize

$$z' = b_1 y_1 + b_2 y_2 + \cdots + b_m y_m \qquad (5.5)$$

Subject to the constraints

$$\begin{aligned} &a_{11} y_1 + a_{21} y_2 + \cdots + a_{m1} y_m \geq c_1 \\ &a_{12} y_1 + a_{22} y_2 + \cdots + a_{m2} y_m \geq c_2 \\ &\vdots \\ &a_{1n} y_1 + a_{2n} y_2 + \cdots + a_{mn} y_m \geq c_n \\ &y_j \geq 0,\ j = 1,2,\cdots m \end{aligned} \qquad (5.6)$$

The matrix representation is

$$\text{Minimize} \quad z' = b^T y \qquad (5.7)$$

$$\text{Subject to} \quad A^T y \geq c$$

$$y \geq 0$$

$$\text{where } y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}$$

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

$$z = c^T x_0 = z' = b^T y_0 \qquad (5.8)$$

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

$$\sum_{i=1}^{m} a_{ij} y_i \geq c_j \qquad (5.9)$$

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.

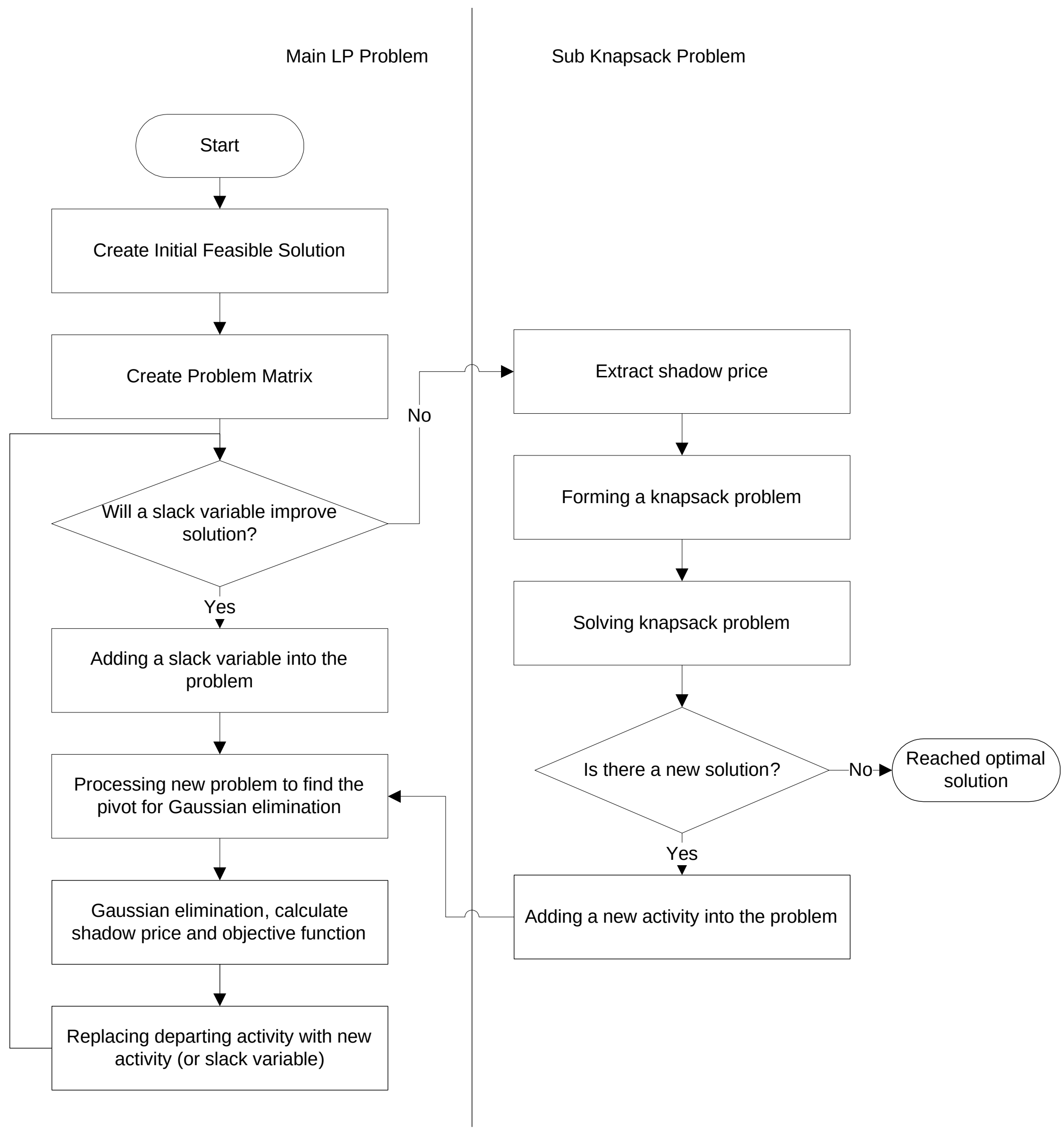

Figure 5.1 Gilmore and Gomory LP Algorithm

CHAPTER 6 SHADOW PRICE GUIDED GENETIC ALGODITHM

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 *j*th solution is defined as $S_j = (a_{1j}, a_{2j}, \ldots, a_{mj})$ where a_{ij} represents *i*th component in *j*th solution. Then, the current solution space is $R = (S_1^T, S_2^T, \ldots, S_n^T)$. Furthermore, we can define a correspondent LP problem as:

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix},\ b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix},\ x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix},\ c = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} \quad (6.1)$$

Optimize $z = c^T x$ (6.2)

Subject to $Ax(\le)(=)(\ge)b$

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 $x + 15 \times z \le 45$ (6.4)

$y + 10 \times z \le 45$ (6.5)

$x^2 + y^2 + z^2 \le 3000$ (6.6)

$x \ge 0; y \ge 0; z \ge 0;$ (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

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

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 $x \ne y \ne z$.

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 $x \le 45, y \le 45, z \le 3$.

There for, we define shadow prices *S* as

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

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

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

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.

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

$p_2 = (15,10,2); f(x, y, z) = 204040; S(x, y, z) = (2560,1080,10);$

$p_3 = (10,15,1); f(x, y, z) = 196200;; S(x, y, z) = (2760,930,20);$

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

	Proposed GA			Classic GA		
Testing	Generations	x,y,z	Fitness	Generations	x,y,z	Fitness
1	171	45.00,31.22,0.00	4896905	181	44.94,31.30,0.00	4889183
2	173	45.00,31.20,0.00	4895037	206	44.51,31.90,0.00	4838689
3	201	45.00,31.20,0.00	4895037	128	45.00,31.20,0.00	4895037
4	218	45.00,31.22,0.00	4896905	218	44.84,31.45,0.00	4878062
5	108	45.00,31.22,0.00	4896905	145	44.77,31.54,0.00	4868991
6	112	44.98,31.25,0.00	4894634	305	45.00,31.20,0.00	4895037
7	173	45.00,31.22,0.00	4896905	210	45.00,31.20,0.00	4895037
8	228	45.00,31.22,0.00	4896905	157	44.67,31.68,0.00	4857306
9	115	45.00,31.22,0.00	4896905	161	45.00,31.22,0.00	4896905
10	270	45.00,31.22,0.00	4896905	384	45.00,31.22,0.00	4896905
Average	176.9		4896304	209.5		4881115

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, medium, 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

	N	Mean	Standard Deviation	Medium	Inter-quartile Range	Min	Max
Proposed GA Algorithm	100	165.3	55.1	163.5	87	104	352
Classic GA Algorithm	100	210.7	79.4	198	118	107	464

Table 6.3
Distribution of the Fitness Values

	N	Mean	Standard Deviation	Medium	Inter-quartile Range	Min	Max
Proposed GA Algorithm	100	4895755	990	4895971	2271	4894634	4896905
Classic GA Algorithm	100	4881970	22295	4893435	22164	4780972	4896905

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

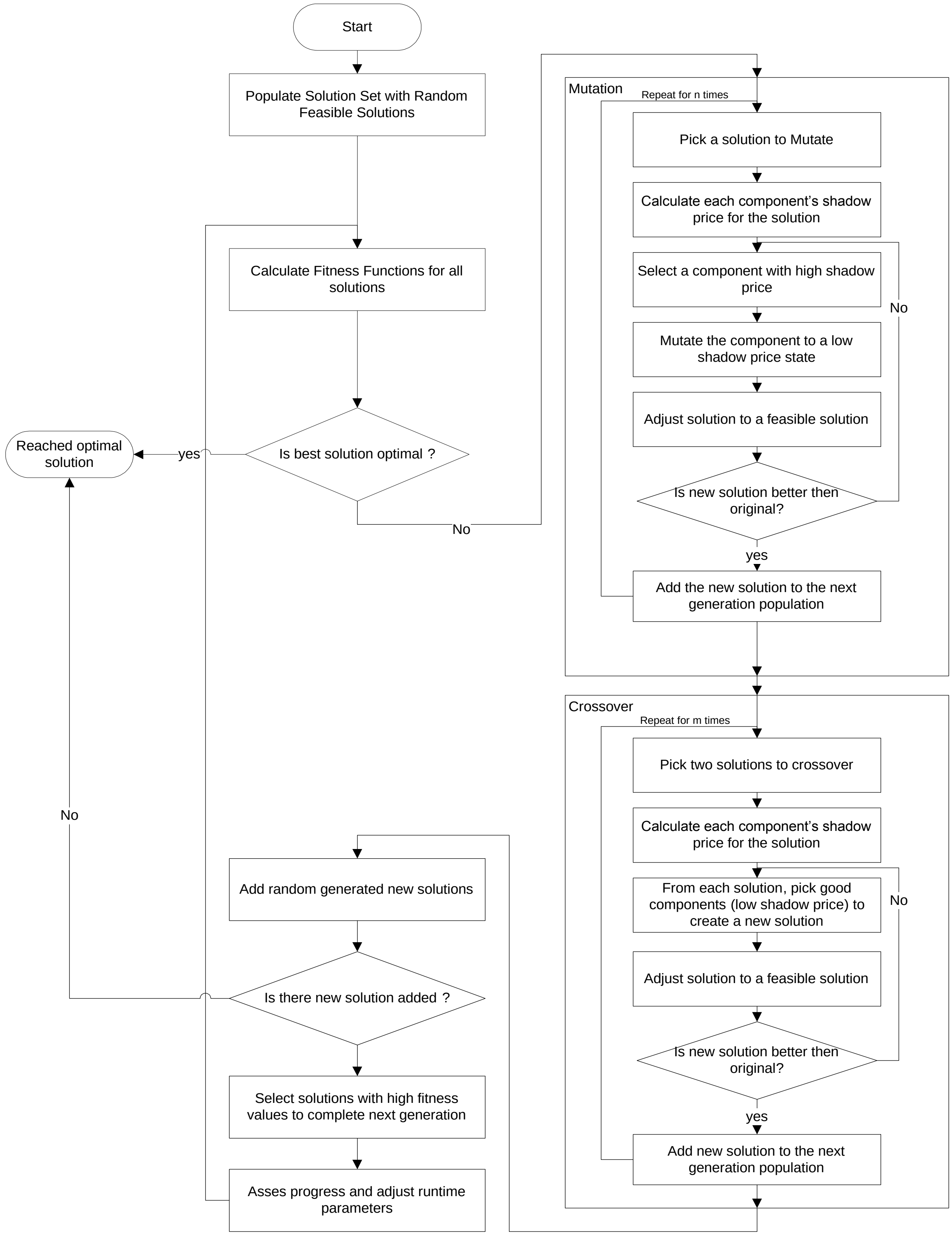

Figure 6.1 New GA Framework with Shadow Price Guided Operators

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_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

$$S_j = \sum_{q=1}^{n}(D_{ij} - D_{qj}) + \sum_{r=1}^{n}(D_{jk} - D_{rj}) \qquad (7.1)$$

where $D_{ij} > D_{qj}$ 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).

