Solving a Class of Time-Dependent Combinatorial Optimization Problems with Abstraction, Transformation and Simulated Annealing

by
Rigoberto Diaz, B.B.A., M.B.A.

Submitted in partial fulfillment
of the requirements for the degree of
Doctor of Professional Studies
in Computing

at

School of Computer Science and Information Systems

Pace University

November 2003

We hereby certify that this dissertation, submitted by Rigoberto Diaz, satisfies the dissertation requirements for the degree of *Doctor of Professional Studies in Computing* and has been approved.

_____-_____
Lixin TaoDate
Chairperson of Dissertation Committee

_____-_____
Michael GarganoDate
Dissertation Committee Member

_____-_____
Fred Grossman Date
Dissertation Committee Member

School of Computer Science and Information Systems
Pace University 2003

Abstract

Solving a Class of Time-Dependent Combinatorial Optimization Problems with Abstraction, Transformation and Simulated Annealing

by
Rigoberto Diaz, B.B.A, M.B.A

Submitted in partial fulfillment
of the requirements for the degree of
Doctor of Professional Studies
in Computing

November 2003

While the operations research community has been working on combinatorial optimization problems for over half a century, most of the problems considered so far have constant event costs. This dissertation is dedicated to efficient solutions to a class of real-world combinatorial optimization problems whose event costs are time-dependent.

A class of time-dependent problems is first identified and abstracted into a mathematical model. Based on some critical observation on the model, a problem transformation algorithm is proposed to significantly shrink the solution space while maintaining equivalency to the original problem. This problem transformation can benefit any solution strategies for this class of problems.

Since the class of problems is NP-hard, a comprehensive literature survey is conducted for the prevailing meta-heuristics for solving NP-hard problems, including local optimization, genetic algorithms, simulated annealing, and tabu search. Simulated annealing is adopted as the base of this research's solution strategy due to its proven convergence to global optimum when its temperature is reduced slowly enough. Comprehensive experiments are conducted to study the sensitivity of the simulated annealing algorithm to the values and strategy of its multiple parameters including initial temperature, cooling schedule, stopping criteria for the same temperature, and stopping criteria for the algorithm.

More than 70 problem instances are generated to evaluate the relative performance of the proposed simulated annealing algorithm against repeated random solutions and one of the published genetic algorithms for the same problem. The size of the problem instances ranges from 4 to 200. Considered performance categories include both solution quality and running time. Experiments show that the proposed simulated annealing algorithm outperforms the genetic algorithm by a factor of 5% to 116% while reducing the latter's running time by a factor of 2 to 145.

Acknowledgements

Acknowledgement body should be formatted with tag “Normal Single-Spaced.”

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

During World War II, British military leaders asked scientists and engineers to analyze several military problems: the development of radar and the management of convoy, bombing, anti-submarine, and mining operations. The application of mathematics and the scientific methods to military operations was called operations research. Today, the term *operations research* (or, often, *management science*) means a scientific approach to decision making, which seeks to determine how best to design and operate a system, usually under conditions requiring the allocation of scarce resources.

Combinatorial optimization is an active branch of operations research that focuses on finding the most cost effective solutions to real-world engineering problems in which solutions are made up of discrete objects or integer values. For example, one of such problems could be how to partition a large VLSI circuit into two or more sub-circuits so each of them can be implemented on a separate chip, the components of the circuit are evenly distributed to the chips, and the connection lines across the chips can be minimized.

While combinatorial optimization problems are very common in the industry design and management problems, most of them are intrinsically hard (NP-hard [8]) to be solved by traditional mathematical approaches. As a matter of fact mathematicians and computer

scientists have proven that for the popular NP-hard problems, no algorithms can ever be designed to generate optimal solutions to real-world problem instances [8]. Therefore the industries have to turn to heuristics to find optimized solutions to these hard real-world problems. Computer scientists and mathematicians have abstracted the common patterns of these heuristics into several *meta-heuristics* as reusable knowledge in problem solving for intractable problems.

Traditionally the costs of events in a combinatorial optimization problem are constants. However many of the real-world industry problems also have event costs whose value change with time. For example, the highway construction costs vary by season. In 2001 Prof. Michael L. Gargano and Prof. William Edelson [1] described a set of combinatorial optimization problems in which the costs of events change with time. In 2002 Joseph DeCicco [2] developed in his dissertation a genetic algorithm solution to two of the problems in [1]. In this dissertation we further study these time-dependent combinatorial optimization problems and contribute to their efficient solutions.

1.1 Problem Statement and Solution Strategies

While there are many time-dependent combinatorial optimization problems, we can classify them into categories based on their common properties, and design a common mathematical model for each of these categories. As a result, any research results on such a mathematical model can be applied to all the problems in that particular category. We call such mathematical models our *problem formulations*.

In this section we informally specify a class of time-dependent problems that we are going to address in this dissertation, outline their example applications, and discuss our fundamental solution strategies.

1.1.1 Problem Statement

Table 1 Problem Statement

We have a set of tasks and a set of workers. Each task must be conducted by one worker. The tasks must be conducted in consecutive time units, one task in one time unit, but in any order. Each worker can bid to work on one task only, but his bidding cost depends on the time unit in which the task will be conducted. Find an optimal assignment of the workers to the tasks and an optimal ordering to conduct these tasks so that the total cost to complete all of these tasks could be minimized.

The above generic problem statement can be used to model many real-world time-dependent combinatorial optimization problems. For example, if we let building highway segments be the tasks, and the bidding companies be the workers, then we have the *Highway Minimum Bidding Problem* in [1] (refer to Sub-Section 3.1.2 on page 19 for details). If we define tasks to be placing a satellite receiver for each group of mutually reachable communications network backbone nodes, and define the involved communications network backbone nodes as the workers, w have the *Minimum Cost Satellite Receiver Placement Problem* in [1] (refer to Sub-Section 3.1.1 on page 18 for details).

1.1.2 Solution Strategies

Given a combinatorial optimization problem, the most important work is to design a proper problem formulation (mathematical model). For the same problem, we may usually define many different problem formulations, each with its own solution space structure. A proper problem formulation is much more important than choosing the right solution strategies.

In this research we first carefully study the properties of a class of time-dependent problems that fit in the specification in Sub-Section 1.1.1 and propose a unified proper problem formulation as the mathematical model for their design of solution strategies. We identify the unique property of this model that the choices for which worker to conduct a task are *independent*. This critical observation is the key to understand this class of problems, and the foundation of this dissertation research. We proposed two problem formulations, one is more straightforward but has a larger solution space, and the other can be derived from the first problem formulation and have a significantly smaller solution space. We proved that these two formulations are fundamentally equivalent, but the latter provides the base for any efficient solution strategies.

Since the problem class at hand is NP-hard [1], we have to use heuristics to find their optimized solutions. We surveyed several prevailing meta-heuristics in the operations research domain including local optimization, genetic algorithms [7], simulated annealing [3][4], and tabu search [5]. We adopted simulated annealing as the base of our solution strategy because it is the only meta-heuristic that has been proven to converge to the global optimal solutions when its temperature is reduced slow enough [3].

1.2 Research Contributions

The major contributions of this research include

- Abstracting and formulating a class of time-dependent problems;
- Designing a problem transformation algorithm to significantly reduce the solution space, and proving the equivalence of the transformed problems to the original ones;
- Designing solution space, solution moves, and solution neighborhood;
- Designing and implementing exhaustive search and local optimization as reference algorithms for performance evaluation;
- Implementing Joseph DeCicco's genetic algorithm [2] as a reference algorithm for performance evaluation;
- Designing a simulated annealing algorithm and studying its sensitivity to various cooling strategies and parameters
- Designing experiments to evaluate both solution quality and run-time for various solution algorithms, and analyzing the resulting data to provide insights

1.3 Dissertation Outline

Chapter 2 surveys the major meta-heuristics for combinatorial optimization, and provides a base for understanding the key ideas of simulated annealing. Chapter 3 is the heart of this dissertation, and it proposes two problem formulations to a class of time-dependent combinatorial optimization problems, constructively prove their equivalence, and

demonstrate the advantages of the new problem formulation. Chapter 3 also provides our contributions in the design of solution moves and solution neighborhood as well as the incremental update of the objective function, which are the common base of most of the algorithms in this dissertation. Chapter 4 designs several reference algorithms for performance evaluation comparison of our simulated annealing algorithm. Chapter 5 provides the design of our simulated annealing algorithm, and conduct sensitivity analysis of the algorithm to its various parameters. Chapter 6 designs experiments and systematically compare the solution quality and running time of all the algorithms designed and implemented in this dissertation. Chapter 7 concludes the dissertation with some observations.

Chapter 2

Combinatorial Optimization Strategies

This chapter surveys the fundamental concepts and meta-heuristics that have been proven effective in solving combinatorial optimization problems [11].

2.1 Fundamental Concepts

2.1.1 Generic Formulation of Combinatorial Optimization

A combinatorial optimization problem is typically specified with a *solution space S*, a *cost function* $f : S \to \Re$ where $\Re$ is the set of real numbers, and a *constraint function* $c : S \to \{true, false\}$. In its general form, a combinatorial optimization problem looks for an optimal solution $s \in S$ such that $c(s)$ is true and $f(s)$ is minimized.

2.1.2 NP-hardness of a Problem

If an algorithm has an exponential time complexity $O(2^n)$, the algorithm cannot be used when problem instance is larger than 100 or so. Since each decimal digit is represented roughly by three binary digits, 2^{100} is roughly equal to 10^{33}. The fastest supercomputer today can process less than 10^{15} floating-point operations per second. Therefore, 10^{33} operations will take $10^{33}/10^{15} = 10^{18}$ seconds, or at least 10^{14} hours, or at least 10^{10} years. Even a supercomputer of the future wouldn't be helpful.

Over the last 40 years computer scientists have identified a set of problems and proved that, with a high probability, no algorithms can solve these problems with a time complexity fundamentally different from $O(2^n)$ [8]. This is an unusual achievement in science since the claim applies to the intelligence of future human development. These problems are called *NP-hard* or *intractable* since there is no hope to come up with efficient algorithms to solve them for practical problem instances.

Unfortunately, many interesting problems in science and engineering belong to this intractable category. The problems that we study in this dissertation are intractable.

Even though we do not have efficient algorithms to solve these intractable problems, industries need good solutions to these types of problems. Any small improvement in the solution quality may imply significant benefits. So the problem becomes: within practical time limits, how can we find optimized, instead of optimal, solutions?

2.1.3 Solution Space, Moves and Neighborhood

Given any combinatorial optimization problem, a set consisting all of its potential solutions is called its *solution space*. Those solutions in the solution space that satisfy the constraint function are called *feasible solutions*.

Given a current feasible solution, a *move* to the solution is an operation that will transform the current solution to another feasible solution.

Give a solution in the solution space, a *neighbor* of the solution is another solution in the solution space that can be reached through the application of one of the defined moves. The *neighborhood* of a solution is made up of all neighbors of the solution. The

neighborhood structure is derived from the move design. A more flexible move set will lead to a larger solution neighborhood.

2.1.4 Local vs. Global Optimal Solutions

If a feasible solution has a cost function value smaller than those for all of its neighbors, this solution is called a *local optimal solution*. Those local optimal solutions having the smallest cost function values among all the local optimal solutions are called the *global optimal solutions*.

2.2 Dominant Solution Meta-Heuristics

For NP-hard problems, like the problems we are working on, we can only obtain optimal solutions for small problem instances. For practical problem instance sizes, heuristics must be used to find optimized solutions within reasonable time frame. A *heuristic* is an algorithm that tries to find good solutions to a problem but it cannot guarantee its success. Most heuristics are not based on rigid mathematical analyses, but on human intuitions, understanding of the properties of the problem at hand, and experiments. The value of a heuristic must be based on performance comparisons among competing heuristics. The most important performance metrics are solution quality and CPU running time.

Over the last half a century people have studied combinatorial optimization heuristics in solving many practical NP-hard problems, and some common problem-solving strategies underlying these heuristics emerged as *meta-heuristics*. A meta-heuristic is basically a pattern or main idea for a class of heuristics. Meta-heuristics represent reusable knowledge in heuristic design, and they can provide valuable starting points for us to design effective new heuristics in addressing new NP-hard problems. But meta-heuristics

are not based on theory. We should not limit ourselves to their guidelines and let them limit our own creativity. Meta-heuristics are not algorithms. To effectively solve a problem based on a meta-heuristic, we need to have deep understanding of the characteristics of the problem, and creatively design and implement the major components of the meta-heuristics. Therefore, using a meta-heuristic to propose an effective heuristic to solve an NP-hard problem is an action of research.

2.2.1 Local Optimization

Local optimization is also called *greedy algorithm* or *hill-climbing*. Starting from a random initial partition, the algorithm keeps migrate to better neighbors in the solution space. If all neighbors of the current partition are worse, then the algorithm terminates. This scheme can only find local optimal solutions that are better than all of their neighbors but they may not be the global optimal solutions. Table 2 shows the local optimization algorithm in pseudo-code.

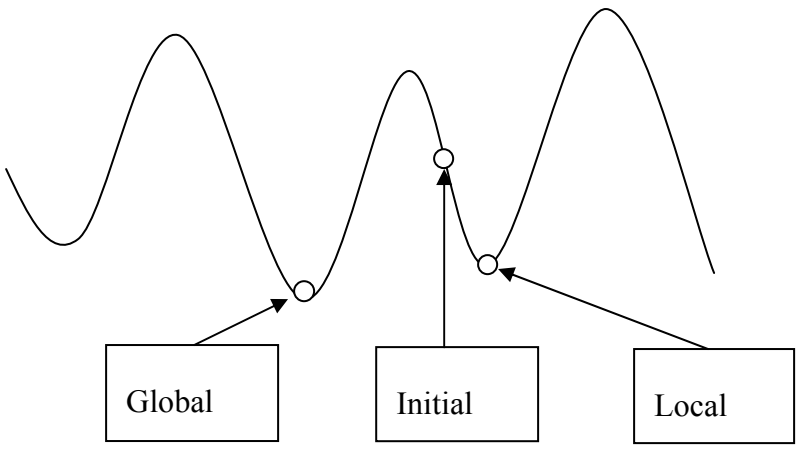

Figure 1 Local vs. Global Solutions

Table 2 Local Optimization

```
Get a random initial solution as the current solution.
While there is an untested neighbor of the current solution
    Find an untested neighbor of the current solution.
    Evaluate the neighbor's cost.
    If the neighbor's cost improves the current cost
        Let the neighbor be the new current solution.
        If the neighbor's cost improves the best one seen so far, record it.
    End If.
End While.
Return the best solution visited.
```

2.2.2 Genetic Algorithm

Genetic algorithm is based on the analogy of combinatorial optimization to the mechanics of natural selection and natural genetics. Its application in combinatorial optimization area started back in early 1960s [7].

In a genetic algorithm, a solution is represented by a coding. The algorithm starts with the generation of a pool of codings for random solutions. This pool is called generation 0. To generate the next generation, parents are randomly selected from the previous generation according to some selection criteria. Every pair of such parent codings could be randomly crossovered (mixed) to generate a new child coding in the new generation. Each such parent could also be randomly mutated (modified) to generate a new child coding. Hopefully the advantages of the parents could be combined to generate a better child, and a mutated parent could lead to unexplored area of the solution space. This generation production process will be repeated until some stopping criteria are met.

Now we can describe a generic genetic algorithm with the following pseudo-code.

Table 3 Genetic Algorithm

```
Generate codings of random solution as generation 0.
While there are improvements to the best cost seen so far in the recent generations do
    Use crossover to generate some children with randomly chosen parents.
    Use mutation to generate some children with randomly chosen parents.
    Let the new generation be made up of the best solutions we have now.
    If the best coding improves the best cost seen so far, record it.
End While.
Return the best solution visited.
```

2.2.3 Simulated Annealing

In 1983 Kirkpatrick and his coauthors proposed to use the analogy of metal annealing process to design combinatorial optimization heuristics [4].

The atoms in metal have their natural home positions. When they are away from their natural positions, they hold energy to pull them back. Metal will be in its softest state when most of its atoms are in their natural home positions and in its hardest state when most of its atoms are far away from their natural home positions. To make a sword with steel, we need first to put the steel in very high temperature so the atoms can randomly move around instead of getting stuck in some foreign positions. The natural force will pull the atoms back to their home positions. Little by little, we lower the temperature until most of the atoms are frozen in their home positions. If the temperature is lowered too fast, some atoms may get stuck in foreign positions. This is the annealing process.

Kirkpatrick viewed the combinatorial optimization process analogous to the metal annealing process, and the optimal solutions analogous to metal in its softest state. Such analogies are not logically justifiable. Basically the physical metal annealing process gave Kirkpatrick some fresh ideas on combinatorial optimization.

The simulated annealing heuristic starts with a random initial solution as its current solution and a high temperature *t*. The heuristic then goes through loop iterations for the same temperature. During each iteration, a random neighbor of the current solution is generated. If the neighbor improves the current cost, then the neighbor becomes the new current partition for the next iteration. If the neighbor worsens the current cost, it will be accepted as the new current solution with a probability. When the temperature is high, the probability is not sensitive to how bad the neighbor is. But when the temperature is low, the probability to accept a worsening neighbor will shrink with the extent of the worsening. When no improvement in solution cost happens for a while, the temperature will be reduced by a very small amount, and the above looping repeats. The process will terminate when some termination criteria is met.

Simulated annealing has been widely applied to solve many combinatorial optimization problems. Simulated annealing is unique among all the other meta-heuristics for combinatorial optimization in that it has been mathematically proven to converge to the global optimum if the temperature is reduced sufficiently slowly. But this theoretical result is not very interesting to practitioners since very few real world problems will be able to afford such excessive execution time.

A simulated annealing heuristic can be described by the following pseudo-code.

