uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Wei Xie

AUTEUR DE LA THÈSE / AUTHOR OF THESIS

M.Sc. (Systems Science)

GRADE / DEGREE

Systems Science

FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

RROS Reliability Redundancy Optimization Solver a Microsoft Excel Add-in

TITRE DE LA THÈSE / TITLE OF THESIS

Dr. Tet Yeap

DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

Dr. John Nash

Dr. Jeffrey Sidney

Gary W. Slater

Le Doyen de la Faculté des études supérieures et postdoctorales / Dean of the Faculty of Graduate and Postdoctoral Studies

RROS Reliability Redundancy Optimization Solver
A Microsoft Excel Add-in

Wei Xie

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
In partial fulfillment of the requirements
For the MSc degree in the System Science Program

System Science Program
University of Ottawa

Your file Votre référence
ISBN: 978-0-494-41686-0
Our file Notre référence
ISBN: 978-0-494-41686-0

Abstract

This thesis proposes an implementation of an improved algorithm for solving nonlinear separable integer programming problems, as they arise in the field of system reliability. The reliability redundancy optimization solver (RROS) improves on a hybrid technique of dynamic programming with depth first search with tighter variable bounds. The implementation takes the form of an Excel add-in, which will be familiar to users with little formal OR/MS training. The user interface design is discussed. Example applications with benchmarks show that the RROS offers accurate solutions in a comparatively efficient way. This thesis further introduces a surrogate method to solve nonlinear problems with non-separable objectives, where a separable constraint is used as a surrogate objective. This is shown to yield solutions to a number of difficult cases.

Acknowledgements

I would like to thank Dr.Kevin Y.K. Ng for his guidance during this study.

I would like to thank Dr. Tet Hin Yeap for his great support. Without him, this work would never have arrived at its final stage.

Special thanks to Dr. John Nash and Dr. Jeffrey Sidney for their detailed and significant comments that guided me improving the quality of this thesis.

I had the pleasure of studying with Mr. Marc-Antoine Parent and Mr. Tuan Nguyen, who became my good friends. We all experienced difficult times during the study but we encouraged and helped each other through. I am indebted to Mr. Parent for his great support and help in editing this thesis, I really enjoyed working and studying with him.

Of course, I am grateful to my husband Eric Zhen Zhang, my parents, and parents-in-law for their patience, support and love. I would like to dedicate this thesis to my new born daughter ShinYao Zhang, who brings me faith, motivation and happiness.

Special thanks to my manager Mr. Eric Milligan for his encouragement and support.

Contents

Introduction to Redundancy Optimization

"People from all walks of life have probably suffered one way or another from unreliable products or services" [Bohoris, 1993], such as an appliance not working, or a car failing to start. Thus, determination of an optimal design is very important in order to economically produce new systems which meet or exceed customers' expectation for reliability, quality and performance. The emergence of system reliability in late 1940s has given rise to international industrial competition. Methods to improve system reliability have been developed, and are continuously applied in all fields, including electric power, defence, transportation, telecommunication, and banking.

The term "system" indicates a collection of components performing a specific function. The performance of a system depends on the performance of its components; the importance of the components may differ [Kuo and Zuo, 2002]. The relationship between system reliability and components' reliability is largely determined by the system's structure, such as series structure, parallel structure, series-parallel structure, parallel-series structure, bridge structure, and general complex structure.

System reliability can be improved, among other ways, by adding redundant components into the system, or by increasing reliability levels for some or all of the system components. Both of these reliability enhancement approaches consume resources, such as cost or weight. Furthermore, in many reliability optimization problems, the resource allocation decisions, such

as the number of components in a subsystem, are constrained to integer values. Maximizing the overall reliability by placing redundant components among various subsystems, subject to limited resource constraints, is known as an integer programming problem. The nonlinearity of the reliability functions and the non-separability associated with some system structures represent great challenges in system reliability redundancy optimization.

A common example that can be found in many electrical and mechanical systems and that will be used in this thesis, is an over-speed protection design for a gas turbine system (shown in Figure **??**). This system has five subsystems in a series ($s1, ..., s5$); each subsystem can have multiple identical redundancy components. The objective is to determine the optimal configuration such that the overall system reliability is maximized, subject to design constraints on the cost, weight, and the product of weight and volume.

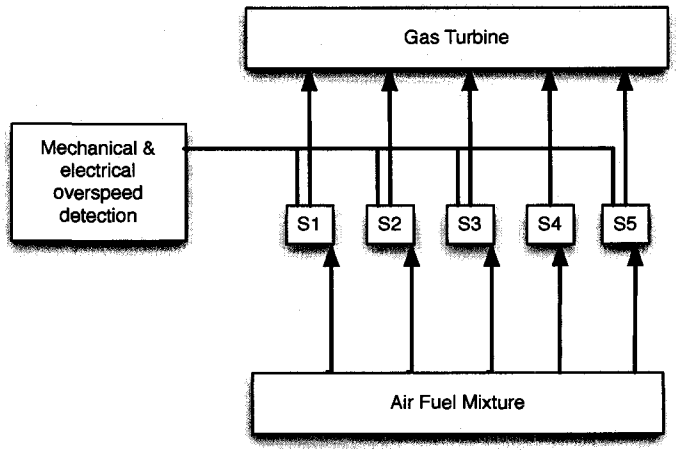

Figure 1: An overspeed protection system for a gas turbine

This thesis is organized as follows. An introduction to reliability redundancy issues, terminology and principal optimization techniques are discussed in Chapter 1. Chapter 2 explains the DP/DFS algorithm, a hybrid algorithm for solving separable integer programming problems [Ng and Sancho, 2001]. It follows by discussing improvement on the algorithm, a

technique called RROS. Further, this thesis introduces the RROS surrogate method for solving reliability problems with non-separable objective functions. Chapter 3 reviews optimization software on the market and describes how to use the implementation of the RROS algorithm in the form of an Excel add-in. It also discusses the main features of the RROS and model implementation. Chapter 4 selects a range of reliability redundancy problems to demonstrate and evaluate the RROS and the RROS surrogate method. Chapter 5 addresses the direction of future studies and concludes this thesis research.

Chapter 1

Reliability Redundancy Optimization

The probability of a system performing a specified function in a designated environment is defined as system reliability. Real-world *systems* can be viewed as integrated collections of components (subsystems or stages), such as hardware, software, or humans, in a specific system configuration (structure). The system exists in an environment and can be influenced by it. The reliability of each subsystem will contribute to the reliability of the whole system [Ireson and Coombs, 1988; Kuo et al., 2001]. The aim of reliability research is to increase customer satisfaction by improving system reliability, which is part of overall system performance.

There are many ways to enhance system reliability, such as improving the reliability of the system's components or using a parallel configuration for less reliable components [Kuo et al., 2001]. Providing redundant components, which are identical or similar to other components and perform the same function, is one option often used in practice. On the other hand, redundancy increases the total cost of the components. Often, reliability redundancy optimization problems are stated as the maximization of system reliability subject to some resource constraints, or the minimization of resource consumption (such as the total cost of system components) while attaining a specific level of system reliability [Majety et al., 1999].

Redundancy may be static or dynamic. Static redundancy elicits immediate corrective action, such as fault masking and error correcting code [Ng and Sancho, 2001]. In dynamic redundancy, standby units can be brought into the system in order to replace the faulty ones. Standby redundancy is said to be cold, warm or hot: cold and warm standby redundancy replaces a problem unit when it fails, or has nearly failed, respectively. Hot standby has a parallel redundancy structure so that all standby units are active simultaneously in the system [Kuo et al., 2001].

The following notation is used in formulating system reliability issues:

	Notation:
X	$x(x_1,...,x_n)$
x_j	number of components at stage j ($1 \leq j \leq n$), a non-negative integer
u_j, l_j	upper bound and lower bound on x_j
R_s or R	system reliability
$r_j(x_j)$	component reliability at stage j
R_0	minimum required system reliability
C , W , P	cost, weight and price constraints
$c_j(x_j)$, $w_j(x_j)$, $p_j(x_j)$	cost, weight and price of x_j components at stage j
j	index of stage
n	number of stages in a reliability system
m	number of resources constraints in a reliability system

1.1 Reliability Model Formulation

A reliability redundancy optimization problem is usually stated in the following form:

Maximize or minimize *Objective function*

Subject to: *Constraints*

This representation contains three elements:

1. The value of *decision variables* $X(x_1, x_2, ..., x_n)$ represents the number of system components at each stage, and their values can be changed to any desired value within each variable's specific operating range.

2. The *objective function* is a criterion measure of the decision variables that is required to be optimized for a better performance [Murty, 1995]. In system reliability, the objective can be maximizing system reliability, or minimizing system cost, weight or price. Any maximization problem can be transformed directly into a minimization problem and vice versa.

   Maximize value of $f(x)$ = $-$(Minimize value of $-f(x)$)

   subject to constraints subject to the same constraints

   In a *single objective* optimization problem, there is only one term which has to be optimized, such as system reliability. However, it is often desirable to simultaneously maximize system reliability and minimize resource consumption in designing a reliable system. Such a *multi-objective* model can be treated as a single-objective one, by choosing one highest priority objective as the one to optimize, or by combining the multiple objectives into a single function.

3. A *constraint* is an essential part of a system reliability problem, such as the system's cost and weight limitations, or a minimal reliability goal. An optimization problem without constraints, which is called an *unconstrained optimization problem*, is much easier to solve than a constrained problem. In some applications, a constrained optimization problem can be solved through transformation into an unconstrained problem.

Solving a reliability redundancy problem involves a process of finding an optimal allocation of system components that minimizes or maximizes the objective function while satisfying the constraints. A solution is considered *feasible* when it satisfies all constraints. An *optimal* solution is feasible solution that has the highest (or lowest, as desired) value for the objective function among all feasible solutions. A *local optimum* solution is a feasible solution which is better than all solutions found in a given region of the problem space, but is still worse than the (global) optimal solution.

Most practical reliability redundancy problems involve nonlinear functions with both objectives and constraints. So they belong to the class of nonlinear integer programming problems, where the solution is limited to integer numbers. Those problems require great computational effort and are much harder to solve than a general linear problem. Chern has noted that even a simple redundancy allocation problem in series systems with linear constraints is NP-hard [Chern, 1992]. The next section introduces concepts of algorithm complexity and effectiveness to explain what this means, before optimization techniques are introduced.

1.2 Algorithm Complexity

An algorithm is a finite step-by-step operation procedure for solving a problem or achieving a result [Sait and Youssef, 1999]. This process may be repeated in an iterative or recursive manner and may eventually terminate. This thesis examines algorithms that are guaranteed to terminate in a finite, well-defined amount of time, using a finite amount of computer memory. *Time complexity* and *space complexity* measure the amount of time and the memory, respectively, that a given algorithm needs to terminate in the worst case. Both measurements are given as functions of a measure of the size of the problem that the algorithm is working on; problem size is traditionally measured as the size of the problem specifications, but this thesis uses simplified

measures, such as the number of variables.

Big-Oh and Big-Ω notations are often used to describe, respectively, the upper bound and the lower bound of the time complexity of algorithms, while Big-Θ notation states a tightly defined growth rate for the algorithm complexity function.

Polynomial-time algorithms and *exponential-time algorithms* were defined in the 1970s to explain the formal properties of algorithms and the inherent complexity of problems. If a problem can be solved within an amount of time that is (in the worst case) a polynomial function of the problem size, where size can be measured in terms of the input length, the number of variables, etc., it is considered to be a class P problem. Similar classes can be defined for other types of functions; for example, some problems require an amount of time that is an exponential function of the problem size.

On the other hand, a class NP problem is defined for a problem where possible solutions are validated; in other words, a program is given the problem, and has to both find and validate an "answer" to that problem. The problem is of class NP if the validation step belongs to the P class. The N in the name NP refers to non-determinism; that is, if the program could "guess" the answer correctly, and validate in polynomial time, it would be done. In practice, the search for the correct answer is the time-consuming part, and requires a combinatorial evaluation that is an exponential function of the input size. So NP problems belong to the exponential time class[Hopcroft and Ullman, 1979]. Clearly, $P \subseteq NP$; but it has not yet been proven that $P \neq NP$. Though a majority of researchers in the field of computational complexity theory believe this to be the case, the possibility that $P = NP$ has not been ruled out.

Some problems can be reduced to another problem through a simple (polynomial-time) transformation. Some problems are such that any NP problem can be reduced to them; those are known as $NP - hard$. If those problems are themselves NP, they are known as

NP – complete. Many classes of integer programming, which this thesis studies, are *NP – hard*. As *NP – hard* problems require very significant computational effort, many researchers have applied approximation methods to optimization problems. Those methods can find adequate sub-optimal solutions in polynomial time. [Carter and Price, 2001; Kuo et al., 2001].

1.3 System Configuration and Reliability Functions

A reliability system normally has more than one component. On the other hand, the problem complexity may vary, even if the size of the system remains the same, since components can be arranged in different ways (structures). Reliability system structures can be mainly catalogued as series systems, parallel systems, series-parallel systems, parallel-series systems, hierarchical series-parallel systems (HSP), k-out-of-n systems (G or F system), and general network systems (not pure series nor pure parallel systems) [Kuo et al., 2001; Ng and Sancho, 2001]. Various unspecified systems also exist in which the system's structure is not explicit. This thesis initially focuses on reliability redundancy issues with series-parallel or with k-out-of-n structures. A surrogate relaxation method is used later to study complex systems including bridge networks.

1. Series systems (Figure 1.1):

   A system with a series configuration fails if any of its n components fails. Hence the system reliability, the probability that the system will operate successfully, is:

   $$R_s = \prod_{j=1}^{n} r_j \tag{1.1}$$

   where r_j is the reliability of an individual component.

2. Parallel systems (Figure 1.2):

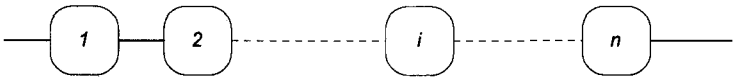

Figure 1.1: A series system

A system with a parallel configuration consists of n components in parallel; the system is successful if any one of its n components is successful. A parallel configuration is also called a redundant configuration [Kuo et al., 2001]. The reliability of a parallel system is:

$$R_s = 1 - \prod_{j=1}^{n}(1 - r_j) \tag{1.2}$$

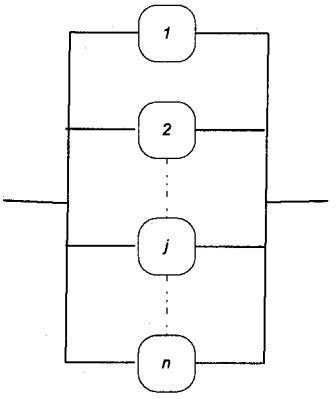

Figure 1.2: A parallel system

3. Series-parallel systems (Figure 1.3):

   A system with a series-parallel configuration consists of k subsystems connected in parallel. Subsystem i has n_i components in series ($i = 1, ..., k$). The system reliability can be represented as:

$$R_s = 1 - \prod_{i=1}^{k}(1 - \prod_{j=1}^{n_i} r_{ij})$$

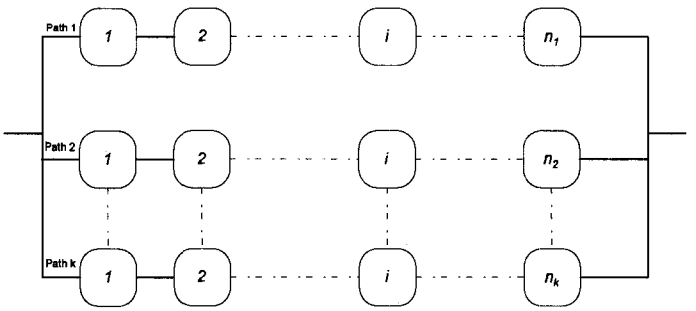

Figure 1.3: A series-parallel system

When the components are identical in each series subsystem, the above system reliability can be formulated as:

$$R_s = 1 - \prod_{i=1}^{k}(1 - r_i^{n_i})$$

4. Parallel-series systems (Figure 1.4):

A parallel-series system consists of k subsystems in series. Subsystem i has n_i com-

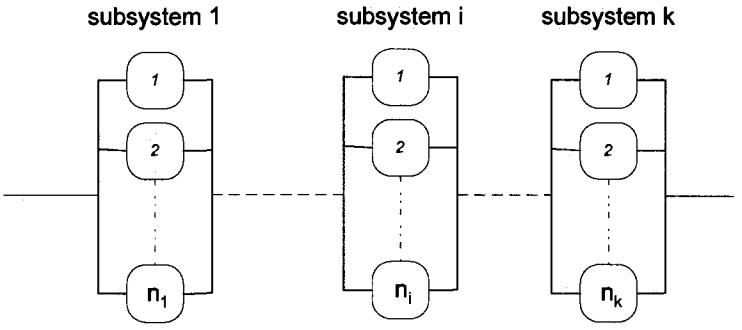

Figure 1.4: A parallel-series system

ponents in parallel. Let r_i be the reliability of subsystem i and r_{ij} be the reliability of component j in subsystem i, $1 \leq j \leq n_i$. Subsystem i has reliability $r_i = 1 - \prod_{j=1}^{n_i}(1 - r_{ij})$ and the parallel-series system reliability is :

$$R_s = \prod_{i=1}^{k}[1 - \prod_{j=1}^{n_j}(1 - r_{ij})]$$

If all components in subsystem i are identical ($r_{ij} = r_i$ for $i = 1,...,k$), then the system reliability is

$$R_s = \prod_{i=1}^{k}[1-(1-r_i)^{n_j}] \simeq 1 - \sum_{i=1}^{k}(1-r_i)^{n_j}$$

[Kuo et al., 2001]

5. K-out-of-n systems:

   A k-out-of-n system requires at least k of its n components to work so that the system can function. This system is also called a *k-out-of-n: G system.* A *k-out-of-n: F system* defines a system which fails whenever at least k of its n components fail. Series and parallel systems are special cases of a k-out-of-n system. For example, a series system is an *n-out-of-n: G system,* while a parallel system is a *1-out-of-n: G system.*

   It is often assumed that the system is able to detect failure and switch components smoothly, as the $(n-k)$ redundant components are either active or in a standby mode. When all the system components are independent and identical with the same reliability r, the reliability of a general k-out-of-n can be written as:

$$R_s = \sum_{j=k}^{n}\binom{n}{j}r^j(1-r)^{n-j}$$

6. Hierarchical series-parallel systems (HSP)

   A HSP system can be viewed as a hierarchical combination of subsystems with a series or parallel configuration (structure). Figure 1.5 illustrates a five-component hierarchical series-parallel system. The reliability of each subsystem can be computed from its subsystem by applying Equations 1.1 and 1.2. The system reliability of Figure 1.5 can be expressed as:

$$R_s = 1 - [1 - r_1(r_2 + r_3 - r_2r_3)](1 - r_4r_5)$$

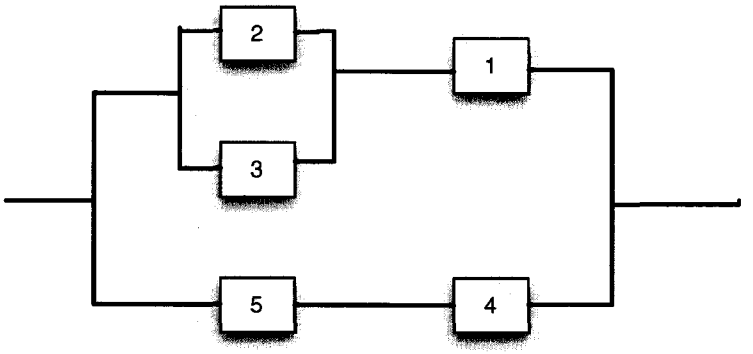

Figure 1.5: Five-component hierarchical series-parallel system.