Table 7.1
Distance Matrix for gr17.tsp

City	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	0	633	257	91	412	150	80	134	259	505	353	324	70	211	268	246	121
2	633	0	390	661	227	488	572	530	555	289	282	638	567	466	420	745	518
3	257	390	0	228	169	112	196	154	372	262	110	437	191	74	53	472	142
4	91	661	228	0	383	120	77	105	175	476	324	240	27	182	239	237	84
5	412	227	169	383	0	267	351	309	338	196	61	421	346	243	199	528	297
6	150	488	112	120	267	0	63	34	264	360	208	329	83	105	123	364	35
7	80	572	196	77	351	63	0	29	232	444	292	297	47	150	207	332	29
8	134	530	154	105	309	34	29	0	249	402	250	314	68	108	165	349	36
9	259	555	372	175	338	264	232	249	0	495	352	95	189	326	383	202	236
10	505	289	262	476	196	360	444	402	495	0	154	578	439	336	240	685	390
11	353	282	110	324	61	208	292	250	352	154	0	435	287	184	140	542	238
12	324	638	437	240	421	329	297	314	95	578	435	0	254	391	448	157	301
13	70	567	191	27	346	83	47	68	189	439	287	254	0	145	202	289	55
14	211	466	74	182	243	105	150	108	326	336	184	391	145	0	57	426	96
15	268	420	53	239	199	123	207	165	383	240	140	448	202	57	0	483	153
16	246	745	472	237	528	364	332	349	202	685	542	157	289	426	483	0	336
17	121	518	142	84	297	35	29	36	236	390	238	301	55	96	153	336	0

We number the cities from 1 to 17. Let's assume we have a tour as

$C_1 \to C_2 \to C_3 \to C_4 \to C_5 \to C_6 \to C_7 \to C_8 \to \ldots \to C_{17} \to C_1$

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

$$S_4 = \sum_{q=1}^{17}(D_{34} - D_{q4}) + \sum_{r=1}^{17}(D_{45} - D_{r4}) = \sum_{q=1}^{17}(228 - D_{q4}) + \sum_{r=1}^{17}(383 - D_{r4}) = 3813 \quad (7.2)$$

$$S_5 = \sum_{q=1}^{17}(D_{45} - D_{q5}) + \sum_{r=1}^{17}(D_{56} - D_{r5}) = \sum_{q=1}^{17}(383 - D_{q5}) + \sum_{r=1}^{17}(267 - D_{r5}) = 2100 \quad (7.3)$$

$$S_6 = \sum_{q=1}^{17}(D_{56} - D_{q6}) + \sum_{r=1}^{17}(D_{67} - D_{r6}) = \sum_{q=1}^{17}(267 - D_{q6}) + \sum_{r=1}^{17}(63 - D_{r6}) = 1697 \quad (7.4)$$

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 $S_5 + S_6 \le S_4 + S_5$. 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.

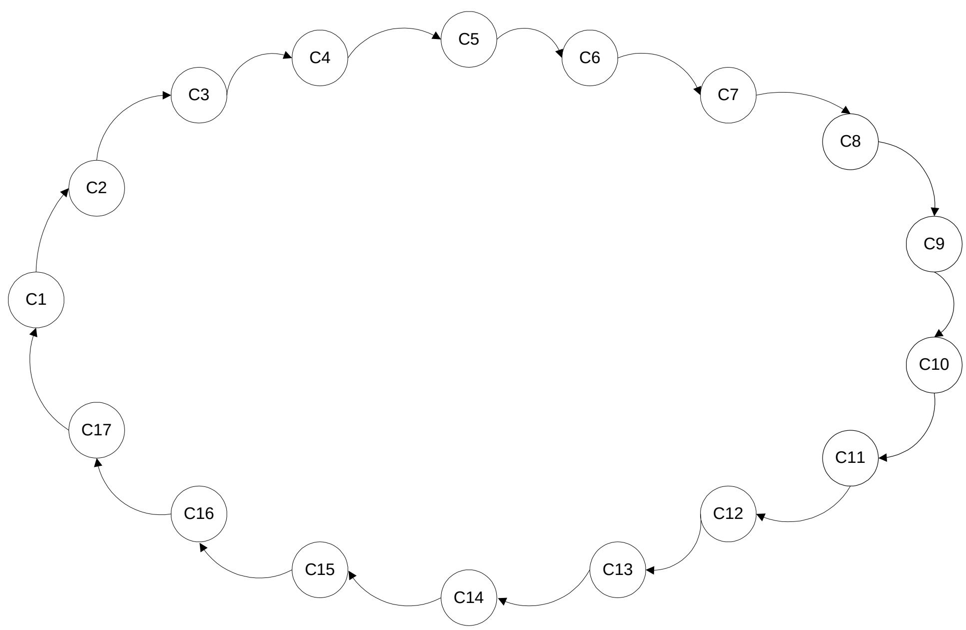

Figure 7.1 A Sample Tour

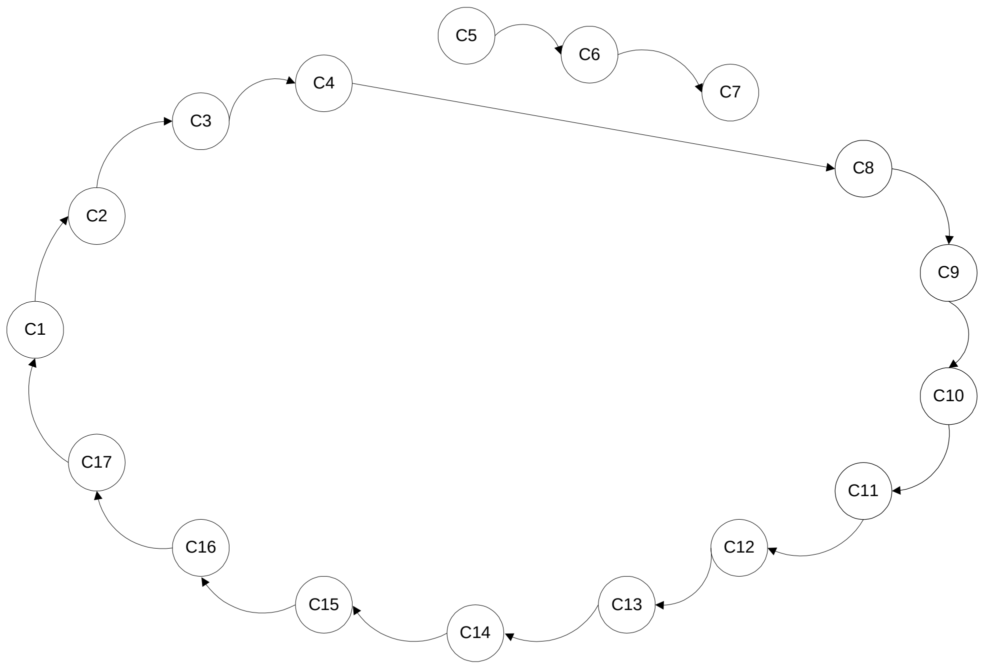

Figure 7.2 Result from Mutation

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, $C_1 \to C_2 \to C_3 \to \ldots \to C_n \to C_1$, we encode it as ($C_1$, C_2, C_3,…, 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 7.2
Comparison with Ray, Bandyopadhyay, and Pal (2004)

	Optimal	Ray, et al. 2004	Our result		
		Best	Best	Average	Avg Time(s)
GR24	1272	1272	1272	1272	0.054
Bayg29	1610	1610	1610	1610	0.097
GR48	5046	5046	5046	5046	0.825
ST70	675	685	675	675	4.834
KroA100	21282	21504	21282	21282	2.987

Table 7.3
Comparison with Zhong, Zhang, and Chen (2007)

	Optimal	**Zhong, et al 2007**			**Our result**		
		Best	Average	Avg Time(s)	Best	Average	Avg Time(s)
Eil51	426	427	433.64	4.06	426	426.1	11.87
Berlin52	7542	7542	7598.76	4.12	7542	7542	0.16
Eil76	538	540	551.72	11.59	538	538	2.70
KroA100	21282	21296	21689.30	23.95	21282	21282	2.99
KroA200	29368	29563	30374.30	198.55	29368	29368	115.87

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

	Optimal	**Wong, et al. 2008**		**Our Result**				
		% from optimal		Distance		% from optimal		Time(s)
		Best	Average	Best	Average	Best	Average	Average
ATT48	10628	0.31	0.83	10628	10628	0	0	0.56
EIL51	426	0.47	0.85	426	426.1	0	0.0002	11.88
EIL76	538	0.19	2.01	538	538	0	0	2.70
EIL101	629	0.95	2.29	629	629	0	0	2.20
KROA100	21282	2.26	3.43	21282	21282	0	0	2.99
KROB100	22141	2.24	3.1	22141	22141	0	0	6.12
KROC100	20749	0.5	1.5	20749	20749	0	0	1.12
KROD100	21294	1.64	3.25	21294	21294	0	0	14.60
KROE100	22068	1.73	2.2	22068	22096.8	0	0.0013	218.14
KROA150	26524	5.03	6.39	26524	26524	0	0	41.08
KROB150	26130	1.55	3.68	26130	26130	0	0	281.92
KROA200	29368	2.02	4.26	29368	29368	0	0	115.87
KROB200	29437	3.1	6.36	29437	29437	0	0	295.23
LIN105	14379	0.32	1.24	14379	14379	0	0	2.48
LIN318	42029	6.32	7.55	42029	42113.6	0	0.0020	1233.42

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

Subject to $\sum_{j \in J} a_{ij} x_j = N_i$ for i=1,2,…n. (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 $j \in J$. (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

Case	Evaluations	Mean fitness	Std. Dev.	Found at
1	1184	1	0	407
2	1184	1	0	740
3	1184	1	0	407
4	2294	0.9995	0.0022	2294
5	2294	0.9998	0.0007	2294

Table 8.2
Test results for the CSP with single stock length

Case	Evaluations	Mean fitness	Std. Dev.	Found at
1a	1184	0.9133	0	296
2a	1184	0.9227	0.0018	1184
3a	1184	1	0	407
4a	1184	0.9642	0	851
5a	2294	0.8479	0.007	2294

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 8.3
Sample problem, the stock length is 14

Item Length	3	4	5	6	7	8	9	10
No. Required	5	2	1	2	4	2	1	3

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.

$$SP_{ij} = l_i \times \frac{L}{\sum l_k} \quad \text{for } k \text{ is the number of items in pattern } j, \quad (8.4)$$

$$S_i = \frac{\sum SP_{ij}}{N_i} \quad \text{for } j \text{ is the number of patterns in the trim.} \quad (8.5)$$

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

$$\text{Maximize } S = \sum_{i=1}^{n} a_i SP_i \,, \qquad (8.6)$$

$$L \geq \sum_{i=1}^{n} a_i l_i \quad . \qquad (8.7)$$

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:

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

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 jth 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.