Table 4 Simulated Annealing

Get a random initial solution π as the current solution.
Get an initial temperature $t > 0$.
While stop criteria not met do
    Perform the following loop L times.
        Let π' be a random neighbor of π.
        Let $\Delta = \text{cost}(\pi') - \text{cost}(\pi)$.
        If $\Delta \leq 0$ (downhill move), set $\pi = \pi'$.
        If $\Delta > 0$ (uphill move), set $\pi = \pi'$ with probability $e^{-\Delta/t}$.
    Set $t = 0.95t$ (reduce temperature).
End While.
Return the best π visited.

Let us have a look of the function of Δ and t in the probability function $e^{-\Delta/t}$. Since $\Delta > 0$ and $t > 0$, $e^{-\Delta/t} = \frac{1}{e^{\Delta/t}}$. When t is much larger than Δ, Δ/t approaches 0, $e^{\Delta/t}$ approaches to 1, and $e^{-\Delta/t}$ approaches to 1. This indicates that when the temperature is high, the heuristic has very high chance to accept worsening neighbors. But when t is much smaller than Δ, Δ/t approaches ∞, $e^{\Delta/t}$ approaches to ∞, and $e^{-\Delta/t}$ approaches to 0. This indicates that when the temperature is very small, the heuristic has very small chance to accept worsening neighbors.

When we compare local optimization and simulated annealing, we find they mainly differ in whether to accept worsening neighbors. For simulated annealing, it starts with random walk in the solution space. When a random neighbor is better, it always takes it. But if the neighbor is worsening, its chance of accepting it is reduced slowly. Simulated annealing is reduced to local optimization when the temperature is very low.

2.2.4 Tabu Search

Tabu search is another meta-heuristic for combinatorial optimization becoming very active in late 1980s [5]. The proponents of tabu search disagree with the analogy of optimization process to metal annealing process. They argued that when a hunter was put in an unfamiliar environment, he will not walk randomly first but zero in to the area that appears most promising in finding games. This is similar to the greedy local optimization process. Only when neighboring areas are all worse than the current area will the hunter be willing to walk through worsening neighboring areas in hope of finding a better local optimum. To avoid being trapped in a loop in the solution space (for example, accepting a worsening neighbor, then returning back to the better starting solution right away), tabu search uses a tabu list to remember the recent moves and avoids repeating them Tabu means prohibition here.

Tabu search differs from simulated annealing in that it is more aggressive and deterministic. A tabu search heuristic starts by generating a random partition as the current partition. It then executes a loop until some stopping criteria are met. During each iteration, the current solution is replaced with its best neighbor that is not tabued on the tabu list.

Theoretically the tabu list should record the solutions visited recently so we can avoid repeating them. But in practice checking each neighbor against each of the recorded solutions will take up too much execution time. Therefore in practice we usually only record some features of the recent solutions, or some features leading to these recent solutions.

A tabu search heuristic can be described by the following pseudo-code

Table 5 Tabu Search.

Get a random initial solution π as the current solution. While stop criteria not met do Let π' be a neighbor of π minimizing $\Delta = cost(\pi') - cost(\pi)$ and not visited in the last *t* iterations. Set $\pi = \pi'$. End While. Return the best π visited.

.

Chapter 3

Abstraction and Transformation of a Class of Time-Dependent Optimization Problems

3.1 Real World Time Dependent Optimization Problems

The traditional combinatorial optimization research focuses on problems in which the cost of each decision-making step is a constant that will not change with time. For example, in the famous *Travel Salesperson Problem*, the cost for the salesperson to travel from one city to another is fixed and does not dependent on the order in which the salesperson will make this trip.

But in the real world there are many problems that do not fit into this category. For example, the cost of building a skyscraper depends on the construction season. Usually, building a skyscraper in the summer will be different from that in the winter due to different environmental working conditions. In the stock market, the sale of the same amount of stocks at different times will usually lead to different returns.

In 2001 Prof. Michael Gargano and Prof. William Edeson [1] first formally introduced the problem of finding optimal solutions for problems in which the costs are time-dependent, outlined five such problems with practical significance, and proposed their solutions based on a novel solution coding for genetic algorithms. The following are two of these five problems.

3.1.1 Minimal Cost Satellite Receiver Placement

Given a directed communication network abstracted as nodes and directed edges, satellite receivers need to be installed on selected nodes to make sure that satellite signal can reach all nodes of the network. The satellite receivers must be installed in successive months, one for each month. The price to install a satellite receiver is a function of both the node and the month involved. Find a subset of nodes and decide on the order for their satellite receiver installation so that the satellite signal can reach all the nodes and the total cost is minimized.

As an example, let us assume that we have a communication network consisting of 10 nodes, as shown in Figure 2.

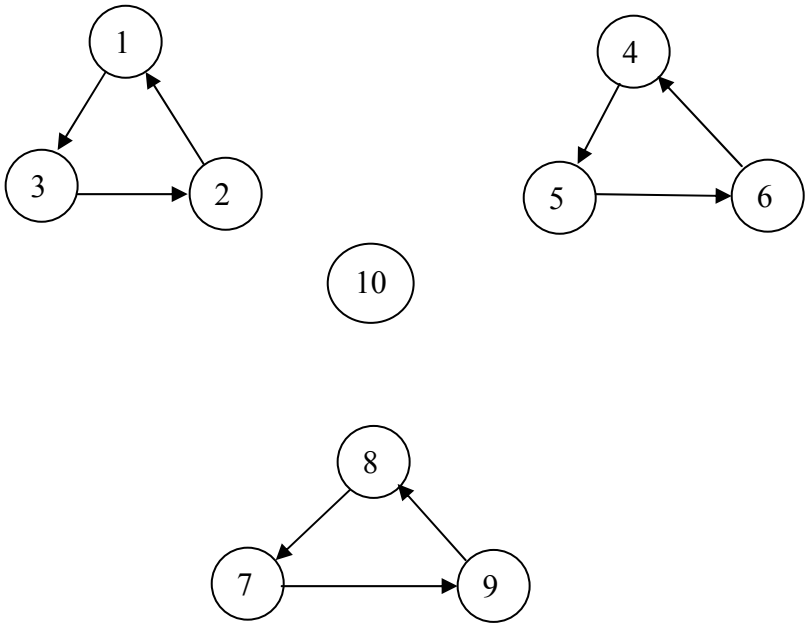

Figure 2 Example Communications Network

Table 6 shows the cost of installing a satellite receiver on a node in four different months. For example, to install a satellite receiver on node 2 in month 3 will cost 86 millions of dollars.

Table 6 Cost Table for Satellite Receiver Installation

Node	Cost (in millions of dollars)			
	Month 1	Month 2	Month 3	Month 4
1	24	45	91	38
2	48	87	86	23
3	60	34	94	71
4	50	83	89	36
5	22	51	51	91
6	46	51	14	97
7	47	26	11	93
8	77	54	34	60
9	64	38	21	35
10	67	49	35	50

The question now is how can we choose the smallest number of nodes to install satellite receivers and conduct the installations in a particular order so that the total cost for the project could be minimized?

3.1.2 Highway Minimum Bidding

There is a need to build s highway segments to connect some cities. There are n ($n \geq s$) companies bidding to build the highway segments. The segments will be built in successive months, one in each month. Each bidding company can only bid on the construction of a particular highway segment, and the cost of the bid vary with months.

Assign the bidding companies to build the highway segments in a particular order so that the total cost is minimized.

As an example, Figure 3 shows a highway network to be built. There are four highway segments involved. Due to budget limits, the four segments must be built in four consecutive months, one month for each segment, but the order of their construction is not important.

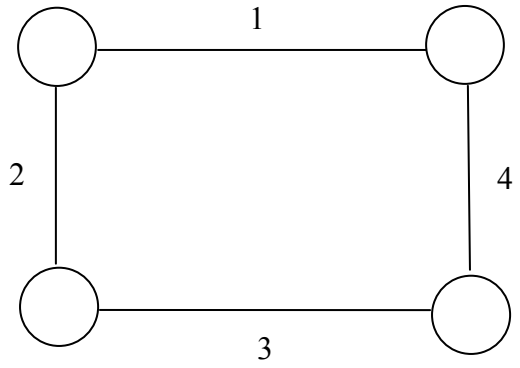

Figure 3 Example Highway Network

There are ten companies bidding on the construction of the four segments, each can only bid on a particular segment. Let us assign unique identification numbers 1, 2, …, 10 to these companies. Table 7 shows the costs for a bidding company to build a highway segment in a particular month. For example, for company 2 to build segment 1 in month 3, the bidding cost is 86. Notice that we purposely used the same cost data as in Table 6 to highlight the similarity of the two different real world problems.

Table 7 Cost Table for Highway Bidding

Highway Segment	Bidding Company	Cost (in millions of dollars)			
		Month 1	Month 2	Month 3	Month 4
1	1	24	45	91	38
	2	48	87	86	23
	3	60	34	94	71
2	4	50	83	89	36
	5	22	51	51	91
	6	46	51	14	97
3	7	47	26	11	93
	8	77	54	34	60
	9	64	38	21	35
4	10	67	49	35	50

We can generalize from these two problems that in the real world there are many similar combinatorial optimization problems whose costs are time-dependent. Efficient solutions to these problems can bring us significant economic benefits.

3.2 Abstraction of a Class of Time-Dependent Optimization Problems

Instead of trying to solve each similar problem, in this section we will abstract a class of such problems into a general problem formulation. Providing efficient solutions to such a problem formulation will enable us to solve all problems that are based on the same abstraction.

3.2.1 Problem Formulation A

Table 8 Problem Formulation A

Problem Formulation A: Let $X = \{x_1, x_2, \ldots, x_n\}$ be a set, $1 < s \le n$ an integer, $\{X_1, X_2, \ldots, X_s\}$ a partition of X, $\wp_s$ the set of all permutations on set $S = \{1, 2, \ldots, s\}$, $\Re^+$ the set of all nonnegative real numbers, and $c : X \times S \to \Re^+$ a given function. Find a vector $v = (x_1, x_2, \ldots, x_s) \in X_1 \times X_2 \times \ldots \times X_s$ and $\pi \in \wp_s$ such that the cost function

$$f(v, \pi) = \sum_{i=1}^{s} c(x_i, \pi(i))$$

is minimized.

In a generic language, this mathematical model identifies a set of *n* workers (*X*) to work on *s* tasks. Each of the *s* tasks must be completed by one worker in a particular time slot. All the *s* tasks must be completed in *s* consecutive time slots. Each worker can only work on one of the *s* tasks. The cost for a worker x to work in time slot *i* is *c*(*x*, *i*). *Problem Formulation A* tries to find one worker for each task (vector *v*) and an order π for these workers to work on the tasks such that the total cost would be minimized.

It has been proven that *Problem Formulation A* specifies a class of NP-hard problems [1].

3.2.2 Application of Problem Formulation A

In this sub-section we show how we can use *Problem Formulation A* to model both of the two problems outlined in Section 3.1. Of course *Problem Formulation A* can also be used to model many other similar real world problems.

3.2.2.1 Minimum cost satellite receiver placement

We can first find strong components of the communication network so that signal can flow between any pair of two nodes in the same strong component, and no node in a strong component can be reached from nodes of another strong component. Let s be the number of the resulting strong components. Obviously we only need to install exactly s satellite receivers, one for each strong component. The problem is now which node in each strong component will be chosen to install a satellite receiver, and the receivers will be installed in which order.

Let X be the set of all the nodes in the strong components, which is partitioned into subsets of nodes belonging to each partition. Let $c(x, i)$ be the cost of installing a receiver on node x in month i. Then a solution to *Problem Formulation A* will find the subset of nodes to install satellite receivers and the order to install the receivers so that the total cost is minimized.

Figure 4 shows the partition of the example communications network in Figure 3 into four strong components, which are represented by dotted ovals.

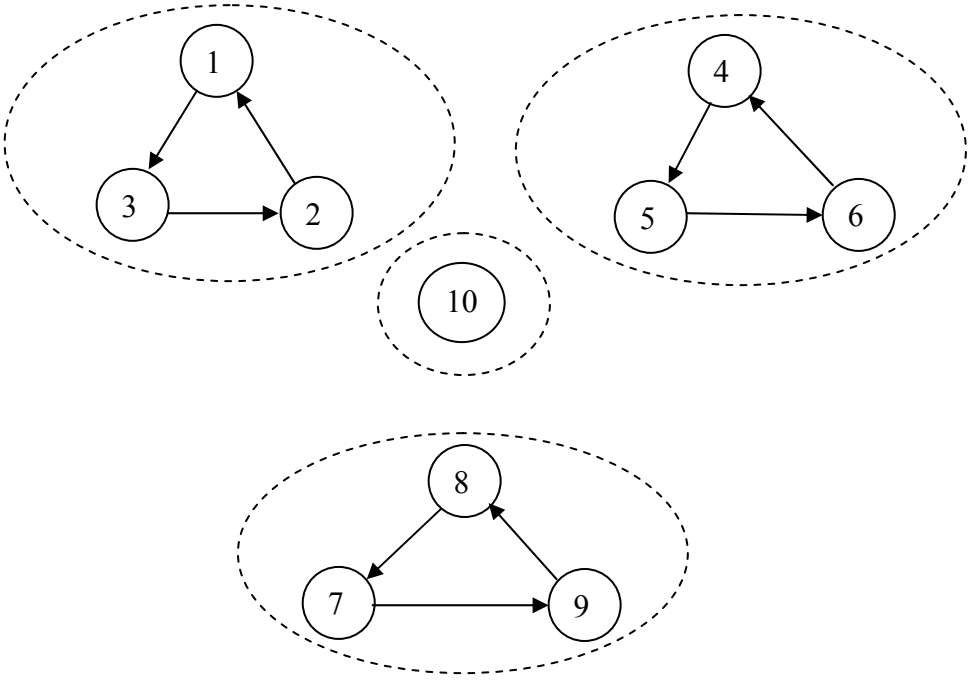

Figure 4 Strong Components of a Communications Network

3.2.2.2 Highway minimum bidding

Let X be the set of all bidding companies, and $c(x, i)$ the cost of letting company x build highway segment x in month i. Then a solution to *Problem Formulation A* will find the best subset of bidding companies to build the highway segments in a particular order so that the total cost is minimized.

3.3 Problem Transformation

In this section we first use the previous highway bidding example to observe some special properties of *Problem Formulation A*, and then propose a scheme to transform *Problem Formulation A* into an equivalent but simpler *Problem Formulation B*. Since the solution space for *Problem Formulation B* is significantly smaller than that for *Problem*

Formulation A, this transformation can benefit any solution strategies. Therefore this transformation represents a major research contribution of this dissertation.

3.3.1 Critical Observation of Problem Formulation A

The most important property of *Problem Formulation A* is the independence of the choice of a representative x_i of a subset X_i in the solution vector *v* for any $1 \le i \le s$. As a result, instead of focusing on both which workers will be chosen to complete the tasks and in which order, we only need to focus on in which order the tasks will be conducted.

We use the earlier highway bidding problem as an example, and refer to the cost data in Table 7. Suppose we decide to build the highway segments in the order of 2, 1, 3, and 4. As indicated by the shaded areas of Table 9, companies 4, 5, and 6 are bidding on segment 2 and their bidding prices for month 1 are 50, 22, and 46 respectively; companies 1, 2, and 3 are bidding on segment 1 and their bidding prices for month 2 are 45, 87, and 34 respectively; companies 7, 8, and 9 are bidding on segment 3 and their bidding prices for month 3 are 11, 34, and 21 respectively; and company 10 is bidding on segment 4 and its bidding price for month 4 is 50. Since we need to build the highway with the minimal total cost, for each segment we choose the bidding company asking for the minimal cost for the assigned month, as indicated by the darker cell in each of the shaded areas in Table 9. Obviously the decision of having company 5 to build segment 2 in month 1 has nothing to do with the decision of having company 3 to build segment 1 in month 2. Therefore, based on the decision of building the highway segments in the order of 2, 1, 3, and 4, the implied solution to the problem is to assign company 5 to build segment 2 in month 1, company 3 to build segment 1 in month 2, and company 7 to build

segment 3 in month 3, and company 10 to build segment 4 in month 4. This assignment leads to a total cost of 22 + 34 + 11 + 50 = 117. We will see later that this solution is not optimal since the segment building order is not optimal.

Table 9 Highway Segment Building Order Implies Solution

Highway Segment	Bidding Company	Cost (in millions of dollars)			
		Month 1	Month 2	Month 3	Month 4
1	1	24	45	91	38
	2	48	87	86	23
	3	60	34	94	71
2	4	50	83	89	36
	5	22	51	51	91
	6	46	51	14	97
3	7	47	26	11	93
	8	77	54	34	60
	9	64	38	21	35
4	10	67	49	35	50

This is the critical observation leading to our introduction of *Problem Formulation B* and the *Translation Theorem*. If the minimal bidding cost in a shaded area has multiple occurrences, the company assignment will not be unique, but the minimal total cost is still unique.

3.3.2 Simplified Problem Formulation B

Based on our critical observation on Problem Formulation A, we introduce the following simplified *Problem Formulation B*.

Table 10 Problem Formulation B

Problem Formulation B: Let $c' : S \times S \to \Re^+$ be a function such that for $i, j \in S$, $c'(i, j) = \min_{x \in X_i} c(x, j)$ (if such x is not unique, choose the one with the smallest index in set X). Find a $\pi \in \wp_s$ such that the cost function

$$f'(\pi) = \sum_{i=1}^{s} c'(i, \pi(i))$$

is minimized. A corresponding vector $v = (x_1, x_2, \ldots, x_s) \in X_1 \times X_2 \times \ldots \times X_s$ can be derived from π by function $r : \wp_s \to X_1 \times X_2 \times \ldots X_s$ such that for any $\pi \in \wp_s$, $r(\pi) = (x_1, x_2, \ldots, x_s)$, and for all $i \in S$, $x_i \in X_i$, and $c(x_i, \pi(i)) = c'(i, \pi(i))$ (if such x is not unique, choose the one with the smallest index in set X).