7. General network systems

   Many reliability systems, such as bridge network systems, can't be represented by series or parallel configuration or by their combinations. These systems are called non-parallel-series systems, or general network systems. Formalizing the reliability of such a general network system is dependent on the system structure. Two examples of general network systems are studied in Chapter 4.

1.4 Optimization Approaches for Reliability Redundancy Optimization

How to find an optimal trade-off efficiently among reliability, cost and other constraints is still an open question. Methods for solving reliability redundancy optimization can be catalogued as *exact solution methods* and *approximation methods*. Kuo and Prasad [2000] presented a summary of the research efforts in this area. Exact solution methods for reliability problems include integer programming, the branch-and-bound method, dynamic programming, and implicit enumeration. For many small-sized redundancy allocation problems, exact methods are able to provide accurate solutions. However, exact solution methods require much computational

effort and memory when the size of the problem increases.

1.4.1 Integer Programming

Integer Programming (IP) problems [Carter and Price, 2001; Trick, 1996] are limited to having integer solutions for decision variables. Most IP methods require that the objective and constraints in a problem be separable, either directly or by transformation, as shown here:

$$\begin{aligned} \text{Minimize} \quad & z = c^T x \\ \text{Subject to:} \quad & Ax \leq b \\ & x \geq 0 \\ & x \in \mathbb{Z}^n \end{aligned}$$

where $\mathbb{Z}^n$ is the set of n-dimensional integer vectors. There are three basic types of IP problems: *pure integer programming, 0-1 problems* and *mixed integer programming*. In a pure integer programming problem, all the decision variables must have positive integer values. If all the decision variables are restricted to zero or one, the problem is called a 0-1 programming problem. When some of decision variables must have integer values and others may hold real values, this problem is described as a mixed integer programming problem.

In general, the formulation of IP problems is similar to that of continuous mathematical programming problems; however, the additional constraint requiring that one or more variables have an integer value makes the problem dramatically more difficult. Most IP problems, especially with nonlinear objective functions, are classified as hard problems [Carter and Price, 2001]. While a general linear programming problem (LP) may be solved in polynomial time, finding an optimal integer solution to the same formulation usually requires an exponential amount of computation time. The next section describes one of the methods used to ease computation of linear programming problems in the system reliability domain.

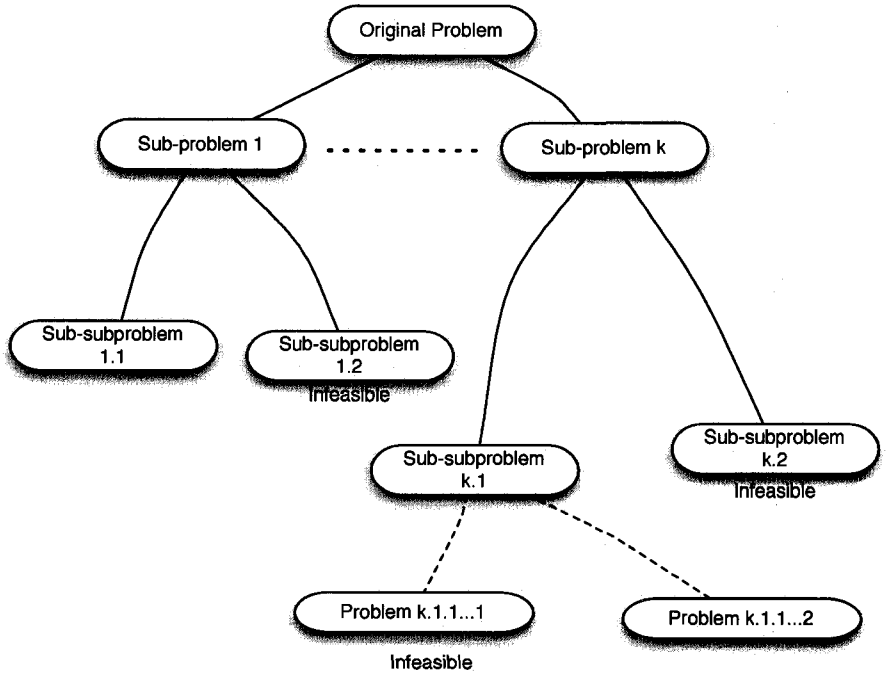

Figure 1.6: A general branch-and-bound tree

Branch-And-Bound

The branch-and-bound technique applies a *divide-and-conquer* strategy to divide a feasible solution region (search space) successively in sub-regions. When the sub-problem defined by the sub-region is still too hard to be solved easily, the splitting process (branch) is repeated until each sub-problem can be solved easily, that is to say, branching is done when all sub-problems either are solved or have no feasible solution [Carter and Price, 2001]. The optimal solution for the original problem is the best feasible solution that is found among all sub-problems.

The containment of the sub-regions naturally forms a tree structure [Hillier and Lieberman, 2005], as shown in Figure 1.6. Each node also corresponds to a sub-problem, and each arc represents a restriction on one of the integer variables (such as $a \leq x \leq b$). That way, the integer solutions are themselves implicitly located in the tree structure, and so the branch-and-bound technique belongs to the larger class of implicit enumeration techniques. The branch-and-bound

technique is often applied to medium-sized general IP problems.

A real problem may have many different branch-and-bound subdivisions; however, a good algorithm will try to find the optimal solution as quickly as possible. The choice of branching strategies (backtracking or jumptracking), bounding strategies and separation rules is critical for algorithm's performance [Carter and Price, 2001].

Three policies are often used to create sub-problems in the branch-and-bound technique [Chinneck, 2003]:

1. A best-first search policy involves choosing the sub-tree where, according to some heuristic, the optimal point is likely to be on the sub-tree;
2. A depth first-search policy only works down the branch created most recently within the given bounds, in order to find a solution quickly;
3. A breadth-first search policy involves expanding sub-problems in the order in which they were created.

The branch-and-bound technique is also used as a basis for various heuristics algorithms to solve NP-hard problems. For example, one may stop the branching process when the gap between the upper and lower bounds becomes smaller than a certain value. This can save significant computational effort.

1.4.2 Dynamic Programming

Dynamic Programming (DP) provides a systematic procedure that decomposes a n-variable problem into n interdependent single-variable sub-problems and solves each sub-problem in

sequence, in such a way that the optimal solution to the original problem can be found [Taha, 1997; Carter and Price, 2001].

R. Bellman developed this method into a systematic tool and pointed out its broad scope in the 1950s . Bellman [1957], and later Bellman and Dreyfus [1958, 1962] first applied DP to maximize system reliability with a single cost constraint. Fyffe, Hines, and Lee [1968] later used a Lagrangian multiplier (λ) within the objective function to combine all the problem constraints into one; they also used the DP method to solve a 14-subsystem reliability problem with both cost and weight constraints. A decade later, Nakagawa and Miyazaki [1981] developed an alternative mean of combining all the constraints by using a surrogate constraint method instead of a Lagrangian multiplier. Today, the DP technique is widely applied in many discrete deterministic optimization problems, including the shortest path problem, the resource allocation problem, and the production plan and control problem, to list only a few.

The main feature of DP is the *principle of optimality* [Bellman, 1957]:

> *"An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision."*

The principle of optimality assumes that an optimization problem can be decomposed into finite stages which are independent; i.e. such that the immediate objective value depends only on the current stage and on the current decision, not on the previous stages. This is a direct and simple consequence of the Markovian property [Hillier and Lieberman, 2005], and in particular it is true in the case where the objective function is separable, i.e. the sum or product of the immediate objective function value incurred at each stage. Figure 1.7 shows the path that is formed by the sequence of decisions made at each stage.

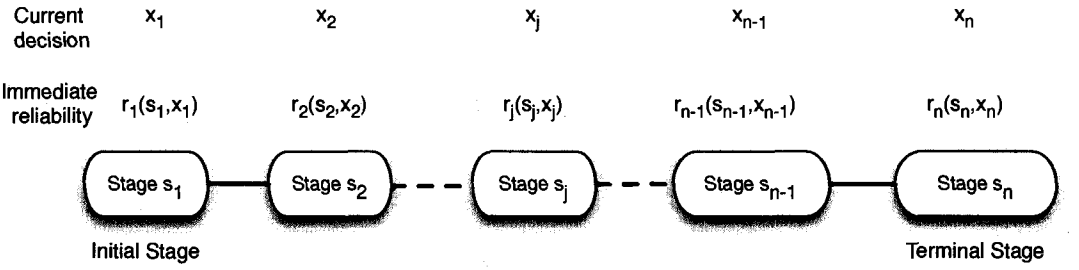

Figure 1.7: The path of a sequential decision process. The objective of this path is $\prod r_j$.

The structure in Figure 1.7 is identical to the structure of the series systems in Figure 1.1. Because of this stage independence, the DP approach can be a good candidate for solving reliability problems with a series configuration. In a series system, the principle of optimality implies that, if s_j is a stage in an optimum path and is obtained by seeking an optimum policy beginning with an initial state s_0, then an optimum path from s_0 to the end s_n will be composed from this path and another optimum path that begins with stage s_j and ends at s_n.

More formally, let x_j be a decision at stage s_j, so that there are $n-j$ stages remaining in the path, where x_j is chosen among a set of possible states, $x_j \in X_j = \{l_j \ldots u_j\}$. The immediate (local) reliability value of choosing x_j at stage s_j is represented by $r_j(s_j,x_j)$. One can represent the total reliability (objective function value) for the last $n-j$ stages (from s_j to s_n) as

$$R_j(s_j,x_j) = \prod_{i=j}^{n} r_i(s_i,x_i) \tag{1.3}$$

The optimal (maximal) reliability for these $n-j$ stages is $R_j^*(s_j) = R_j(s_j,x_j^*)$ where $x_j^* \in X_j$ is the optimal solution for the state s_j. According to the principle of optimality, $R_j^*(s_j)$ is determined only by the decision $r_j(s_j,x_j)$ at current stage s_j and the optimal reliability R_{j+1}^* from succeeding stages (s_{j+1} to s_n), so

$$R_j^*(s_j) = \max_{x_j \in X_j} \{r_j(s_j,x_j) * R_{j+1}^*(s_{j+1})\} \tag{1.4}$$

The optimal solution of the final stage is given by the simpler equation

$$R_n^* = \max_{x_n \in X_n} \{r_n(s_n, x_n)\} \tag{1.5}$$

Working backward from the final stage R_n^* to the initial state R_1^* using backwards recursion, the optimal reliability for the n-stage problem can be found.

A DP procedure can also be performed in a forward direction, where the computation process starts from the first stage and the recursion is based on completed decision stages. Both forward and backward recursions can produce the same optimization solution. However, most DP applications use backward recursion because it is more efficient computationally [Taha, 1997].

DP has provided an excellent procedure for reliability problems with separable objective functions and constraints. On the other hand, DP alone is not generally used for reliability optimization in complex systems, which have non-convex and non-separable objective functions, due to the expensive computational effort it requires. Moreover, for the same reason, optimization problems with more than two constraints are harder to solve by DP.

1.4.3 Approximation Algorithms

Approximation algorithms are generally classified according to two methods: heuristic and meta-heuristic. Almost all heuristic methods begin with a feasible solution, and yield at each step a feasible solution with an improved objective function value. This process is repeated iteratively until the current solution is accepted as close enough to optimal. Optimization methods in this family are more efficient than the exact optimization methods discussed earlier in this chapter, especially for large-sized nonlinear integer programming problems. However,

quite often they will not yield the optimal solution, but only a satisfactory solution (usually a local optimum).

Meta-heuristic methods provide a high-level framework which use a heuristic method as a guiding step in a generalized search strategy. The following sections describe three well-known meta-heuristic methods: genetic algorithms, simulated annealing, and tabu search.

Genetic Algorithms

Genetic Algorithms (GA) imitate a biological evolutionary process, where solution features are inherited and improved through the parent-children relationship [Obitko, 1998; Coit and Smith, 1995]. This type of algorithm carries out a multi-dimensional search by generating and selecting multiple solutions at each stage. The GA approach is flexible and can deal with both discrete and continuous optimization problems [Kuo et al., 2001]. It is also better suited to investigating large-sized NP problems than the DP and IP approaches. On the other hand, GA may not be able to find the optimal solution for some reliability optimization problems. Also, an effective GA depends on the setting of search parameters, such as crossover and mutation operators; settings that lead to an efficient search often have to be found experimentally [Obitko, 1998].

A genetic algorithm starts with a set of feasible or infeasible solutions (first generation) in an initial population. The motivation to form a new generation is that the new generation should be better than the old one. Not all solutions of the current generation can form offspring; each solution in a given generation must be evaluated by fitness tests, which form the main heuristic step. Through these tests, some solutions are selected as "parents" for the next generation, while others are discarded [Obitko, 1998; Kuo et al., 2001]. Then, crossover and mutation operators generate a new set of solutions (subsequent generation) derived from the selected

parents. The crossover operator impacts the rate of convergence, and the mutation operator prevents the algorithms from converging prematurely. This reproduction process is conducted iteratively, and stops when some stopping criteria are satisfied. Coit and Smith [1995] has adapted this method to solve series-parallel reliability redundancy allocation problems.

Simulated Annealing

Simulated Annealing (SA) takes an analogy between the search for a minimum in an optimization problem and the annealing process in which a melted solid slowly cools down to a minimum energy level. This slow and gradual cooling process allows the molecules in the solid at each temperature to experience many random transitions and find their low energy levels until the solid attains a thermal equilibrium. This thermal equilibrium has been described mathematically as a Boltzmann distribution: when the temperature decreases, the probability that the solid can remain in a state with a higher energy level decreases.

SA simulates this annealing process. A genetic SA algorithm starts from an initial solution and computes the initial value of the objective function. By using a random transition from the initial solution, SA generates (one or many) new solutions and computes the new objectives. If the objective value of the new solution(s) is better than the previous one(s), SA accepts the new solution(s). Otherwise, the new solution(s) still can be accepted based on a given probability (which corresponds to the temperature of the annealing process). SA iterates the process, slowly decreasing the temperature, until the stopping criteria are satisfied.

The annealing schedule, which describes the probability that an unimproved solution will be accepted, is critical to the algorithm's performance. A suitable annealing schedule has an initial temperature high enough to allow to cover the search space and avoid being trapped in a sub-optimal local optimum region (this ability would correspond to a "melting point" of the

system), and proceeds to a lower temperature so as to lock into a region around and optimum, and converge to it with precision at the "freezing point".

Compared with other meta-heuristic methods, SA has the advantage of being able to avoid sub-optimal local optima, thanks to the ability to accept possible solutions which do not improve the objective value based on probability. On the other hand, the performance of a SA algorithm depends on the appropriateness of its neighbourhood structure, of its data structure, and its cooling schedule and parameters [Kuo et al., 2001]. All of these require skill and effort from the user to implement the algorithm.

Cardoso et al. [1994] developed the NESA (Non-Equilibrium Simulated Annealing) algorithm to improve the rate of convergence and reduce the computation time. With NESA, an equilibrium condition is not required; in other words, the temperature can be reduced while an improved solution is obtained. Ravi et al. [1997] employed the NESA technique to solve bridge, mixed series-parallel and complex reliability problems.

Tabu Search

Tabu search (TS) is a local search technique that uses memory structures to enhance search performance. A TS algorithm begins with an initial feasible solution, and then moves to a better solution (closer to optimal) in the neighbourhood of the previous (or initial) solution. At each step, a new search neighbourhood is computed around the new solution, which excludes solutions in the *tabu list*. This process is repeated until some stopping criteria are satisfied. A tabu list usually includes solutions that have been visited in recent moves (previous memory). Some tabu list structures also prohibit solutions with certain attributes and label the attributes tabu-active. Potential solutions that contain tabu-active attributes are then excluded in the new neighbourhood.

TS can be a very effective search technique, as attributes-based tabu lists exclude many solutions. On the other hand, forbidden attributes make a good number of solutions tabu, some of which might be of excellent quality. To avoid missing such good solutions, *aspiration criteria* are used to override the tabu state and include such solutions in the allowed set. A commonly used aspiration criterion will change the label of an attribute from tabu-active to non-active if there is a solution with that attribute which is closer to optimal than the best solutions found so far [Kuo et al., 2001].

1.5 Relaxation Techniques in Integer Programming

Everett [1963] proposed a Generalized Lagrange Multiplier (GLM) method to relax a problem with some complicated constraints. Greenberg and Pierskalla [1970] then introduced "surrogate mathematical programming", which uses the GLM framework to construct a surrogate objective function. Glover [1975] also presented the surrogate duality theory to improve the standard duality approaches and compare them with Lagrangean duality theory. Magee and Glover [1996] provided a summary of the Lagrangean relaxation, Lagrangean dual techniques and surrogate duality methods that are applied in the Integer Programming field.