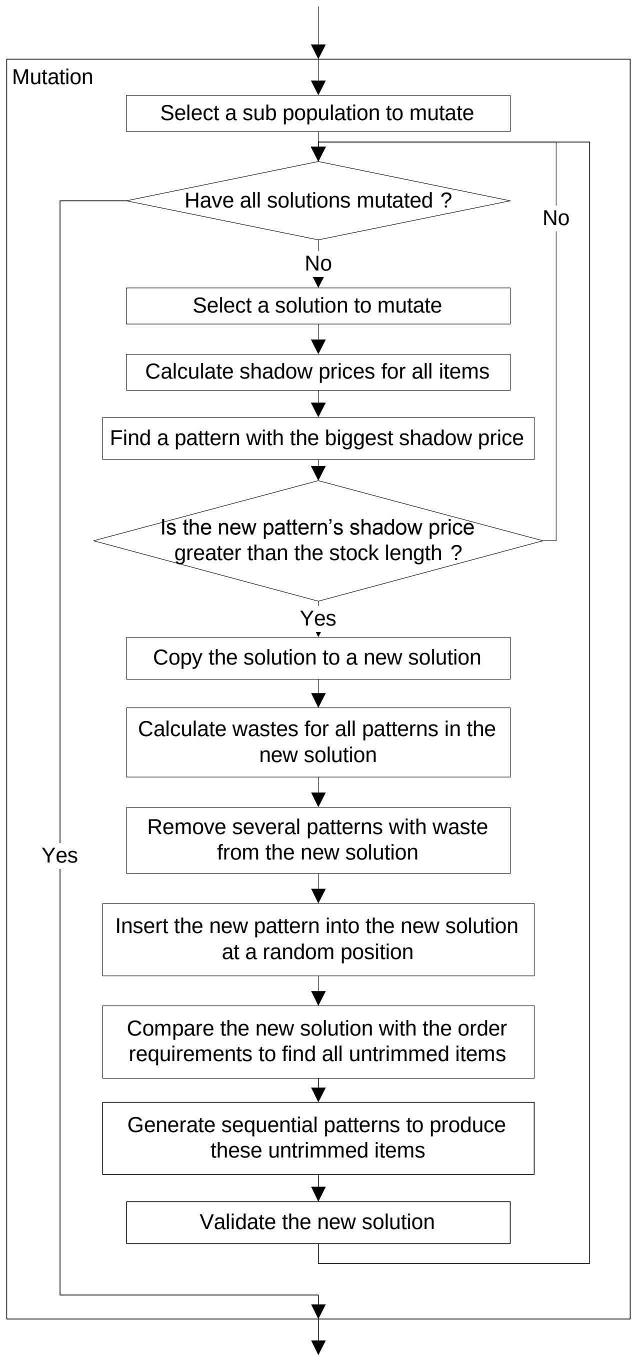

Figure 8.1 Algorithm B's mutation operator

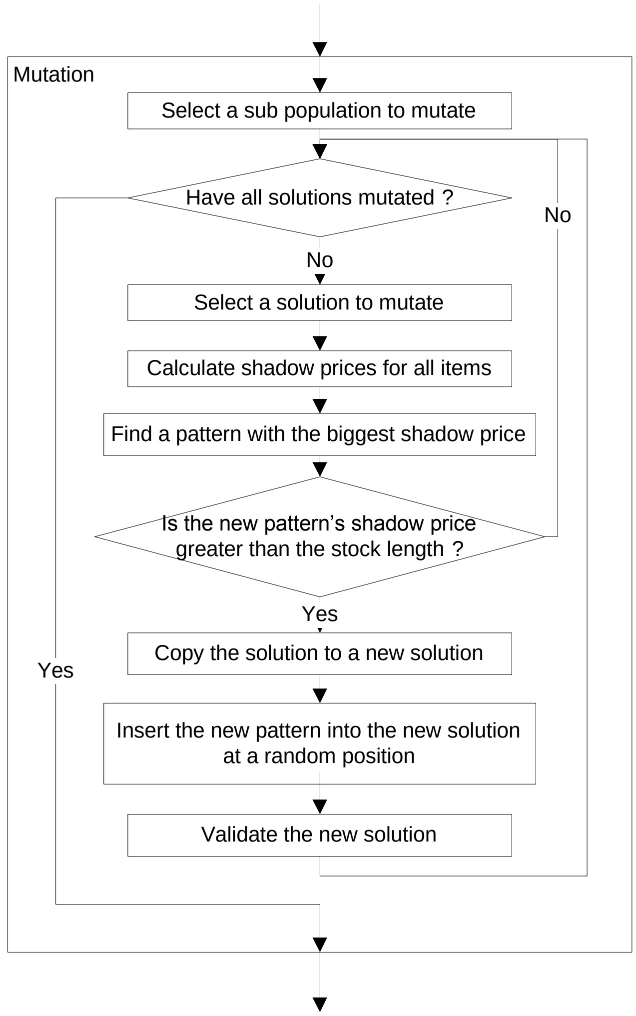

Figure 8.2 Algorithm C's mutation operator

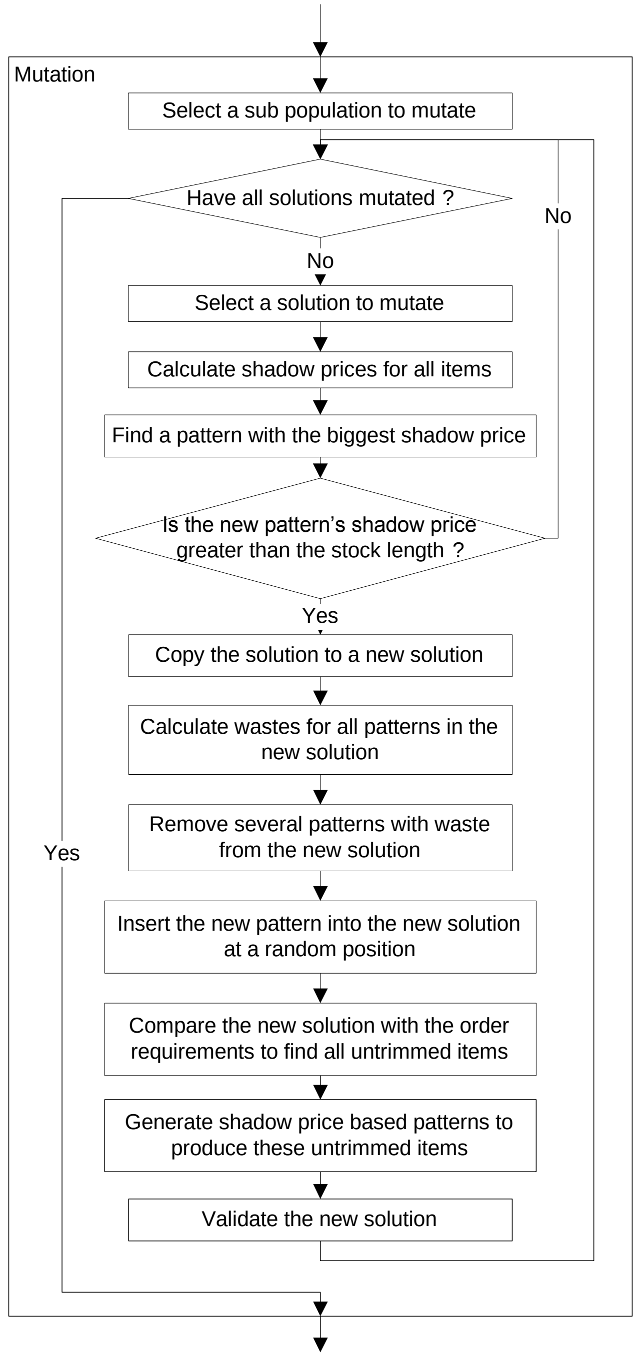

Figure 8.3 Algorithm D's mutation operator

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 8.4
Test case summary

Case	Size Count	Total Items
1a	8	20
2a	8	50
3a	8	60
4a	8	60
5a	18	126
6a	18	200
7a	24	200
8a	24	400
9a	36	400
10a	36	600

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 8.5
Mean Fitness Value Comparison

	Other Algorithms								**Our New Algorithms**				
Case	Lu Pure-ACO	Lu ACO-MCSP	Hinterding Group Based	Hinterding Order Based	Liang EP	Alg. A	**Avg.**	**Max**	Alg. B	Alg. C	Alg. D	**Avg.**	**Max**
1a	0.8056	0.9133	0.9133	0.9133	0.9133	0.9133	**0.8954**	**0.9133**	0.9133	0.9133	0.9133	**0.9133**	**0.9133**
2a	0.8912	0.9231	0.9227	0.9198	0.9231	0.9237	**0.9173**	**0.9237**	0.9237	0.9237	0.9237	**0.9237**	**0.9237**
3a	0.9921	1	1	1	1	1	**0.9987**	**1**	1	1	1	**1**	**1**
4a	0.9113	0.9638	0.9642	0.9588	0.964	0.9642	**0.9544**	**0.9642**	0.9642	0.9642	0.9642	**0.9642**	**0.9642**
5a	0.8312	0.8481	0.8479	0.8489	0.8568	0.8649	**0.8496**	**0.8649**	0.8638	0.8647	0.8657	**0.8647**	**0.8657**
6a					0.889	0.9389	**0.9140**	**0.9389**	0.9472	0.9475	0.9483	**0.9477**	**0.9483**
7a					0.9529	0.9796	**0.9663**	**0.9796**	0.983	0.9821	0.9853	**0.9835**	**0.9853**
8a					0.884	0.9567	**0.9204**	**0.9567**	0.975	0.9749	0.9786	**0.9762**	**0.9786**
9a					0.9003	0.9701	**0.9352**	**0.9701**	0.9836	0.9808	0.9942	**0.9862**	**0.9942**
10a					0.899	0.9735	**0.9363**	**0.9735**	0.9879	0.9878	0.9986	**0.9914**	**0.9986**

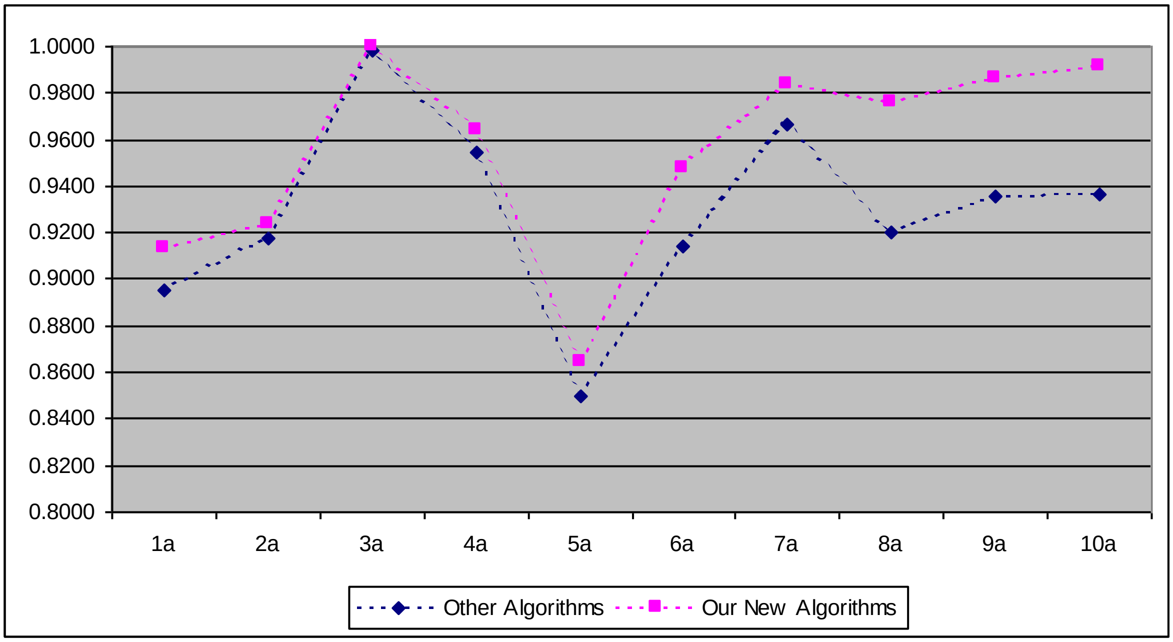

Figure 8.4 Average Mean Fitness Value Comparison

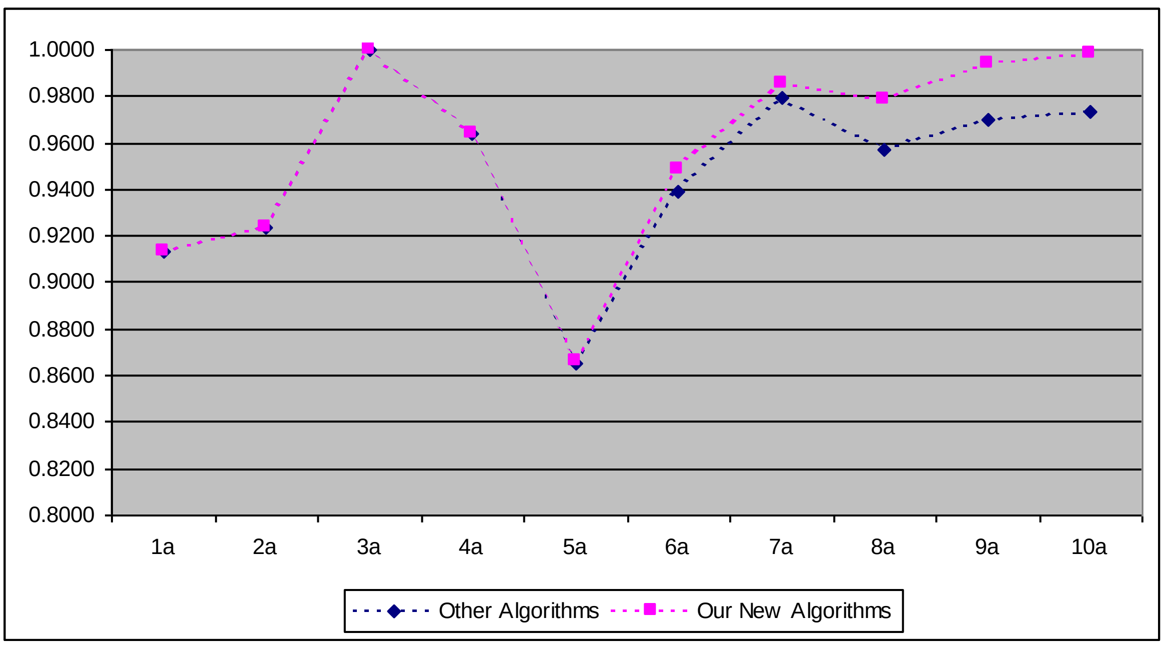

Figure 8.5 Maximum Mean Fitness Value Comparisons

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.*

	Other Algorithms					**Our New Algorithms**				
Case	Chiong EP	Liang EP	Alg. A	**Avg.**	**Min**	Alg. B	Alg. C	Alg. D	**Avg.**	**Min**
1a	3	3	3	**3**	**3**	3	3	3	**3**	**3**
2a	13	13	13	**13**	**13**	13	13	13	**13**	**13**
3a	0	0	0	**0**	**0**	0	0	0	**0**	**0**
4a	11	11	11	**11**	**11**	11	11	11	**11**	**11**
5a	11370*	11966	11450	**11622**	**11450**	11450	11450	11450	**11450**	**11450**
6a	240.6	309.4	120.2	**223.4**	**120.2**	103	103	103	**103**	**103**
7a	84	189.6	84	**119.2**	**84**	84	84	84	**84**	**84**
8a	308	788	200	**432**	**200**	104	92	92	**96**	**92**
9a	250	730	142	**374**	**142**	94	106	22	**74**	**22**
10a	190	1037.2	166	**464.4**	**166**	118	130	10	**86**	**10**