In a generic language, this mathematical model identifies a set of *n* workers (*X*) to work on *s* tasks. Each of the *s* tasks must be completed by one worker in a particular time slot. All the *s* tasks must be completed in *s* consecutive time slots. Each worker can only work on one of the *s* tasks. The cost for a worker x to work in time slot *i* is *c*(*x*, *i*). For any $1 \le i, j \le s$, where *i* represents a task and *j* represents a time slot, $c'(i, j)$ is the minimal cost for completing task *i* in time slot *j*. *Problem Formulation B* tries to find the best order to complete the *s* tasks to minimize the total cost, and it also provides an algorithm to derive from such an order the best task assignment to the workers.

3.3.3 Problem Transformation Theorem

The following Transformation Theorem explains that, given any problem instance in the form of *Problem Formulation A*, we can solve it by first solving its simpler version in *Problem Formulation B*.

Table 11 Problem Transformation Theorem

Problem Transformation Theorem: Given any set $X = \{x_1, x_2, \ldots, x_n\}$ and function $c : X \times S \rightarrow \Re^+$. Any solution $\pi \in \wp_s$ to Problem *Formulation B* and its corresponding vector $v = r(\pi)$ provides a solution to the corresponding *Problem Formulation A*.

Proof: Problem A seeks v and π that makes

$$\min_{\substack{v=(x_1,x_2,\ldots,x_s)\in X_1\times X_2\times\ldots\times X_s \\ \pi\in\wp_s}} \sum_{i=1}^{s} c(x_i, \pi(i))$$

true, which is equivalent to

$$\min_{\pi\in\wp_s} \sum_{i=1}^{s} \min_{x\in X_i} c(x, \pi(i))$$

or

$$\min_{\pi\in\wp_s} \sum_{i=1}^{s} c'(i, \pi(i))$$

which is exactly the specification for Problem B. □

3.3.4 Transformation Example

Given a highway bidding problem in *Problem Formulation A* with cost data in Table 7 on page 21, we can derive the following simpler cost table for function $c'(i, j)$ shown in Table 12.

Table 12 Simplified Cost Table for Highway Bidding

Highway Segment	Cost (in millions of dollars)			
	Month 1	Month 2	Month 3	Month 4
1	24	35	86	23
2	22	51	14	36
3	47	26	11	35
4	67	49	35	50

Based on *Problem Formulation B*, the solution space for this problem instance is made up of all permutations of numbers in {1, 2, 3, 4}, and the solution space has exactly $4! = 4 \times 3 \times 2 \times 1 = 24$ feasible solutions. We can easily enumerate all these 24 feasible solutions and find (2, 4, 3, 1) is the optimal solution to *the Problem Formulation B* version of the problem instance at hand, and the cost of this optimal solution is 22 + 49 + 11 + 23 = 105. The physical meaning of this solution means that to minimize the total project cost, we should build highway segment 2 in month 1, build highway segment 4 in month 2, build highway segment 3 in month 3, and build highway segment 1 in month 4.

Now we can recover the corresponding optimal bidding company assignment by referring to Table 7 on page 21. Companies 4, 5, and 6 bid for building highway segment 2 in month 1 for costs 50, 22, and 46 respectively. Since 22 is the minimal cost here, company 5 should be assigned to build highway segment 2 in month 1. Company 10 is the only company bidding for building highway segment 4 in month 2 and it asks for a cost of 49.

Therefore company 10 should be assigned to build highway segment 4 in month 2. Companies 7, 8, and 9 bid for building highway segment 3 in month 3 for costs 11, 34, and 21 respectively. Since 11 is the minimal cost here, company 7 should be assigned to build highway segment 3 in month 3. Companies 1, 2, and 3 bid for building highway segment 1 in month 4 for costs 38, 23, and 71 respectively. Since 23 is the minimal cost here, company 2 should be assigned to build highway segment 1 in month 4. Again the total cost based on Table 7 is the same 22 + 49 + 11 + 23 = 105.

3.4 Benefit of Problem Transformation

3.4.1 Exhaustive Search

Given a *Problem Formulation A* model with *n* workers and *s* tasks, each solution is made up of an element of $X_1 \times X_2 \times \cdots X_s$ and a permutation π of numbers in {1, 2, …, *s*}. The former has $|X_1| \cdot |X_2| \cdot \cdots \cdot |X_s|$ possibilities. The latter has $s! = s \times (s-1) \times \ldots \times 2 \times 1$ possibilities. Therefore, the solution space for *Problem Formulation A* has a size of $|X_1| \cdot |X_2| \cdot \cdots \cdot |X_s| \cdot s!$.

On the another hand, for the equivalent *Problem Formulation B*, each solution is made up of a permutation of *s* numbers in {1, 2, …, *s*}, so the corresponding solution space has a size of *s*!.

Therefore, if we conduct exhaustive search to find an optimal solution, we can expect a speedup of $\frac{|X_1| \cdot |X_2| \cdot \cdots \cdot |X_s| \cdot s!}{s!} = |X_1| \cdot |X_2| \cdot \cdots \cdot |X_s|$ if we adopt *Problem Formulation B* instead of *Problem Formulation A*.

3.4.2 Heuristic Solution Searches

As explained in Chapter 2, most heurist algorithms are based on searching a selected subset of a solution space to find an optimized solution. They mainly differ in which subset to look into. Without knowledge on the structure of a solution space, we have to assume each area of the same size has similar chance of containing the optimal solution. Under this assumption, if we adopt *Problem Formulation B*, the chance for a heuristic search algorithm to find an optimal solution is $|X_1| \cdot |X_2| \cdot \cdots \cdot |X_s|$ times of that based on the *Problem Formulation A* version of the same problem instance.

3.5 Common Foundations for Solution Searches

3.5.1 Solution Space Neighborhood Design

Most algorithms for solving NP-hard problems are based on a loop and a current solution. During each iteration of the loop, we perturb the current solution a little to get a neighbor of it, and decide whether to accept the neighbor as the new current solution based on some criteria. Therefore it is important to design a solution neighborhood for a current solution.

First we need to decide what moves (perturbations) are suitable for the problem solutions at hand. The moves are problem specific. There are some general guidelines for choosing moves for a problem:

- *The reachability property*. The moves should allow the algorithm to visit any feasible solution in the solution space, through a series of steps, starting with any current solution

- *The incrementally updateable property*. Ideally, the moves should support the incremental update of the objective function. Evaluating the objective function is in general a time consuming process. Suppose we know the cost of the current solution. After we apply a local move or perturbation to the current solution, hopefully we can derive the cost of the resulting solution by some simple modifications to the current cost, instead of evaluating the objective function entirely. Such incremental cost update can benefit any solution techniques, and is one of the key factors for success.

For *Problem Formulation B*, each solution is a permutation of numbers in {1, 2, …, *s*}. We choose to use swapping two values in the current permutation as our only move. It is easy to see that this swapping enjoys the reachability property. In the next subsection we will show that this move also supports incremental cost updates.

Give a solution in the solution space, a *neighbor* of the solution is another solution in the solution space that can be reached through the application of one of the defined moves. The *neighborhood* of a solution is made up of all neighbors of the solution. The neighborhood structure is derived from the move design. A more flexible move set will lead to a larger solution neighborhood.

3.5.2 Incremental Cost Update

Most combinatorial optimization heuristics are based on a current solution and iterations. During each iteration, the current solution will be perturbed to obtain a neighboring solution, and the cost of this neighbor will decide the proper action of the heuristic. Since this cost evaluation has important impacts in heuristic running time, we hope to

incrementally update the old cost for the current solution to obtain the new cost for the neighboring solution. The following *Incremental Cost Update Theorem* tells us this is possible and how to do it.

Table 13 Incremental Cost Update Theorem

Incremental Cost Update Theorem: For *Problem Formulation B*, given a solution (permutation) $\pi \in \wp_s$ and its corresponding cost $f'(\pi)$, applying a perturbation of swapping values at positions i and j in solution π, where $1 \le i < j \le s$, will result in a neighboring solution π' with cost $f'(\pi')$ satisfying the following relationship

$$f'(\pi') = f(\pi) - c'(i, \pi(i)) - c'(j, \pi(j)) + c'(i, \pi(j)) + c'(j, \pi(i)).$$

Proof: Based on our *Problem Formulation B*, given any permutation π on set {1, 2, …, s}, the cost function $f'(\pi)$ is defined as

$$f'(\pi) = \sum_{i=1}^{s} c'(i, \pi(i)) = c'(i, \pi(i)) + c'(j, \pi(j)) + \sum_{\substack{1 \le k \le s \\ k \ne i, k \ne j}} c'(k, \pi(k))$$

After swapping values at positions i and j, we effectively swapped values of $\pi(i)$ and $\pi(j)$. Therefore the only terms that need updating are $c'(i, \pi(i))$ and $c'(j, \pi(j))$, which should be replaced by values of $c'(i, \pi(j))$ and $c'(j, \pi(i))$ respectively. □

For example, let π = (2, 1, 4, 3) be the current solution for the example problem instance in Table 12 on page 29. We can check that $f'(\pi) = c'(1, 2) + c'(2, 1) + c'(3, 4) + c'(4, 3)$ =

35 + 22 + 35 + 35 = 127. If we swap the values in positions 2 and 3 in π, the resulting solution will be π' = (2, 4, 1, 3), and its cost will be $f'(\pi') = c'(1, 2) + c'(2, 4) + c'(3, 1) + c'(4, 3)$ = 35 + 36 + 37 + 35 = 143. But we can also obtain this new cost by applying incremental update

$$f'(\pi') = f'(\pi) - c'(i, \pi(i)) - c'(j, \pi(j)) + c'(i, \pi(j)) + c'(j, \pi(i)) \quad \text{or}$$

$$f'(\pi') = 127 - c'(2, 1) - c'(3, 4) + c'(2, 4) + c'(3, 1) = 127 - 22 - 35 + 36 + 47 = 143.$$

The time complexity of evaluating the cost function $f'(\pi')$ is $O(s)$ or linear function of *s*, but the time complexity for our incremental cost update is $O(1)$ or constant. When *s* is large, the benefit of our incremental update can be significant.

Chapter 4

Reference Algorithms

To evaluate the relative performance of our simulated annealing algorithm, we design and implement several reference algorithms including exhaustive search, repeated random solutions, and a genetic algorithm. This chapter describes the design of these algorithms.

4.1 Enumeration of All Combinations

For the implementation of exhaustive search, we need to systematically enumerating all permutations of elements of set {1, 2, …, *s*}. Our implementation is based on Jeffrey A. Johnson's algorithm [10], which is outlined below. This algorithm is efficient, and it takes no extra auxiliary memory space, except for a few constant number of integer variables, for computing successive permutations in lexicographic order.

Table 14 Return Next Permutation in Lexicographic Order

1. Let p[] hold the current permutation.
2. Find the *key*, the first index from right that points to a value smaller than its right neighbor.
3. If no such key can be found, p[] is reversely sorted and represents the last permutation in lexicographic order, return.
4. Find *newKey* so p[newKey] is the smallest value to the right of p[key] that is larger than p[key].
5. Swap p[key] and p[newKey]; p[newKey] is now the new key.
6. Reverse the values to the right of the key.
7. Return p[] holding the next permutation in lexicographic order.

4.2 Design and Implementation of Exhaustive Search

For small problem instances of an NP-hard problem, we may still want to design algorithms to find optimal solutions. The optimal solutions found by such algorithms can serve as a comparison reference point for evaluating the effectiveness of heuristics.

The basic approach for finding optimal solutions is *exhaustive search*. This is a brute force approach. We systematically enumerate all feasible solutions, evaluate the objective function value (cost) for each of them, and report the ones with the minimal (maximal) costs.

The more advanced approach for finding optimal solutions is called *branch-and-bound*. It is similar to exhaustive search. It incrementally constructs all feasible solutions. It uses a *bound function* to find the best cost that may be produced from a partial solution. When the bound function finds that a partial solution cannot produce any solutions better than the best solution seen so far, that partial solution will not be further explored, thus saving

time in generating and evaluating all solutions based on that partial solution. The bound function is problem specific and it is the key for the performance improvement of branch-and-bound over exhaustive search.

For the problems in this research, our exhaustive search algorithm is based on the following pseudo-code.

Table 15 Exhaustive Search for Time-Dependent Problems

```
Let π = (1, 2, …, s) be the first permutation.
do
    Let cost = f'(π).
    If cost improves the best one seen so far, record it.
    Let π be the next permutation in lexicographic order.
While π still has next permutation
Return the best partition visited.
```

Our experiments show that, on a computer with a 1.5 GH CPU, this exhaustive search algorithm can solve our time-dependent problem instances efficiently up to around s = 15.

4.3 Design and Implementation of Repeated Random Solutions

For larger problem instances, the use of exhaustive search cannot produce optimal solutions as reference points for performance evaluation. However, we should expect any reasonable heuristic to perform better than randomly generated solutions since randomly generated solutions are not exploring any property or structure of the problem; it is mindless.

One possible basic performance evaluation for a heuristic could be repeatedly generating random solutions for as long as the heuristic does, and see which produces better solutions in the same amount of time.

Our repeated random solution algorithm is based on the following pseudo-code. The running time of the algorithm depends on the value of parameter *L*.

Table 16 Repeated Random Solutions for Time-Dependent Problems

```
Repeat L times
    Generate a random solution π..
    Evaluate its cost f'(π).
    If the cost improves the best one seen so far, record it.
End Repeat.
Return the best solution visited.
```

4.4 Implementation and Enhancement of Joseph DeCicco's Genetic Algorithm

Table 17 shows the outline of Joseph DeCicco's genetic algorithm [2] to the highway bidding problem based on *Problem Formulation A*. Since the original implementation has no proper documentation and was not properly organized and designed for the best running time, it was re-implemented in this research with special attention to its efficiency in running time. Each solution has both the *v* and π components of a solution to the *Problem Formulation A*. All parameter values, solution coding, algorithms for selection, crossover, and mutation as well as other algorithm details, not implementation details, strictly follow the original design in [2].

Table 17 Joseph DeCicco's Genetic Algorithm

```
Generate 50 codings of random solutions, and sort them in the generation table
according to their costs.
While there are improvements to the best cost seen so far in the last 50 iterations do
    Use crossover to generate 40 children with randomly chosen parents,
        and insert them into the generation table according to their costs.
    Use mutation to generate 10 children with randomly chosen parents,
        and insert them into the generation table according to their costs.
    The generation table only keeps the best 50 codings.
    If the best coding improves the best cost seen so far, record it.
End While.
Return the best partition visited.
```

In the original genetic algorithm described by Joseph DeCicco [2], the generation table allows duplicate codings. Experiments show that after a few generations the generation table may be filled up with many duplicate codings, thus significantly reducing the diversifying process of the algorithm. We use a Boolean variable to control the uniqueness of the generation table. For making this algorithm more competitive, we enforced a uniqueness of codings in the generation table when we make performance comparisons in Chapter 6.

Chapter 5

Simulated Annealing: Algorithm Design and Sensitivity Analysis

In this chapter we describe the design and implementation of our simulated annealing algorithm for time-dependent problems based on the simulated annealing meta-heuristic. We will also design experiments to conduct sensitivity analysis of the heuristic to its various parameter values.

5.1 Simulated Annealing Algorithm

The simulated annealing meta-heuristic described by David S. Johnson [3] will be used as the base of our solution to the class of time-dependent combinatorial problems that can be modeled with *Problem Formulation B*. Given a positive integer *s* for the number of tasks that need be conducted, we use $\wp_s$, the set of all permutations on set {1, 2, …, *s*}, as the solution space. Given a current solution $\pi = (x_1, x_2, \cdots, x_i, \cdots, x_j, \cdots, x_s)$ where $x_k \in \wp_s$ for all $k \in$ {1. 2. …, *s*} and $1 \le i < j \le s$,, we define a move on it is to swap x_i and x_j that lead to a neighboring solution $\pi' = (x_1, x_2, \cdots, x_j, \cdots, x_i, \cdots, x_s)$. A random neighbor of π is obtained if the values of *i* and *j* are randomly chosen in the range 1 to *s*. The algorithm is described in Table 18.

Table 18 Simulated Annealing for Time-Dependent Problems

Get a random initial solution π as the current solution.
Let temperature $t = t_0$, the initial temperature.
While there are improvements of the best cost in the last k iterations do
    While there are improvements of the best cost in the last l iterations do
        Perform the following loop l times.
            Let π' be a random neighbor of π.
            Let $\Delta = f'(\pi') - f'(\pi)$.
            If $\Delta \leq 0$ (downhill move), set $\pi = \pi'$.
            If $\Delta > 0$ (uphill move), set $\pi = \pi'$ with probability $e^{-\Delta/t}$.
    End While.
    Set $t = r \cdot t$ (reduce temperature).
End While.
Return the best π visited.

There are four parameters that we need to configure:

1. Initial temperature t_0. A too large value for t_0 will lead to wasted random walk in the solution space at the beginning of the algorithm execution thus prolong the algorithm's running time without the benefits of improving the solution qualities. A too small value for t_0 will let the algorithm get stuck in a local optimum.

2. Temperature reduction ratio r. Ratio r should be a real number between 0.0 and 1.0. If it is too small, the temperature will be lowed very slowly, leading to prolonged algorithm execution. On the other hand, if r is too large, the temperature will be reduced too fast and the current solution can get stuck in a local optimum.

3. Number l of consecutive non-improvement iterations before the temperature is reduced. If l is too large, the algorithm may waste execution time in a prolonged

non-aggressive solution search. If *l* is too small, then the current solution may not have a chance to settle down to a stable good solution.

4. Number *k* of consecutive non-improvement iterations before the algorithm is terminated. If *k* is too large, execution may be extended without quality benefit. If *k* is too small, then the algorithm may terminate too soon before better solutions could be obtained.

These four parameters are not independent. As a matter of fact, they have close inter-dependence. It is a big challenge to find optimized values for them so that the resulting algorithm can perform well on a large set of potential problem instances.

5.2 Experiment Design on Parameter Sensitivity Analysis

In Section 6.1 we describe a complete set of 70 benchmark problem instances used for performance evaluation across various solution algorithms. These problem instances have values of *s* ranging from 4 to 200. In this chapter, we choose the following seven files from that benchmark set for simulated annealing parameter sensitivity analysis. The best costs with an asterisk * have been proven optimal.