1.5.1 Relaxation in General

The motivation to relax a hard problem, such as an Integer Programming problem, is that a relaxed problem may be much easier to solve. The relaxed problem often widens the search boundary, and allows additional solutions that improve the original objective value, without disallowing any of the original solutions. Thus, the optimal objective of the relaxed problem can provide an upper bound for the original problem, if the objective is to be maximized; or

provide a lower bound for the original problem, if the objective is to be minimized.

The basic relaxation strategy is to identify a relaxed problem that has the characteristic of being much easier to solve than the original problem, of being tight (strong) enough to provide good solution candidates for the original problem, or at least of being able to provide a good bound for the original objective.

There are three common conditions that occur when relaxation techniques are applied [Magee and Glover, 1996].

1. If there is no feasible solution for the relaxed problem, there is no feasible solution for the original problem.

2. The optimal objective value for the relaxed problem is able to provide an "optimistic estimate" for the optimal objective value in the original problem. It is also able to provide a bound for the original objective.

3. The optimal solution for the relaxed problem can be the optimal solution for the original problem, in the case where it is feasible in the original problem and it provides the same objective values for both problems.

The relaxation techniques such as Lagrangean relaxations can be applied to reduce restrictive constraint(s) on a hard problem, or to relax an objective function. A relaxed objective function gives an optimistic estimate for the original objective value. Any objective function that allows Condition 2 above qualifies as a relaxed objective. In Condition 3, if the optimal solution from the relaxed objective provides the same objective value, then it is the optimal for the original problem. This can be applied to the problems even when the relaxed objective is different from the original objective [Magee and Glover, 1996].

1.5.2 Lagrangian Relaxation

Lagrangian Relaxation is a specific method to relax the complicated (hard) constraints of an original problem and to generate a set of relaxed (easy) problems. The solutions to the relaxed problems are different from the solutions to the original problem. However, these solutions can represent the upper bounds of the original maximization problem [Magee and Glover, 1996]. Consider the following linear integer programming problem:

$$\begin{gathered}\text{Max } \sum_{j=1}^{n} c_j x_j \\ \text{Subject to: } Ax \leq b \\ Dx \leq e \\ x \in \mathbb{Z}^n\end{gathered}$$

There are two groups of inequality constraints: assume that $Ax \leq b$ is an easy constraint, while $Dx \leq e$ (i.e. $\sum_{j=1}^{n} d_j x_j \leq e$) is hard to satisfy as a constraint. One way to make this integer problem easier to solve is to incorporate the hard constraint into the objective function as a *penalty* term. It is then possible to ignore it as a constraint. [Carter and Price, 2001]. Ignoring the hard constraints $Dx \leq e$ and adding a penalty to the objective is described as saying that the hard constraints are dualized [Magee and Glover, 1996]. The following relaxed objective $L(x,u)$, which includes a dual price u, defines a *Lagrangian* relaxation.

$$\begin{gathered}\text{Maximize } L(x,u) = \sum_{j=1}^{n} c_j x_j - u(\sum_{j=1}^{n} d_j x_j - e) \\ \text{Subject to: } Ax \leq b,\ u \geq 0,\ x \in \mathbb{Z}^n\end{gathered}$$

Suppose a fixed positive penalty value is set for u; the Lagrangian function $L(x,u)$ can be rewritten as a function of x:

$$\begin{aligned}\text{Maximize} \quad & L(x,u) = \sum_{j=1}^{n} (c_j - ud_j)x_j + ue \\ \text{Subject to:} \quad & Ax \leq b,\ u \geq 0,\ x \in \mathbb{Z}^n\end{aligned}$$

The original hard problem has been transformed into an easy integer programming problem for any fixed value for u. It is obvious that finding a small u can maximize the Lagrangian function. As a result, a "min-max" problem to find the optimal $L(x,u)$ can be stated as:

$$\begin{array}{lll} \text{Minimize} & \text{maximize} & L(x,u) \\ u \geq 0 & & x \in \mathbb{Z}^n \end{array}$$

Then, the above "min-max" problem can be solved for any fixed value of u. u is initially set with a small value, such as 0; the resulting solution x is tested against the hard constraints $Dx \leq e$. Any constraint violation triggers an incremental increase in u. The Lagrangian relaxation is then updated with a new u, and a new solution x to the Lagrangian problem is generated. This iteration process continues until a solution is found where all the constraints are satisfied.

This constraints relaxation technique can be used to bound sub-problems in branch-and-bound, to provide a worst-case bound on heuristic solutions, or to provide a relaxation solution (near feasible) to initialize a heuristic search [Magee and Glover, 1996].

Chapter 2

The RROS Algorithm and Surrogate Method

This thesis proposes an optimization tool, called "reliability redundancy optimization solver" (RROS) for a class of nonlinear integer optimization problems with separable objective functions. The RROS integrates and extends the hybrid "dynamic programming/depth-first search" (DP/DFS) algorithm [Ng and Sancho, 2001], with an upper-bound technique [Prasad and Kuo, 2000]. This combined algorithm is implemented in a Microsoft Visual Basic Application environment.

This thesis further develops a unique surrogate method to extend the scope of the RROS application. This way, an optimization problem with non-separable objective function can be converted into a surrogate problem, which can be solved by the RROS application.

This chapter firstly explains the DP/DFS algorithm, and discusses the RROS's improvement on the existing algorithm. An example is used to demonstrate the RROS process. This chapter then explains the RROS surrogate method, and finally provides an example to describe its

procedures.

2.1 The DP/DFS algorithm

The hybrid "dynamic programming/depth-first search" (DP/DFS) algorithm was originally developed by Ng and Sancho and applied in LINGO 6 to solve reliability redundancy problems. The DP/DFS algorithm guarantees to find an optimal solution (if one exists) for a class of nonlinear integer problems with separable and monotonic objective functions [Ng and Sancho, 2001].

The DP/DFS algorithm chooses a key constraint from the original constraints and treats this constraint as a knapsack constraint to apply the DP technique. This yields an upper bound for the objective function, which helps constrain a depth-first search for near-optimal solutions. These solutions are ranked and the best one which also satisfies the remaining constraints is the optimal solution to the original problem. If no feasible solution is found in the range, then the search depth will be iteratively marked up. The choice of the key constraint will impact the size of the search. Ng and Sancho [2001] apply the DP to all constraints so as to find the tightest upper bound to constrain the search.

The DP/DFS algorithm deals with the following separable nonlinear integer programming problem:

$$max \prod_{j=1}^{n} r_j(x_j),\ where\ 0 \leq r_j(x_j) \leq 1, x \in \mathbb{N}^n \tag{2.1}$$

or

$$max \sum_{j=1}^{n} r_j(x_j),\ where\ 0 \leq r_j(x_j),\ x \in \mathbb{Z}^n \tag{2.2}$$

Subject to:

$$\sum_{j=1}^{n} P_{ij}(x_j) \leq d_i, \qquad i = 1,2,\ldots,L_1 \tag{2.3}$$

$$\sum_{j=1}^{n} Q_{kj}(x_j) \geq c_k, \qquad k = 1,2,\ldots,L_2 \tag{2.4}$$

The DP/DFS algorithm assumes the following conditions [Ng and Sancho, 2001]:

1. d_i and c_k are non-negative constants for all i's and k's;
2. $r_j(x_j)$ is a non-decreasing function;
3. $P_{ij}(0) = Q_{kj}(0) = 0$;
4. $P_{ij}(x_j)$ and $Q_{kj}(x_j)$ are monotonic increasing functions of x_j which approaches ∞ as $x_j \longrightarrow \infty$.

The third and the fourth conditions require that all the constraints of the research problem are monotonic; so that the dynamic programming technique can be applied.

The following algorithm is described for a problem whose objective function is in additive form (System 2.2). The algorithm can also be applied to a problem whose objective is in product form, by taking the logarithm on the object function and thus converting it into an objective function in additive form.

The DP/DFS algorithm is illustrated in Figure 2.1, and the steps are described as follows [Ng and Sancho, 2001]:

A: Determine upper and lower bounds on all decision variables from the system constraints.

B: For each constraint i in System 2.3, define a relaxed knapsack problem with the objective (System 2.2), then calculate the Dynamic Programming solution $f_i(d_i)$.

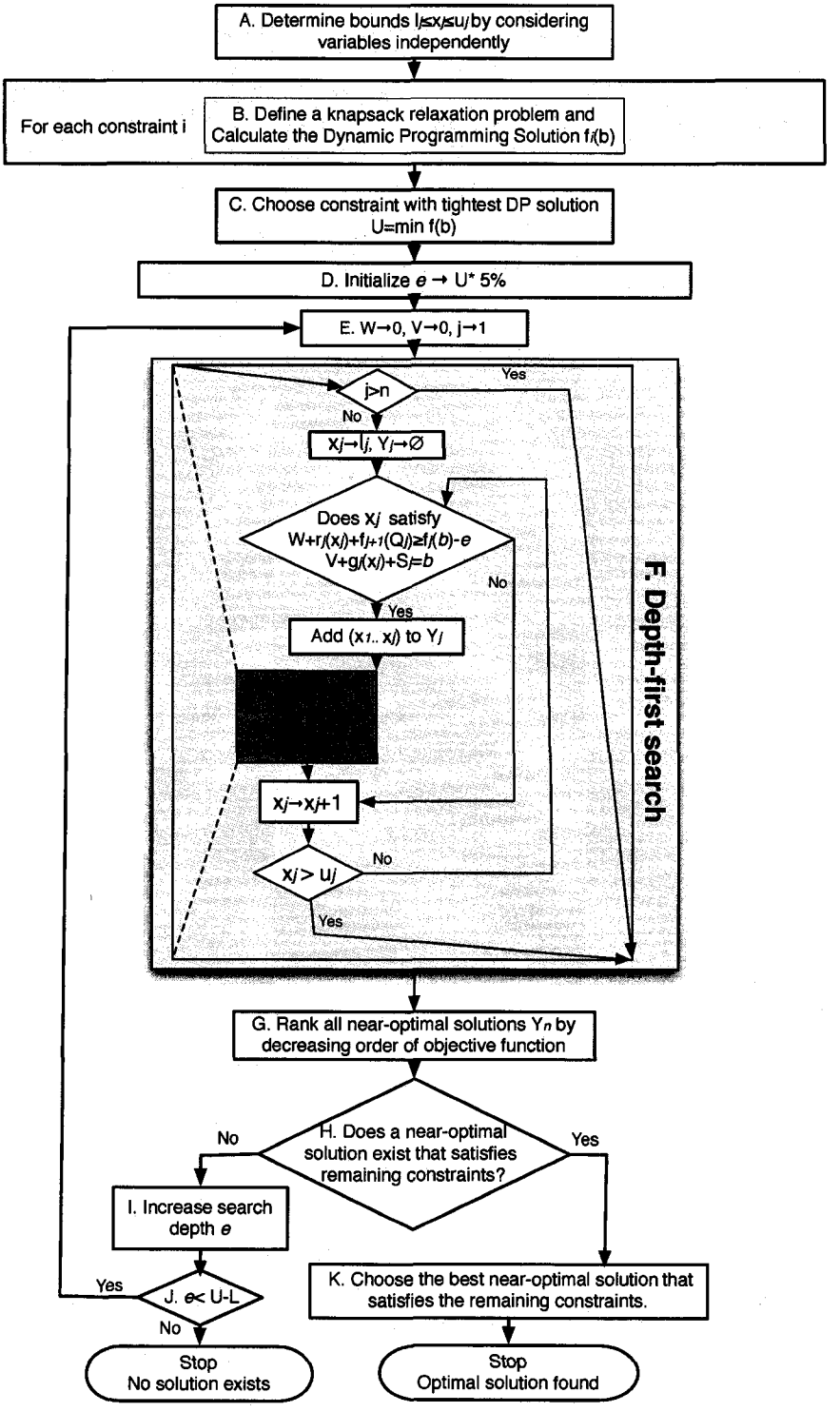

Figure 2.1: Flow chart of the DP/DFS algorithm

C: Select the constraint whose DP solution has the tightest (smallest) objective function value as the key constraint. For example, the selected key constraint (System 2.5 from System 2.3) together with the objective function (System 2.2) creates a knapsack relaxation problem where one tries to maximize System 2.2 subject to constraint i (from System 2.3):

$$\sum_{j=1}^{n} g_j(x_j) \leq d_i = b \tag{2.5}$$

If $f_j(T)$ is the maximum return from stages j to n for T resources available (T being bounded by b), then using the principle of optimality yields the recursive relation

$$f_j(T) = \max_{x_j \in \{l_j \ldots u_j\}} [r_j(x_j) + f_{j+1}(T - g_j(x_j))] \tag{2.6}$$

for $j \in \{1 \ldots n\}$, with $f_{n+1}(T)$ is defined to be 0. Then $f_1(b)$ is the DP solution for Systems 2.5 and 2.2.

D: Initialize the search depth (e)

Let $U = f_1(b)$ represent the maximum objective value of the relaxed knapsack problem (Systems 2.2 and 2.5). The available search depth is then bracketed by the maximum value U and the minimum value of the knapsack problem. The latter is determined by the level of lower bounds to the decision variables. However, it is not efficient to enumerate all the possible solutions in the above range, so the DP/DHS introduces an arbitrary lower bound ($U - e$) to improve algorithm performance. For example, the search depth e can be selected to be $5\% * U$.

E: Introduce temporary intermediate variables V and W, initialized to zero.

F: Depth-first-search (recursion) Apply depth-first search technique through recursion on

the number of decision variables (j represents the current number of variables, and the recursion depth).

At each recursion (when $j > 1$), start with a value $(x_1 \ldots x_{j-1})$ in the set of solutions Y_{j-1} defined in the previous recursion step, then find all integers x_j within the bounds $(l_j \leq x_j \leq u_j)$ that satisfy

$$V + g_j(x_j) + S_j = b, \tag{2.7}$$

$$W + r_j(x_j) + f_{j+1}(Q_j) \geq U - e \tag{2.8}$$

In Equation 2.7, S_j is the spare in the knapsack after packing it with items x_1 through x_j, and Q_j is the largest integer less than or equal to S_j that satisfies System 2.8.

Those x_j that do not satisfy Systems 2.7 and 2.8 are discarded at this step. For each remaining x_j, add $(x_1 \ldots, x_{j-1}, x_j)$ to the set Y_j (solutions defined at level j), and apply the recursion to that value using the following updated values for V and W: $V \rightarrow V + g_{j-1}(x_{j-1})$, $W \rightarrow W + r_{j-1}(x_{j-1})$.

The recursion is ended when all variables have been exhausted ($j > n$).

G: Rank the objective values ($\sum r_j(x_j)$) in decreasing order by implementing all possible solutions $(x_1, x_2, \ldots, x_n) \in Y_n$.

H,K: If at least one solution $(x_1, x_2, \ldots, x_n) \in Y_n$ exists that satisfies all the constraints, then the optimal solution is given by the best objective value that satisfies the remaining constraints (System 2.3 and 2.4).

H,I,J: If no $(x_1, x_2, \ldots, x_n) \in Y_n$ exists that satisfies all the constraints and $e < (U - L)$, then incrementally increase the value of e, and repeat Step E.

J: If $e \geq (U - L)$, the search has been exhausted without finding solutions to the problem; therefore, no solution exists.

The above algorithm finds all near-optimal solutions for objective function values in the interval $[U-e,U]$.

2.2 The RROS algorithm

The RROS algorithm follows the structure of the above DP/DFS technique, but improves slightly on some of its steps, in particular: determining tighter upper bounds for decision variables, constraining initial search depth, and selecting the key constraint heuristically.

- In step A, the RROS employs a technique developed by Prasad and Kuo [2000] to calculate tighter initial upper bounds of the decision variables. Specifically, this technique applies System 2.9 to each less-than-or-equal type of constraints (including equal, less-than, and less-than-or-equal constraints) in System 2.3.

$$x_j \leq \min\{u_j, \max[k : P_{ij}(k) - P_{ij}(l_j) \leq d_i - \sum_{j=1}^{n} P_{ij}(l_j), 1 \leq i \leq L_1]\} \qquad (2.9)$$

  If the upper bounds are already given and they are tighter than the calculated ones, the RROS will take the given bounds as upper bounds for the decision variables.

  The RROS also initializes the lower bounds, if the lower bounds are not given. The lower bounds of all decision variables will be set to 0 if the objective function is in additive form (System 2.2) or to 1 if it is in product form (System 2.1).

- In step D, the DP/DFS selects the initial search depth at a level of $5\% * U$. This setting does not consider the lower bounds on objective value $\sum_{j=1}^{n} r_j(l_j)$, which can be calculated by the lower bounds of the decision variables. If the available search depth, bracketed by the upper and lower bounds on objective value, is less than the programmed one (e), the depth-first-search technique that employed in the DP/DFS will be as costly as a

full enumeration practice. Therefore, the RROS algorithm constrains the initial search depth by this known lower bounds on objective value, specifically, $e = \min[0.05 * U, U - \sum_{j=1}^{n} r_j(l_j)]$.

- In step C, the DP/DFS selects the constraint with the smallest DP solution as key constraint. The choice of key constraint has very high impact on the efficiency of the depth-first search in step F. The question of a procedure for constraint choice is not explicitly mentioned in Ng and Sancho [2001], which give results for all cases. However, they state that one should use the tightest of the key constraints, which suggests that DP should be applied to all constraints and the constraint that gives the fewer upper-bound should be selected.

  The heuristic used in the RROS is simply to choose the constraint with the least upper bound. This choice minimizes the time spent in the DP part of the algorithm, possibly at the expense of time spent in the DFS part of the algorithm. However, since the RROS has tighter bound calculations in step A, less options will be examined in the DFS overall, and the impact of that choice is mitigated.

2.2.1 An example application of the RROS

[**Problem** $S1$]: The turbine example mentioned in the introduction is used to demonstrate the RROS algorithm and to explain its improvements over the DP/DFS algorithm.

In Figure **??** on page ??, x_j components are arranged in parallel in subsystem j, for $j = 1,2,3,4,5$. The reliability of subsystem j is $1-(1-r_j)^{x_j}$ (r_j is the reliability for component in subsystem j). The total cost of components is $\sum c_j(x_j + exp(x_j/4))$, the total weight of components is $\sum w_j \cdot x_j \cdot exp(x_j/4)$, and the total product of weight and volume is $\sum p_j \cdot x_j^2$.