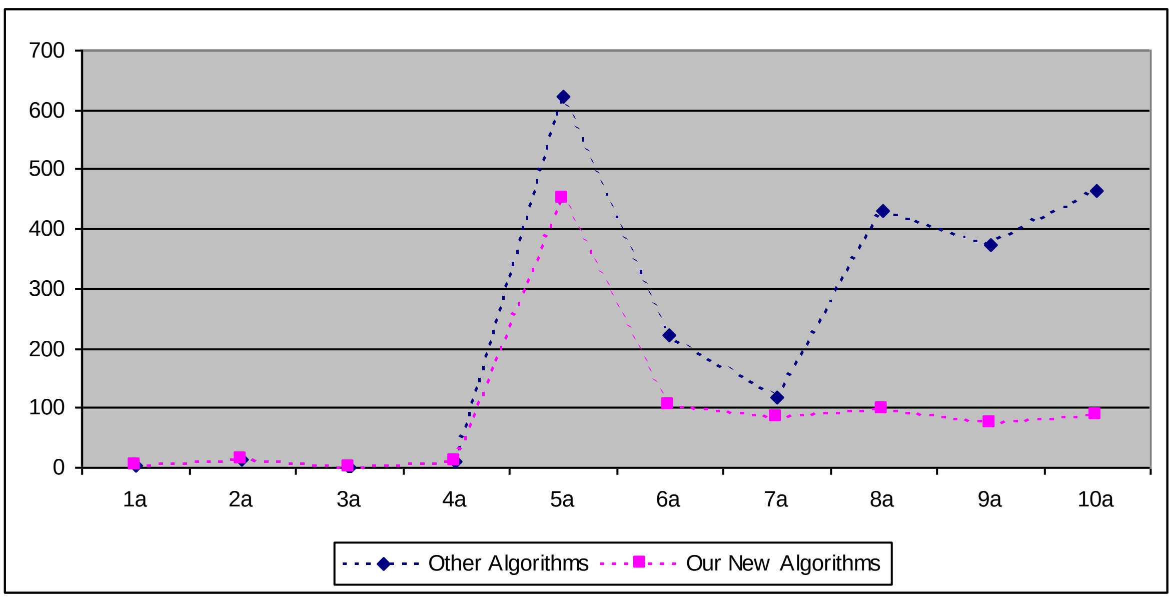

Figure 8.6 Average Total Waste Comparisons

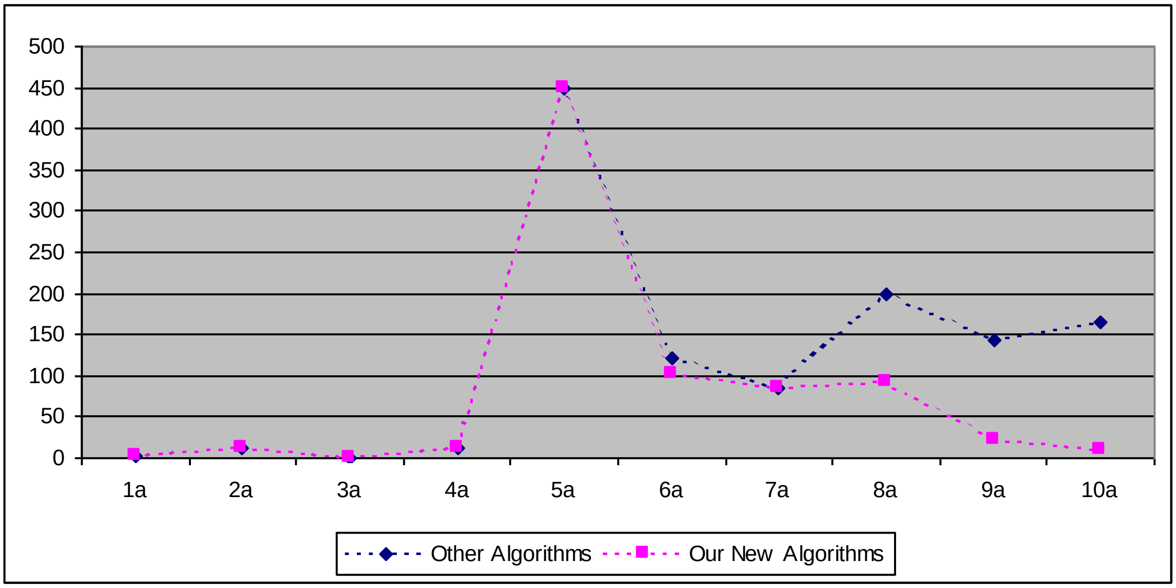

Figure 8.7 Minimum Total Waste Comparisons

Table 8.7
Number of Stocks with Waste Comparison

	Other Algorithms					Our New Algorithms				
Case	Chiong EP	Liang EP	Alg. A	**Avg.**	**Min**	Alg. B	Alg. C	Alg. D	**Avg.**	**Min**
1a	2.8	2	2	**2.3**	**2**	2	2	2	**2**	**2**
2a	4.7	4	4	**4.2**	**4**	4	4	4	**4**	**4**
3a	0	0	0	**0.0**	**0**	0	0	0	**0**	**0**
4a	3.2	1.02	1	**1.7**	**1**	1	1	1	**1**	**1**
5a	27.1	22.8	22.2	**24.0**	**22.2**	22.4	22	22	**22.1**	**22**
6a	26.5	29.96	23.5	**26.7**	**23.5**	21.1	21.1	21	**21.1**	**21**
7a	6.6	7.48	4	**6**	**4**	2.5	2.7	1.8	**2.3**	**1.8**
8a	27.4	56.24	30.3	**38.0**	**27.4**	19.9	21.2	16.6	**19.2**	**16.6**
9a	17.6	48.54	23.7	**29.9**	**17.6**	14.1	15.6	5.3	**11.7**	**5.3**
10a	11.4	73.06	31.7	**38.7**	**11.4**	13	12.2	1	**8.7**	**1**

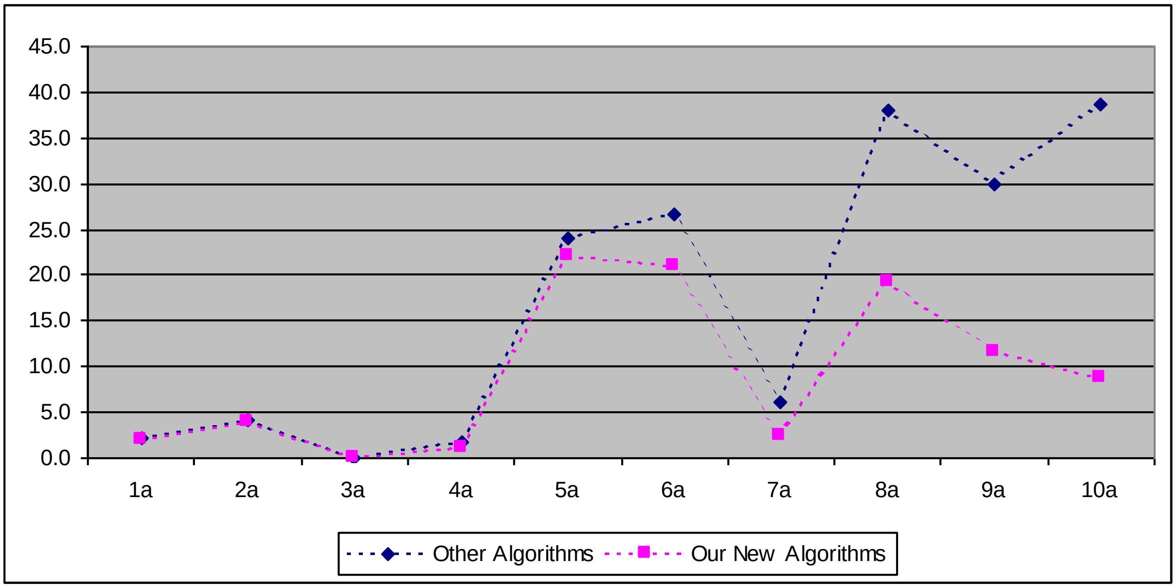

Figure 8.8 Average Number of Stocks with Waste Comparisons

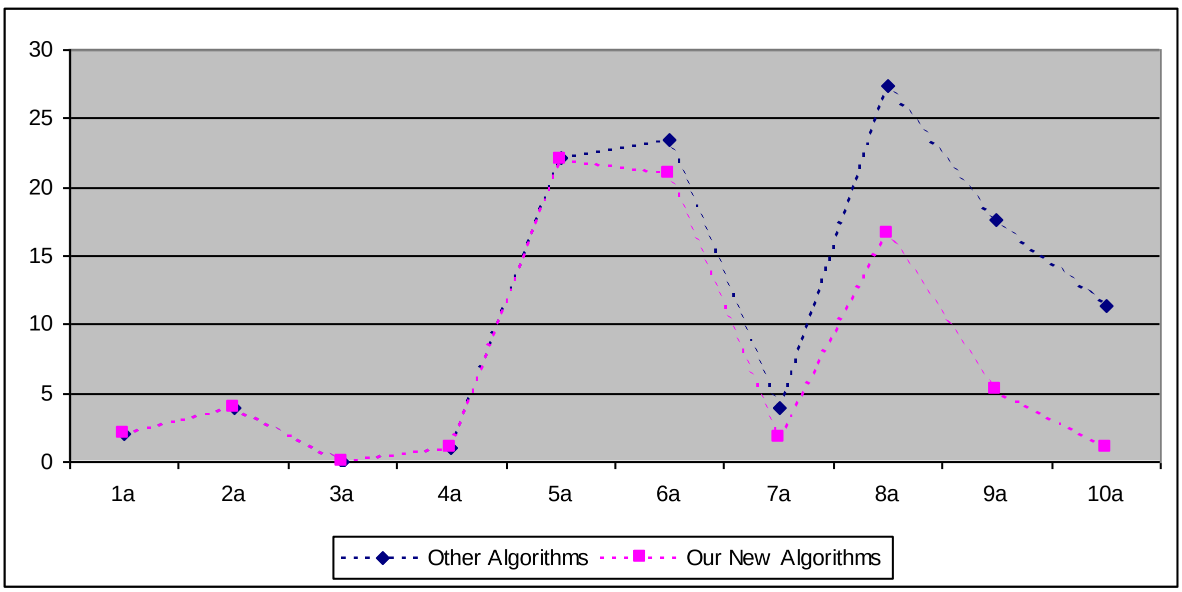

Figure 8.9 Minimum Number of Stocks with Waste Comparisons

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. Figures8.10 and 8.11 present them in chart.

Table 8.8
Speed Comparison

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

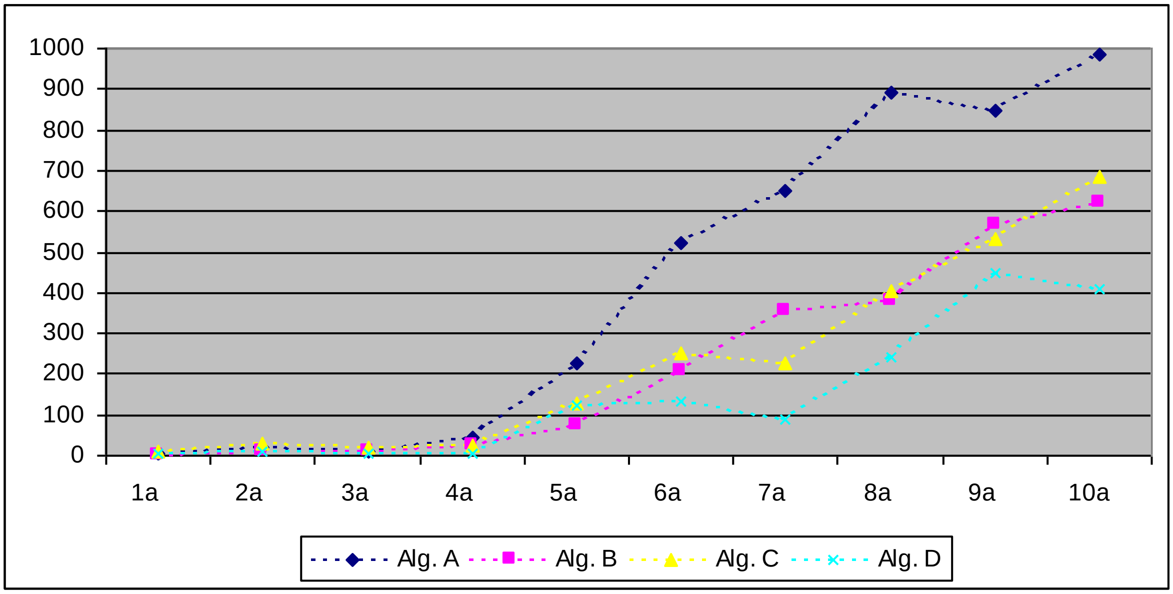

Figure 8.10 Best Solution Found Generation Comparisons

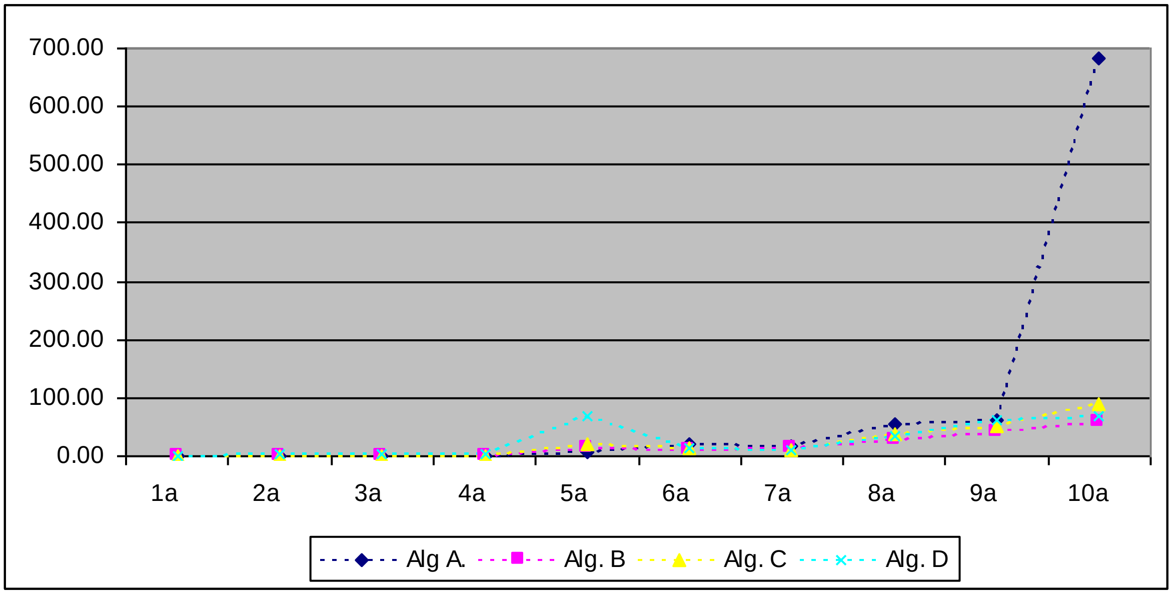

Figure 8.11 Time (s) Comparisons

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.