Table 19 Simulated Annealing Parameter Training Data Set

File Name	*s*	*n*	Best Cost
Data4c0	4	10	105*
Data10c0	10	37	110*
Data15c0	15	60	176
Data20c0	20	94	213
Data50c0	50	196	518
Data100c0	100	423	1007

Data200c0	200	835	2001

The objective of the remainder of this chapter is to find a single set of values for the four parameters of our simulated annealing algorithm so that it can perform competitively across all the 70 benchmark problem instances. Basically we are deriving general knowledge from the seven training problem instances, and then apply the knowledge to a ten-time large problem instance set to verify whether our derived knowledge is general and reusable.

We run the SA algorithm with various parameter settings on a Pentium III PC with a 1.5 GH CPU and a 512 MB main memory.

5.3 Parameter Sensitivity Analysis

We start with test runs of the SA algorithm on the seven training problem instances. We decided the effective ranges for the four parameters are as follows:

Table 20 Simulated Annealing Parameter Value Ranges

t_0	r	l	k
5, 10, 15, 20, 25, 30, 35, 40, 45, 50	0.9, 0.95, 0.99, 0.995, 0.9995	100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1100, 1200, 1300, 1400, 1500, 1600, 1700, 1800, 1900, 2000	20, 40, 60, 80, 100, 120, 140, 160, 180, 200, 220, 240, 260, 280, 300, 320, 340, 360, 380, 400

There are a total of 20,000 possible combinations for the values of these parameters. A driver was designed and implemented to use each of the parameter value combinations to run each of the seven training problem instances. A total of continuous 55 CPU hours were used to generate the best cost and running time for each of the seven problem

instances with each of the parameter value combinations. Table 21 shows for each training problem instance the best ten parameter combinations.

Table 21 Sample Data for SA Parameter Tuning

File Name	Best Cost	Time (ms)	t_0	r	l	k
Data4c0	105	0	5	0.9	100	20
Data4c0	105	0	5	0.99	100	40
Data4c0	105	0	5	0.995	100	20
Data4c0	105	0	5	0.9995	100	20
Data4c0	105	0	10	0.9	100	20
Data4c0	105	0	10	0.95	200	20
Data4c0	105	0	10	0.99	100	20
Data4c0	105	0	10	0.995	100	20
Data4c0	105	0	10	0.9995	100	20
Data4c0	105	0	15	0.9	100	40
Data10c0	110	0	15	0.9	100	20
Data10c0	110	0	15	0.9995	100	40
data10c0	110	0	20	0.95	100	20
Data10c0	110	0	20	0.995	100	20
Data10c0	110	0	20	0.995	100	40
Data10c0	110	0	35	0.9	100	20
Data10c0	110	0	40	0.9	100	20
Data10c0	110	0	40	0.99	100	20
Data10c0	110	0	40	0.9995	100	20
Data10c0	110	0	50	0.95	100	20
Data15c0	176	10	5	0.9995	100	20
Data15c0	176	10	5	0.9995	200	20
Data15c0	176	10	10	0.9	100	60
Data15c0	176	10	10	0.995	100	40
Data15c0	176	10	10	0.9995	100	20
Data15c0	176	10	15	0.9	100	40
Data15c0	176	10	15	0.95	100	60
Data15c0	176	10	15	0.9995	100	40
Data15c0	176	10	25	0.95	100	40
Data15c0	176	10	25	0.9995	100	40
Data20c0	213	20	30	0.9995	300	20
Data20c0	213	40	5	0.9995	1000	20
Data20c0	213	40	25	0.9995	100	180
Data20c0	213	40	35	0.9995	800	20
Data20c0	213	50	10	0.995	100	200
Data20c0	213	50	15	0.9995	100	220
Data20c0	213	50	35	0.95	300	60
Data20c0	213	51	35	0.9995	100	200
Data20c0	213	60	15	0.995	1200	20

Data20c0	213	70	15	0.9995	200	160
Data50c0	519	911	35	0.9995	1200	320
Data50c0	519	1402	35	0.995	1100	340
Data50c0	520	150	35	0.9995	300	100
Data50c0	520	371	20	0.9995	800	180
Data50c0	520	981	10	0.9	700	240
Data50c0	520	1112	50	0.9995	1700	280
Data50c0	520	1682	30	0.95	2000	180
Data50c0	520	1742	40	0.95	1700	220
Data50c0	521	131	50	0.9995	1600	20
Data50c0	521	150	10	0.9995	200	240
Data100c0	1007	1452	25	0.9	1400	120
Data100c0	1009	911	35	0.99	600	100
Data100c0	1009	1372	5	0.9	600	160
Data100c0	1009	1652	45	0.95	500	320
Data100c0	1009	2053	45	0.9995	1300	260
Data100c0	1009	2493	15	0.9	1000	380
Data100c0	1009	3085	20	0.9	2000	160
Data100c0	1010	431	10	0.9995	1600	40
Data100c0	1010	601	50	0.9995	700	140
Data100c0	1010	621	40	0.95	1300	20
Data200c0	2001	2473	20	0.9995	500	400
Data200c0	2001	2674	25	0.9995	700	280
Data200c0	2001	3274	30	0.99	1000	220
Data200c0	2001	3656	50	0.9995	800	340
Data200c0	2001	3926	40	0.99	1300	160
Data200c0	2001	4106	20	0.99	1000	180
Data200c0	2001	4226	40	0.9995	2000	220
Data200c0	2001	4246	30	0.9995	1400	380
Data200c0	2001	4377	20	0.99	1400	180
Data200c0	2001	4587	45	0.995	600	300

Figure 5 through Figure 22 show the trends of solution cost and running time as functions of r and l for all the combinations of t_0 = 10, 15, 20 and k = 20, 40, 60. For each combination for t_0 and k, r varies from 0.9, 0.95, 0.99, 0.995, to 0.9995, and l varies from 100, 200, to 300.

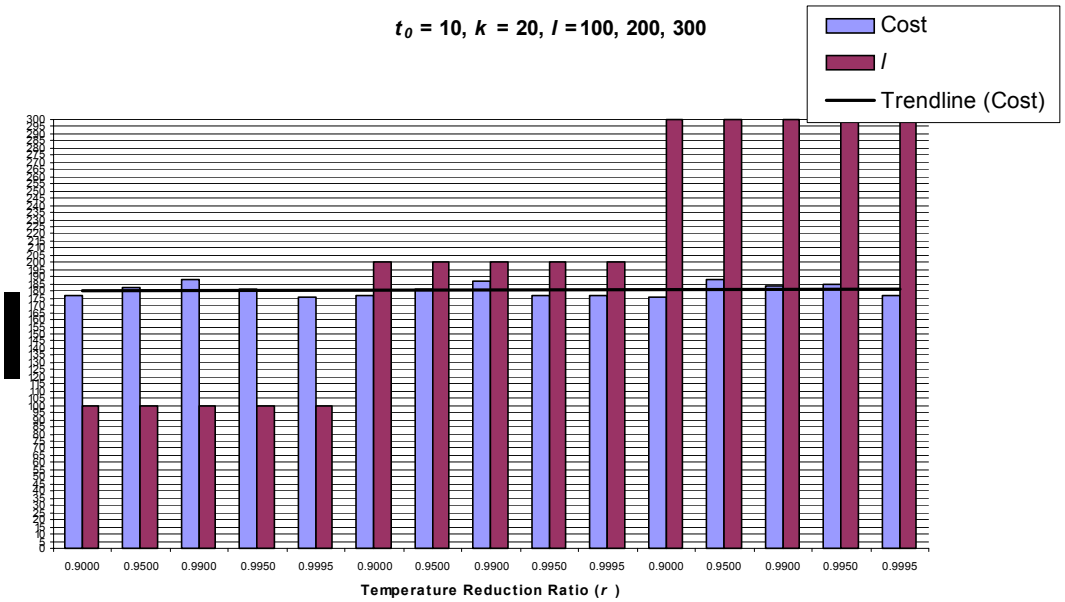

Figure 5 Cost as a Function of r and l (t_0 = 10, k = 20)

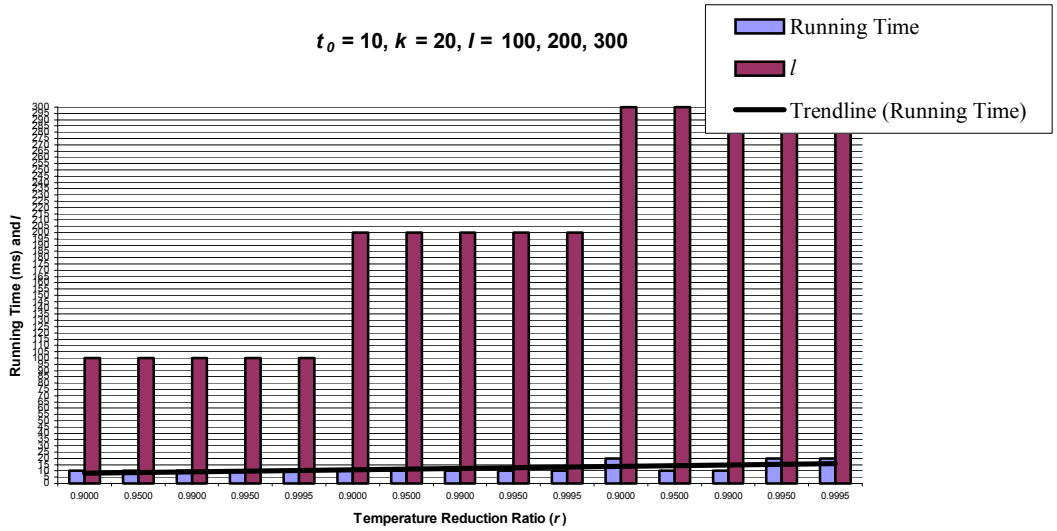

Figure 6 Running Time as a Function of r and l (t_0 = 10, k = 20)

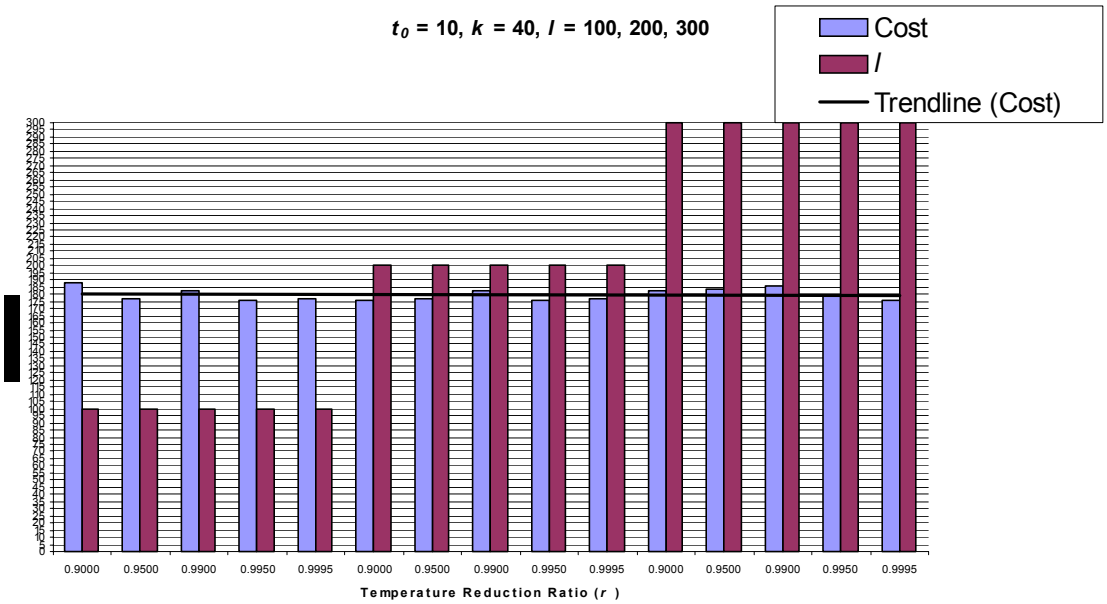

Figure 7 Cost as a Function of r and l (t_0 = 10, k = 40)

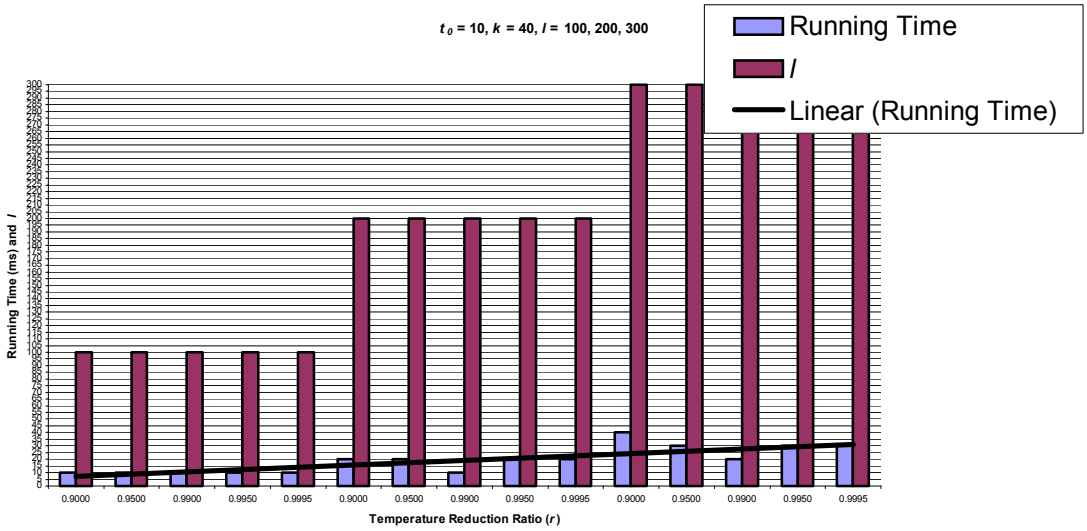

Figure 8 Running Time as a Function of r and l (t_0 = 10, k = 40)

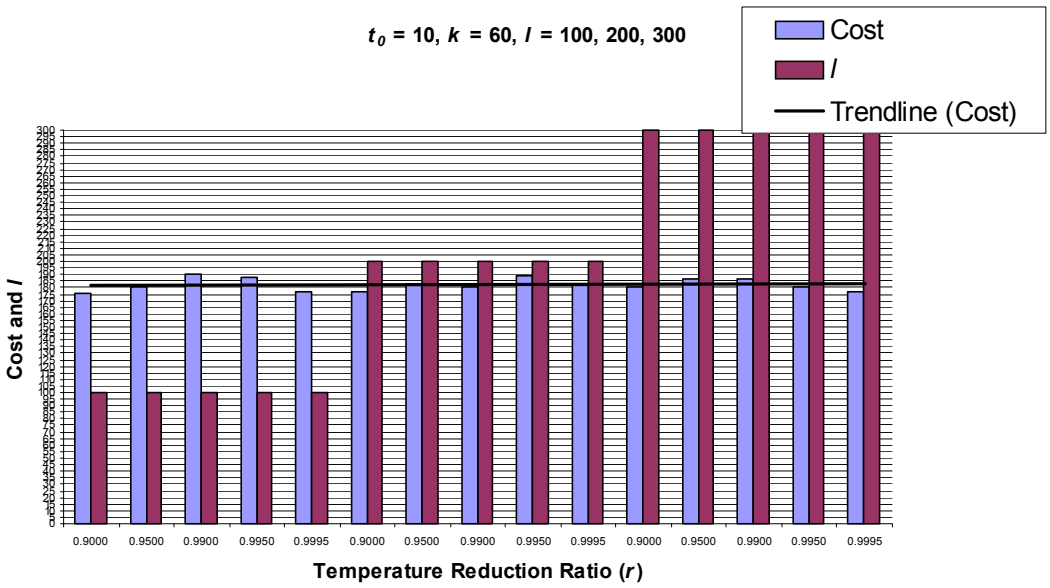

Figure 9 Cost as a Function of r and l (t_0 = 10, k = 60)

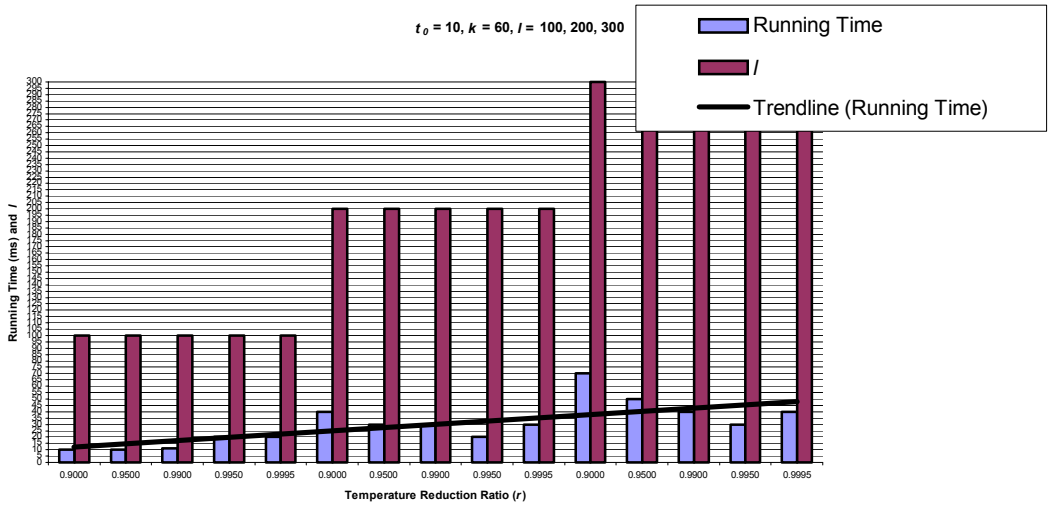

Figure 10 Running Time as a Function of r and l (t_0 = 10, k = 60)