This reliability problem can be formulated as follows:

$$max \prod_{j=1}^{n}[1-(1-r_j)^{x_j}],\ where\ x \in \mathbb{N}^n \tag{2.10}$$

subject to

$$\sum_{j=1}^{n} p_j \cdot x_j^2 \leq P \tag{2.11}$$

$$\sum_{j=1}^{n} c_j \cdot (x_j + exp(x_j/4)) \leq C \tag{2.12}$$

$$\sum_{j=1}^{n} w_j \cdot x_j \cdot exp(x_j/4) \leq W \tag{2.13}$$

This problem has been used by many researchers to demonstrate their approaches, such as Tillman and Liittschwager [1967]; Tillman et al. [1977]; Prasad and Kuo [2000]; Ravi et al. [1997], to name only a few.

The input data for $S1$ adapted from Ng and Sancho [2001] are given in Table 2.1.

Problem	R_j	p_j	P	c_j	C	w_j	W
$S1$	[0.80,0.85,0.90,0.65,0.75]	[1,2,3,4,2]	110	[7,7,5,9,4]	175	[7,8,8,6,9]	200

Table 2.1: Input data for $S1$

The RROS chooses System 2.11 (with the smallest RHS) as the key constraint, and creates a knapsack problem (Systems 2.10 and 2.11). By solving this knapsack problem, the DP solution U is found to be 0.93314. An arbitrary 5% of the DP solution is set by default to define the search depth ($0.93314 * 5\% = 0.04666$), so that the depth-first-search technique can be used. Then the RROS enumerates all near-optimal solutions in the search range $[0.88648, 0.93314]$ and a total of 22 near-optimal solutions are found in this range. After ranking all the 22 reliabilities (objective values) in decreasing order, and testing satisfaction on the remaining

constraints (Systems 2.12 and 2.13), the RROS finds the optimal solution $(3,2,2,3,3)$ and its reliability 0.90447. The overall process for solving problem $S1$ by the RROS takes 3.3 CPU seconds.

Key constraint	DP sol. U	Search depth	# of near-optimal solutions	Total CPU Time(sec.)	DP Time(sec.)	DFS Time(sec.)
System 2.11	0.93314	$5\% * U$	22	3.3	2.4	0.9
System 2.12	0.95045	$5\% * U$	66	5.5	3.7	1.8
System 2.13	0.90447	$5\% * U$	12	4.5	4	0.5

Table 2.2: RROS performance in Problem $S1$
* All three tests provide the same optimal solution [3,2,2,3,3] with reliability 0.90447.
* All testing problems in this thesis are run on a Sony desktop equipped with a 2.8G Pentium 4 Processor and 1Gb memory.

Table 2.2 illustrates how RROS performance is influenced by the choice of key constraint. System 2.13 gives the smallest DP solution, 0.90447, compared with 0.93314 (by System 2.11) and 0.95045 (by System 2.12). With the same arbitrary proportion of the DP solution (e.g. 5%) as the search depth, the test using System 2.13 as the key constraint searches less area than the others. Due to the limited search range, the number of near-optimal solutions (12) found in this test is less than in others (22 and 66). However, in Table 2.2, the computing performance for this test (using System 2.13) is not the best. This is because the total performance considers both dynamic programming and depth-first-search calculation performance. System 2.13 gives the best performance in the DFS, but takes the longest time to calculate DP solution.

Here is where the specificity of the RROS algorithm plays a role:

- In step A, the upper bounds that RROS calculates are tighter than the DP/DFS, they are $1 \leq x_1 \leq 5$, $1 \leq x_2 \leq 5$, $1 \leq x_3 \leq 5$, $1 \leq x_4 \leq 5$, $1 \leq x_5 \leq 5$. On the other hand, the upper bounds that DP/DFS finds, are $1 \leq x_1 \leq 6$, $1 \leq x_2 \leq 5$, $1 \leq x_3 \leq 5$, $1 \leq x_4 \leq 6$, $1 \leq x_5 \leq 5$ [Ng and Sancho, 2001], which are not as tight.

- The changes in step D do not impact this example.

- In step C, the DP/DFS algorithm suggests choosing System 2.13 as the key constraint, which has the smallest upper-bounds on the objective, after applying DP to all three constraints [Ng and Sancho, 2001].

  The RROS picks System 2.11 because it has the lowest constraint upper bound in the right hand side. The impact is best understood in Table 2.2:

  The DP/DFS algorithm would calculate DP for each system, for a total time of 10.1 seconds, and spend another 0.5 seconds in the DFS for System 2.13. By choosing System 2.11 directly, the RROS implementation spends 0.9 seconds in the DFS, but only 2.4 seconds in the DP for a total of 3.3 seconds.

Both the DP/DFS and the RROS calculate the same result: [3,2,2,3,3] with reliability 0.90447. The RROS slightly improves calculation performance on the DP/DFS algorithm. Unfortunately, the results of Ng and Sancho [2001], as stated in Table 2.3, could not be directly compared with the RROS performance. This is because the computer and language are too different to allow direct comparisons. Comparisons can be based on a re-implementation of the original algorithm; but in the above examples, the tightened bounds that the RROS calculates in step A do not impact the number of solutions found, and do not have a measurable impact on total execution speed.

Key constraint	DP sol. U	Search depth	# of near-optimal sol.	Adjusted CPU Time (second)*
System 2.11	0.93314	3.55%U	16	6
System 2.12	0.949014	5.2%*U	60	11
System 2.13	0.90447	0.5%*U	5	2

Table 2.3: DP/DFS performance in Problem *S1* [Ng and Sancho, 2001]
* Ng and Sancho [2001] test DP/DFS on a PC equipped with a Pentium II Processor 266MHz. The adjusted CPU time is based on the 2.8G Processor CPU that the RROS tested. This is a very rough comparison without considering the difference of computer memory, hard disk speed and etc.

2.3 The RROS Surrogate Method

As discussed in the above section, the RROS has been restricted to dealing with a subset of nonlinear integer programming problems. This is due to the fact that the embedded DP technique can only be applied to models with separable and monotonic objective function. This thesis further extends the scope of the RROS by integrating a surrogate relaxation method within the RROS algorithm.

In Chapter 1, the lagrangian relaxation method was introduced to convert a hard problem with complicated constraints into a set of relaxed problems with easy constraints. The goal of the RROS surrogate method is to convert a hard problem with a non-separable objective into a relaxed (surrogate) problem that has a separable objective function (surrogate objective), so that the RROS can be applied to solve this surrogate problem. The near-optimal solutions for the surrogate problem, calculated by the RROS, provide good optimal solution candidates for the original problem. One of the constraints of the hard problem, however, has to satisfy the monotonicity condition that the RROS requires for the objective function.

With the RROS surrogate method, one of the separable constraints of the original problem serves as the surrogate objective in the surrogate problem. The surrogate problem has the same constraints as the original problem, therefore, the two problems share the same feasible solutions. In addition to providing a good "guess" as to the optimal solution for the original problem, the RROS surrogate method establishes a procedure to determine the optimal for the original problem. This procedure can be outlined as follows:

Step 1 Generate a surrogate discrete problem $SP1$ by choosing a separable constraint as an objective.

Step 2 The RROS solves $SP1$ and provides a set of near-optimal solutions within a search

depth e. Apply these near-optimal solutions to the original objective function (non-separable) and rank the objective value. The solution, which has the best objective value and satisfies all the original constraints, is considered as the integer optimal within the interval $[U-e,U]$.

Step 3 Generate a continuous surrogate problem $SP2$ in order to find the continuous optimum within the range of the $SP1$ that the RROS has not searched in Step 2: $[L,U-e]$. Use a continuous linear programming method to find the continuous optimum of $SP2$.

Step 4 Compare the values of the original objective function, generated by the integer optimal solution from Step 2 and by the continuous optimal solution from Step 3. If the former provides a greater objective value than the latter, then this integer solution from Step 2 is the optimal for the original problem. Otherwise, the search depth e has to be marked up and resume search from step 2.

More precisely, consider a reliability redundancy optimization problem (Systems 2.14 to 2.18) in which its objective R_s has a non-separable function:

$$max\ R_s(x) \tag{2.14}$$

Subject to the following conditions: one separable constraint

$$\sum_{j=1}^{n} c_j(x_j) \leq C \tag{2.15}$$

and a series of non-separable constraints:

$$P_i(x) \leq P_i^* \tag{2.16}$$

$$Q_i(x) \leq Q_i^* \tag{2.17}$$

$$x \in \mathbb{Z}^n \tag{2.18}$$

The surrogate RROS method assumes the following conditions, inherited from the RROS:

1. P_i^* , Q_i^* and C are non-negative constants for all i's;
2. $P_i(0) = Q_i(0) = 0$;
3. $c_j(x_j)$ are monotonic increasing functions, and $\lim_{x_j \to \infty} c_j(x_j) = \infty$
4. For any fixed value of $x = (x_1, \ldots x_n) \in \mathbb{R}^n$, if any one variable x_j varies independently, $P_i(x_1, \ldots x_j, \ldots x_n)$ is a monotonic increasing function in x_j; also, $\lim_{x_j \to \infty} P_i(x_1, \ldots x_j, \ldots x_n) = \infty$; and the same holds for Q_i.

The algorithm of the RROS surrogate method is illustrated in Figure 2.2 and the detailed steps are described as follows.

Step 1 Transform the original problem into surrogate problem $SP1$, using System 2.15 as a basis. The surrogate problem $SP1$ can be described as $max \sum_{j=1}^{n} \frac{c_j(x_j)}{C}$, Subject to System 2.15 to 2.18. This way, the maximum value of the surrogate objective is constrained by 1. $SP1$ is in a form which can be treated by the RROS algorithm.

Note that most reliability redundancy problems include cost and weight constraints, which are generally separable. They are good candidates to become the objective function of the surrogate problem. In general, there is no guarantee that the optimum of the surrogate problem will be an optimum for the original problem. However, in many cases, the optimum of a constraint problem lies near the boundary of the constraints. Therefore, to search for optima of the objective function, it is a good heuristic to first look the near-optimals of the surrogate objective function (one of the original constraints).

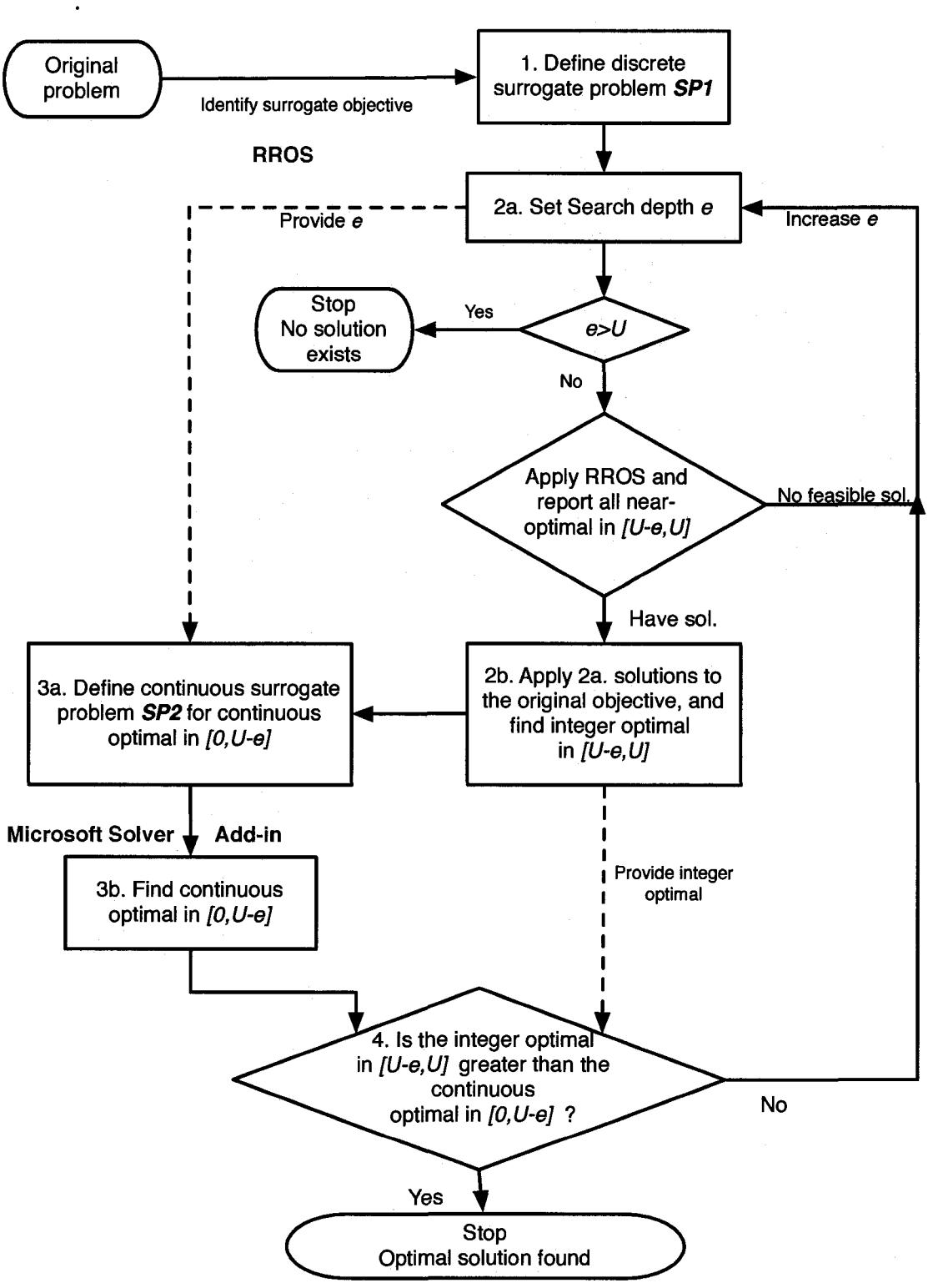

Figure 2.2: The basic scheme of the RROS surrogate method

When there is more than one separable constraint in the original problem, the best surrogate objective is the one that can define the smallest upper bounds for decision variables.

Step 2a Apply the RROS algorithm to solve the surrogate problem $SP1$. The constraint 2.15 chosen as the surrogate objective is also used as the key constraint. As both the objective function and the key constraint are monotonic and separable, the RROS can find all the near-optimal solutions for $SP1$ within a specific range $[U-e, U]$. In other words, these solutions represent feasible near-optimal choices in a search space defined by

$$\sum_{j=1}^{n} c_j(x_j) \in [C * \frac{U-e}{U}, C]$$

Because the surrogate problem $SP1$ shares the constraints of the original problem, the feasible solutions of $SP1$ are also feasible solutions in the original problem.

If there is no feasible solution for $SP1$, then the search depth e is increased and Step 2a repeats. If $e > U - L$ and there is still no feasible solution for $SP1$, then the original problem has no feasible solution according to Condition 1 of the relaxation technique.

Step 2b Apply the feasible solutions generated in Step 2a to the original objective function (System 2.14), and rank the objective values in a decreasing order. The solution with the best objective value is recorded as $O_N^{\geq e}$.

Step 3a Define a new surrogate problem $SP2$ which allows continuous solutions. $SP2$ shares the same objective function as the original problem, and keeps most of its constraints, except the discreteness constraint (System 2.18) and the separable constraint (System 2.15). In $SP2$, the original separable constraint (System 2.15) is replaced by $\sum_{j=1}^{n} c_j(x_j) \leq C * \frac{U-e}{U}$. This upper bound in $SP2$ is the same as the lower bound of the surrogate objective in $SP1$, so that the range $[L, U-e]$, that has not been searched in

$SP1$ can be searched in $SP2$.

Step 3b Use a continuous linear programming method, such as the Generalized Reduced Gradient method [Solver], to find the continuous optimal solution of system $SP2$. Record that optimum as $O_R^{\leq e}$.

Step 4 Compare $O_N^{\geq e}$ and $O_R^{\leq e}$. If the former is greater than the latter, it means that the best integer solution in $[U-e,U]$ is better than the best continuous solution in $[L,U-e]$; of course that also means that it is better than the best integer solution in $[L,U-e]$. So $O_N^{\geq e}$ is the global optimum. Otherwise, increase the search depth e and resume from step 2a.

2.3.1 An example application of the surrogate method

The following example $SR1$ is used to demonstrated the RROS surrogate method.

[**Problem** $SR1$]: In many redundancy allocation problems, both redundancy levels and component choices are determined at the system design stage. Consider the maximization of the system reliability of a 3-stage series system with redundancy units in parallel, as in Figure 1.4 on page 11. Different types of components can be used as design alternatives at each stage. Chern and Jan [1986]; Majety et al. [1999]; Ng and Sancho [2001] have studied this problem (Systems 2.19 to 2.23).

$$max\ R_s = \prod_{i=1}^{s}[1-\prod_{j=1}^{k_j}(1-r_{ij})^{x_{ij}}] \tag{2.19}$$

subject to:

$$\text{Weight Constraint: } \sum_{i=1}^{s}\sum_{j=1}^{k_j} w_{ij}\cdot x_{ij} \leq W_0 \tag{2.20}$$

$$\text{Cost Constraint: } \sum_{i=1}^{s}\sum_{j=1}^{k_j} c_{ij}\cdot x_{ij} \leq C_0 \tag{2.21}$$

Type		$i=1$	$i=2$	$i=3$
	Reliability	0.69	0.88	0.78
$j=1$	weight	4	3	4
	cost	3	5	11
	Reliability	0.75	0.8	0.7
$j=2$	weight	6	5	6
	cost	13	7	5

Table 2.4: Input data for $SR1$ (W_0=15, C_0=20)

$$\sum_{i=1}^{k_i} x_{ij} \geq 1 \tag{2.22}$$

$$x_{ij} \in \mathbb{N} \tag{2.23}$$

where $s = 3$ is the number of subsystems, k_i is the number of design alternatives (different component types) in subsystem i, $x_{ij} \in \mathbb{N}$ is the number of redundant components of type j in subsystem i, and r_{ij} is the reliability of components of type j in subsystem i.