$$\text{Maximize} \quad f = 1 - \frac{1}{m+1}\left(\sum_{j=1}^{m}\sqrt{\frac{w_j}{L}} + \sum_{j=1}^{m}\frac{v_j}{m} + \left(\frac{p}{m}\right)^2\right). \qquad (8.9)$$

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 8.9
Mean fitness value and number of stocks used

		A	B	C	D	A	B	C	D
Case	Items	Mean Fitness				Stocks Used			
6a	200	0.9328	0.9465	0.9478	0.9483	79.3	79	79	79
7a	200	0.9710	0.9820	0.9819	0.9863	68	68	68	68
8a	400	0.9551	0.9730	0.9743	0.9783	143.9	143.2	143.1	143
9a	400	0.9699	0.9813	0.9784	0.9942	150	149.8	150	149
10a	600	0.9716	0.9895	0.9868	0.9984	216.2	215.8	215.9	215

Table 8.10
Total waste, number of stocks with waste, and distinct pattern count

	A	B	C	D	A	B	C	D	A	B	C	D
Case	Total Waste				Stocks with Waste				Distinct Pattern Count			
6a	128.8	103	103	103	25.3	21.4	21.1	21	18	16	15	15
7a	84	84	84	84	7	2.5	2.9	1.5	24	22	22	24
8a	200	116	104	92	32.1	20.7	21.3	16.7	33	27	28	30
9a	142	118	142	22	24.7	15.4	16.5	5.4	44	39	37	43
10a	154	106	118	10	35.5	10.8	14.6	1.4	56	43	42	50

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 volt-age 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, …, 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, …, n, and k=1, 2, …, 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_k^i on computer i is $\frac{R_k^i}{S_k^i}$. The total execution time for m_i tasks P_k^i on computer i is defined as $T_i = \sum_{k=1}^{m_i} \frac{R_k^i}{S_k^i}$. For example (Table 7.3.1), m_i tasks P_k^i for $m_1 = 4$, $m_2 = 4$, and $m_3 = 3$ on three processors,

Table 9.1
A Sample Task Schedule

Processor 1	P_1^1	P_2^1	P_3^1	P_4^1
Processor 2	P_1^2	P_2^2	P_3^2	P_4^2
Processor 3	P_1^3	P_2^3	P_3^3	

The energy for P_k^i on computer i is $E_k^i = C_i R_k^i [S_k^i]^{\alpha_i - 1}$ (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, …, n, and k=1, 2, …, m_i.

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

The optimization problem for P1 is

$$\text{Minimize } E = \sum_{i=1}^{n} \sum_{k}^{m_i} C_i R_k^i [S_k^i]^{\alpha_i - 1} \qquad (9.3)$$

Constraints: $1 \leq m_i \leq m - n + 1$, $m = \sum_{i=1}^{n} m_i$, $m > n$, $\sum_{k=1}^{m_i} \frac{R_k^i}{S_k^i} \leq T$ and $a_i \leq S_k^i \leq b_i$ where a_i is the minimum speed and b_i is the maximum speed of computer i, respectively, for i=1, 2, …, n, and k=1, 2, …, 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*.

$$\text{Minimize } T = \max_{i \in n}(T_i) \tag{9.4}$$

$$T_i = \sum_{j=1}^{k} \left(\frac{C_i \left(R_j^i\right)^{\alpha_i}}{E_j^i} \right)^{\frac{1}{\alpha_i - 1}} \tag{9.5}$$

$$E \geq \sum_{i=1}^{n} E_i \tag{9.6}$$

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

$$T_i = \sum_{j=1}^{k} \frac{R_j^i}{S_j^i} \tag{9.7}$$

$$E_i = C_i \sum_{j=1}^{k} \left(R_j^i \left(S_j^i\right)^{\alpha_i - 1} \right) \tag{9.8}$$

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:

$$T_i = \left(\frac{C_i\left(\sum_{j=1}^{k} R_j^i\right)^{\alpha_i}}{E_j^i}\right)^{\frac{1}{\alpha_i - 1}} \qquad (9.5')$$

$$T_i = \frac{\sum_{j=1}^{k} R_j^i}{S_i} \qquad (9.7')$$

$$E_i = C_i\left(\sum_{j=1}^{k} R_j^i\right)(S_i)^{\alpha_i - 1} \qquad (9.8')$$

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.

$$S_i = \left(\frac{E_i}{C_i\left(\sum_{j=1}^{k} R_j^i\right)}\right)^{\frac{1}{\alpha_i - 1}} \qquad (9.9)$$

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
End

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 α is in the exponent. Since α is a positive number and greater or equal to 3, α can generate bigger impact to the execute time and energy consumption. While comparing 2 processors with same tasks, the one with bigger α 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 α since it may cause much small increase to the execution time. We define the shadow prices as the combination of execution time and α. We want to mutate the task from a long execution time processor to a processor with short execution time and a smaller α.

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
   - 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
   - Energy exchange mutation operation
     a) Randomly select two processors
     b) Exchange energy assignments between them
     c) Validate the new solution
   - 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
   - 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
   - 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
   - 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 α value
     f) Create a subset from short run time processors to establish a small α 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

3.3 Filter and build next generation
End While
End

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 α. 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

ID	Processor	Inst. / Second (MIPS/MHZ)	Inst. /clock cycle	Year	Min Speed (MIPS)	OverClocking Improvement (%)	Max Speed (MIPS)	C	Φ
1	DEC Alpha 21064 EV4	300 / 150	2.7	1992	300	0.09	327	84	0.65
2	Intel Pentium III	1,354 / 500	2.7	1999	1354	0.15	1557	7	0.75
3	AMD Athlon	3,561 / 1.2	3	2000	3561	0.23	4380	8	0.57
4	AMD Athlon XP 2400+	5,935 / 2.0	3	2002	5935	0.14	6766	86	0.68
5	Pentium 4 Extreme Edition	9,726 / 3.2	3	2003	9726	0.07	10407	74	0.8
6	AMD Athlon FX-57	12,000 / 2.8	4.3	2005	12000	0.09	13080	50	0.83
7	AMD Athlon 64 3800+ X2 (Dual Core)	14,564 / 2.0	7.3	2005	14564	0.13	16457	60	0.73
8	ARM Cortex A8	2,000 / 1.0	2	2005	2000	0.18	2360	52	0.56
9	Xbox360 IBM "Xenon" Triple Core	19,200 / 3.2	6	2005	19200	0.2	23040	55	0.51
10	AMD Athlon FX-60 (Dual Core)	18,938 / 2.6	7.3	2006	18938	0.17	22157	62	0.63
11	Intel Core 2 Extreme X6800	27,079 / 2.93	9.2	2006	27079	0.21	32766	19	0.66
12	Intel Core 2 Extreme QX6700	49,161 / 2.66	18.5	2006	49161	0.16	57027	22	0.6
13	PS3 Cell BE (PPE only)	10,240 / 3.2	3.2	2006	10240	0.23	12595	45	0.94
14	P.A. Semi PA6T-1682M	8,800 / 2.0	4.4	2007	8800	0.25	11000	11	0.75
15	Intel Core 2 Extreme QX9770	59,455 / 3.2	18.6	2008	59455	0.17	69562	92	0.55
16	Intel Core i7 Extreme 965EE	76,383 / 3.2	23.9	2008	76383	0.18	90132	11	0.65
17	AMD Phenom II X4 940 Black Edition	42,820 / 3.0	14.3	2009	42820	0.15	49243	42	0.56
18	AMD Phenom II X6 1090T	68,200 / 3.2	21.3	2010	68200	0.12	76384	77	0.91
19	Intel Core i7 Extreme Edition i980EE	147,600 / 3.3	44.7	2010	147600	0.14	168264	29	0.71
20	IBM 5.2-GHz z196	52,286 / 5.2	10.05	2010	52286	0.15	60129	66	0.69

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 ϕ 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 ϕ for each processor. To improve the quality of random number, we used public available true random number generating services (Random,

Table 9.3
Energy Consumption Comparison

		Gmax=500			Gmax=1000			Gmax=1500			Gmax=2000		
Cp	Ct	Ega	Esga	Ega-Esga	Ega	Esga	Ega-Esga	Ega	Esga	Ega-Esga	Ega	Esga	Ega-Esga
10	500	1.76E+20	1.37E+20	3.90E+19	1.36E+20	1.36E+20	4.90E+15	1.36E+20	1.36E+20	6.50E+15	1.36E+20	1.36E+20	4.51E+15
10	1000	1.71E+21	9.76E+20	7.31E+20	6.39E+20	5.78E+20	6.05E+19	5.78E+20	5.78E+20	8.50E+16	5.78E+20	5.78E+20	1.46E+16
10	1500	5.99E+21	3.89E+21	2.10E+21	2.01E+21	1.15E+21	8.59E+20	1.06E+21	9.19E+20	1.43E+20	9.20E+20	9.17E+20	2.10E+18
10	2000	6.18E+21	4.59E+21	1.59E+21	2.10E+21	1.30E+21	8.00E+20	1.22E+21	7.44E+20	4.77E+20	7.85E+20	6.39E+20	1.46E+20
10	3000	5.22E+21	3.82E+21	1.40E+21	2.31E+21	1.55E+21	7.51E+20	1.39E+21	9.05E+20	4.89E+20	9.34E+20	5.69E+20	3.65E+20
10	5000	1.63E+22	1.23E+22	4.00E+21	7.11E+21	5.05E+21	2.06E+21	4.37E+21	2.91E+21	1.46E+21	3.16E+21	1.95E+21	1.21E+21
20	500	4.62E+19	2.40E+19	2.22E+19	1.11E+19	4.30E+18	6.79E+18	3.85E+18	2.63E+18	1.22E+18	2.66E+18	2.58E+18	8.00E+16
20	1000	1.40E+21	7.66E+20	6.29E+20	2.49E+20	1.07E+20	1.42E+20	8.43E+19	3.11E+19	5.32E+19	3.45E+19	1.40E+19	2.05E+19
20	1500	1.73E+22	1.25E+22	4.81E+21	1.31E+21	6.75E+20	6.35E+20	3.52E+20	1.89E+20	1.63E+20	1.68E+20	6.96E+19	9.83E+19
20	2000	1.79E+22	1.15E+22	6.41E+21	2.44E+21	1.22E+21	1.22E+21	7.30E+20	3.68E+20	3.61E+20	3.01E+20	1.48E+20	1.54E+20
20	3000	4.58E+21	3.65E+21	9.28E+20	1.61E+21	1.28E+21	3.27E+20	7.94E+20	4.48E+20	3.46E+20	4.26E+20	2.07E+20	2.20E+20
20	5000	1.65E+22	1.54E+22	1.17E+21	7.22E+21	6.15E+21	1.07E+21	3.74E+21	2.89E+21	8.56E+20	2.36E+21	1.53E+21	8.36E+20
30	500	1.12E+19	5.23E+18	5.97E+18	3.06E+18	1.68E+18	1.37E+18	1.62E+18	1.09E+18	5.26E+17	1.16E+18	1.04E+18	1.24E+17
30	1000	1.21E+20	7.27E+19	4.85E+19	3.44E+19	1.67E+19	1.76E+19	1.55E+19	7.38E+18	8.08E+18	8.65E+18	4.80E+18	3.85E+18
30	1500	3.78E+20	2.47E+20	1.31E+20	1.10E+20	6.17E+19	4.85E+19	5.45E+19	2.71E+19	2.74E+19	2.98E+19	1.38E+19	1.60E+19
30	2000	4.73E+20	3.25E+20	1.48E+20	1.52E+20	1.01E+20	5.17E+19	6.90E+19	3.44E+19	3.46E+19	3.97E+19	2.08E+19	1.89E+19
30	3000	3.60E+20	2.81E+20	7.94E+19	1.61E+20	1.13E+20	4.82E+19	8.19E+19	4.83E+19	3.36E+19	4.88E+19	2.64E+19	2.24E+19
30	5000	8.72E+20	7.07E+20	1.65E+20	5.06E+20	3.72E+20	1.35E+20	2.94E+20	2.13E+20	8.08E+19	1.83E+20	1.18E+20	6.44E+19
40	500	5.23E+18	2.58E+18	2.65E+18	1.44E+18	5.98E+17	8.39E+17	6.11E+17	3.67E+17	2.44E+17	4.23E+17	3.24E+17	9.97E+16
40	1000	6.01E+19	4.30E+19	1.71E+19	1.56E+19	7.44E+18	8.20E+18	6.76E+18	2.93E+18	3.83E+18	3.31E+18	1.73E+18	1.58E+18
40	1500	2.08E+20	1.36E+20	7.28E+19	5.62E+19	3.14E+19	2.48E+19	2.47E+19	1.18E+19	1.29E+19	1.25E+19	5.80E+18	6.73E+18
40	2000	2.31E+20	1.58E+20	7.36E+19	8.31E+19	5.01E+19	3.30E+19	3.52E+19	1.80E+19	1.71E+19	1.73E+19	8.61E+18	8.71E+18
40	3000	1.71E+20	1.38E+20	3.22E+19	8.04E+19	5.30E+19	2.74E+19	3.96E+19	2.38E+19	1.58E+19	2.47E+19	1.12E+19	1.34E+19
40	5000	3.75E+20	3.47E+20	2.85E+19	2.36E+20	1.76E+20	5.96E+19	1.54E+20	9.92E+19	5.46E+19	9.81E+19	5.95E+19	3.85E+19
50	500	2.90E+18	1.30E+18	1.60E+18	7.67E+17	4.10E+17	3.58E+17	3.89E+17	2.49E+17	1.40E+17	2.75E+17	2.24E+17	5.15E+16
50	1000	2.66E+19	1.61E+19	1.06E+19	7.09E+18	3.64E+18	3.45E+18	3.35E+18	1.61E+18	1.74E+18	1.93E+18	1.06E+18	8.74E+17
50	1500	6.23E+19	4.26E+19	1.97E+19	1.99E+19	1.18E+19	8.18E+18	1.01E+19	5.02E+18	5.10E+18	5.94E+18	2.87E+18	3.07E+18
50	2000	7.55E+19	5.53E+19	2.02E+19	2.67E+19	1.80E+19	8.74E+18	1.37E+19	7.01E+18	6.71E+18	7.65E+18	4.04E+18	3.62E+18
50	3000	6.24E+19	4.85E+19	1.39E+19	2.94E+19	1.77E+19	1.18E+19	1.47E+19	8.73E+18	5.96E+18	9.73E+18	5.20E+18	4.53E+18
50	5000	1.47E+20	1.34E+20	1.34E+19	8.20E+19	6.43E+19	1.77E+19	5.18E+19	3.54E+19	1.63E+19	3.63E+19	2.04E+19	1.59E+19