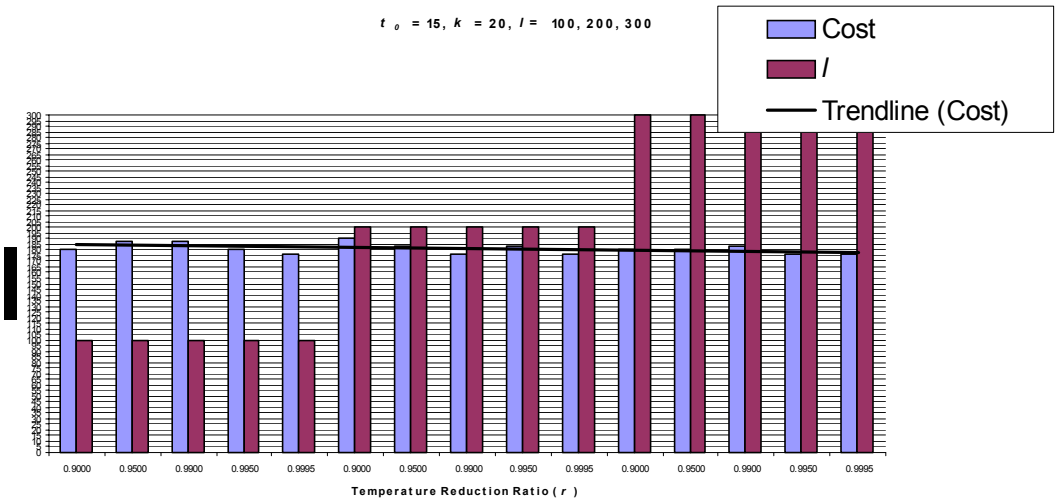

Figure 11 Cost as a Function of r and l ($t_0 = 15$, $k = 20$)

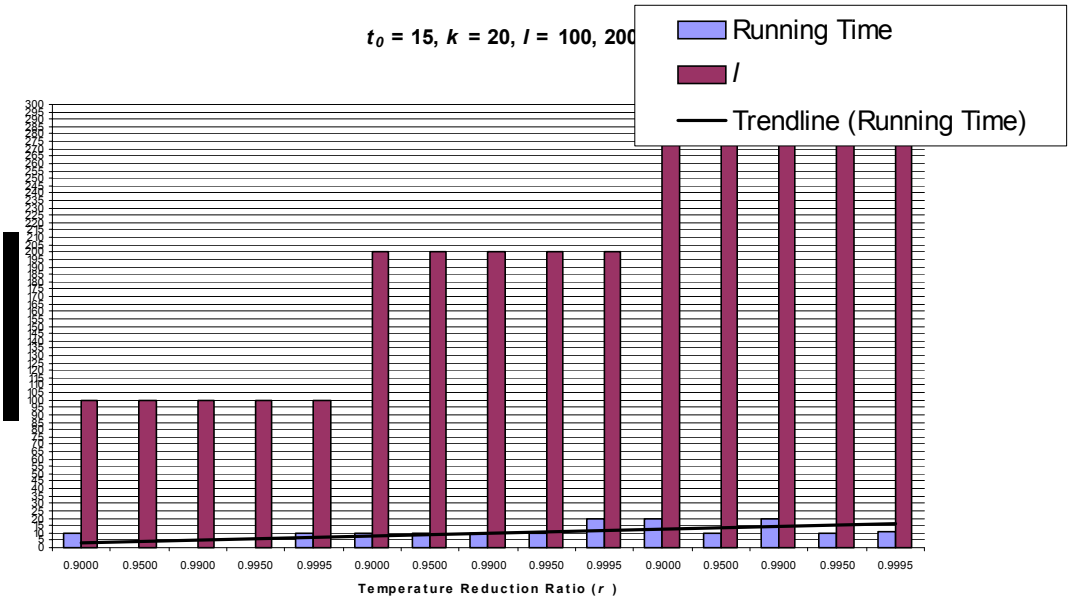

Figure 12 Running Time as a Function of r and l ($t_0 = 15$, $k = 20$)

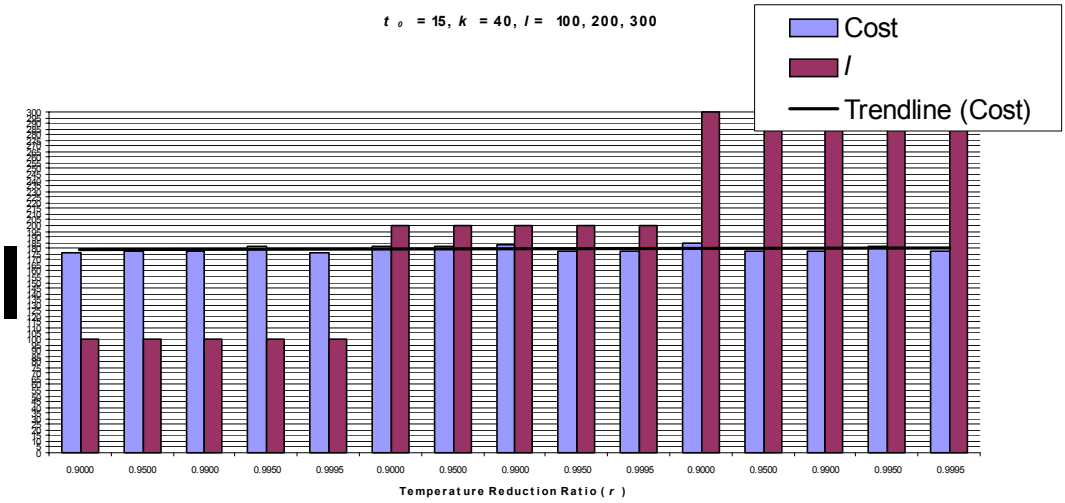

Figure 13 Cost as a Function of r and l ($t_0 = 15$, $k = 40$)

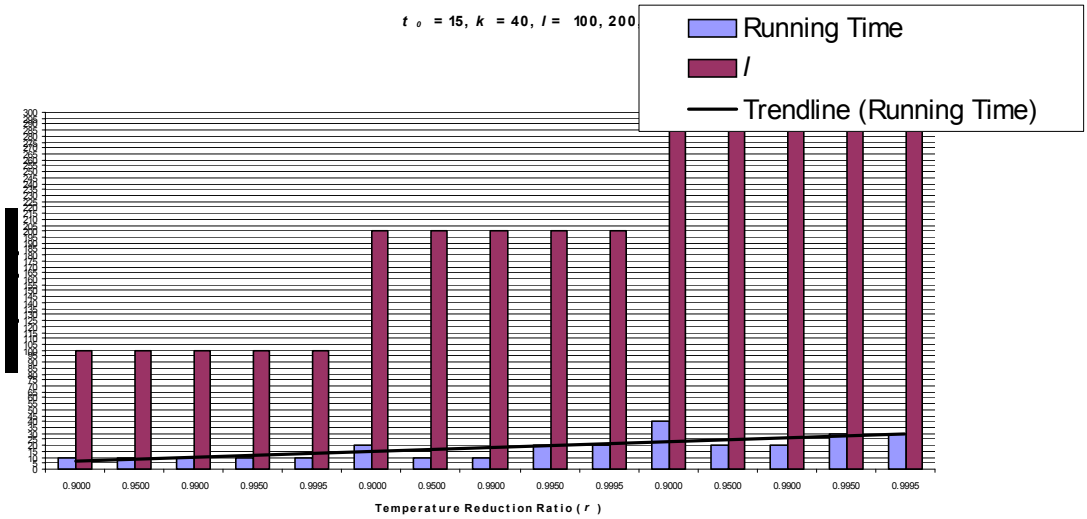

Figure 14 Running Time as a Function of r and l ($t_0 = 15$, $k = 40$)

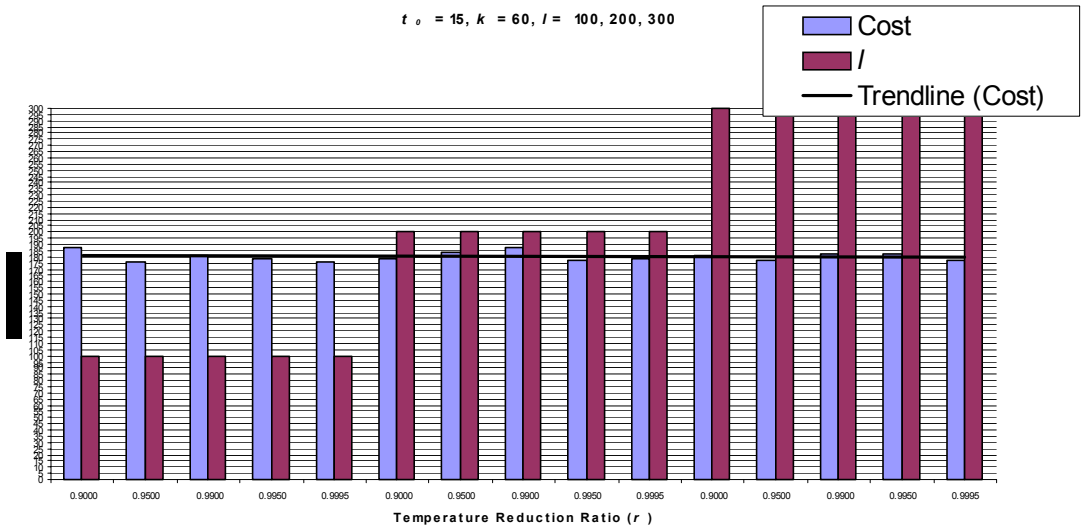

Figure 15 Cost as a Function of r and l ($t_0 = 15$, $k = 60$)

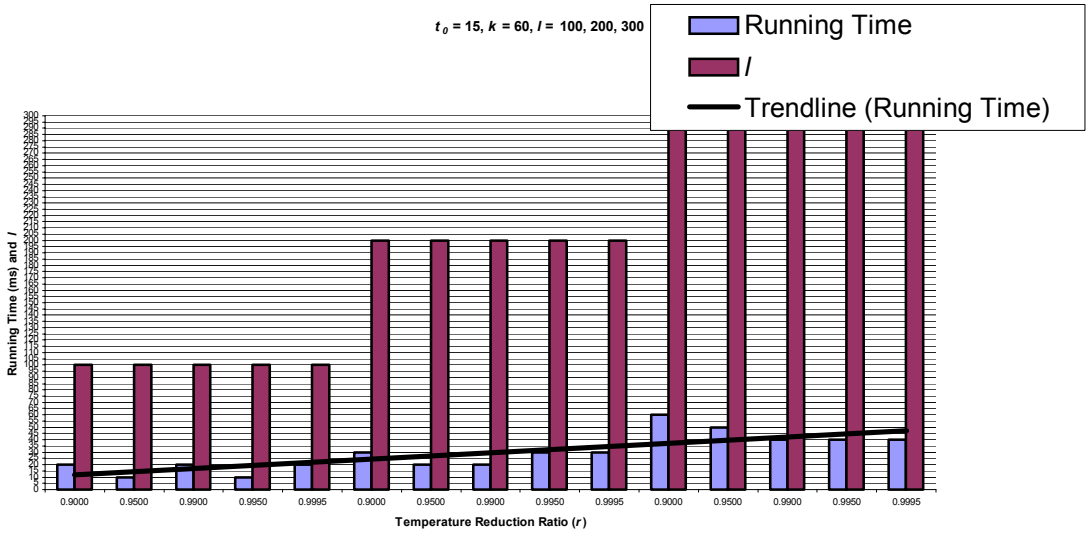

Figure 16 Running Time as a Function of r and l ($t_0 = 15$, $k = 60$)

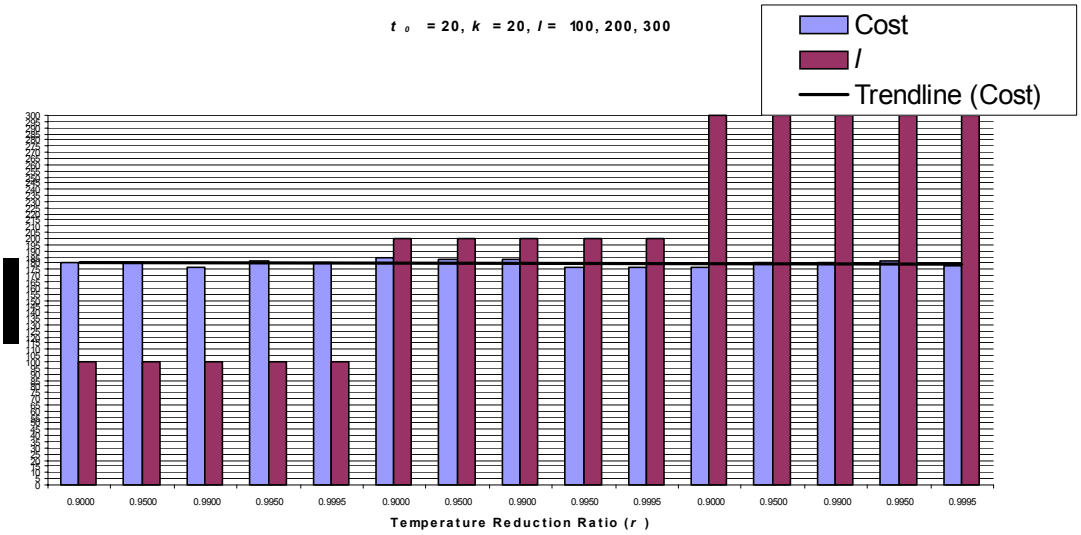

Figure 17 Cost as a Function of r and l ($t_0 = 20$, $k = 20$)

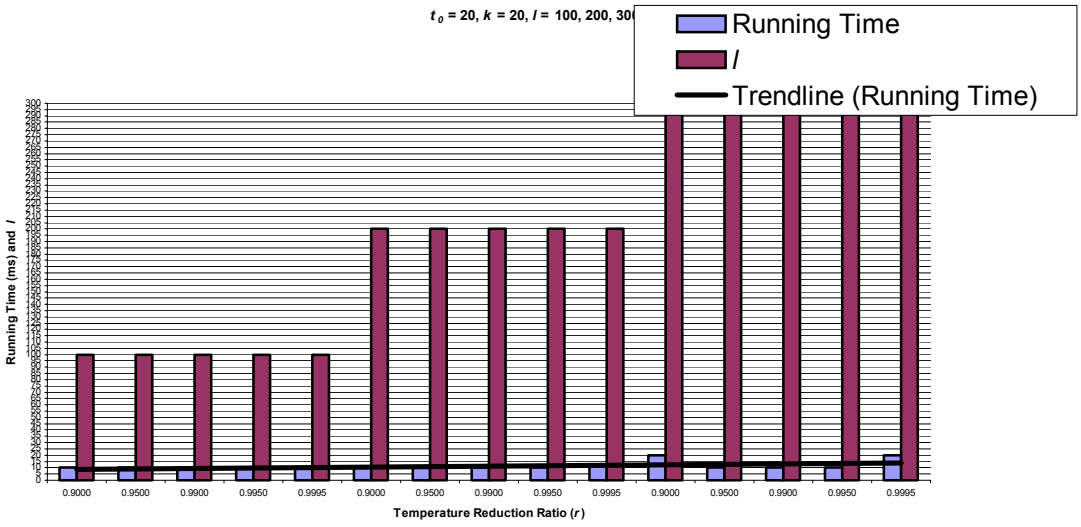

Figure 18 Running Time as a Function of r and l ($t_0 = 20$, $k = 20$)

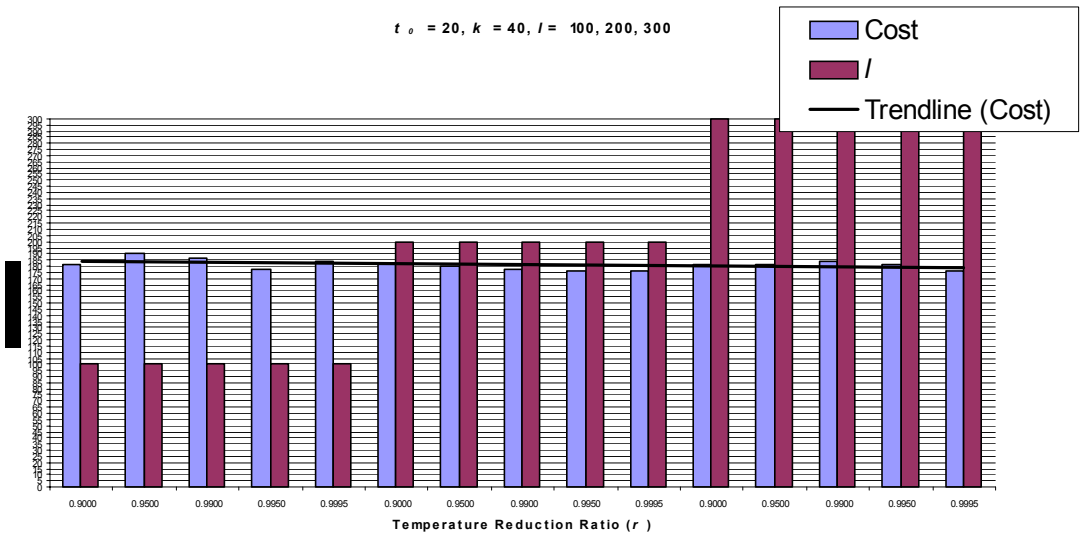

Figure 19 Cost as a Function of r and l ($t_0 = 20$, $k = 40$)