The input data for $SR1$ is given in Table 2.4. As $SR1$ has a non-separable objective function (System 2.19), it cannot be solved by the techniques used for separable objective functions. By applying the RROS surrogate method, the original problem is converted into a relaxed surrogate problem $SP1$. The objective in $SP1$ uses one of the resource constraints in the original problem, such as the cost constraint (System 2.21). $SP1$ keeps all the constraints from the original problem, as follows:

$$max \sum_{i=1}^{s} \sum_{j=1}^{k_j} \frac{c_{ij}}{C_0} \cdot x_{ij} = [(3x_{11} + 13x_{12}) + (5x_{21} + 7x_{22}) + (11x_{31} + 5x_{32})]/20 \tag{2.24}$$

subject to System 2.20 to 2.23.

The following steps describe how the RROS surrogate method is applied to the problem $SR1$, and the performance data of each step is summarized in Table 2.5:

1. To define SP1, the cost constraint is used as the key constraint. The RROS first calculates the upper bounds for decision variables, $[3,1,4,2,1,2]$, and the DP solution (U) for $SP1$ is 1. By setting the search depth $e = 5\% * U = 0.05$, the RROS finds 226 near-optimal solutions in the interval [0.95,1] ($[U-e,U]$). The RROS then applies the 226 solutions to the original objective function (System 2.19) to calculate the system reliability for each solution. Finally, it ranks all the 226 reliabilities and determines the optimal one that satisfies other constraints (Systems 2.20 and 2.22). The optimal solution to the original problem, where the cost constraint (System 2.21) has a lower bound $C_0 * \frac{U-e}{U} = 19$ (namely, $19 \leq \sum_{i=1}^{s} \sum_{j=1}^{k_j} c_{ij} \cdot x_{ij} \leq C_0$), is found as $[1,0,1,0,1,0]$ with reliability 0.4736. This step takes 10 seconds.

2. Create a continuous surrogate problem $SP2$ (Systems 2.19-2.22) and modify the RHS of System 2.21 from 20 to $20 * (1-0.05) = 19$. The discrete constraint (System 2.23) in the original problem is ignored in $SP2$. The continuous optimum solution is calculated by the Microsoft Solver Add-in, which quickly finds the optimum for $SP2$ with objective value 0.5127. This continuous optimal represents the best reliability which can be found in the range where the cost constraint (System 2.21) value is less than 19 (namely, $\sum_{i=1}^{s} \sum_{j=1}^{k_j} c_{ij} \cdot x_{ij} \leq 19$), and which step 1 has not yet searched. This continuous optimal 0.5127 is greater than the integer optimal (0.4736) from step 1.

3. Increase e from 0.05 to 0.1 and repeat step 1. The newly generated integer optimal (0.473616) is the same as the one obtained from step 1. At this point, the RROS searches the area where the cost constraint is $18 \leq \sum_{i=1}^{s} \sum_{j=1}^{k_j} c_{ij} \cdot x_{ij} \leq C_0$. The total number of near-optimal solutions found in this step is 304, computed in 14 seconds.

4. Modify problem $SP2$ by changing the RHS of System 2.21 from 19 to $20 * (1-0.1) = 18$, and repeat step 2. The newly generated continuous optimal for $SP2$ is 0.4639, which is less than the integer optimal generated from steps 1 and 3. In other words, there is

	Test Problem	Program	Optimal reliability	CPU time (second)	# of Solutions
Step 1	$SP1$,$SR1$ ($e = 0.05 * U$)	RROS surrogate	0.4736	10	226 near-optimal sol.
Step 2	$SP2$ ($C_0 = 19$)	Microsoft Solver	0.5127		
Step 3	$SP1$,$SR1$ ($e = 0.1 * U$)	RROS surrogate	0.4736	14	304 near-optimal sol.
Step 4	$SP2$ ($C_0 = 18$)	Microsoft Solver	0.4639		

Table 2.5: The RROS surrogate process for problem $SR1$

no continuous solution in the range (where $\sum_{i=1}^{s}\sum_{j=1}^{k_j} c_{ij} \cdot x_{ij} \leq 18$) that can give better objective value than the integer solution $[1,0,1,0,1,0]$ (with objective value 0.4736) found when $18 \leq \sum_{i=1}^{s}\sum_{j=1}^{k_j} c_{ij} \cdot x_{ij} \leq C_0$. Stop the process and report the integer optimal solution [1,0,1,0,1,0] with reliability 0.4736.

To check results, a full enumeration process gives the same solution [1,0,1,0,1,0] as the RROS surrogate method at steps 1 and 3.

Chapter 3

A Microsoft Excel Add-in Solution: RROS

At the time when software was supported primarily on mainframe computing systems, optimization was thought of as "a highly specialized field and practiced only by those with advanced knowledge of mathematics and computer programming languages" [Ragsdale, 1998]. The ongoing worldwide availability of PCs and the continuous improvement in computation performance have dramatically changed the optimization field. Various optimization software tools have been developed; therefore normal users are able to use appropriate software tools to optimize their own models without professional help.

This chapter first compares three major types of optimization software on the market and then introduces a Microsoft Excel Add-in solution (RROS) for reliability redundancy problems. The RROS application implements the RROS algorithm, and was developed using Microsoft's Visual Basic Application (VBA). It has many features, such as the ability to build models automatically in Excel and to solve nonlinear, non-separable redundancy problems in a reasonable time. This chapter also discusses the modelling procedure, software structure, and

graphical user interface (GUI).

3.1 Why An Excel add-in?

3.1.1 Overview of Optimization Software Tools

There is no single type of software that is capable of solving all kinds of optimization problems. In fact, most program can only solve certain types of problems. As a result, the number of optimization software packages is numbered in the hundreds [Mittelmann, 2007] and users have to be very knowledgeable about mathematical models in order to select appropriate software.

On the other hand, software vendors tend to focus on specific user groups and provide specific optimization packages. Thus, three types of software are designed to meet the need of three user groups, namely *optimizers*, *developers*, and *end-users* [Fourer, 1996].

- *Procedure and class libraries* are software packages that allow *optimizers* to build and practice mathematical models in the user's own development environment. For example, IMSL C Numerical Libraries provides a comprehensive collection of approximately 300 mathematical and statistical functions (subroutines). *Optimizers* can write a program which calls one or more specific function(s) by using a general-purpose programming language (such as C, C++, or Fortran). The functions solve the optimization problem and send the result back to the user's own program. The advantages of this software class are flexibility and low cost. Optimizers can format input parameters appropriately and generate output reports in a desired form without learning a new language or package (software). However, mathematical and programming expertise are required on the part of users. Moreover, code written for a specific problem cannot be directly transferred to solve other similar problems.

- *Application development environments* generally have high-level interfaces that allow *developers* to build complete optimization applications in an easy fashion, so that they are freed from programming work. For example, ILOG Inc. has created ILOG OPL Studio, which supports quick application development and deployment [ILOG].

- *Spreadsheet Add-ins* provide additional functionality to spreadsheet programs such as Excel. The Add-ins are often written in the VBA language which is built into Excel, so that the additional functions (Add-ins) can be loaded when spreadsheet programs start up. It is an excellent way of increasing the power of Excel. That is why Add-ins have become an ideal vehicle for distributing custom functions for normal *end-users*. Frontline Systems Inc. has been successful in the spreadsheet optimization domain, by bundling its Solver product with Microsoft Excel [Carter and Price, 2001][Solver].

Overall, the software industry has been pushed to provide more user friendly products by the high level of competition in the field and by an expanded clientele. The added difficulty comes from demands for easier user interfaces and for expanded mathematical functionality.

3.1.2 Spreadsheets and Add-ins

Since the first electronic spreadsheet for microcomputers, VisiCalc, was released in 1979 [Power, 2004], spreadsheets have been universally recognized as the most convenient and versatile tools for Operation Research/Management Science (OR/MS) people to analyze and implement quantitative models. Eudoxus System Ltd. explains the relationship between spreadsheet and optimization as follows [Eudoxus]:

> *Given the way in which [spreadsheets] are used to calculate derived quantities from the values in other cells, it was inevitable that they should be extended so*

> *that the values in cells could float so as to enable the value in the target cell to be maximized. In this way the spreadsheet itself has become another format for expressing a [mathematical programming] matrix.*

The spreadsheet framework offers a tabular (row and column) paradigm for problem parameters, which makes it easy to input, to create and to read a model. The fact that spreadsheets such as Excel and Lotus 1-2-3 are able to seamlessly integrate optimization add-ins has enabled intense research on spreadsheet optimization. Not only has the use of spreadsheets expanded the scope of optimization applications, it has also enabled more and more users to solve many optimization problems without special OR/MS training.

Pros and Cons of Spreadsheet Optimization

Spreadsheets have been universally accepted because they are able to convey quantitative methodologies in a language that most people easily understand. Users generally tend to stay and work in their familiar and flexible environment, which has an easy-to-use interface. Spreadsheets are capable of performing many routine jobs, including repetitive calculations, statistics and optimizations.

In 1986, *OR/MS Today* reviewed an early spreadsheet optimizer, *What's Best!* (by Lindo System Inc.) and described the software as a "breakthrough" product [Savage, 1997]. Since then, Excel's capability to integrate add-ins seamlessly has allowed users to enjoy more customized applications. Also, the Solver products have spawned many applications in industry, institutes and government.

On the other hand, there are serious drawbacks to spreadsheet models, such as the difficulty to detect errors and limitations to computational expressiveness. Savage pointed out that about 90% of spreadsheet models used in business contain errors, and that those errors can be potentially dangerous; yet 95% of spreadsheet users are confident that their models do not

contain errors [Savage, 1997]. In many cases, the visible part of a worksheet does not provide clues to detect mistakes which happen in an underlying equation. When spreadsheets become larger, the problems become even less manageable [System Modelling]. Another drawback to the spreadsheet model is that their tabular structure is not conducive to good documentation.

Moreover, many types of algorithms are more difficult to implement in spreadsheets than in full programming languages or in professional optimization applications; and the execution time of spreadsheet can be much higher, especially when dealing with large optimization problems. Also, it is difficult to change or modify a model structure when the spreadsheet model has been built. Inserting a row inside an existing model might result in unpredictable changes in the whole model, as the formula definitions are changed in distant cells, possibly off-screen. Thus, a careful design of the model is required to avoid undesirable changes.

3.1.3 A survey of Spreadsheet Optimization Software

According to the *OR/MS Today* 2002 survey [Grossman, 2002], the spreadsheet optimization market is dominated by three big vendors: Frontline Systems Inc., LINDO Systems and Palisade Corp. The main areas of competition in the market are software stability, accuracy, implementations speed, user interface and the ability to handle large sized problems. It has been noted that the three above companies have focused their attention on industry end-users, but not on the OR/MS professionals. The trend of spreadsheet optimization is to enable managers and users without special OR/MS training to solve optimization problem in their own workgroups.

FrontLine Systems Inc. created the *Excel Standard Solver Add-In* that is packaged with each copy of Microsoft Excel and provides the Solver/Optimizer for Lotus 1-2-3 and Quattro Pro users. In addition, the company offers other more evolved Solver products to solve more difficult models, such as nonlinear problems [Solver]. LINDO System distributes *What's Best!*,

which is the original spreadsheet solver [Lindo]. Palisade Corp. also introduced spreadsheet optimization add-ins to the market [Palisade]. The company has long provided many Excel simulation products.

Some free of charge spreadsheet optimization tools are available on the Internet. *SolverTable* is one of the most notable add-ins. Created by Dr. S. Christian Albright at Indiana University [Albright], *SolverTable* is the only spreadsheet parametric optimization tool with a portable code and good documentation. Appendix A lists available add-ins products on the market with computational capacity, price, and other main features.

It is important to point out that spreadsheet add-ins, like other software packages, are limited by the embedded algorithms. Fylstra et al. from Frontline System Inc., notes that the GRG algorithm embedded in Excel Solver may fail to find a feasible solution even though one exists, or it may return a local optimum which is not global [Fylstra et al., 1998].

3.2 The RROS: Reliability Redundancy Optimization Solver Solution

The RROS application was created by the motivation that users without OR/MS training can solve some non-linear integer problems in an easy way. Without the application, applying the existing RROS algorithm would require users to fully understand the algorithm and to master some professional mathematical programs. Without a complex working environment, it is very time consuming to enter the formulae and data of a model.

The RROS application uses Microsoft Excel as modelling platform; therefore it does not require users to install or learn a special program such as MatLab or R. This application implements the RROS algorithm, as discussed in Chapter 2. One of this application's objectives

is to provide an easy to use tool, so that users can create an optimization model in a fast and reliable way, with minimal training time.

The RROS application combines the functions of a graphical user interface (GUI), an algebraic modelling language, and the RROS algorithm. The spreadsheet's formula language itself acts as an algebraic language used to define models. Through the RROS's GUI, users can generate a model by filling in dialog boxes in a wizard interface, and then specify an objective function and a number of constraints by inputting values and formulae in Excel cells. The RROS algorithm is then applied to find an optimal solution. In addition to reporting optimal solution values and summarizing key performance data in a tabular form, the RROS application also provides a sensitivity analysis of the solution values.

3.2.1 Main Features of the GUI

As discussed in the last section, user interfaces and the embedded mathematical algorithms are the two main problem areas for optimization software. The majority of Microsoft Excel users are not OR/MS professionals, and may not have modelling experience. Unfortunately, most current approaches such as Frontline Solver products can only optimize a pre-existing spreadsheet model, but do not provide any help in formatting the models themselves [Albright, 2001]. The RROS, however, is designed for end-users and allows them to input/update models with minimal effort.

The RROS collects basic information regarding the optimization problem parameters. From there, it automatically initializes a spreadsheet with an model skeleton for the specification of the decision variables, the objective function and the constraints. The user can fill the model skeleton with that information, in the form of traditional Excel formulae. This usage of the formula input method makes model construction particularly convenient; also, it allows the

RROS system to leverage Excel for calculations, instead of parsing user input.

This Chapter uses Problem $S1$ (discussed in Chapter 2) as an example to demonstrate the RROS application.

1. The RROS initializes a model using a wizard interface

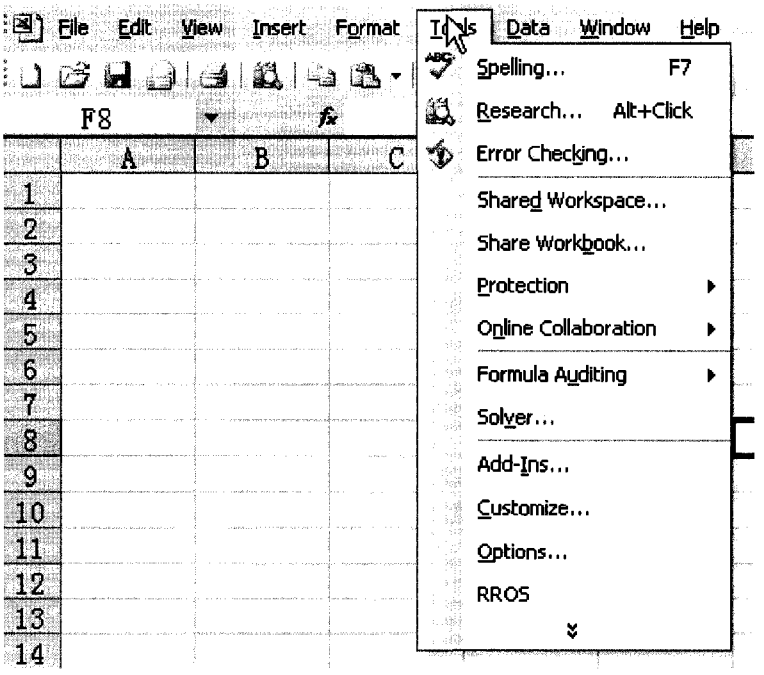

Figure 3.1: *RROS* command on the *Tools* menu

A click on the *RROS* command from the *Tools* menu (Figure 3.1) displays an Excel dialog box (Figure 3.2). Through the dialog box, the RROS collects the basic information required to create a model: : specifically the number of variables, the number of constraints, and the form of the objective function. The objective function can be in either the *multiplicative* or *additive* format. In reliability optimization, the maximization of the logarithm of system reliability (additive form) is equivalent to the maximization of the system reliability (multiplicative form). The data type of the input data is verified.

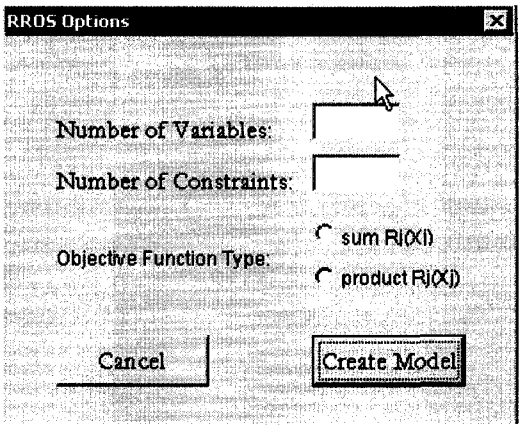

Figure 3.2: Dialog box for data input

Problem $S1$ has 5 decision variables and 3 constraints. Its objective function is in multiplicative form. Figure 3.3 shows the resulting Excel model skeleton that The RROS creates automatically in a compact tabular form.

2. To assist users, the generated model is displayed in colours, which help to distinguish between the model data, the decision variables, and the model formulae. Additionally, a note is available for each functional cell so that users are guided when inputting contents.

   In Figure 3.3, blocks of *light blue* Excel cells are meant for input data on the coefficients of the objective function and constraints, the relational operator for the constraints, the resource (RHS) values of the constraints, and (optionally) initial bounds on the decision variables. The *dark blue* Excel cells are meant for input of the objective and constraint formulae. *Yellow* cells are reserved for the output of the optimal solution and for the RROS summary report on the solution.

3. The RROS allows users to fill initial data and formulae into the generated model skeleton with a new method that reduces the input effort significantly.