2010) instead of using C# library to generate pseudo random numbers. Table 3 list the minimum speed, overclocking improvement, maximum speed, constants C and ϕ 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 (*Gmax*). For each combination of CPU count (*Cp*) and task count (*Ct*), it lists GA energy consumption (*Ega*), SGA energy consumption (*Esga*), and there difference (*Ega*-*Esga*). Since the objective is to minimize energy usage, *Ega*-*Esga* 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 (*Gga*, *Gsga*) and time used (*Tga*, *Tsga*). It also computes the difference (*Gga*-*Gsga*, *Tga*-*Tsga*). 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

		Generation			Time		
Cp	Ct	Gga	Gsga	Gga-Gsga	Tga	Tsga	Tga-Tsga
10	500	633	403	230	1.297	0.834	0.463
10	1000	814	600	214	4.124	3.058	1.066
10	1500	936	715	221	8.935	6.622	2.313
10	2000	1001	717	284	14.571	10.745	3.826
10	3000	1113	808	305	29.792	21.956	7.836
10	5000	1022	805	217	68.884	54.008	14.876
20	500	941	650	291	1.871	1.31	0.561
20	1000	820	675	145	3.342	2.716	0.626
20	1500	707	587	120	4.826	4.014	0.812
20	2000	832	691	141	8.211	6.919	1.292
20	3000	1058	837	221	19.544	16.187	3.357
20	5000	1007	817	190	42.62	34.748	7.872
30	500	967	634	333	1.755	1.182	0.573
30	1000	943	685	258	3.308	2.53	0.778
30	1500	962	755	207	5.337	4.171	1.166
30	2000	1003	762	241	7.899	6.1	1.799
30	3000	1084	832	252	13.893	10.435	3.458
30	5000	1141	930	211	29.71	23.837	5.873
40	500	965	652	313	1.876	1.292	0.584
40	1000	961	733	228	3.195	2.396	0.799
40	1500	965	742	223	4.673	3.791	0.882
40	2000	1027	793	234	6.848	5.138	1.71
40	3000	1131	849	282	12.402	9.776	2.626
40	5000	1222	916	306	26.614	20.318	6.296
50	500	964	613	351	1.973	1.292	0.681
50	1000	948	688	260	2.962	2.158	0.804
50	1500	1056	761	295	4.674	3.372	1.302
50	2000	1009	807	202	5.829	4.62	1.209
50	3000	1087	856	231	10.858	8.578	2.28
50	5000	1179	919	260	21.367	16.899	4.468

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 $(T_{ga}\text{-}T_{spga})x100/T_{ga}$ for 10 Processors

G_{max}	*R= 500*	*R= 1000*	*R= 1500*	*R= 2000*	*R= 3000*	*R= 5000*
500	44	25	15	8	5	1
1000	85	56	30	22	12	2
1500	84	76	51	37	19	8
2000	74	78	71	50	27	13
3000	57	70	77	75	45	24
5000	23	58	68	72	76	50

Table 9.6 SPGA Time Improvement over GA $(T_{ga}\text{-}T_{spga})x100/T_{ga}$ for 20 Processors

G_{max}	*R= 500*	*R= 1000*	*R= 1500*	*R= 2000*	*R= 3000*	*R= 5000*
500	76	58	49	39	28	26
1000	69	79	70	60	55	41
1500	57	79	79	74	62	49
2000	49	78	81	80	73	56
3000	42	70	76	80	79	67
5000	31	56	69	75	78	77

Table 9.7 SPGA Time Improvement over GA $(T_{ga}\text{-}T_{spga})x100/T_{ga}$ for 30 Processors

G_{max}	*R= 500*	*R= 1000*	*R= 1500*	*R= 2000*	*R= 3000*	*R= 5000*
500	88	75	61	49	25	18
1000	90	86	79	72	54	36
1500	85	90	86	80	68	53
2000	81	87	90	86	82	68
3000	69	82	88	89	88	79
5000	46	76	82	86	90	86

Table 9.8 SPGA Time Improvement over GA $(T_{ga}\text{-}T_{spga})x100/T_{ga}$ for 40 Processors

G_{max}	*R= 500*	*R= 1000*	*R= 1500*	*R= 2000*	*R= 3000*	*R= 5000*
500	81	67	56	45	31	39
1000	81	83	74	68	60	53
1500	76	85	84	79	73	61
2000	70	83	87	85	80	66
3000	58	79	84	86	86	74
5000	43	71	79	82	85	85

Table 9.9 SPGA Time Improvement over GA (T_{ga}-T_{spga})x$100/T_{ga}$ for 50 Processors

G_{max}	*R= 500*	*R= 1000*	*R= 1500*	*R= 2000*	*R= 3000*	*R= 5000*
500	88	74	69	63	52	37
1000	90	85	81	76	72	61
1500	88	91	89	84	80	72
2000	85	89	91	88	84	76
3000	80	85	89	91	90	83
5000	61	80	85	88	91	89

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}-$STsp_{ga}$

P_c	*R= 500*	*R= 1000*	*R= 1500*	*R= 2000*	*R= 3000*	*R= 5000*
10	3	3	10	18	33	60
20	3	8	12	15	36	52
30	5	7	10	13	21	42
40	6	8	12	17	31	43
50	8	10	12	16	20	38

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}-$STsp_{ga}$. 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}

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

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,

$$\text{Minimize } f = (P + S) * C + L \tag{10.1}$$

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

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 step 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

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

Target Efficiency	0.99

Item Length	404	408	473	527	545	576	584	585	597	604
No. Required	58	159	105	7	76	1	226	7	42	20
Item Length	606	636	690	780						
No. Required	62	20	9	284						

Table10.2

GA Result of Sample CSP

Run	Time (s)	Waste	Efficiency
1	3839	6387	0.9900
2	1345	6323	0.9901
3	1967	6339	0.9901
4	2154	6387	0.9900
5	1323	6355	0.9901
6	2629	6371	0.9901
7	1737	6307	0.9902
8	1534	6291	0.9902
9	1405	6387	0.9900
10	2075	6067	0.9905
Average	2001	6321	0.9901

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

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

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

Index	Pattern	Pattern length	Stock Length	Sets
1	545,780	1325	1328	76
2	408,584	992	992	89
3	473,597	1070	1072	35
4	606,780	1386	1392	19
5	408,473,606	1487	1488	43
6	408,473,604	1485	1488	20
7	404,780	1184	1184	58
8	780,780	1560	1568	50
9	690,780	1470	1472	9
10	585,597	1182	1184	7
11	408,473,527	1408	1408	7
12	576,780	1356	1360	1
13	584,584	1168	1168	68
14	636,780	1416	1424	20
15	584,780	1364	1376	1
Efficiency		0.998234		

Table 10.5

Convert LP Solutions to Integer Using Stock 1392

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

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

Table 10.7

Comparison Study on Stock Count Variations

Problem				Pure GA Algorithm			Our New Algorithm		
Name	Total Stock Count	Width Count	Item Count	Time (s)	Efficiency	Waste	Time (s)	Efficiency	Waste
base	10	36	400	1030	0.9954	164	53	0.9954	164
b12s	12	36	400	1681	0.9954	164	53	0.9955	162
b14s	14	36	400	2562	0.9954	164	104	0.9951	176
b16s	16	36	400	3935	0.9951	176	136	0.9951	176
b18s	18	36	400	4366	0.9954	164	98	0.9947	189.5
b20s	20	36	400	4516	0.9951	176	91	0.9958	149

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 10.8

Production Problem Run Result

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

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

Adra, S. F., Dodd, T. J., Griffin, I. A., & Fleming, P.J. (2009). Convergence Acceleration Operator for Multiobjective Optimization. *IEEE Transactions on Evolutionary Computation*, 13(4), 825-847.

Aktin, T., & Özdemir, R.G. (2009). An integrated approach to the one-dimensional cutting stock problem in coronary stent manufacturing. *European Journal of Operational Research*, Vol. 196, Issue 2, 737-743.

Alba, E., & Tomassini, M. (2002), Parallelism and evolutionary algorithms, *IEEE Trans. Evolutionary Computation,* vol. 6, issue 5, pp. 443-462.

Alves, C., & Carvalho, J.M. (2008). A stabilized branch-and-price-and-cut algorithm for the multiple length cutting stock problem. *Computers & Operations Research*, Vol. 35, Issue 4, 1315-1328.

Ang, K. H., Chong, G., & Li, Y. (2002). Preliminary statement on the current progress of multi-objective evolutionary algorithm performance measurement. *Congress on Evolutionary Computation*, *2*, 1139-1144.

Belov, G., & Scheithauer, G. (2006). A branch-and-cut-and-price algorithm for one-dimensional stock cutting and two-dimensional two-stage cutting. *European Journal of Operational Research,* Vol. 171, Issue 1, 85-106.

Benkhider, S., Baba-Ali, A.R., & Drias, H. (2007). A new generationless parallel evolutionary algorithm for combinatorial optimization. *IEEE Congress on Evolutionary Computation 2007*. 4691 – 4697.

Berntsson, J., & Tang, M.(2003). A convergence model for asynchronous parallel genetic algorithms. *The 2003 Congress on Evolutionary Computation*, 4, 2627 – 2634.

Bersini, H. (2002). The immune and chemical crossovers. *IEEE Transactions on Evolutionary Computation*, 6(3), 306–313.

Bianchi, L.,Gambardella, L.M., & Dorigo, M. (2002). An ant colony optimization approach to the probabilistic traveling salesman problem, *Proceedings of the PPSN-VII, Seventh International Conference on Parallel Problem Solving from Nature, Lecture Notes in Computer Science*. Springer Verlag, Berlin, Germany,

Bredstrom, D., Carlsson, D., & Ronnqvist, M. (2005). *A hybrid algorithm for distribution problems. IEEE Transactions on Intelligent Systems, 20(4),* 19 – 25.

Cabusao G, Mochizuki M, Mashiko K, Kobayashi T, Singh R, Nguyen T, Wu P (2010) Data center energy conservation utilizing a heat pipe based ice storage system CPMT Symposium Japan, 2010 IEEE. doi: 10.1109/CPMTSYMPJ.2010.5680287. Publication Year: 2010, Page(s): 1 - 4

Cantu-Paz, E. (1997). Design Efficient Master_Slave Parallel Genetic Algorithms. http://citeseerx.ist.psu.edu/viewdoc/download? doi=10.1.1.28.4982&rep=rep1&type=pdf. Accessed March 2009. Also available at: *Proceeding of the Third Annual Conference on Genetic Programming*, 455-460.

Caponetto, R., Fortuna, L., Fazzino, S., & Xibilia, M.G. (2003). Chaotic sequences to improve the performance of evolutionary algorithms. *IEEE Transactions on Evolutionary Computation, 7(3),* 289-304.