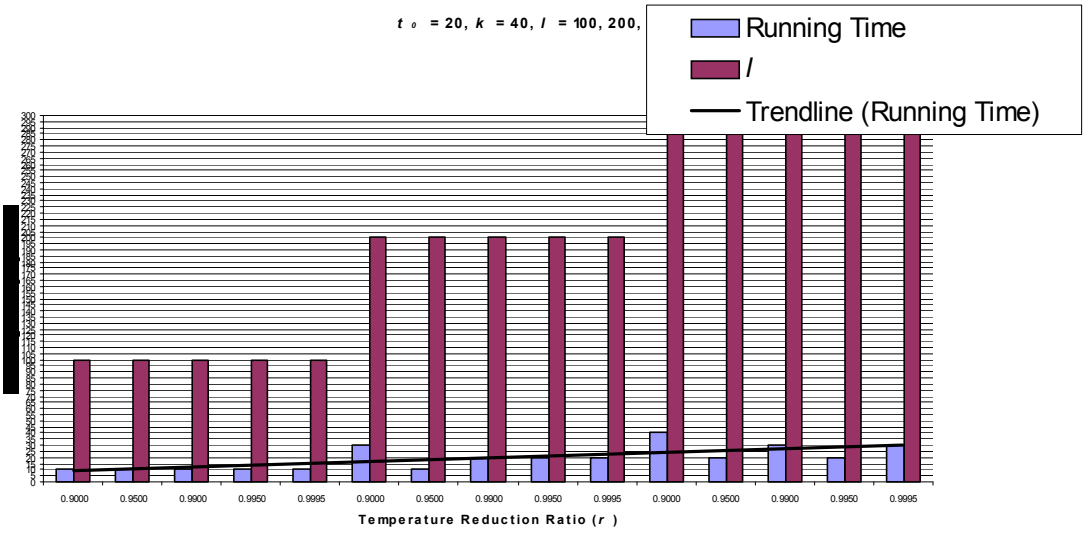

Figure 20 Running Time as a Function of r and l ($t_0 = 20$, $k = 40$)

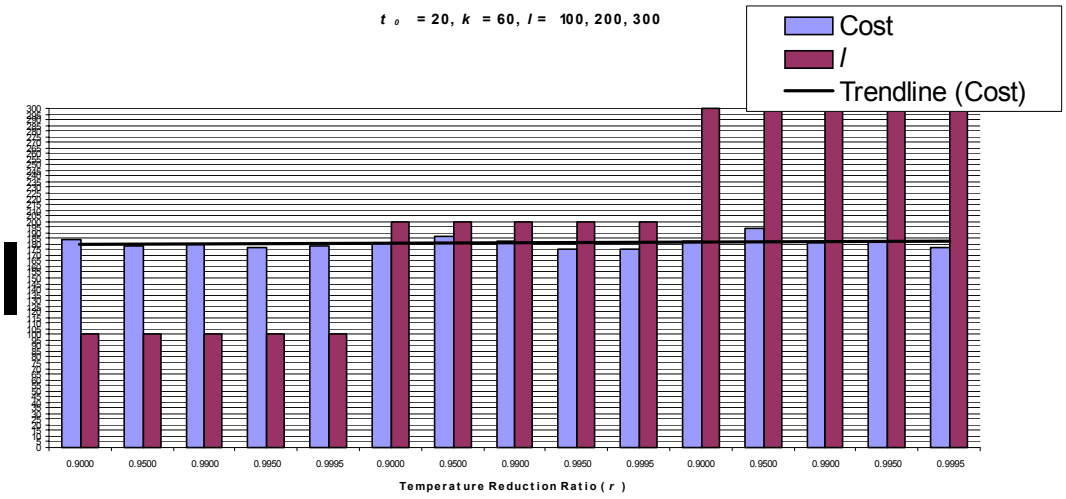

Figure 21 Cost as a Function of r and l ($t_0 = 20$, $k = 60$)

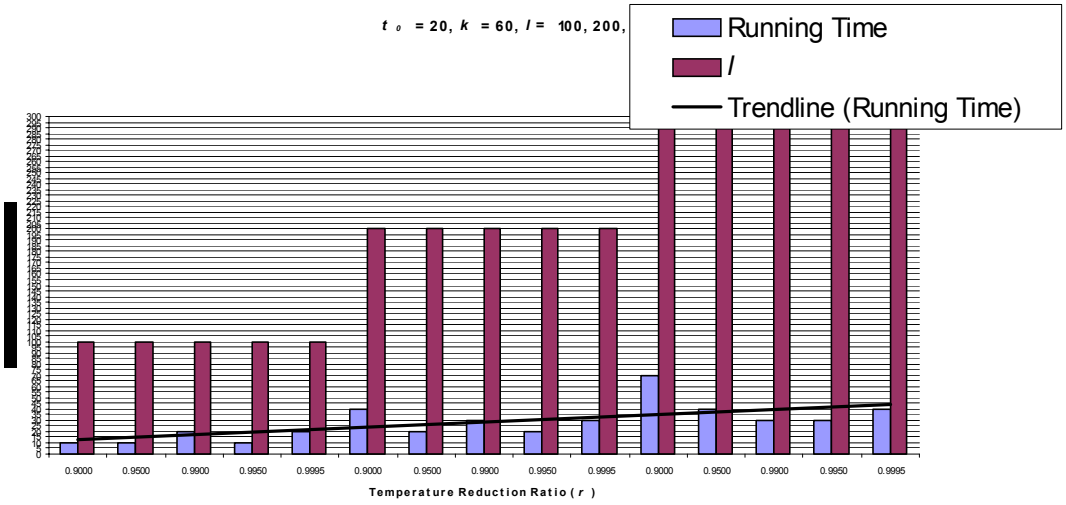

Figure 22 Running Time as a Function of r and l ($t_0 = 20$, $k = 60$)

After carefully balancing the solution quality and running time, we decided to adopt the following parameter values for the simulated annealing algorithm for performance evaluation with all of the 70 benchmark problem instances.

Table 22 Chosen Parameter Values for Simulated Annealing

t_0	r	l	k
15	0.9995	1000	40

Chapter 6

Comparative Study

Since heuristics for solving combinatorial optimization problems are not based on theoretical analysis, the only objective way to evaluate their performance is by conducting comparative study based on a large enough set of benchmark problem instances.

In this chapter, we first define the experimental environment and problem instances. Then a thorough performance evaluation will be conducted to compare both the solution quality and running time for five different heuristics for solving the same time-dependent problems.

6.1 Experiment Design

All experiments will be conducted on a Pentium III PC with a 1.5 GH CPU and a 512 MB main memory.

The following 70 problem instances will be used for performance evaluations. Their value of s varies from 4 to 200; value of n varies from 10 to 835. There are ten problem instances for each combination of the values of s and n.

Table 23 Benchmark Problem Instances

File Name	*s*	*n*	Best Cost	Algorithms leading to best cost
data4c0	4	10	105*	sa, es
data4c1	4	20	70*	es,sa,ga,rr,lo
data4c2	4	16	68*	es,sa,ga,rr
data4c3	4	18	52*	es,sa,ga,rr
data4c4	4	17	65*	es,sa,ga,rr,lo
data4c5	4	19	67*	es,sa,ga,rr,lo
data4c6	4	23	62*	es,sa,ga,rr,lo
data4c7	4	17	61*	es,sa,ga,rr
data4c8	4	15	86*	es,sa,ga,rr
data4c9	4	16	62*	es,sa,ga,rr,lo
data10c0	10	37	110*	es,sa
data10c1	10	30	124*	es,sa
data10c2	10	40	122*	es,sa
data10c3	10	41	133*	es,sa
data10c4	10	38	129*	es,sa
data10c5	10	40	119*	es,sa
data10c6	10	42	131*	es,sa
data10c7	10	39	136*	es,sa
data10c8	10	46	122*	es,sa
data10c9	10	42	145*	es,sa
data15c0	15	60	176	sa
data15c1	15	61	193	sa
data15c2	15	53	192	sa
data15c3	15	56	177	sa
data15c4	15	51	183	sa
data15c5	15	56	181	sa
data15c6	15	62	169	sa
data15c7	15	56	180	sa
data15c8	15	64	183	sa
data15c9	15	55	176	sa
data20c0	20	94	213	sa
data20c1	20	76	245	sa
data20c2	20	74	229	sa
data20c3	20	75	226	sa
data20c4	20	73	241	sa
data20c5	20	84	222	sa
data20c6	20	81	219	sa
data20c7	20	87	219	sa
data20c8	20	71	240	sa
data20c9	20	80	225	sa
data50c0	50	196	518	sa
data50c1	50	212	520	sa
data50c2	50	202	529	sa

data50c3	50	182	535	sa
data50c4	50	191	521	sa
data50c5	50	200	534	sa
data50c6	50	199	531	sa
data50c7	50	205	530	sa
data50c8	50	205	520	sa
data50c9	50	197	528	sa
data100c0	100	423	1007	sa
data100c1	100	396	1034	sa
data100c2	100	404	1029	sa
data100c3	100	404	1027	sa
data100c4	100	411	1031	sa
data100c5	100	408	1036	sa
data100c6	100	377	1030	sa
data100c7	100	386	1037	sa
data100c8	100	392	1030	sa
data100c9	100	410	1026	sa
data200c0	200	835	2001	sa
data200c1	200	814	2027	sa
data200c2	200	813	2033	sa
data200c3	200	804	2041	sa
data200c4	200	797	2039	sa
data200c5	200	784	2047	sa
data200c6	200	785	2040	sa
data200c7	200	804	2041	sa
data200c8	200	801	2030	sa
data200c9	200	814	2037	sa

In Table 23, the best costs are the costs for the corresponding problem instances ever found during this research. If the cost values are marked with asterisk *, then they are the optimal costs proven by exhaustive search. The rightmost column lists the heuristics that achieved the listed best costs. The following abbreviations for heuristic names are used in this chapter:

Table 24 Heuristic Name Abbreviations

Abbreviation	*Full Name*
ES	***Exhaustive Search***
LO	***Local Optimization***
RR	***Repeated Random***
SA	***Simulated Annealing***
GA	***Genetic Algorithm***

6.2 Simulated Annealing vs. Genetic Algorithm

For this experiment, we compare the performances of *simulated annealing* and *genetic algorithm*. For each of the 70 benchmark problem instances, we use each of the above two algorithms to run 10 times and report the best cost, standard deviation of the cost, percentage of cost deterioration from the best known cost for the problem instance; average running time, and standard deviation of the running time.

Table 25 compares the solution quality between simulated annealing and genetic algorithm. We observe that the cost value of simulated annealing outperforms that for genetic algorithm, except for the trivial cases where $s = 4$, by a factor of 5% to 116%. The larger the problem instances, the more simulated annealing out performed genetic algorithm more improvement simulated annealing can provide. The standard deviation columns for simulated annealing shows that the algorithm is producing very stable solution quality for all problem instances. On the another hand, the standard deviation columns for genetic algorithm shows that each run of GA for the same problem instance has wide variation for the quality of the resulting solution.

Table 25 Solution Quality Comparison between SA and GA

File Name	SA Best Cost	Standard Deviation of SA Cost	SA Cost Deterioration Percentage	GA Best Cost	Standard Deviation of GA Cost	GA Cost Deterioration Percentage
data4c0	105	0	0%	105	0	0%
data4c1	70	0	0%	70	6.42	0%
data4c2	68	0	0%	68	0	0%
data4c3	52	0	0%	52	0	0%
data4c4	65	0	0%	65	0	0%
data4c5	67	0	0%	67	0	0%
data4c6	62	0	0%	62	0.32	0%

data4c7	61	0	0%	61	0	0%
data4c8	86	0	0%	86	0	0%
data4c9	62	0	0%	62	0	0%
data10c0	110	0	0%	116	16.78	5.45%
data10c1	124	0	0%	134	12.28	8.06%
data10c2	122	0.95	0%	133	18.23	9.02%
data10c3	133	0	0%	144	10.54	8.27%
data10c4	129	0	0%	147	11.65	13.95%
data10c5	119	0	0%	140	8.12	17.65%
data10c6	131	0	0%	137	13.45	4.58%
data10c7	136	0	0%	154	9.85	13.24%
data10c8	122	0.63	0%	141	12.85	15.57%
data10c9	145	0.32	0%	156	9.53	7.59%
data15c0	176	0.63	0%	232	15.15	31.82%
data15c1	193	2.86	0%	250	12.94	29.53%
data15c2	192	2.71	0%	236	26.31	22.92%
data15c3	177	2.11	0%	223	25.58	25.99%
data15c4	183	2.13	0%	216	26.06	18.03%
data15c5	181	2.22	0%	222	25.07	22.65%
data15c6	169	3.37	0%	241	15.21	42.60%
data15c7	180	1.16	0%	230	26.77	27.78%
data15c8	183	2.37	0%	222	25.84	21.31%
data15c9	176	0.82	0%	222	13.40	26.14%
data20c0	215	2.41	0.94%	311	42.46	46.01%
data20c1	245	2.49	0%	348	22.23	42.04%
data20c2	229	3.96	0%	335	26.39	46.29%
data20c3	226	2.90	0%	329	28.87	45.58%
data20c4	241	3.79	0%	303	36.10	25.73%
data20c5	222	2.91	0%	323	29.80	45.50%
data20c6	219	2.56	0%	300	27.71	36.99%
data20c7	219	3.37	0%	304	14.88	38.81%
data20c8	240	3.89	0%	338	28.14	40.83%
data20c9	225	4.14	0%	319	29.28	41.78%
data50c0	529	6	2.12%	958	92.19	84.94%
data50c1	520	7.44	0%	931	53.87	79.04%
data50c2	529	5.87	0%	914	75.02	72.78%
data50c3	535	6.45	0%	1013	48.72	89.35%
data50c4	521	3.37	0%	999	40.04	91.75%
data50c5	534	5.62	0%	930	87.40	74.16%
data50c6	531	6.46	0%	922	78.40	73.63%
data50c7	530	7.88	0%	977	46.30	84.34%
data50c8	520	7.94	0%	952	61.16	83.08%
data50c9	528	5.18	0%	919	74.26	74.05%
data100c0	1025	7.44	1.79%	2021	126.31	100.70%
data100c1	1034	5.28	0%	2151	84.73	108.03%
data100c2	1029	8.80	0%	2203	82.15	114.09%
data100c3	1027	8.93	0%	2127	128.10	107.11%
data100c4	1031	4.70	0%	2016	175.76	95.54%

data100c5	1036	6.35	0%	1987	118.26	91.80%
data100c6	1030	9.22	0%	2238	120.92	117.28%
data100c7	1037	6.93	0%	2193	105.34	111.48%
data100c8	1030	7.13	0%	2116	107.70	105.44%
data100c9	1026	5.48	0%	2152	99.55	109.75%
data200c0	2030	10.68	1.45%	4562	170.27	127.99%
data200c1	2027	13.35	0%	4594	140.53	126.64%
data200c2	2033	9.96	0%	4529	175.34	122.77%
data200c3	2041	10.56	0%	4623	157.85	126.51%
data200c4	2039	9.87	0%	4541	188.92	122.71%
data200c5	2047	6.04	0%	4792	86.15	134.10%
data200c6	2040	11.04	0%	4538	214.04	122.45%
data200c7	2041	6.60	0%	4573	162.38	124.06%
data200c8	2030	14.29	0%	4573	195.64	125.27%
data200c9	2037	10.92	0%	4420	213.42	116.99%

Table 26 compares the running time of simulated annealing and genetic algorithm for all the 70 benchmark problem instances. For each problem instances, we run each of the algorithms ten times and report the average running time and the standard deviation of these ten running times. It can be observed from the table that simulated annealing improves the running time of genetic algorithm by a factor of 2 to 145; the larger the problem instances, the more improvement in running time. It can also be observed from the table that the genetic algorithm has huge standard deviation of its running time, which implies very unstable performance.

Table 26 Running Time Comparison between SA and GA

File Name	SA Ave Time	Standard Deviation of SA Time	GA Ave Time	Standard Deviation of GA Time
data4c0	8	6.32	20	14.91
data4c1	7	4.83	17	6.75
data4c2	5	5.27	14	5.16
data4c3	6	5.16	14	5.30
data4c4	6	5.16	15	5.27
data4c5	8	4.22	18	6.32

data4c6	7	4.91	16	5.16
data4c7	8	4.22	16	5.26
data4c8	7	4.83	12	4.22
data4c9	5	5.27	15	5.27
data10c0	10	0	45	13.50
data10c1	11	3.16	70	54.33
data10c2	12	4.43	89	70.78
data10c3	12	4.22	89	64.39
data10c4	11	5.68	52	13.15
data10c5	7	6.75	61	23.69
data10c6	11	3.16	56	34.69
data10c7	13	4.77	52	35.53
data10c8	11	3.16	64	34.31
data10c9	11	5.68	141	233.25
data15c0	15	5.27	97	59.52
data15c1	15	7.07	315	373.41
data15c2	14	5.16	165	93.14
data15c3	15	5.27	112	51.55
data15c4	14	5.16	298	245.26
data15c5	15	5.27	248	171.24
data15c6	14	5.30	135	76.70
data15c7	14	5.16	162	94.64
data15c8	17	4.83	138	170
data15c9	15	5.27	134	137.92
data20c0	18	4.28	303	209.88
data20c1	18	4.22	524	464.56
data20c2	18	4.22	380	235.80
data20c3	15	5.38	278	131.87
data20c4	15	5.27	390	296.54
data20c5	18	4.22	321	187.26
data20c6	18	4.22	369	265.18
data20c7	18	4.28	709	1258.21
data20c8	17	4.83	379	336.31
data20c9	19	5.68	319	233.76
data50c0	33	4.77	4368	3461.11
data50c1	32	9.19	3338	5154.31
data50c2	35	7.00	2826	1766.25
data50c3	32	7.89	3144	5306.64
data50c4	33	9.57	4084	6741.39
data50c5	33	8.23	4005	5489.07
data50c6	35	8.57	2782	1056.25
data50c7	39	11.12	5387	8880
data50c8	34	5.16	3036	1229.09
data50c9	31	7.37	2909	1585.80
data100c0	73	17.65	14261	12070.83
data100c1	82	13.24	8115	6465.45
data100c2	82	12.19	10660	5501.57
data100c3	78	18.15	12896	5836.05

data100c4	82	12.19	6840	3997.60
data100c5	72	21.80	16038	19472.37
data100c6	84	15.96	9987	8308.54
data100c7	86	30.64	11865	9382.15
data100c8	78	25.83	7008	2369.92
data100c9	74	9.73	11789	9600.68
data200c0	198	45.89	31768	17982.08
data200c1	179	36.30	27262	10885.68
data200c2	192	28.81	38870	24670.55
data200c3	167	27.82	34574	29842.35
data200c4	205	42.38	47288	41698.07
data200c5	183	23.60	48851	32810.93
data200c6	195	39.64	32730	44931.52
data200c7	185	33.20	51656	70257.49
data200c8	206	80.50	51656	22993.9
data200c9	229	87.98	33423	38083.9

6.3 Comparison between Simulated Annealing and Repeated Random Solutions

For a heuristic to prove its value in combinatorial optimization, it must show that it can produce better solutions than repeatedly generated random solutions in the same amount of running time. In this and the following two sections, we conduct this type of performance evaluation for both simulated annealing and genetic algorithm.