   It has never been difficult to input a linear program model; on the other hand, nonlinear programming is quite different. Nonlinear models come in many different forms, and

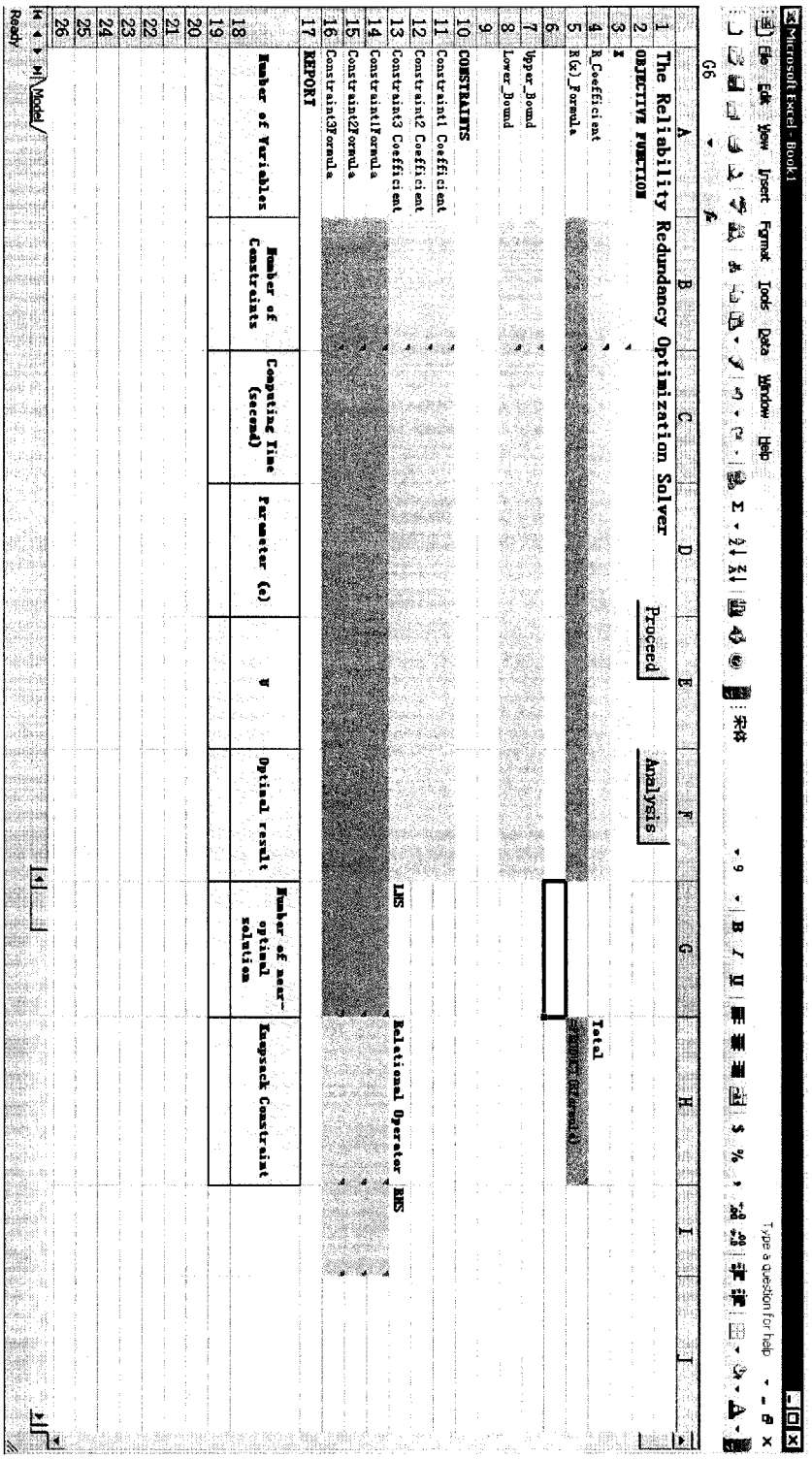

Figure 3.3: Example of the model input for Problem *S*1

there is no standard form in which to specify a nonlinear model. With that in mind, the RROS keeps the task of inputting formulae (for objective function and constraints) to a minimum.

In the example problem $S1$, the objective function $R_s = \prod_{j=1}^{n} R_j = \prod_{j=1}^{n} [1-(1-r_j)^{x_j}]$ is the product of stage reliability R_j for all j where each R_j is of the same format $(1-(1-r_j)^{x_j})$.

When users input the first term R_1, the RROS application takes advantage of the repeated form of R_j, and generates $R_2, ..., R_n$ automatically for the remaining given $n-1$ stages (Figure 3.4). Similarly, only the first term of each constraint needs to be specified and the RROS automatically generates the remaining terms. On the other hand, users have the option to modify any formula generated by the RROS in the event that terms of the objective function or constraint are not similar. For example, consider the following constraint $5x_1 + 5x_2^3 + 6x_3 \leq 80$, where x_2 has a different format from the first and third terms. Users only need to modify the Excel cell corresponding to the second term, namely, x_2^3 instead of x_2. The term involving x_3 has been generated by the RROS as it is of the same format as the first term.

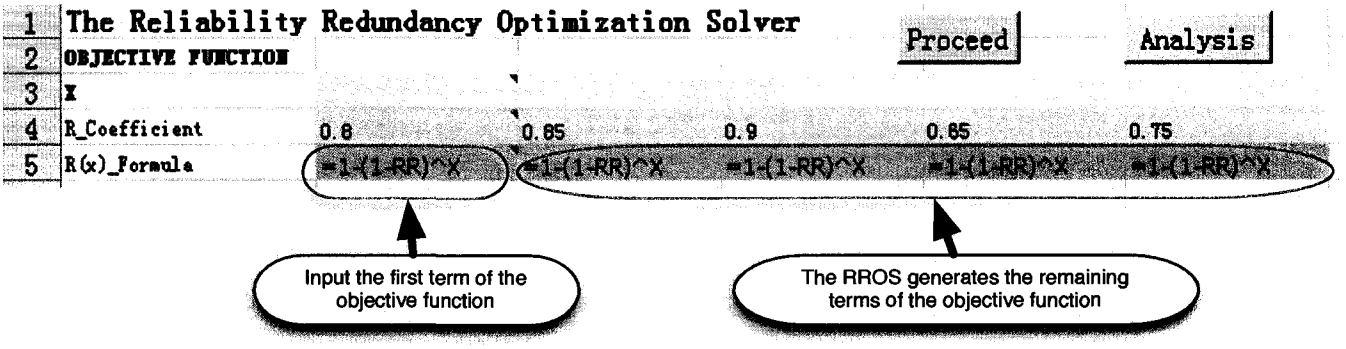

Figure 3.4: Input objective function formulae

In addition, Excel cells are grouped as sets and declared as *Range* objects in the RROS application. This enables users to input formulae without concern for the exact cell

location, in view of the fact that formulae are copied down or across. Table 3.1 explains the relationship between the mathematical model (System 2.10-2.13) and the Excel model (Figure 3.6) for this problem of five variables with three constraints. In this case, the RROS initializes a set of cells (`B5:F5`) which hold R_j as a Range object with a name `RR`. This way, when entering the formula $R_j = (1 - r_j)^{x_j}$ in cell `B5`, users can simply type "`=(1-RR)^ X`" instead of "`=(1-B4)^ B3`", where X is the Range name of the decision variables (located in range (`B3:F3`)). And the identical formulae are generated in cells (`C5:F5`) as described above. By default, the RROS generates constraints formulae in an additive format (Figure 3.5); users can modify the formulae if they are in other formats. In the modelling process, the algebraic formulae appear directly

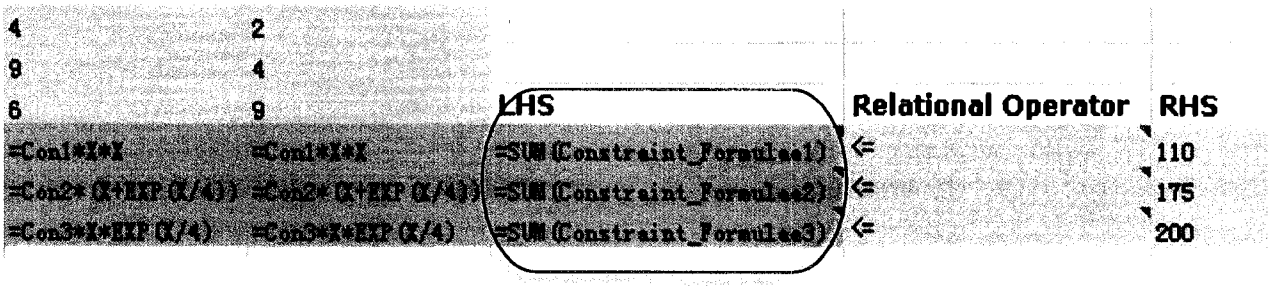

Figure 3.5: The RROS generates the LHS of a constraint

in the Excel cells, rather than the numerical results. It can help users to check model formulae frequently, and to avoid possible typographical errors. A complete input model is shown in Figure 3.6. Once a model is generated, however, users are not allowed to change the model structure, such as the number of constraints or the number of variables. If there is a modification of the model structure, users should create a new model.

4. The RROS provides default values for all parameters. In many cases, end-users may not know what input is acceptable or how to set appropriate parameters. For example, they

Name	Mathematical Format	Excel Cells	Range Name	Block color
Decision variables	x_j	B3:F3	X	yellow
Coefficient of the objective function	R_j	B4:F4	RR	light blue
Formulae of the objective function	$r_j(x_j)$	B5:F5	RFomula	dark blue
Objective function	$\prod_{j=1}^{n}(1-(1-r_j)^{x_j})$	H5	Total	dark blue
Coefficient of constraint 1	p_j	B11:F11	Con1	light blue
Formulae of constraint 1	$p_j x_j^2$	B14:F14	Constraint_Formula1	dark blue
Left Hand Side (LHS) of constraint 1	$\sum_{j=1}^{n} p_j x_j^2$	G14	LHS1	dark blue
Relational operator of constraint 1	$\leq$	H14	Operator1	light blue
Right Hand Side (RHS) of constraint 1	P	I14	RHS1	light blue
Coefficient of constraint 2	c_j	B12:F12	Con2	light blue
Formulae of constraint 2	$c_j(x_j+exp(x_j/4))$	B15:F15	Constraint_Formula2	S dark blue
Left Hand Side (LHS) of constraint 2	$\sum_{j=1}^{n} c_j(x_j+exp(x_j/4))$	G15	LHS2	dark blue
Relational operator of constraint 2	$\leq$	H15	Operator2	light blue
Right Hand Side (RHS) o of constraint 2	C	I15	RHS2	light blue
Coefficient of constraint 3	w_j	B13:F13	Con3	light blue
Formulae of constraint 3	$w_j x_j exp(x_j/4))$	B16:F16	Constraint_Formula3	dark blue
Left Hand Side (LHS) of constraint 3	$\sum_{j=1}^{n} w_j x_j exp(x_j/4))$	G16	LHS3	dark blue
Relational operator of constraint 3	$\leq$	H16	Operator3	light blue
Right Hand Side (RHS) of constraint 3	W	I16	RHS3	light blue

Table 3.1: The relationship between the mathematical model and the Excel model

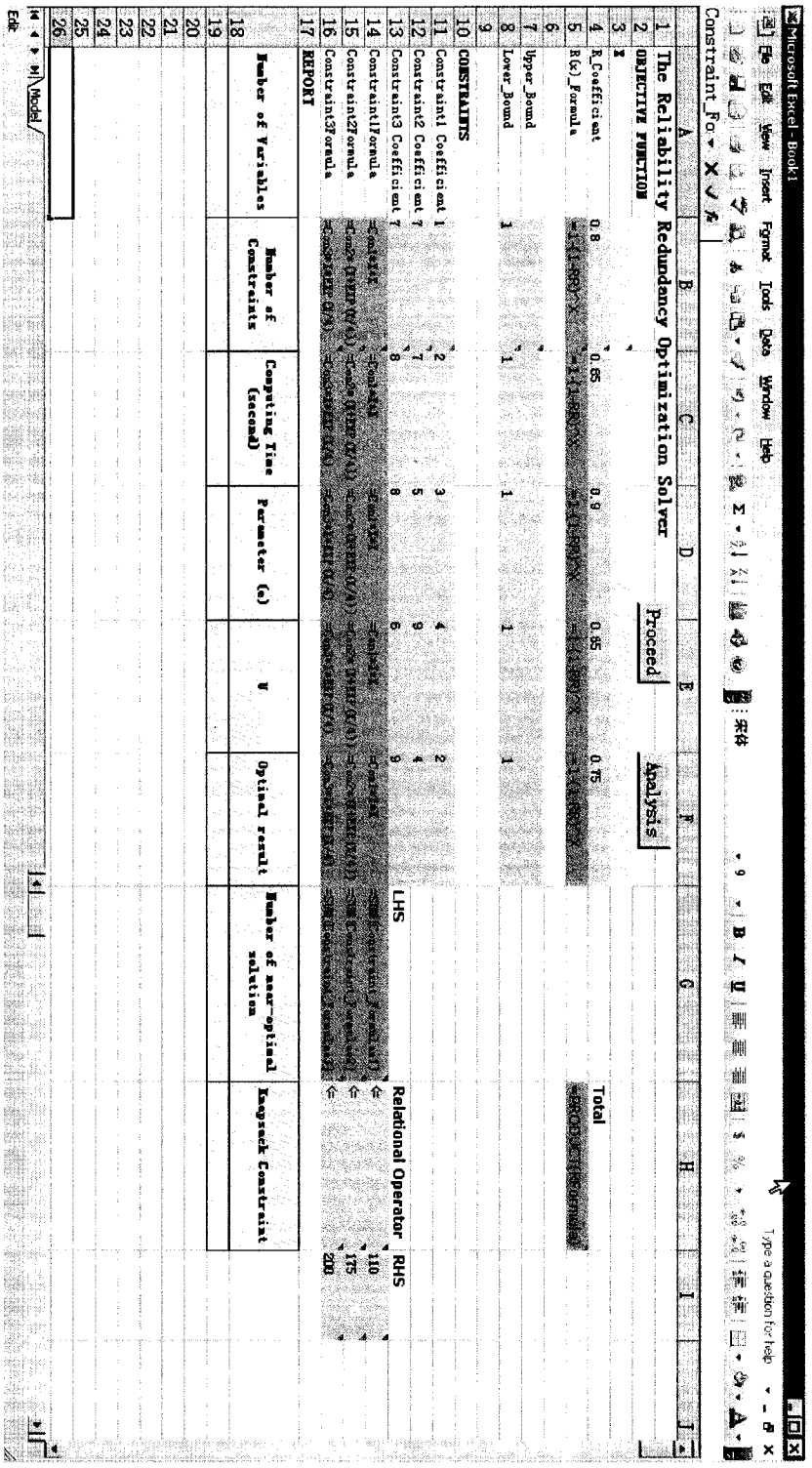

Figure 3.6: Data input for problem $S1$

may not know what the bounds of the decision variables are.

In the Figure 3.6, cell sets (B7:F7) and (B8:F8) represent the upper and lower bounds on decision variables, respectively. Generally, tighter bounds improve computing performance. Depending on the given problem, users may have the bounds information, and they can input the bounds value into the corresponding Excel cells. On the other hand, the RROS calculates the bounds based on the theory outlined in Chapter 2, and then compares with the given bounds. Tighter bounds will then be derived for the following calculation.

When users complete the model input step, and click on the *Proceed* button in the spreadsheet, a parameter dialog box appears, as shown in Figure 3.7. This dialog box allows users to define the key constraint and the search depth. Users do not have to input values for these parameters; the RROS will calculate the parameters according to the default parameters discussed in Chapter 2 and proceed with the solution report.

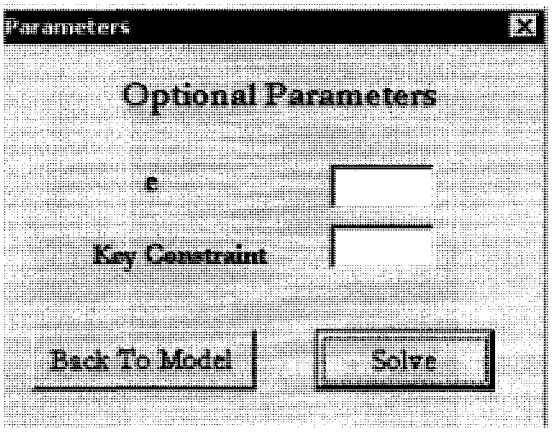

Figure 3.7: Parameter dialog box

5. The RROS provides two analysis reports: the optimal solution summary report and the feasibility analysis of the near-optimal solutions .

   The summary report lists the optimal solution, CPU computing time, the DP solution

of the relaxed knapsack problem U, the arbitrary chosen search depth e, the number of near-optimal solutions and the key constraint. This report can be found in the *yellow* Excel cells with *red* borders, as shown in Figure 3.8.

In integer programming applications, finding an optimal solution to a model is not the only requirement. Often, it is desirable to know what happens if a certain change is made on the right-hand sides of the constraints. Sometimes, the right-hand sides of the constraints may not be known with certainty. Decision makers need to determine how the optimal solution behaves as the right-hand sides are varied in the model.

With those requirements in mind, the RROS provides a simple method to conduct sensitivity analysis. By clicking on the *analysis* button (Figure 3.6), the RROS displays a feasibility analysis report in another worksheet (named *report*), as shown in Figure 3.9. The report lists the lattice of near-optimal solution points, together with their corresponding constraint resource values.