Chang, P.C., Wu, I.W., Shann, J.J., & Chung, C.P, (2008). ETAHM: An energy-aware task allocation algorithm for heterogeneous multiprocessor, 45th *ACM/IEEE Design Automation Conference*, 2008, pp. 776 – 779.

Cherri, A.C., Arenales, M.N., & Yanasse, H.H. (2009). The one-dimensional cutting stock problem with usable leftover – A heuristic approach. *European Journal of Operational Research,* Vol. 196, Issue 3, 897-908.

Chiong, R., Chang, Y.Y., Chai, P.C., & Wong A. L. (2008). A selective mutation based evolutionary programming for solving Cutting Stock Problem without contiguity. *IEEE Congress on Evolutionary Computation,* 1671 – 1677. doi: 10.1109/CEC.2008.4631015

Choi, I.C., Kim, S.I., & Kim, H.S. (2003). A genetic algorithm with a mixed region search for the asymmetric traveling salesman problem, *Computer Operations Research*. 30 (5), 773–786.

Church, k., Greenberg, A., Hamilton, J. (2008) On delivering embarrassingly distributed cloud services, In *ACM HotNets VII, 2008*. http://www.techrepublic.com/whitepapers/on-delivering-embarrassingly-distributed-cloud-services/2388125

Consortium for School Networking Initiative, (2010), "Some Facts About Computer Energy Use",http://www.cosn.org/Initiatives/GreenComputing/InterestingFacts/tabid/4639/Default.aspx, Accessed Oct. 2010.

Cui, Y., & Lu, Y. (2009). Heuristic algorithm for a cutting stock problem in the steel bridge construction. *Computers & Operations Research,* Vol. 36, Issue 2, 612-622.

Cui, Y., & Yang, Y. (2010). A heuristic for the one-dimensional cutting stock problem with usable leftover. *European Journal of Operational Research,* Vol. 204, Issue 2, 245-250.

Dantzig, G. B. (1963). Linear Programming and Extensions. *Princeton University Press*.

Deng, G.Q., Huang, Z.C., & Tang, M. (2007). Research in the Performance Assessment of Multi-objective Optimization Evolutionary Algorithms. *International Conference on Communications, Circuits and Systems,* 915-918.

El-Araby, E. E., Yorino, N., & Zoka, Y. (2005). Optimal procurement of VAR ancillary service in the electricity market considering voltage security. *IEEE International Symposium on Circuits and Systems, 5*, 5290-5293.

El-Araby, E. E., Yorino, N., & Sasaki, H. (2002). A comprehensive approach for FACTS devices optimal allocation to mitigate voltage collapse. *IEEE/PES Conference and Exhibition Transmission and Distribution.*

Falkenauer, E., & Delchambre, A. (1992). A genetic algorithm for bin packing and line balancing. *Proceedings of 1992 IEEE International Conference on Robotics and Automation*, Vol. 2, 1186 – 1192. doi: 10.1109/ROBOT.1992.220088

Feng, C., Wang, X., & Li, F. (2009). Optimal maintenance scheduling of power producers considering unexpected unit failure. *Generation, Transmission & Distribution, IET, 3*, 460-471.

Garg, S., Konugurthi, P., & Buyya, R. (2009) A Linear Programming Driven Genetic Algorithm for Meta-Scheduling on Utility Grids. Accessed in Aug 2009 at: http://arxiv.org/ftp/arxiv/papers/0903/0903.1389.pdf.

Gilmore, P.C., & Gomory, R. E. (1961). A Linear Programming Approach to the Cutting-Stock Problem. *Operations Research,* 9, 849–859.

Gilmore, P.C., & Gomory, R.E. (1963). A linear programming approach to the cutting-stock problem, Part II. *Operations Research,* 11, 863–888.

Gilmore, P.C., & Gomory, R.E. (1965). Multistage cutting stock problems of two and more dimensions. *Operations Research* 13, 94–120.

Gilmore, P.C., & Gomory, R.E. (1966). The theory and computation of knapsack functions. *Operations Research*, 14, 1045–1074.

Green Computing (2010). Retrieved from http://en.wikipedia.org/wiki/Green_computing.

Hamann H.F., López, V, Stepanchuk A (2010) Thermal zones for more efficient data center energy management Thermal and Thermomechanical Phenomena in Electronic Systems (ITherm), 2010 12th IEEE Intersociety Conference on doi: 10.1109/ITHERM.2010.5501332. Publication Year: 2010, Page(s): 1 – 6

Hasegawa, M., Ikeguchi, T., & Aihara, K. (2002). Solving large scale traveling salesman problems by chaotic neurodynamics, *Neural Networks* 15 (2), 271–283.

Hayashi, T., Takeuchi, A., & Nozaki, Y. (2008). A method for determining optimum design for fuel-cell-based energy network. *IEEE 30th International Conference on Telecommunications Energy*, 1-7.

Hinterding R., & Khan L. (1994). Genetic algorithms for cutting stock problems: with and without contiguity. *Progress in evolutionary computation. Lecture notes in artificial intelligence, vol. 956*, 166-186.

Hinterding, R. (1997). Self-adaptation using multi-chromosomes. *IEEE International Conference on Evolutionary Computation*, 87 – 91.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems.* University of Michigan Press.

Holland, J. H. (1992). *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. The MIT Press.

Hung, K.S., Su, S.F., & Lee, Z.J. (2007). Improving ant colony optimization algorithms for solving traveling salesman problems, *Journal of Advanced Computational Intelligence and Intelligent Informatics* 11 (4) , 433–434.

Ishibuchi, H., Nojima, Y., & Tsutomu, D. (2006). Comparison between Single-Objective and Multi-Objective Genetic Algorithms: Performance Comparison and Performance Measures. *IEEE Congress on Evolutionary Computation*, 1143-1150.

Iyengar M, Schmidt R, Caricari J, Reducing energy usage in data centers through control of Room Air Conditioning units Thermal and Thermomechanical Phenomena in Electronic Systems (ITherm), 2010 12th IEEE Intersociety Conference on doi: 10.1109/ITHERM.2010.5501418. Publication Year: 2010, Page(s): 1 – 11

Jelodar, M.S.; Kamal, M.; Fakhraie, S.M.; Ahmadabadi, M.N.(2006). SOPC-Based Parallel Genetic Algorithm. *IEEE Congress on Evolutionary Computation*, 2800 - 2806.

Kaur, D. and Murugappan, M.M. (2008) Performance enhancement in solving Traveling Salesman Problem using hybrid genetic algorithm. *Annual Meeting of the North American of Fuzzy Information Processing Society*. 1 – 6

Kirkpatrick, S., Gelatt, C. & Vecchi, M. (1983) Optimization by simulated annealing, *Science* 220, 671–680.

Kolman, B., & Beck, R.E. (1980) Elementary Linear Programming with Applications, *Academic Press, Inc. Chapter 2,3.*

Koza, J.R., Keane, M.A., Streeter, M.J., Mydlowec, W., Yu, J., & Lanza, G. (2005). Genetic Programming IV: Routine Human-Competitive Machine Intelligence. *Springer*.

Koduru, P., Das, S., Welch, S.M., Roe, J., & Lopez-Dee, Z.P. (2005). A co-evolutionary hybrid algorithm for multi-objective optimization of gene regulatory network models. *Genetic and Evolutionary Computation Conference*. 393–399.

Koduru, P., Dong, Z.S., Das, S., Welch, S.M., Roe, J.L., & Charbit, E., (2008), A Multiobjective Evolutionary-Simplex Hybrid Approach for the Optimization of Differential Equation

Models of Gene Networks, *IEEE Trans. Evolutionary Computation*, vol. 12, issue 5, pp. 572-590.

Kulkarni, A.J. & Tai, K. (2009) Probability Collectives: A multi-agent approach for solving combinatorial optimization problems, *Applied Soft Computing*, to be published.

Laszewski, G. von, Wang, L., Younge, A.J., He, X., (2009). Power-aware scheduling of virtual machines in DVFS-enabled clusters, IEEE Intl. Conf. on Cluster Computing and Workshops, 2009, pp.1-10, doi: 10.1109/CLUSTR.2009.5289182.

Lee, Z.-J., Lee, C.-Y. & Su, S.-F.(2002) An immunity-based ant colony optimization algorithm for solving weapon–target assignment problem, *Applied Soft Computing*, 2 (1), 39-47

Lee, C.-H., Park, S.-H., & Kim, J.-H. (2000). Topology and migration policy of fine-grained parallel evolutionary algorithms for numerical optimization. *Proceedings of the 2000 Congress on Evolutionary Computation*, 1, 70 – 76.

Lee, C.-H., Park, K.-H., & Kim, J.-H. (2001). Hybrid parallel, evolutionary algorithms for constrained optimization utilizing PC clustering. *Proceedings of the 2001 Congress on Evolutionary Computation*, 2, 1436 – 1441.

Leou, R.-C. (2008). An economic analysis model for the energy storage systems in a deregulated market. *IEEE International Conference on Sustainable Energy Technologies*, 744–749.

Levine, J., & Ducatelle, F. (2004). Ant colony optimization and local search for bin packing and cutting stock problems. Retrieved December 10, 2009, from http://www.palgrave-journals.com/jors/journal/v55/n7/full/2601771a.html

Li, K. (2008). Performance Analysis of Power-Aware Task Scheduling Algorithms on Multiprocessor Computers with Dynamic Voltage and Speed. *IEEE Transactions on Parallel and Distributed Systems*, 19(11), 1484-1497.

Li, X., & Kirley, M. (2002) The effects of varying population density in a fine-grained parallel genetic algorithm. *Proceedings of the 2002 Congress on Evolutionary Computation*. 2, 1709 – 1714.

Li, Y., Liu, Y., & Qian, D., (2009). A Heuristic Energy-aware Scheduling Algorithm for Heterogeneous Clusters, 2009 *15th Intl. Conf. on Parallel and Distributed Systems (ICPADS)*, pp. 407 - 413, doi:10.1109/ICPADS.2009.33.

Liang, C.H., Chung, C.Y., Wong, K.P., & Duan, X.Z. (2007), Parallel Optimal Reactive Power Flow Based on Cooperative Co-Evolutionary Differential Evolution and Power System Decomposition, *IEEE Trans. Power Systems*, vol. 22, issue 1, pp. 249-257.

Liang, K.-H., Yao, X., Newton, C., & Hoffman, D. (2002). A new evolutionary approach to cutting stock problems with and without contiguity. *Computers & Operations Research*, Vol. 29, Issue 12, 1641-1659.

Liu, Y., Chu, C.B., & Wang, K.L. (2008). A heuristic procedure based on column generation to solve a cutting stock problem. *IEEE International Conference on Industrial Engineering and Engineering Management.*

Liu, Y., Yang, H., Luo, R., & Wang, H., (2006). Combining Genetic Algorithms Based Task Mapping and Optimal Voltage Selection for Energy-Efficient Distributed System Synthesis, *2006 Intl. Conf. on Communications, Circuits and System* , vol. 3, pp. 2074 - 2078, doi: 10.1109/ICCCAS.2006.285087.

Lu, Q., Wang, Z., & Chen, M. (2008). An Ant Colony Optimization Algorithm for the One-Dimensional Cutting Stock Problem with Multiple Stock Lengths. *Fourth International Conference on Natural Computation, Vol. 7*, 475-479. doi: 10.1109/ICNC.2008.208

Mantovani, J.R.S., Modesto, S.A.G., & Garcia, A.V. (2001). VAr planning using genetic algorithm and linear programming. Generation, Transmission and Distribution, 148(3), 257-262.

Massa, A., Franceschini, D., Franceschini, G., Pastorino, M., Raffetto, M., & Donelli, M. (2005) Parallel GA-based approach for microwave imaging applications, *IEEE Trans. Antennas and Propagation,* vol. 53, issue 10, pp. 3118-3127.

Matsumura, T., Nakamura, M., Miyazato, D., Onaga, K., & Okech, J.(1997). Effects of chromosome migration on a parallel and distributed genetic algorithm. Third International Symposium on Parallel Architectures, Algorithms, and Networks, 357 – 361.

Miao, L., Qi, Y., Hou, D., & Dai, Y.H., (2007). Energy-Aware Scheduling Tasks on Chip Multiprocessor, *Third International Conference on Natural Computation*, 2007, vol. 4, pp. 319 - 323, doi: 10.1109/ICNC.2007.356.

Miao, L., Qi, Y., Hou, D., Dai, Y.H., & Shi, Y., (2008). A multi-objective hybrid genetic algorithm for energy saving task scheduling in CMP system, IEEE Intl. Conf. on Systems, Man and Cybernetics, 2008, pp. 197–201, doi: 10.1109/ICSMC.2008.4811274.

Nagata, Y., & Kobayashi, S. (1999). An analysis of edge assembly crossover for the traveling salesman problem. *1999 IEEE International Conference on Systems, Man, and Cybernetics, 3.* 628 – 633.

Nelder, J. A., & Mead, R. (1965). A simplex method for function minimization. *Computer Journal,* 7(4), 308–313.

Noman, N., & Iba, H. (2008). Accelerating Differential Evolution Using an Adaptive Local Search. *IEEE Transactions on Evolutionary Computation, 12(1),* 107-125.