Table 27 shows the comparison of solution quality of simulated annealing in Table 25 with the *Repeated Random* heuristic, for each problem instance.

Table 27 Comparison between Simulated Annealing and Repeated Random Solutions

File Name	SA Best Cost	RR Cost for Same Time	SA Cost Improvement	SA Cost Improvement %
data4c0	105	105	0	0
data4c1	70	70	0	0
data4c2	68	68	0	0
data4c3	52	52	0	0

data4c4	65	65	0	0
data4c5	67	67	0	0
data4c6	62	62	0	0
data4c7	61	61	0	0
data4c8	86	86	0	0
data4c9	62	62	0	0
data10c0	110	124	14	11.29
data10c1	124	144	20	13.89
data10c2	122	155	33	21.29
data10c3	133	157	24	15.29
data10c4	129	152	23	15.13
data10c5	119	155	36	23.23
data10c6	131	142	11	7.75
data10c7	136	153	17	11.11
data10c8	122	131	9	6.87
data10c9	145	164	19	11.59
data15c0	176	246	70	28.46
data15c1	193	240	47	19.58
data15c2	192	267	75	28.09
data15c3	177	250	73	29.20
data15c4	183	261	78	29.89
data15c5	181	218	37	16.97
data15c6	169	229	60	26.20
data15c7	180	241	61	25.31
data15c8	183	243	60	24.69
data15c9	176	227	51	22.47
data20c0	215	318	103	32.39
data20c1	245	382	137	35.86
data20c2	229	380	151	39.74
data20c3	226	374	148	39.57
data20c4	241	351	110	31.34
data20c5	222	331	109	32.93
data20c6	219	336	117	34.82
data20c7	219	323	104	32.20
data20c8	240	319	79	24.76
data20c9	225	346	121	34.97
data50c0	529	1039	510	49.09
data50c1	520	1037	517	49.86
data50c2	529	1035	506	48.89
data50c3	535	1087	552	50.78
data50c4	521	1016	495	48.72
data50c5	534	1059	525	49.58
data50c6	531	1025	494	48.20
data50c7	530	1032	502	48.64
data50c8	520	1028	508	49.42
data50c9	528	1029	501	48.69
data100c0	1025	2164	1139	52.63
data100c1	1034	2232	1198	53.67

data100c2	1029	2218	1189	53.61
data100c3	1027	2250	1223	54.36
data100c4	1031	2160	1129	52.27
data100c5	1036	2189	1153	52.67
data100c6	1030	2253	1223	54.28
data100c7	1037	2236	1199	53.62
data100c8	1030	2291	1261	55.04
data100c9	1026	2143	1117	52.12
data200c0	2030	4683	2653	56.65
data200c1	2027	4485	2458	54.80
data200c2	2033	4580	2547	55.61
data200c3	2041	4631	2590	55.93
data200c4	2039	4595	2556	55.63
data200c5	2047	4669	2622	56.16
data200c6	2040	4702	2662	56.61
data200c7	2041	4668	2627	56.28
data200c8	2030	4695	2665	56.76
data200c9	2037	4682	2645	56.49

6.4 Comparison between Genetic Algorithm and Repeated Random Solutions

Table 28 compares the solution quality between genetic algorithm and repeated random solutions. The repeated random heuristic was executed for as long as the average running time of the genetic algorithm for each particular problem instance. The table reports for each problem instance the best GA cost for ten runs, the repeated random cost for the same running time, cost improvement by GA over repeated random, and percentage of cost improvement of GA over repeated random. It can be seen from the table that genetic algorithm is slightly worse than repeated random for most of the problem instances.

Table 28 Comparison between Genetic Algorithm and Repeated Random Solutions

File Name	GA Best Cost	RR Cost for Same Time	GA Cost Improvement	GA Cost Improvement %
data4c0	105	105	0	0.00
data4c1	70	70	0	0.00
data4c2	68	68	0	0.00

data4c3	52	52	0	0.00
data4c4	65	65	0	0.00
data4c5	67	67	0	0.00
data4c6	62	62	0	0.00
data4c7	61	61	0	0.00
data4c8	86	86	0	0.00
data4c9	62	62	0	0.00
data10c0	116	127	11	8.66
data10c1	134	143	9	6.29
data10c2	133	125	-8	-6.40
data10c3	144	139	-5	-3.60
data10c4	147	153	6	3.92
data10c5	140	133	-7	-5.26
data10c6	137	142	5	3.52
data10c7	154	148	-6	-4.05
data10c8	141	130	-11	-8.46
data10c9	156	156	0	0.00
data15c0	232	231	-1	-0.43
data15c1	250	236	-14	-5.93
data15c2	236	265	29	10.94
data15c3	223	225	2	0.89
data15c4	216	237	21	8.86
data15c5	222	262	40	15.27
data15c6	241	226	-15	-6.64
data15c7	230	238	8	3.36
data15c8	222	225	3	1.33
data15c9	222	237	15	6.33
data20c0	311	297	-14	-4.71
data20c1	348	337	-11	-3.26
data20c2	335	337	2	0.59
data20c3	329	333	4	1.20
data20c4	303	332	29	8.73
data20c5	323	323	0	0.00
data20c6	300	326	26	7.98
data20c7	304	290	-14	-4.83
data20c8	338	341	3	0.88
data20c9	319	316	-3	-0.95
data50c0	958	946	-12	-1.27
data50c1	931	919	-12	-1.31
data50c2	914	945	31	3.28
data50c3	1013	984	-29	-2.95
data50c4	999	921	-78	-8.47
data50c5	930	964	34	3.53
data50c6	922	934	12	1.28
data50c7	977	933	-44	-4.72
data50c8	952	951	-1	-0.11
data50c9	919	963	44	4.57
data100c0	2021	1991	-30	-1.51

data100c1	2151	2140	-11	-0.51
data100c2	2203	2082	-121	-5.81
data100c3	2127	2082	-45	-2.16
data100c4	2016	2014	-2	-0.10
data100c5	1987	2043	56	2.74
data100c6	2238	2157	-81	-3.76
data100c7	2193	2135	-58	-2.72
data100c8	2116	2081	-35	-1.68
data100c9	2152	2019	-133	-6.59
data200c0	4562	4346	-216	-4.97
data200c1	4594	4431	-163	-3.68
data200c2	4529	4411	-118	-2.68
data200c3	4623	4497	-126	-2.80
data200c4	4541	4453	-88	-1.98
data200c5	4792	4547	-245	-5.39
data200c6	4538	4550	12	0.26
data200c7	4573	4493	-80	-1.78
data200c8	4573	4453	-120	-2.69
data200c9	4420	4395	-25	-0.57

Chapter 7

Conclusion

This research focused on efficient solutions for a class of time-dependent combinatorial optimization problems. The major contributions of this research include:

1. Mathematical modeling of a class of time-dependent combinatorial optimization problems described in [1];
2. Reducing problem complexity by problem transformation;
3. Designing an efficient simulated annealing algorithm that outperforms the best existing algorithm for this class of problems with significant improvement in solution quality and phenomenal reduction in running time.

The following are potential future work to extend on this research:

1. Generalizing the problem class to include problems that don't exhibit the independency property described in Subsection 3.3.1;
2. Improving the simulated annealing algorithm by taking advantage of the latest research advancement for this approach;
3. Investigating alternative meta-heuristics, including tabu search, in solving this class of problems.

Appendix A

Java Source Code for All Algorithms

SimulatedAnnealing.Java

```
public class SimulatedAnnealing {

  int run(int bestSolution[], Utilities u) {
    int p[] = new int[u.linkNbr+1];       // current solution
    // retrieve the Random object in class Utilities
    java.util.Random ro = u.getRandom();
    int debugLevel = u.getDebugLevel();
    // use a random solution as the current solution
    u.randomSolution(p);
    int currentCost = u.cost(p);
    int bestCost = currentCost;
    u.copyArray(p, bestSolution);
    int neighbor[] = new int[p.length];
    double t = t0;   // initial temperature; parameter to adjust
    int nonImprovementTemperatureNbr = 0;
    while (nonImprovementTemperatureNbr < k) { // while not yet frozen
      nonImprovementTemperatureNbr++;
      int nonImprovementIterationNbr = 0;
      while (nonImprovementIterationNbr < l) {
        nonImprovementIterationNbr++;
        u.copyArray(p, neighbor);
        // make a random swap; neighbor[] is now a neighbor of p[];
        // return updated cost
        int newCost = u.randomSwap(neighbor, currentCost);
        int delta = newCost - currentCost;
        double acceptProbability = Math.exp(-delta/t);
        if ((delta <= 0) || (ro.nextDouble() < acceptProbability)) {
           // take neighbor as new current solution
           u.copyArray(neighbor, p);
           currentCost = newCost;
           if (currentCost < bestCost) {
             // update the best solution seen so far
             bestCost = currentCost;
             u.copyArray(p, bestSolution);
             nonImprovementTemperatureNbr = 0;
             nonImprovementIterationNbr = 0;
           }
        }
      }
      t = r*t;   // reduce temperature}
    }
    return bestCost;
```

```
  }

  public static void main(String args[]) {
    String dataFileName = "data4.txt";   // default data file name
    if (args.length == 1)
      dataFileName = args[0];            // command-line data file name
    Utilities u = new Utilities();
    u.setDebugLevel(0);                  // how much debug info to
                                         // print; 0 means nothing
    u.readData(dataFileName);            // read data for a problem
                                         // instance
    u.problemTransformation();           // transform the problem to a
                                         // simpler one
    int bestCost;
    int bestSolution[] = new int[u.linkNbr + 1];
    SimulatedAnnealing sa = new SimulatedAnnealing();
    u.startRun();
    bestCost = sa.run(bestSolution, u);
    u.endRun();
    u.report("Simulated annealing", bestCost, bestSolution);
  }
}
```

GeneticJoe.Java (Genetic Algorithm)

```
// Implement the GA algorithm described in Joseph DeCicco's DPS 2002
// dissertation "Sensitivity Analysis of Certain Time Dependent Matroid
// Base Models Solved by Genetic Algorithms"
import java.util.Random;

public class GeneticJoe {

  // objects/values to be retrieved from Utilities object
  Utilities u;       // Utility object
  Random r;          // Random object
  int bidAlloc[][];  // Each row corresponding to a link.
                     // Links are numbered 1, 2, ...
                     // Each row specifies number of bids for a link,
                     // and the corresponding successive starting and
                     // ending bid IDs
  int cost[][];      // cost[i][j] is the cost for letting bid i build
                     // its bidding link at stage j
  int linkNbr;       // number of highway links to build

  // global objects for GA
  int p[];           // a generic array for permutations of time stages
                     // or selected bid IDs
  Object generation[] = null;  // table of members of a generation
  int debugLevel = 0;  // volume of debug info printed; 0 means minimal

  // GA parameters for adjustment
  int generationSize = 50;          // number of code members in a
                                    // generation
  int childrenNumber = 50;          // number of new children in a new
                                    // generation
  int crossoverNumber = 40;         // number of children generated by
                                    // crossover
  int mutateNumber = 10;            // number of children generated by
                                    // mutation
  int selectBestNumber = 50;        // number of best members selected
                                    // as parents
  int maxMutatePositions;           // max number of mutation positions
                                    // for a member
  int nbrNoGainGen4terminate = 50;  // number of non-improvement
                                    // generations before the algorithm
                                    // ends

  boolean banDuplicate = true;      // false: allow duplicate child
                                    // during crossover and mutation
                                    // true:  don't allow duplicate
                                    // child during crossover and
                                    // mutation
                                    // To truely follow Joe's
                                    // dissertation, use false

  int run(int bestSolution[], Utilities u1) {
    u = u1;
    debugLevel = u.getDebugLevel();
```

```
r = u.getRandom();                  // retrieve Random object
linkNbr = u.getLinkNbr();           // retrieve highway link number
maxMutatePositions = linkNbr;       // max number of positions of a
                                    // member for mutation
// [0] for cost, [1..linkNbr] for bid IDs, [linkNbr+1..2*linkNbr]
// for permutation
// allocate space for best member seen so far
int gaBestMember[] = new int[2*linkNbr+1];
bidAlloc = u.getBidAlloc();         // retrieve bid allocation
                                    // table
cost = u.getCost();                 // retrieve original (Problem
                                    // A) cost table
p = new int[u.linkNbr+1];

int bestCost = 9999;                // impossible bad cost to be
                                    // replaced right away
int noGainIterations = 0;           // number of no-improvement
                                    // successive generations seen
                                    // now
generation = new Object[generationSize];
int generationSeqNumber = 0;        // initial one is generation 0
int generation4lastImprovement = 0; // the generation in which the
                                    // last cost reduction was made
generateInitialGeneration();
while (noGainIterations <= nbrNoGainGen4terminate) {
  generationSeqNumber++;
  if (debugLevel > 0) {
    System.out.println("Start generation " + generationSeqNumber);
    printGeneration(10);  // print the best 10 members of the
                          // current generation
  }
  crossover(crossoverNumber);
  mutate(mutateNumber);
  int bestMember[] = (int[])generation[0];  // retrieve the best
                                            // member in current
                                            // generation
  int cost = bestMember[0];
  if (cost < bestCost) {
    bestCost = cost;
    u.copyArray(bestMember, gaBestMember);
    generation4lastImprovement = generationSeqNumber;
    noGainIterations = 0;     // reset the count for successive
                              // non-improvement iterations
  }
  else
    noGainIterations++;
}
System.out.print("GA best cost = " + bestCost + ", bid-ID
                 permutation ");
member2permutation(gaBestMember, bestSolution, false);
u.printSolution(bestSolution);
System.out.println("GA finished " + generationSeqNumber +
   " generations. The best cost was found in generation " +
   generation4lastImprovement);
// the returned cost and solution may be better than the ones found
// by GA
```

```
  // since multiple different GA members may be converted into the
  // same permutation
  member2permutation(gaBestMember, bestSolution, true);
  return u.cost(bestSolution);
}


// print the first (best) n members of the current generation
// order: member sequence number, cost, coding
void printGeneration(int n) {
  System.out.println("Best " + n + " member(s) in current
     generation:");
  for (int i = 0; i < n; i++) {
    int temp[] = (int[])generation[i];
    System.out.print(i + ": cost= " + temp[0] + " : ");
    for (int j = 1; j <= 2*linkNbr; j++)
      System.out.print(temp[j] + ", ");
    member2permutation(temp, p, false);
    System.out.print(": ");
    for (int j = 1; j <= linkNbr; j++)
      System.out.print(p[j] + ", ");
    System.out.println();
  }
}

// If permuteTimeStage is true, return in p[] the permutation of time
// stages according to code[]
// Otherwise, return in p[] the permutation of bid IDs in
// code[1..linkNbr] according to code[]
void member2permutation(int code[], int p[], boolean
                        permuteTimeStage) {
  for (int i =0; i <= linkNbr; i++)
    p[i] = 0; // reset p[]
  for (int i = 1; i <= linkNbr; i++) {
    int j = 0;                    // index for empty position in p[];
                                  // properly set in do loop
    int count = code[linkNbr+i];  // count = number of empty
                                  // positions in p[] from left
    do {
      j++;
      while (p[j] > 0)
        j++;
      count--;
    } while ((count > 0) && (j < linkNbr));
    if (j > linkNbr) {
      System.out.println("member2permutation() generates invalid
         permutation");
      System.exit(-1);
    }
    if (permuteTimeStage == false)
      p[j] = code[i];  // generate permutation of selected bid IDs
    else
      p[j] = i;        // generate permutation of highway links {1,
                       // 2, ..., linkNbr}
  }
  if (permuteTimeStage == false)
```

```
    return;
  int temp[] = new int[linkNbr+1];
  for (int i = 0; i <= linkNbr; i++)
    temp[i] = p[i];
  // p[i] = j : hihgway link j will be built in time stage i
  for (int i = 1; i <= linkNbr; i++)
    p[temp[i]] = i;
  // p[i] = j : hihgway link i will be built in time stage j
}

// objective function of the original Problem A
int codeCost(int code[]) {
  int temp[] = new int[linkNbr+1];
  member2permutation(code, temp, false); // generate permutation of
                                         // time stages
  int totalCost = 0;
  for (int i = 1; i <= linkNbr; i++)
    totalCost += cost[temp[i]][i];
  return totalCost;
}

// allocate member space and generate random value for generation 0;
// duplicates not allowed
// sort the members based on costs
void generateInitialGeneration() {
  for (int i = 0; i < generationSize; i++) {
    // generation[*][0] is the cost of solution generation[*]
    int temp[] = new int[2*linkNbr+1];
    do {
      generateRandomeCode(temp);
    } while (isDuplicate(temp, 0, i-1));  // generate a new member
    temp[0] = codeCost(temp);
    insert(temp, i);
  }
}

// return true iff code[] duplicates a member in generation[from..to]
boolean isDuplicate(int code[], int from, int to) {
  if (to < 0 || from > to)
    return false;  // table generation[from..to] is empty
  int cost = codeCost(code);
  boolean foundDuplicate = false;
  for (int i = from; i <= to; i++) {
    int temp[] = (int[])generation[i];
    if (temp[0] != cost)
      continue;                // if generation[i] has different
                               // cost, it cannot be a duplicate
    int j;
    for (j = 1; j <= 2*linkNbr; j++)
      if (temp[j] != code[j])
        break;
    if (j > 2*linkNbr) {
      foundDuplicate = true;
      break;
    }
  }
```

```
  return foundDuplicate;
}

// generate the coding of a random solution
void generateRandomeCode(int c[]) {
  // generate the first linkNbr random link IDs
  for (int i = 1; i <= linkNbr; i++) {
    int start = bidAlloc[i][1];
    int end = bidAlloc[i][2];
    c[i] = r.nextInt(end-start+1) + start;  // c[i] is the random bid
                                            // ID for building link i
  }
  // Generate the next linkNbr of permutation numbers
  // First number is in 1..linkNbr, next in 1..linknbr-1, ... The
  // last must be 1.
  for (int i = 1; i <= linkNbr; i++) {
    c[linkNbr+i] = r.nextInt(linkNbr + 1 - i) + 1;
  }
}

// generation[0..i-1] is already sorted relative to generation[][0]
// insert member c and make generation[] remain sorted
void insert(int c[], int i) {
  generation[i] = c;
  while ((i > 0) && ((int[])generation[i-1])[0] >
    ((int[])generation[i])[0]) {
    // swap generation[i-1] and generation[i]
    Object temp = generation[i-1];
    generation[i-1] = generation[i];
    generation[i] = temp;
    i--;
  }
}

// conduct crossover n times to generate n new unique children
void crossover(int n) {
  int child[];
  int parent1, parent2;
  int from, to;
  int parent1code[], parent2code[];
  for (int i = 0; i < n; i++) {
    do {
      parent1 = r.nextInt(selectBestNumber);
      // parent1 and parent2 always differ: this is different from
      // Joe's dissertation
      do {   // I made this change to avoid generation of duplicate
             // members
             // To truely implement Joe's algorithm, reset variable
             // banDuplicate to false
        parent2 = r.nextInt(selectBestNumber);
      } while (parent1 == parent2);
      parent1code = (int[])generation[parent1];
      parent2code = (int[])generation[parent2];
      from = r.nextInt(linkNbr*2-1) + 1;          // from is in
                                                  // [1..2*linkNbr-1]
      to = r.nextInt(linkNbr*2-from) + from + 1;  // to is in
```

```
                                                    // [from+1..2*
                                                    // linkNbr]
      child = (int[])generation[generationSize-1]; // reuse the space
                                                    // of the worst
                                                    // member
      u.copyArray(parent1code, child);
      for (int j = from; j <= to; j++)
        child[j] = parent2code[j];
      child[0] = codeCost(child);
    } while (isDuplicate(child, 0, generationSize-2) &&
             banDuplicate);
    // if banDuplicate = false, crossover may generate duplicate
    // members in generation
    insert(child, generationSize-1); // insert child[] into
                                     // generation[]
    if (debugLevel > 0) {
      System.out.println("Mate " + parent1 + " and " + parent2 +
                         " from " + from + " to " + to + ", child
                         cost = " + child[0]);
      System.out.println("parent1 " + parent1 + ":");
      u.printArray(parent1code);
      System.out.println("parent2 " + parent2 + ":");
      u.printArray(parent2code);
      System.out.println("child:");
      u.printArray(child);
    }
  }
}

// conduct mutatation n times to generate n new unique children
void mutate(int n) {
  for (int i = 0; i < n; i++) {
    // reuse the space of the worst member
    int child[] = (int[])generation[generationSize-1];
    int parent = r.nextInt(selectBestNumber);
    int parentCode[] = (int[])generation[parent];
    u.copyArray(parentCode, child);
    int mutateTime = r.nextInt(maxMutatePositions) + 1;
    do {  // new child must not duplicate any current member; differ
          // from Joe's dissertation
          // To truely implement Joe's algorithm, reset variable
          // banDuplicate to false
      for (int j = 0; j < mutateTime; j++) {
        int k = r.nextInt(2*linkNbr) + 1;  // allow the same
                                           // position chosen
                                           // multiple times
        if (k <= linkNbr)
          perturbBidID(child, k);
        else
          perturbPermutation(child, k);
      }
      child[0] = codeCost(child);
    } while (isDuplicate(child, 0, generationSize-2) &&
             banDuplicate);
    // if banDuplicate = false, mutate may generate duplicate members
    // in generation
```

```
      if (debugLevel > 0) {
        System.out.println("Mutate parent " + parent + " at " +
                           mutateTime + " position(s)," +
                          " child cost = " + child[0]);
        System.out.println("parent " + parent + ":");
        u.printArray(parentCode);
        System.out.println("child:");
        u.printArray(child);
      }
      insert(child, generationSize-1);  // insert new child
    }
  }

  // perturb the left half of code for bid IDs
  void perturbBidID(int code[], int k) {
    int from = bidAlloc[k][1];   // starting bid ID for building link k
    int to = bidAlloc[k][2];     // ending bid ID for building link k
    code[k] = r.nextInt(to - from + 1) + from;  // choose a random bid
                                                // for building link k
  }

  // perturb the right half of code for permutation
  void perturbPermutation(int code[], int k) {
    int i = k - linkNbr;
    code[k] = r.nextInt(linkNbr - i + 1) + 1; // code[linkNbr + i] is
                                              // in {1, 2, ...,
                                              // linkNbr-i+1}
  }


  public static void main(String args[]) {
    String dataFileName = "data4.txt";          // default data file
                                                // name
    if (args.length == 1)
      dataFileName = args[0];                   // command-line data
                                                // file name
    Utilities u = new Utilities();
    u.readData(dataFileName);                   // read data for a
                                                // problem instance
    u.problemTransformation();                  // transform the
                                                // problem to a
                                                // simpler one
    u.setDebugLevel(1);                         // print some debug
                                                // info
    GeneticJoe ga = new GeneticJoe();
    int bestCost;
    int bestSolution[] = new int[u.linkNbr + 1];
    u.startRun();
    bestCost = ga.run(bestSolution, u);
    u.endRun();
    u.report("Genetic (solution transformed)", bestCost, bestSolution);
  }
}
```