Figure 3.9 can be used as follows. First, the current reliability value is 0.904467 (Excel cell `F14`, in the highlighted row). It would remain feasible even if the product of weight and volume constraint were tightened from 110 to 84, the cost constraint were tightened from 175 to 147 and the weight constraint were tightened from 200 to 193. On the other hand, if the weight constraint were relaxed from its current value of 200 to 220, then the optimum reliability value could be as high as 0.922163 (Excel cell `F7`). The best way to see this would be to filter the report for values weight below 220, using the Excel filter functionality, and to pick the highest reliability. Conversely, if the decision makers decided to tighten the cost constraint to 160, then they would have to allow a weight of 217 in order to attain an optimum reliability of 0.922163 (Excel cell `F6`). The CPU time required to generate the feasibility reports is minimal: For example, less than 1 second

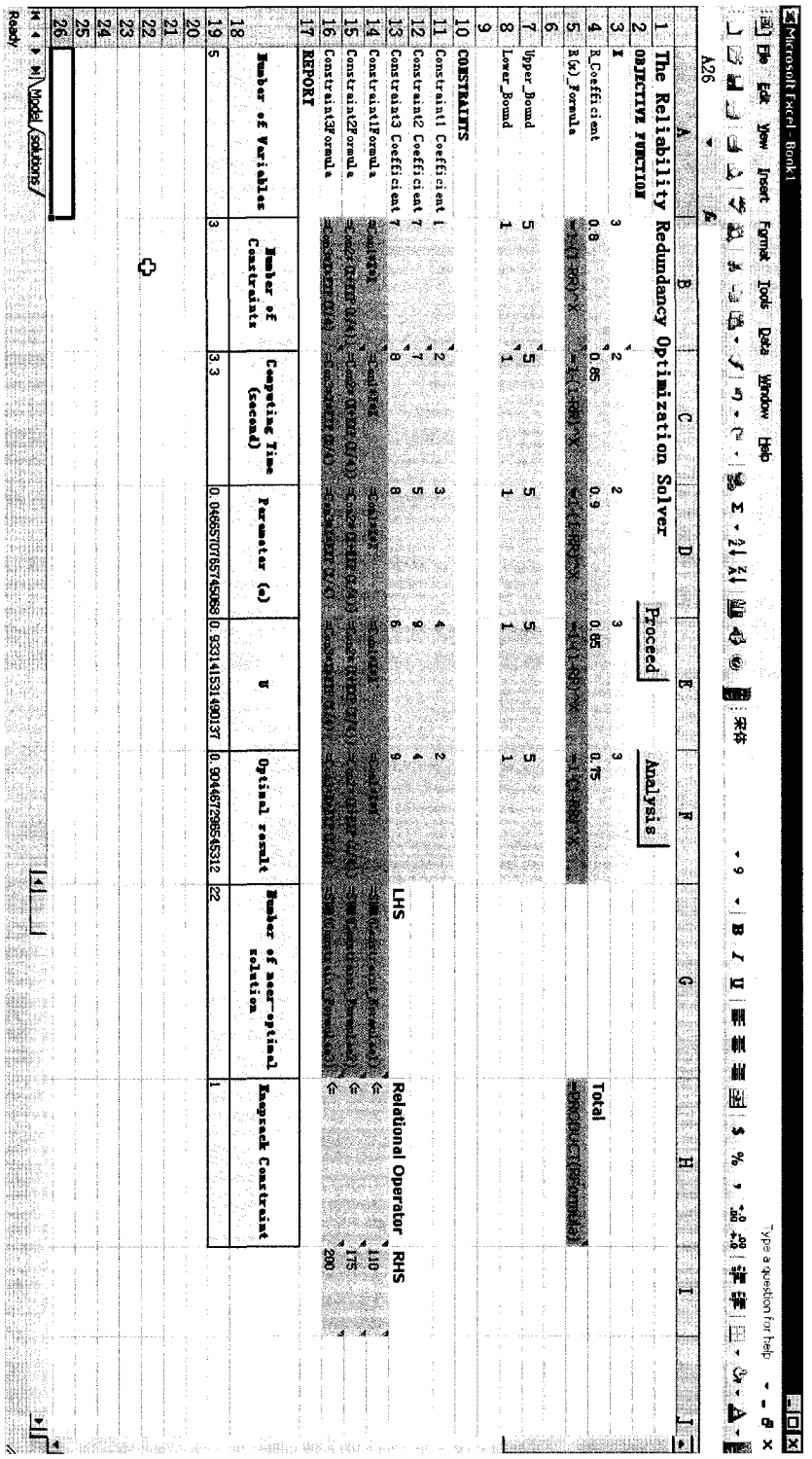

Figure 3.8: Performance report for the Problem $S1$

	A	B	C	D	E	F	G	H	I
1	X1	X2	X3	X4	X5	Objective Function	LHS(1)<= 110	LHS(2)<= 175	LHS(3)<= 200
2	3	3	2	3	4	0.933142	107	162.8077	257.6087
3	3	3	3	3	3	0.930547	108	163.744	241.338
4	5	3	2	3	3	0.929303	109	180.016	294.6145
5	4	3	2	3	3	0.928113	100	167.6116	248.5644
6	3	4	2	3	3	0.924818	107	167.6116	253.0866
7	3	3	2	3	3	0.922163	93	156.4026	216.9095
8	4	2	2	3	4	0.92114	104	163.7388	264.8351
9	4	2	3	3	3	0.918578	105	164.675	248.5644
10	3	2	2	3	4	0.915235	97	152.5298	233.1802
11	3	2	3	3	3	0.91269	98	153.466	216.9095
12	5	2	2	3	3	0.91147	99	169.7381	270.1861
13	4	2	2	3	3	0.910303	90	157.3336	224.136
14	3	2	2	3	3	0.904467	83	146.1247	192.4811
15	2	3	2	3	4	0.90304	102	152.5298	236.2338
16	2	2	2	4	3	0.900777	106	150.2582	198.2389
17	2	3	3	3	3	0.900529	103	153.466	219.9631
18	2	4	2	3	3	0.894985	102	153336	231.7117
19	2	2	3	3	4	0.893763	107	149.5932	236.2338
20	2	3	2	3	3	0.892416	88	146.1247	195.5346
21	4	2	2	4	2	0.892198	108	165.872	223.7867
22	4	3	3	3	2	0.891953	105	169.0799	245.5109
23	2	2	2	3	5	0.888316	110	149.3401	271.0126
24									
25									
26									

Model solutions

Enter

Figure 3.9: Sensitivity analysis report

for the problem with 15 decision variables; and 7 seconds and 16 seconds, for typical 20 and 25 variable problems, respectively.

3.3 The RROS Application Design

The design of the RROS is broken up into three distinct modules: GUI, optimization, and analysis. A systematic view of the dynamics between the user and the application is represented by he sequence diagram in Figure 3.10. Through the GUI, the user initializes a model with the basic parameters and the GUI module creates the model skeleton. The user then inputs the

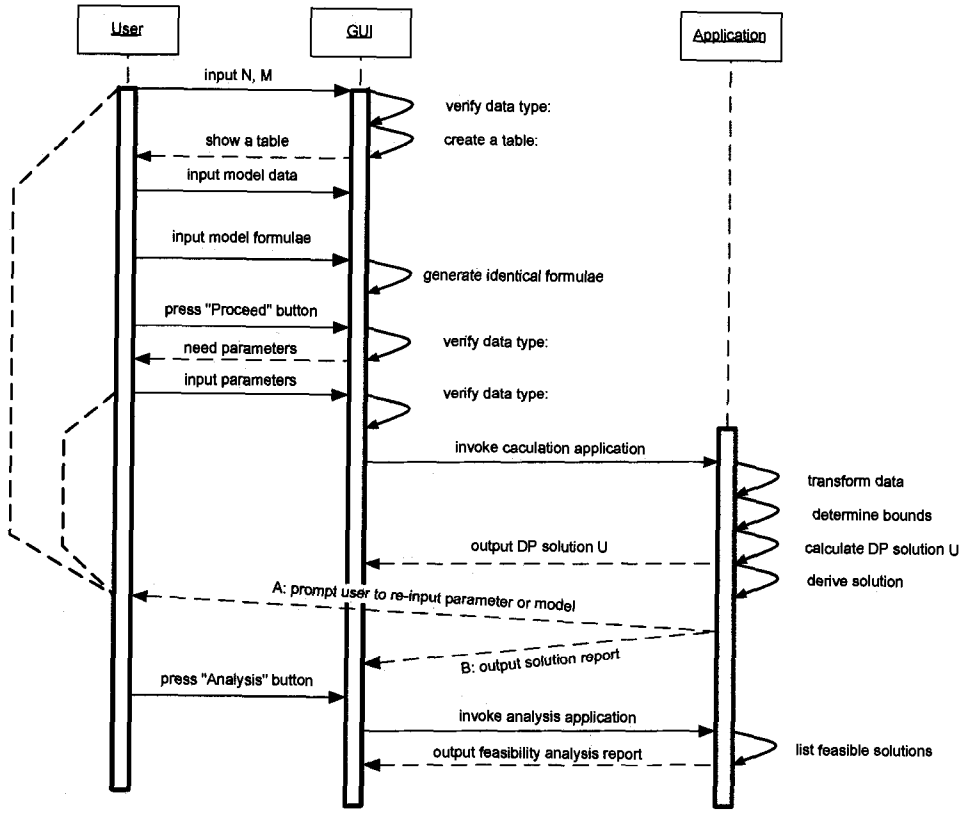

Figure 3.10: The system sequence diagram represents a best use case scenario for the interaction between User, GUI and Application

model data, including formulae, coefficient and RHS of the objective function and constraints; and optionally bounds to the decision variable. During that stage, the GUI also copies formulae to adjacent cells. When the user clicks on the *Proceed* button, the GUI verifies the input data type. The user may input optimisation parameters, e and the key constraint; again, the GUI verifies the parameter data. The GUI then invokes the optimization application to find the optimal solution. If an optimal solution is found, the optimization module sends the solution to the GUI and the GUI then displays it in specified Excel cells; otherwise, the application will prompt the user to modify the parameters or the model. Otherwise, the GUI will prompt the user to modify the input data. Similarly, the *Analysis* button will invoke the analysis module which will report results through the GUI.

The major functions in the RROS application are programmed in VBA. Figure 3.11 shows that the RROS can also be decomposed in three components: input, calculation and output.

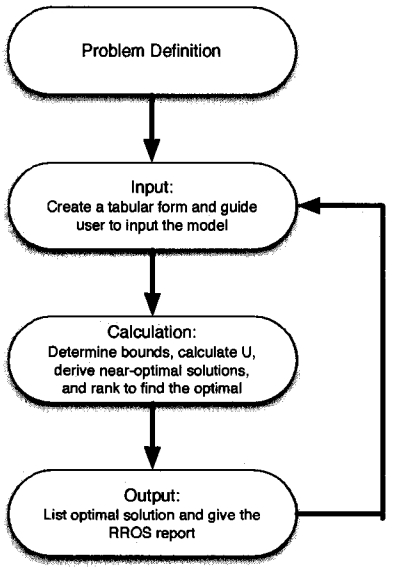

Figure 3.11: Flow chart of the RROS application

The input section initializes a tabular model, names specific ranges, assists users to input model formulae, and validates input data. A *Proceed* button can finalize the input process, and invoke the calculation section.

The calculation section applies the RROS algorithm, which determines an optimal solution (if one exists) so that all the constraints are satisfied and the objective function is maximized, if there is one.

The RROS first collects the model data and determines a key constraint. Normally, an equality constraint has higher priority than an inequality one. If there is no equality constraint, an upper-bound constraint (of the form $P_{ij} \leq d_i$) with the smallest upper bound (RHS) value will be selected as the key constraint. Users' input on the key constraint overrides the RROS's choice.

The lower bounds are set by default to 1 for models with an additive objective function and 0 for models with a multiplicative objective function. The RROS then chooses the lower bounds as the tightest (highest) between the default lower bounds and the user's input. The upper bounds are first computed by the RROS based on the Prased and Kuo technique [Prasad and Kuo, 2000]. In a similar process, the calculated upper bounds are compared with the input upper bounds; and the tighter ones (lowest) are chosen by the RROS.

The backward recursion technique of dynamic programming is employed in order to get the upper bound of the objective function. The RROS then derives all near-optimal solutions in the range $[U-e, U]$ and ranks the objective values generated by these solutions in decreasing order. Of course, those solutions satisfy the key constraint, but may not satisfy the other constraints. The next step is to test those solutions with the remaining constraints, including non-decreasing and general constraints. The best solution that satisfies all the constraints is the optimal solution to the problem.

The output section not only provides optimization results; it also generates the feasibility report. The optimization summary report is highlighted in yellow and includes the decision variable values, the optimal objective function value, the number of variables, the number of constraints, the computing time, search depth, DP solution and the choice on the key constraint.

Chapter 4

Illustrative Examples and Results

This chapter selects a range of reliability redundancy problems to demonstrate and evaluate the RROS and the RROS surrogate method. The testing problems are either solved directly by the RROS or worked out through the RROS surrogate method.

It has been noted that the overall complexity of a nonlinear integer programming problem is not determined only by the size of the problem; the degree of nonlinearity on the objective and the constraint functions also contributes significantly to its complexity. A small-sized problem can be more difficult to solve than a relatively large-sized problem because of a high degree of nonlinearity. Most problems, discussed in this chapter, have objective functions and constraints with high nonlinearity. For this reason, this chapter also provides exact solutions for problems with less than 10 variables, using a full enumeration method. The enumeration results and computing performance are compared with the RROS.

The overall results indicate that the RROS offers accurate solutions in a relatively efficient way. All testing problems in this chapter are run on a Sony desktop equipped with a 2.8GHz Pentium 4 processor and 1Gb 400MHz DDR memory.

4.1 Problems with Separable Objective Function

A total of 15 problems (listed in Table 4.1) are tested here. The number of decision variables in these problems ranges from three to sixty.

Problem Class	Test	Number of Variables	Number of Constraints	Reference
Series	$S1$	5	3	[Tillman et al., 1977; Prasad and Kuo, 2000; Kuo and Prasad, 2000; Ravi et al., 1997; Ng and Sancho, 2001]
Series	$S2$	10	3	[Ng and Sancho, 2001]
Series	$S3$	4	2	[Ravi et al., 1997]
Series	R1	15	4	Randomly Generated [Prasad and Kuo, 2000]
Series	R2	15	4	Randomly Generated [Prasad and Kuo, 2000]
Series	R3	20	4	Randomly Generated [Prasad and Kuo, 2000]
Series	R4	20	4	Randomly Generated [Prasad and Kuo, 2000]
Series	R5	25	4	Randomly Generated [Prasad and Kuo, 2000]
Series	R6	30	4	Randomly Generated [Prasad and Kuo, 2000]
Series	R7	40	4	Randomly Generated [Prasad and Kuo, 2000]
Series	R8	50	4	Randomly Generated [Prasad and Kuo, 2000]
Series	R9	60	4	Randomly Generated [Prasad and Kuo, 2000]
k-out-of-n	$KN1$(B=75)	3	3	[Nakagawa and Miyazaki, 1981; Ng and Sancho, 2001]
k-out-of-n	$KN1$(B=70)	3	3	[Nakagawa and Miyazaki, 1981; Ng and Sancho, 2001]
k-out-of-n	$KN2$	4	3	[Prasad and Kuo, 2000]

Table 4.1: List of test problems with separable objective functions that the RROS can solve directly

4.1.1 Series-parallel Problems with Identical Components in Each Subsystem

[**Problem** $S2$]: Similar to Problem $S1$ discussed in Chapter 2, Problem $S2$ has a series-parallel configuration, but has 10 variables. The input data taken from Ng and Sancho [2001] is provided in Table 4.2.

The RROS calculates upper bounds for decision variables: they are [6,6,6,6,6,5,6,6,5,5]; on

r_j	[0.80, 0.82, 0.60, 0.78, 0.72, 0.80, 0.85, 0.90, 0.65, 0.75]	
p_j, P	[1, 3, 4, 4, 3, 5, 2, 3, 5, 2]	200
c_j, C	[7, 6, 5, 8, 6, 3, 7, 5, 9, 4]	300
w_j, W	[6, 7.5, 7.5, 7.5, 5.5, 4, 8, 8, 4, 9]	300

Table 4.2: Input data for Problem $S2$

the other hand, Ng and Sancho [2001] reports the upper bounds as [7,6,6,6,7,6,6,6,6,6], which are not as tight as bounds that RROS generated.

Table 4.3 compares the RROS calculation performance by choosing different key constraints manually. The RROS chooses the constraint 2.11, based on P, which also has the shortest execution time. When one forces the RROS to use the cost constraint (System 2.12), it can not find a feasible solution when the search depth is set at its default value of 5% of the DP solution ($e = 5\% * U$). In that case, the optimal solution for $S2$ is less than the lower bound of the search range (0.7989=0.84103*95%). So for this problem, users would have to either choose another key constraint or to manually increase the search depth. Both the test using the product of weight and volume constraint (System 2.11) as the key constraint and the test using the weight constraint (System 2.13) provide the same optimal solution [3,2,3,2,3,2,2,2,3,2] with a resulting reliability of 0.69705.

Key constraint	DP sol. U	Search depth	# of near-optimal solutions	CPU time (sec.)
System 2.11 (P)	0.70829	$5\% * U$	36	15
System 2.12 (C) *	0.84103	$5\% * U$	326	62
System 2.13 (W)	0.70132	$5\% * U$	23	18

Table 4.3: RROS performance in Problem $S2$
* The test using System 2.12 as the knapsack constraint could not find a feasible solution while $e = 5\% * U$

In both Problems $S1$ and $S2$, the test using the product of weight and volume constraint

(System 2.11) as the key constraint has a slightly greater DP solution and more near-optimal solutions; however, it features better computation performance. Even though the number of variables and the input data for the both problems are different, the two problems share the same system structure (configuration); this structure determines the pattern of their computation performance.

[Problem $S3$]: a 4-variable series-parallel system can be formulated as follows:

$$max \prod_{j=1}^{n}[1-(1-r_j)^{x_j}] \tag{4.1}$$

subject to

$$\sum_{j=1}^{n} c_j \cdot x_j \leq C \tag{4.2}$$

$$\sum_{j=1}^{n} w_j \cdot x_j \leq W \tag{4.3}$$

R_j	[0.8, 0.7, 0.75, 0.85]	
c_j, C	[1.2, 2.3, 3.4, 4.5]	56
w_j, W	[5, 4, 8, 7]	120

Table 4.4: Input data for Problem $S3$

In Problem $S3$, the upper bounds for the decision variables are determined to be $[20, 20, 13, 10]$ by the RROS. The loose upper bounds require considerably more computing effort, even though the size of the problem is only 4 variables. Both tests in Table 4.5 give the same optimal solution $[5, 6, 5, 4]$ for a system reliability of 0.9975. This result matches the result generated from a full enumeration test. The enumeration takes the CPU 210 seconds and finds a total of 4163 feasible solutions.