Ortiz-Garcia, E.G., Martinez-Bernabeu, L., Salcedo-Sanz, S., Florez-Revuelta, F., Perez-Bellido, A.M., & Portilla-Figueras, A. (2009). A Parallel evolutionary algorithm for the hub location problem with fully interconnected backbone and access networks, *IEEE Congr. Evolutionary Computation*, pp. 1501-1506, 18-21 May 2009.

Paenke, I., Branke, J., & Jin, Y.C. (2006). Efficient search for robust solutions by means of evolutionary algorithms and fitness approximation. *IEEE Transactions on Evolutionary Computation*, 10(4). 405-420.

Page, A.J., & Naughton, T.J., (2005). Dynamic Task Scheduling using Genetic Algorithms for Heterogeneous Distributed Computing, *19th IEEE Intl. Conf. on Parallel and Distributed Processing*, 2005. DOI: 10.1109/IPDPS.2005.184

Pandey, S., Dong, S.Q., Agrawal, P., & Sivalingam, K. (2007). A Hybrid Approach to Optimize Node Placements in Hierarchical Heterogeneous Networks. *IEEE Conference on Wireless Communications and Networking*, 3918-3923.

Poldi, K.C., & Marcos, M.N. (2009). Heuristics for the one-dimensional cutting stock problem with limited multiple stock lengths. *Computers & Operations Research,* Vol. 36, Issue 6, 2074-2081.

Random.org, http://www.random.org. Accessed Oct. 2010.

Ray, S.S., Bandyopadhyay, S. & Pal, S.K.(2004) New operators of genetic algorithms for traveling salesman problem. ICPR 2004. *Proceedings of the 17th International Conference on Pattern Recognition, 2004.* 2, 497 - 500.

Regis, R. G., & Shoemaker, C. A. (2004). Local function approximation in evolutionary algorithms for the optimization of costly functions. *IEEE Transactions on Evolutionary Computation, 8(5).* 490 - 505.

Robin, F., Orzati, A., Moreno, E., Otte, J. H., & Bachtold, W. (2003). Simulation and evolutionary optimization of electron-beam lithography with genetic and simplex-downhill algorithms. IEEE Transaction on Evolution Computation, 7(1), 69–82.

Scott, S.D.; Samal, A.; Seth, S. (1995). HGA: A Hardware-Based Genetic Algorithm. Field-Programmable Gate Arrays. *Proceedings of the Third International ACM Symposium on FPGA*, 53 – 59.

Shen, G. and Zhang, Y. Q. (2010-1) "A Novel Genetic Algorithm", *The 9th International FLINS Conference on Foundations and Applications of Computational Intelligence* (FLINS2010), Aug. 2-4, 2010.

Shen, G. and Zhang, Y. Q. (2010-2) "Solving the Stock Reduction Problem with the Genetic Linear Programming Algorithm", *The 2010 International Conference on Computational and Information Sciences (ICCIS2010),* Dec. 17 - 19, 2010.

Shen, G. and Zhang, Y. Q. (2011-1) "A New Evolutionary Algorithm Using Shadow Price Guided Operators", *Applied Soft Computing*, vol. 11, issue 2, pp. 1983-1992, DOI 10.1016/j.asoc.2010.06.014

Shen, G. and Zhang, Y. Q. (2011-2) "A Shadow Price Guided Genetic Algorithm for Energy Aware Task Scheduling on Cloud Computers", *International Conference on Swam Intelligence (ICSI) 2011*, 522-529.

Shen, G. and Zhang, Y. Q. (2012-1) "An Evolutionary Linear Programming Algorithm for Solving the Stock Reduction Problem", *International Journal of Computer Applications in Technology (IJCAT),* 2012.

Shen, G. and Zhang, Y. Q. (2012-2) "Shadow Price Based Genetic Algorithms for the Cutting Stock Problem", *International Journal of Artificial Intelligence and Soft Computing (IJAISC)* , 2012.

Shen, G. and Zhang, Y. Q. (2012-3) "Power Consumption Constrained Task Scheduling Using Enhanced Genetic Algorithms", *Evolutionary Based Solutions for Green Computing*, Springer, 2012.

Singh, H.K., Isaacs, A., Nguyen, T.T., Ray, T., & Yao, X. (2008). Performance of infeasibility driven evolutionary algorithm (IDEA) on constrained dynamic single objective optimization problems. *IEEE Congress on Evolutionary Computation*, 3127-3134.

Singh, H. K., Isaacs, A., Ray, T., & Smith, W. (2008). Infeasibility Driven Evolutionary Algorithm (IDEA) for Engineering Design Optimization. *21st Australasian Joint Conference on Artificial Intelligence*, 104–115.

Simoncini, D., Collard, P., Verel, S., & Clergue, M. (2007). On the influence of selection operators on performances in cellular Genetic Algorithms. *IEEE Congress on Evolutionary Computation*, 4706 - 4713.

Song, X., Chu, C.B., Nie, Y.Y., & Bennell, J.A. (2006). An iterative sequential heuristic procedure to a real-life 1.5-dimensional cutting stock problem. *European Journal of Operational Research*, Vol. 175, Issue 3, 1870-1889.

Syswerda, G. (1991) Schedule Optimization Using Genetic Algorithm. In *Handbook of Genetic Algorithms.* Van Nostrand Reinhold.

Tan, T. G., Teo, J., & Lau, H.K. (2007). Performance Scalability of a Cooperative Coevolution Multiobjective Evolutionary Algorithm. *International Conference on Computational Intelligence and Security*, 119-123.

Tian, L., & Arslan, T., (2003). A genetic algorithm for energy efficient device scheduling in real-time systems, 2003 Congress on Evolutionary Computation , pp.242-247.

Tournament Selection (2010). Retrieved from http://en.wikipedia.org/wiki/Tournament_selection

Tsai, H.-K., Yang, J.-M., & Kao, C.-Y. (2002). Solving traveling salesman problems by combining global and local search mechanisms. *Congress on Evolutionary Computation, 2002.* 2, 1290 – 1295.

Tseng, L.-Y., & Chen, S.-C. (2009). Two-Phase Genetic Local Search Algorithm for the Multimode Resource-Constrained Project Scheduling Problem. *IEEE Transactions on Evolutionary Computation*, 13*(4)*, 848-857.

TSPLIB (2009). http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/ Accessed Dec. 2009.

Tsutsui, S., Goldberg, D. E., & Sastry, K. (2001). Linkage learning in real coded GAs with simplex crossover. *5th International Conference on Artificial Evolution*, 73–84.

US Environmental Protection Agency. (2010). "EPA Report on Serverand Data Center Energy Efficiency", August 2007. http://www.energystar.gov/ia/partners/prod_development/downloads/EPA_Datacenter_Report_Congress_Final1.pdf . Accessed Oct. 2010.

U.S. Energy Information Administration Independent Statistic and Analysis, (2010). "Renewable Energy Consumption and Electricity Preliminary 2006 Statistics", http://www.eia.doe.gov/cneaf/solar.renewables/page/prelim_trends/rea_prereport.html, accessed Oct. 2010.

Vishwanathan, N. & Wunsch, D.C. (2001). ART/SOFM: a hybrid approach to the TSP, *INNS–IEEE International Joint Conference on Neural Networks* (IJCNN'01), 4.

Wang, K.-P., Huang, L., Zhou, C.-G., & Pang, W., (2003). Particle swarm optimization for traveling salesman problem, *Proceedings of the Second International Conference on Machine Learning and Cybernetics*, 1583–1585.

Wang, L., Laszewski, G. von, Dayal, J., He, X., & Furlani, T. R., (2009). Thermal Aware Workload Scheduling with Backfilling for Green Data Centers, the 28th IEEE Intl. Conf. on Performance Computing and Communications, Dec 2009, doi: 10.1109/PCCC.2009.5403821. (2009)

Wang, L., Laszewski, G.V., Dayal, J., & Wang, F. (2010). Towards Energy Aware Scheduling for Precedence Constrained Parallel Tasks in a Cluster with DVFS, *2010 10th IEEE/ACM Intl. Conf. on Cluster, Cloud and Grid Computing (CCGrid)* , pp. 368 - 377, doi: 10.1109/CCGRID.2010.19

Wang, Q.L., Xu, G.X., Dai, Y.M., Zhao, B.Z., Yan, L.G., & Kim, K.M. (2009). Design of Open High Magnetic Field MRI Superconducting Magnet With Continuous Current and Genetic Algorithm Method. *IEEE Transactions on Applied Superconductivity*, 19(3), 2289-2292.

Wang, C., Zhang, J., Yang, J. Hu, C., & Liu, J., (2005), A modified particle swarm optimization algorithm and its applications for solving travelling salesman problem, *Proceedings of the International Conference on Neural Networks and Brain*, ICNN, 2, 689–694.

Watson, J., Ross, C., Eisele, V., Denton, J., Bins, J., Guerra, C., Whitley, D., & Howe, A. (1998). The Traveling Salesrep Problem, edge assembly crossover, and 2-opt. http://jason.denton.googlepages.com/tsp_ga.pdf (1998).

Wikipedia 2-opt (2009) http://en.wikipedia.org/wiki/2-opt, Accessed Dec. 2009.

Wilipedia, Instructions Per Secound, (2010). http://en.wikipedia.org/wiki/Instructions_per_second, accessed Oct. 2010.

Wong, L.-P., Low, M.Y.H., & Chong, C.S. (2008) A Bee Colony Optimization Algorithm for Traveling Salesman Problem. *AICMS (2008),* 818 – 823.

Xie, Y., Wang, Z. & S. Wei, (2005). An efficient algorithm for nonpreemptive periodic task scheduling under energy constraints, 6th Intl. Conf. on ASIC, pp. 128-131, doi: 10.1109/ICASIC.2005.1611282.

Xuan, W., & Li, Y. (2005) Solving Traveling Salesman Problem by Using A Local Evolutionary Algorithm. *IEEE International Conference on Granular Computing.* 1, 318 – 321.

Yahoo Green, (2010). Sustainable Energy 101, http://green.yahoo.com/global-warming/globalgreen-140/sustainable-energy-101.html, accessed Oct. 2010.

Yanasse, H.H., & Lamosa, M.J.P. (2007). An integrated cutting stock and sequencing problem. *European Journal of Operational Research*, Vol. 183, Issue 3, 1353-1370.

Yanasse, H.H., & Limeira, M.S. (2006). A hybrid heuristic to reduce the number of different patterns in cutting stock problems. *Computers & Operations Research*, Vol. 33, Issue 9, 2744-2756.

Yang, B., Li, C., Huang, L., Tan, Y., & Zhou, C. (2009). Solving One-dimensional Cutting-Stock Problem Based on Ant Colony Optimization. *Fifth International Joint Conference on INC, IMS and IDC,* 1188 - 1191. doi: 10.1109/NCM.2009.233.

Yang K., & Liu, X.B.(2008). Improving the Performance of the Pareto Fitness Genetic Algorithm for Multi-Objective Discrete Optimization. *International Symposium on Computational Intelligence and Design, 2,* 394-397.

Yang, J. & Zhuang, Y., (2010). An improved ant colony optimization algorithm for solving a complex combinatorial optimization problem, *Applied Soft Computing*, 10 (2), 653-660

Yue, Q., & Gao, L. (2009). Genetic annealing algorithm for cutting stock problem in furniture industry. *IEEE 10th International Conference on Computer-Aided Industrial Design & Conceptual Design*, 87-91. doi: 10.1109/CAIDCD.2009.5374979.

Yuen, S.Y., & Chow, C. K. (2009). A Genetic Algorithm That Adaptively Mutates and Never Revisits. *IEEE Transactions on Evolutionary Computation*, *13(2),* 454-472.

Zhang, H., & Ishikawa, M. (2005). Performance Improvement of Hybrid Real-Coded Genetic Algorithm with Local Search and Its Applications. *International Conference on Computational Intelligence for Modeling, Control and Automation, and International Conference Intelligent Agents, Web Technologies and Internet Commerce*, *1*. 1171-1176,

Zhang, Q., & Li, H. (2007). MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition. *IEEE Transactions on Evolutionary Computation*, *11(6),* 712-731.

Zhang, L.M., Li, K., & Zhang, Y.Q., (2010). Green Task Scheduling Algorithms with Speeds Optimization on Heterogeneous Cloud Servers, *2010 IEEE/ACM Intl. Conf. on Green Computing and Communications (GreenCom2010)*, pp. 76-80.

Zhao, F., Dong, J., Li, S., & Yang, X. (2008) An improved genetic algorithm for the multiple traveling salesman problem. *Control and Decision Conference*. 1935 – 1939.

Zhao, T., Man, Z., Wan, Z., & Bi, G. (2008) A CGS-MSM Parallel Genetic Algorithm Based on Multi-agent. *Second International Conference on Genetic and Evolutionary Computing*. 10 – 13.

Zhi, X.H., Xing, X.L., Wang, Q.X., Zhang, L.H., Yang, X.W., Zhou, C.G., & Laing, Y.C., (2004) A discrete PSO method for generalized TSP problem, *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, 4, 2378–2383.

Zhong, W., Zhang, J., and Chen, W. (2007) A novel Discrete Particle Swarm Optimization to Solve Traveling Salesman Problem, *Evolutionary Computation, (2007)*, 3283 – 3287