Random Repeat.Java

```
public class RepeatRandom {

  int run(int bestSolution[], int times, Utilities u) {
    int p[] = new int[u.linkNbr+1];
    int bestCost = 9999;                  // impossible bad cost, to be
                                          // replaced
    for (int i = 0; i < times; i++) {
      u.randomSolution(p);
      int cost = u.cost(p);
      if (cost < bestCost) {
        bestCost = cost;
        u.copyArray(p, bestSolution);
      }
    }
    return bestCost;
  }

  public static void main(String args[]) {
    String dataFileName = "data4.txt";           // default data file
                                                 // name
    if (args.length == 1)
      dataFileName = args[0];                    // command-line data
                                                 // file name
    Utilities u = new Utilities();
    u.readData(dataFileName);                    // read data for a
                                                 // problem instance
    u.problemTransformation();                   // transform the
                                                 // problem to a
                                                 // simpler one
    int bestCost;
    int bestSolution[] = new int[u.linkNbr + 1];
    RepeatRandom rr = new RepeatRandom();
    u.startRun();
    bestCost = rr.run(bestSolution, 100, u);    // 100 will be adjusted
    u.endRun();
    u.report("Repeat random solution 100 times", bestCost,
       bestSolution);
  }
}
```

Exhaustive Search.Java

```
public class ExhaustiveSearch {

  int run(int bestSolution[], Utilities u) {
    int p[] = new int[u.linkNbr+1];
    p[0] = 999;   // p[0] is not used; 999 is a dummy value for
                  // permute() to avoid p[0]
    for (int i = 1; i <= u.linkNbr; i++)  // generate the first
                                          // permutation of {1, 2, ...,
                                          // linkNbr}
      p[i] = i;
    u.copyArray(p, bestSolution);
    int bestCost = 9999;   // an impossible bad cost so it will be
                           // replaced
    do {
      int cost = u.cost(p);
      if (cost < bestCost) {
        bestCost = cost;
        u.copyArray(p, bestSolution);
      }
    } while (Permutator.permute(p));   // p becomes the next
                                       // permutation in lexicographic
                                       // order
    return bestCost;
  }

  public static void main(String args[]) {
    String dataFileName = "data4.txt";    // default data file name
    if (args.length == 1)
      dataFileName = args[0];             // command-line data file
                                          // name
    Utilities u = new Utilities();
    u.readData(dataFileName);             // read data for a problem
                                          // instance
    u.problemTransformation();            // transform the problem to a
                                          // simpler one
    int bestCost;
    int bestSolution[] = new int[u.linkNbr + 1];
    ExhaustiveSearch es = new ExhaustiveSearch();
    u.startRun();
    bestCost = es.run(bestSolution, u);
    u.endRun();
    u.report("Exhaustive search", bestCost, bestSolution);
  }
}
```

Utilities.Java

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

public class Utilities {

  int linkNbr;   //  Number of links
  int bidNbr;    //  Number of bids
  int bidAlloc[][];  // Each row corresponding to a link.
                     // Links are numbered 1, 2, ...
                     // Each row specifies number of bids for a link,
                     // and the corresponding successive starting and
                     // ending bid IDs
  int cost[][];  // cost[i][j] is the cost for letting bid i build its
                 // bidding link at stage j
  int cost1[][]; // cost1[i][j] is the lowest cost for building link i
                 // at stage j
  int cost2[][]; // cost2[i][j] is the bid ID corresponding to
                 // cost1[i][j]
  Random r = null;   // Random number generator
  long startTime;    // mark the start of a run
  long endTime;      // mark the end of a run
  int debugLevel = 0;  // control the amount of idebug info to be
                       // printed; 0 means minimal

  public Utilities() {  // constructor
    // Initialize random number generator with current time
    long randomSeed = System.currentTimeMillis();
    r = new Random(randomSeed);
  }

  public int getDebugLevel() {
    return debugLevel;
  }

  public void setDebugLevel(int v) {
    debugLevel = v;
  }

  public Random getRandom() {
    return r;
  }

  public int getLinkNbr() {
    return linkNbr;
  }

  public int[][] getBidAlloc() {
    return bidAlloc;
  }

  public int[][] getCost() {
    return cost;
```

```
}

void startRun() {
  startTime = System.currentTimeMillis();
}

void endRun() {
  endTime = System.currentTimeMillis();
}

long elapsedTime() {
  return endTime - startTime;
}

// print the contents of a 1-D array
void printArray(int d[]) {
  for (int i = 0; i < d.length; i++)
    System.out.print(d[i] + ", ");
  System.out.println();
}

// prints p[1..linkNbr]; avoids p[0]
void printSolution(int p[]) {
  for (int i = 1; i < p.length; i++)
    System.out.print(p[i] + " ");
  System.out.println();
}


// exchange the value of p[l] and p[r]
void swap(int p[], int l,int r) {
  int temp = p[l];
  p[l] = p[r];
  p[r] = temp;
}

// copy values of from[] into to[]
void copyArray(int from[], int to[]) {
  for (int i = 0; i < to.length; i++)
    to[i] = from[i];
}

// cost of a permutation of highway build sequence; objective
// function for Problem B
int cost(int p[]) {
  int cost = 0;
  for (int i = 1; i <= linkNbr; i++) {
    cost += cost1[i][p[i]];
  }
  return cost;
}

// incrementally update cost after swapping values of p[] in
// positions i and j
int updateCost(int p[], int cost, int i, int j) {
  return cost - cost1[i][p[i]] - cost1[j][p[j]] + cost1[i][p[j]] +
```

```
          cost1[j][p[i]];
}

// print to screen the execution results of an algorithm
void report(String message, int bestCost, int bestSolution[]) {
  System.out.print(message + ", best cost = " + bestCost);
  System.out.print(", solution: ");
  int bidSequence[] = new int[linkNbr+1];
  for (int i = 1; i <= linkNbr; i++)
    bidSequence[bestSolution[i]] = cost2[i][bestSolution[i]];
  for (int i = 1; i <= linkNbr; i++)
    System.out.print(bidSequence[i] + " ");
  System.out.println();
  System.out.println("Elapsed time = " + elapsedTime());
  System.out.println("-------------------------------------");
}

// transform Problem A to simpler Problem B
void problemTransformation() {
  cost1 = new int[linkNbr+1][linkNbr+1];
  cost2 = new int[linkNbr+1][linkNbr+1];
  for (int i = 1; i <=linkNbr; i++) {     // i for link
    for (int j = 1; j <= linkNbr; j++) {  // j for time stage
      int from = bidAlloc[i][1];
      int to = bidAlloc[i][2];
      int minimum = 99999;                // an impossibly large
                                          // number
      int minBidID = linkNbr + 1;         // bid miniBidNbr should
                                          // have minimum bid
      for (int k = from; k <= to; k++) {
        if (cost[k][j] < minimum) {
          minimum = cost[k][j];
          minBidID = k;
        }
      }
      cost1[i][j] = minimum;
      cost2[i][j] = minBidID;
    }
  }
}

// read the data for a problem instance
void readData(String fileName) {
  BufferedReader file = null;
  String line;  // Current input line
  try {
    file = new BufferedReader(new FileReader(fileName));
    // Read bid number and link number on line 1
    line = file.readLine().trim();
    StringTokenizer st = new StringTokenizer(line);
    bidNbr = Integer.parseInt(st.nextToken().trim());
    linkNbr = Integer.parseInt(st.nextToken().trim());
    // Allocate space for bidAlloc[][] and cost[][]
    bidAlloc = new int[linkNbr+1][3];    // link ID starts from 1.
                                         // bidAlloc[0][] is not used
    cost = new int[bidNbr+1][linkNbr+1]; // cost[0][] and cost[][0]
```

```
                                                // are not used
      // Skip the blank line 2
      line = file.readLine();
      // Start to read the bid allocation table
      for (int i = 1; i <= linkNbr; i++) {
        line = file.readLine().trim();
        st = new StringTokenizer(line);
        for (int j = 0; j < 3; j++)
          bidAlloc[i][j] = Integer.parseInt(st.nextToken().trim());
      }
      // Skip the blank line
      line = file.readLine();
      // Start to read the cost table
      for (int i = 1; i <= bidNbr; i++) {
        line = file.readLine().trim();
        st = new StringTokenizer(line);
        for (int j = 1; j <= linkNbr; j++)
          cost[i][j] = Integer.parseInt(st.nextToken().trim());
      }
    }
    catch(Exception e) {}
    finally {
      try {
        if (file != null)
          file.close();
      }
      catch (Exception e) {}
    }
  }

  // print to screen the contents of problem instance data
  // as well as the derived data in cost1[][] and cost2[]
  void printCosts() {
    System.out.println("bidNbr = " + bidNbr + ", linkNbr = " +
                       linkNbr);
    System.out.println();
    System.out.println("Bid Allocation Table:");
    System.out.println();
    for (int i = 1; i <= linkNbr; i++) {
      for (int j = 0; j < 3; j++)
        System.out.print(bidAlloc[i][j] + " ");
      System.out.println();
    }
    System.out.println();
    System.out.println("Cost Table:");
    System.out.println();
    for (int i = 1; i <= bidNbr; i++) {
      for (int j = 1; j <= linkNbr; j++)
        System.out.print(cost[i][j] + " ");
      System.out.println();
    }
    System.out.println();
    System.out.println("Cost1 Table:");
    System.out.println();
    for (int i = 1; i <= linkNbr; i++) {
      for (int j = 1; j <= linkNbr; j++)
```

```
        System.out.print(cost1[i][j] + " ");
      System.out.println();
    }
    System.out.println();
    System.out.println("Cost2 Table:");
    System.out.println();
    for (int i = 1; i <= linkNbr; i++) {
      for (int j = 1; j <= linkNbr; j++)
        System.out.print(cost2[i][j] + " ");
      System.out.println();
    }
    System.out.println();
  }

  // Randomly swap two values of p[1..linkNbr]], and return the updated
  // cost
  int randomSwap(int p[], int cost) {
    int x = r.nextInt(linkNbr) + 1;
    int y;
    do {
      y = r.nextInt(linkNbr) + 1;
    } while (x == y);
    cost = updateCost(p, cost, x, y);
    swap(p, x, y);
    return cost;
  }



  // generate in p[] a random solution
  void randomSolution(int p[]) {
    p[0] = 999;  // p[0] is not used, and 999 is a dummy value
    for (int i = 1; i <= linkNbr; i++)
      p[i] = i;
    for (int i = 0; i < 5*linkNbr; i++) { // randomly sway values in
                                          // p[]
    // swap number is a parameter and can be adjusted; larger the
    // better but slower
      int x = r.nextInt(linkNbr) + 1;
      int y = r.nextInt(linkNbr) + 1;
      swap(p, x, y);
    }
  }

}
```

References

[1] Michael L. Gargano, William Edelson, "Optimal Sequenced Matroid Bases Solved by Genetic Algorihms with Feasibility Including Applications," *Congressus Numerantium* 150, 2001, pp. 5-14.

[2] Joseph DeCicco, "Sensitivity Analysis of Certain Time Dependent Matroid Base Models Solved by Genetic Algorithms," DPS dissertation, CSIS, Pace University, New York, May 11, 2002.

[3] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon, "Optimization by Simulated Annealing: an Experimental Evaluation; Part I, Graph Partitioning," *Operations Research*, vol. 37, issue 6 (Nov.-Dec.), 1989, pp. 865-892.

[4] S. Kirkpatrick, C. D. Celatt Jr., and M. P. Vecchi, "Optimization by Simulated Annealing," *Science* 220, May 13, 1983, pp. 671-680.

[5] F. Glover, "Tabu Search - Part 1," *Operations Research Society of America Journal on Computing*, vol. 1, no. 3, summer 1989, pp. 190-206.

[6] F. Glover and G. A. Kochenberger, *Handbook of Metaheuristics*, Kluwer Academic Publishers, 2003.

[7] D. E. Goldberg, *Genetic Algorithms in Search, Optimization & machine Learning*, Addison-Wesley, 1989.

[8] Michael R. Garey and David S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, New York, 1979.

[9] Wayne L. Winston, *Operations Research: Applications and Algorithms*, 3rd edition, Duxbury Press, 1994.

[10] Jeffrey A. Johnson, "SEPA: A Simple, Efficient Permutation Algorithm," Brigham Young University-Hawaii Campus, http://www.cs.byuh.edu/~johnsonj/permute/soda_submit.html (current November 2003).

[11] Lixin Tao, "Research Incubator: Combinatorial Optimization," CSIS, Pace University, NY, http://csis.pace.edu/~lixin/dps (current November 2003)