Key constraint	DP sol. U	Search depth	# of near-optimal solutions	CPU time (sec.)
System 4.2	0.99773	$5\% * U$	1019	19
System 4.3	0.99747	$5\% * U$	825	27

Table 4.5: RROS performance in Problem $S3$

Table 4.6 compares the result obtained by the RROS algorithm with those reported in the literature. This is given for indicative purposes only; it is impossible to normalize those results strictly, as they come from different software and hardware architectures, the details of which are often omitted in the literature. In addition, variation in the values of key parameters used in each method often play a critical role in the computation performance. However, when computer time and speed are given, a normalized time is given, which seems to show that RROS implementation compares favourably with other implementations.

Problem	Method	Reference	Programming Language	CPU (MHz)	Time (sec.)	Adjusted time (s)*	Parameters
	RROS	RROS	VBA	2800	3	3	$e = 5\% * U$
$S1$	DP/DFS	[Ng and Sancho, 2001]	LINGO6	266	48	5	$e = 0.5\% * U$
	SA	[Ravi et al., 1997]**			37		
	I-NESA	[Ravi et al., 1997]			16		
	RROS	RROS	VBA	2800	15	15	
$S2$	DP/DFS	[Ng and Sancho, 2001]	LINGO6	266	190	18	
	RROS	RROS	VBA	2800	19	19	
$S3$	SA	[Ravi et al., 1997]			9		
	I-NESA	[Ravi et al., 1997]			3		

Table 4.6: Problem $S1$: performance comparison of different methods
* This is a normalized execution time, using the CPU speed of the RROS testing machine.
* Ravi et al. [1997] does not provide the CPU type and the programming language.

The testing results for $S1$, $S2$ and $S3$ are summarized in Table 4.7.

Test	lower bounds l_j	upper bounds u_j	Optimal solution	Reliability
$S1$	[1,1,1,1,1]	[5,5,5,5,5]	[3,2,2,3,3]	0.90447
$S2$	[1,1,1,1,1,1,1,1,1,1]	[6,6,6,6,6,5,6,6,5,5]	[3,2,3,2,3,2,2,2,3,2]	0.697051
$S3$	[1,1,1,1]	[20,20,13,10]	[5,6,5,4]	0.9975

Table 4.7: The RROS solutions for $S1$, $S2$ and $S3$

[**Problems** *R*1-*R*9]: These 9 testing problems are randomly generated to demonstrates the RROS's ability to solve large-sized problems [Prasad and Kuo, 2000]. The generated problems can be formalized by Systems 4.4 to 4.8. The input data for the 9 problems and the optimal solutions generated by the RROS are provided in Appendix B.

$$max \prod_{j=1}^{n}[1-(1-r_j)^{x_j}] \tag{4.4}$$

subject to

$$\sum_{j=1}^{n} \alpha_j \cdot x_j^2 \leq (1+\frac{\theta}{100}) \cdot \sum_{j=1}^{n} \alpha_j \cdot l_j^2 \tag{4.5}$$

$$\sum_{j=1}^{n} \beta_j \cdot exp(x_j/2) \leq (1+\frac{\theta}{100}) \cdot \sum_{j=1}^{n} \beta_j \cdot exp(l_j/2) \tag{4.6}$$

$$\sum_{j=1}^{n} \gamma_j \cdot x_j \leq (1+\frac{\theta}{100}) \cdot \sum_{j=1}^{n} \gamma_j \cdot l_j \tag{4.7}$$

$$\sum_{j=1}^{n} \delta_j \cdot \sqrt{x_j} \leq (1+\frac{\theta}{100}) \cdot \sum_{j=1}^{n} \delta_j \cdot \sqrt{l_j} \tag{4.8}$$

$1 \leq x_j \leq 10$ for $j = 1, 2, ..., n$

The component reliability r_j is generated from a uniform distribution in [0.95, 1.0]; the coefficients $\alpha_j, \beta_j, \gamma_j, \delta_j$ are generated from uniform distributions in [6, 10], [1, 5], [11, 20], and [21, 40], respectively. θ is a resource parameter that determines the level of additional resources other than the minimal requirement (based on l_j, the lower bound of x_j). When $\theta = 0$, there is only one feasible solution for the problem, that is, $x_j = l_j$. The difficulty of the problem grows significantly when θ increases. Table 4.8 lists the test results and RROS performance.

In Table 4.8, both problem *R*3 and *R*4 have 20 variables and they both have the same formulae and coefficients, except that *R*4 has greater resource values (RHS of the constraints), $\theta = 300$. For example, in R3, it takes the CPU 46 seconds to find 100 near-optimal solutions

Test	# of variables	θ	x_j bound	Searching range $[U-e,U]$	e/U ratio	# of near optimal soln	CPU time (sec.)
$R1$	15	33	[1,2]	[0.6598,0.6977]	5%	21	7
$R2$	15	50	[1,3]	[0.6862,0.7224]	5%	93	22
$R3$	20	33	[1,3]	[0.5844,0.6151]	5%	100	46
$R4$	20	300	[1,8]	[0.9550,0.9795]	2.5%	1138	758
$R5$	25	33	[1,3]	[0.6084,0.6404]	5%	150	112
$R6$	30	100	[1,6]	[0.6922,0.7064]	2%	91	486
$R7$	40	33	[1,4]	[0.3891,0.3991]	2.5%	223	560
$R8$	50	33	[1,4]	[0.3610,0.3710]	2.7%	622	976
$R9$	60	33	[1,4]	[0.2280,0.2330]	2.1%	282	3766

Table 4.8: RROS Performance on the randomly generated problems

bracketed within interval [0.5844, 0.6151]. However, when θ increases from 33 (Test R3) to 300 (Test 4), even with $2.5\% * U$ search depth, Test $R4$ requires nearly 4 minutes to find the optimal.

Problem $R5$ has 25 variables and the CPU takes approximately 2 minutes to find the optimal, which is less than problem $R4$ requires. The decision variables in $R5$ are modestly bounded ($1 \leq x_j \leq 3$) because its RHS values are less than those of its counterpart, $R4$ ($1 \leq x_j \leq 8$).

$R9$ represents a problem with 60 variables. The decision variables are modestly bounded ($1 \leq x_j \leq 4$). Setting e to be 2.1% of DP solution, it takes the RROS approximately CPU 63 minutes to generate the optimal solution among 282 near-optimal solutions. A full enumeration process for $R9$ would identify $4^{60} \geq (2^{32})^3$ possible solutions and test each for constraint satisfaction in order to find the optimal.

Table 4.9 compares the calculation performance of the RROS algorithm and Prasad and Kuo [2000]

Algorithm	Programming Language	CPU	CPU Time (sec.)
RROS Alg.	VBA	Intel Pentium 4 2.8 GHz	560
Prasad&Kuo Alg. [Prasad and Kuo, 2000]	FORTRAN 77	POWER CHALLENGE L 6x10000	22
Lawer &Bell Alg. [Prasad and Kuo, 2000]	FORTRAN 77	POWER CHALLENGE L 6x10000	1843

Table 4.9: Problem $R7$ (40 variables): performance comparison between the RROS and Prasad and Kuo [2000]

4.1.2 Optimal Component Choice and Redundancy in a Series System (k-out-of-n problem)

[Problem $KN1$]: Consider maximizing the system reliability for a system which is comprised of three subsystems operating in a series. This system involves choosing the most reliable component out of four candidates in subsystem 1, adding redundant components in parallel in subsystem 2 and using a 2-out-of-(x_3+1): G configuration in subsystem 3. This problem has been studied by Nakagawa and Miyazaki [1981] and Ng and Sancho [2001].

$$max \prod_{j=1}^{3} R_j'(x_j) \tag{4.9}$$

subject to

$$4 \cdot exp(\frac{0.02}{1-R_1'(x_1)}) + 5 \cdot x_2 + 2 \cdot (x_3+1) \leq 45 \tag{4.10}$$

$$5 + exp(x_1/8) + 3 \cdot (x_2 + exp(x_2/4)) + 5 \cdot (x_3 + 1 + exp(x_3/4)) \leq B \tag{4.11}$$

$$10 + 8 \cdot x_2 \cdot exp(x_2/4) + 6 \cdot x_3 \cdot exp(x_3/4) \leq 240 \tag{4.12}$$

where $R_1'(x_1) = 0.88, 0.92, 0.98, 0.99$ for $x_1 = 1,2,3,4$ respectively,

$$R_2'(x_2) = 1 - (1-0.81)^{x_2}$$

$$R_3'(x_3) = \sum_{k=2}^{x_3+1} \binom{x_3+1}{k} \cdot 0.77^k \cdot (1-0.77)^{x_3+1-k}$$

$$B = 70,75$$

[**Problem** *KN*2]: Similar to problem *KN*1, problem *KN*2 has 4 subsystems in a series. This system involves choosing the most reliable component out of 6 in subsystem 1, using a 2-out-of (x_3+1): G configuration in subsystem 3, and using parallel structures in subsystem 2 and 4. Prasad and Kuo have studied this following problem in [Prasad and Kuo, 2000]. Specifically,

$$max \prod_{j=1}^{4} R_j(x_j)$$

subject to

$$10 \cdot exp(\frac{0.02}{1-R_1 \cdot (x_1)}) + 10 \cdot x_2 + 6 \cdot x_3 + 15 \cdot x_4 \leq 150$$

$$10 \cdot exp(x_1/2) + 4 \cdot exp(x_2) + 2 \cdot (x_3 + exp(x_3/4)) + 6 \cdot x_4^2 \leq 320$$

$$40 \cdot x_1^2 + 6 \cdot exp(x_2) + 3 \cdot x_3 \cdot exp(x_3/4) + 8 \cdot x_4^3 \leq 2400$$

where $R_1(x_j) = 0.94, 0.95, 0.96, 0.965, 0.97, 0.975$ for $x_1 = 1,2,3,4,5,6$ respectively,

$$R_2(x_2) = 1 - (1-0.75)^{x_2}$$

$$R_3(x_3) = \sum_{k=2}^{x_3+1} \binom{x_3+1}{k} \cdot 0.88^k \cdot (1-0.88)^{x_3+1-k}$$

$$R_4(x_4) = 1 - (1-0.95)^{x4}$$

Table 4.10 lists the solutions obtained by the RROS for Problem *KN*1 and *KN*2. The RROS finds the same optimal solution for Problem *KN*1 as the one that Nakagawa and Miyazaki [1981] reports finding through a heuristic approach; and it finds the same optimal solution for

Problem $KN2$ as the one reported by Prasad and Kuo [2000].

Test	l_j	u_j	Searching range $[U-e,U]$	# of near optimal sol.	Optimal solution	Reliability	CPU time (sec.)
$KN1$ (B=75)	[1,1,1]	[30,6,6]	[0.9290,0.9779]	12	[3,4,5]	0.9756	4
$KN1$ (B=70)	[1,1,1]	[29,6,5]	[0.9269,0.9757]	9	[3,3,5]	0.9701	4
$KN2$	[1,1,1,1]	[6,4,15,6]	[0.9225,0.9711]	257	[6,3,4,2]	0.9565	7

Table 4.10: The RROS report on Problem $KN1$ and $KN2$

4.2 Problems with Non-separable Objective Functions

Section 2.3.1 on page 43 already showed an example of a 3-stage series system with redundancy units in parallel, which had non-separable objective function. This section will show how the surrogate method can be applied to complex systems, whose architecture cannot be described as hierarchical.

[**Problem** $CR1$]: Figure 4.1 shows a typical bridge system with 5 components. The system can be formulated as maximizing reliability R_s subject to the total weight, the total cost and the total product of weight and volume constraints (Systems 4.15 to 4.17).

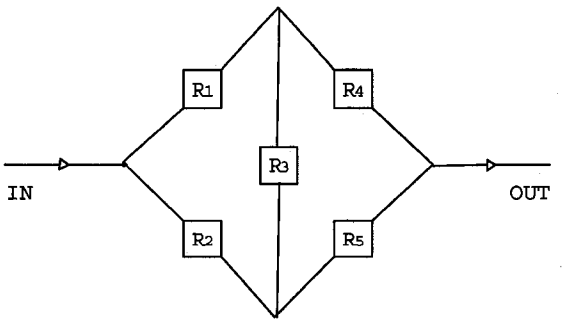

Figure 4.1: A bridge network

$$R_S = R_1 \cdot R_4 + R_2 \cdot R_5 + R_2 \cdot R_3 \cdot R_4 + R_1 \cdot R_3 \cdot R_5$$

$$+2R_1 \cdot R_2 \cdot R_3 \cdot R_4 \cdot R_5 - R_2 \cdot R_3 \cdot R_4 \cdot R_5$$
$$-R_1 \cdot R_3 \cdot R_4 \cdot R_5 - R_1 \cdot R_2 \cdot R_4 \cdot R_5$$
$$-R_1 \cdot R_2 \cdot R_3 \cdot R_5 - R_1 \cdot R_2 \cdot R_3 \cdot R_4 \quad (4.13)$$

$$R_j = 1-(1-r_j)^{x_j} \; j \in \{1,2,3,4,5\} \quad (4.14)$$

$$\sum_{j=1}^{n} p_j \cdot x_j^2 \leq P \quad (4.15)$$

$$\sum_{j=1}^{n} c_j \cdot (x_j + exp(x_j/4)) \leq C \quad (4.16)$$

$$\sum_{j=1}^{n} w_j \cdot x_j \cdot exp(x_j/4) \leq W \quad (4.17)$$

The input data for $CR1$ is given in Table 4.11:

R_j	[0.80,0.85,0.90,0.65,0.75]	
p_j,P	[1,2,3,4,2]	110
w_j,W	[7,7,5,9,4]	175
c_j,C	[7,8,8,6,9]	200

Table 4.11: Input data for Problem $CR1$

The upper bounds for the decision variables in $CR1$ are [6,5,5,5,5] and the lower bound for each variable is 0. Applying the RROS surrogate method, described in Chapter 2, the weight constraint (4.17) is used as the surrogate objective, which defines, as in section 2.3.1, a surrogate problem $SP1$. A continuous surrogate problem $SP2$ is defined in the same manner as before. Table 4.12 shows the steps for the RROS surrogate process. The optimal solution [2,3,2,3,3] generated by the RROS surrogate method matches the one obtained by an enumeration practice. Note that, as the problem is comparatively small, the enumeration of solutions is faster than using the surrogate method; but with more variables or more relaxed bounds, it is likely not to

	Problem	Program	Optimal reliability	CPU time (second)	# of Solutions
Step 1	$SP1$,$CR1$ ($e = 0.05 * U$)	RROS surrogate	0.99918784	20	406 near-optimal sol.
Step 2	$SP2$ ($W_0 = 190$)	Microsoft Solver	0.99943647	<2sec.	
Step 3	$SP1$,$CR1$ ($e = 0.1 * U$)	RROS surrogate	0.99918784	38	705 near-optimal sol.
Step 4	$SP2$ ($W_0 = 180$)	Microsoft Solver	0.99926410	<2sec.	
Step 5	$SP1$,$CR1$ ($e = 0.15 * U$)	RROS surrogate	0.99918784	73	926 near-optimal sol.
Step 6	$SP2$ ($W_0 = 170$)	Microsoft Solver	0.99902871	<2sec.	
Control	$CR1$	Enumeration	0.99918784	22	495 feasible sol.

Table 4.12: The RROS surrogate process for problem $CR1$

be the case, as was shown with problems such as $R9$ in the preceding section.

[**Problem** $CR2$]: Figure 4.2 illustrates a complex system with 7 components.

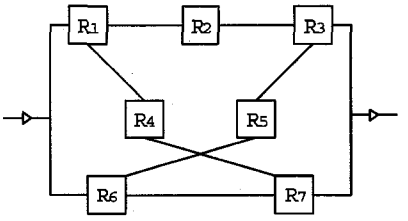

Figure 4.2: 7-component complex network

The problem can be described as maximizing the system reliability subject to the total product of weight and volume ($\sum p_j \cdot x_j^2 \leq P$) and the total weight ($\sum w_j \cdot x_j \leq W$). The system reliability is given by the following equations:

$$\begin{aligned} R_S &= R_4 \cdot R_5 \cdot \Psi_1 + R_4 \cdot (1 - R_5) \cdot \Psi_2 \\ &+ R_5 \cdot (1 - R_4) \cdot \Psi_3 \\ &+ (1 - R_4) \cdot (1 - R_5) \cdot \Psi_4 \end{aligned} \tag{4.18}$$

$$\begin{aligned}
\Psi_1 &= R_1 \cdot R_2 \cdot R_3 + R_1 \cdot R_7 + R_3 \cdot R_6 \\
&+ R_6 \cdot R_7 - R_1 \cdot R_2 \cdot R_3 \cdot R_7 - R_1 \cdot R_2 \cdot R_3 \cdot R_6 \\
&- R_1 \cdot R_6 \cdot R_7 - R_3 \cdot R_6 \cdot R_7 + R_1 \cdot R_2 \cdot R_3 \cdot R_6 \cdot R_7 \qquad (4.19) \\
\Psi_2 &= R_1 \cdot R_2 \cdot R_3 + R_1 \cdot R_7 + R_6 \cdot R_7 \\
&- R_1 \cdot R_2 \cdot R_3 \cdot R_7 - R_1 \cdot R_6 \cdot R_7 \qquad (4.20) \\
\Psi_3 &= R_1 \cdot R_2 \cdot R_3 + R_3 \cdot R_6 + R_6 \cdot R_7 \\
&- R_1 \cdot R_2 \cdot R_3 \cdot R_6 - R_3 \cdot R_6 \cdot R_7 \qquad (4.21) \\
\Psi_4 &= R_1 \cdot R_2 \cdot R_3 + R_6 \cdot R_7 \\
&- R_1 \cdot R_2 \cdot R_3 \cdot R_6 \cdot R_7 \qquad (4.22) \\
R_j &= 1 - (1 - r_j)^{x_j} \; j \in \{1,2,3,4,5,6,7\} \qquad (4.23)
\end{aligned}$$

The input data is given in Table 4.13:

R_j	[0.80,0.70,0.80,0.90,0.80,0.80,0.70]	
p_j,P	[4,6,9,1,9,3,1]	60
w_j,W	[2,9,10,5,2,1,2]	70

Table 4.13: Input data for Problem *CR*2

The upper bounds for decision variables in *CR*2 are [3, 3, 2, 7, 2, 4, 7] and the lower bound for each variable is 0. The product of weight and volume is used as the surrogate objective. The optimal solution [1, 0, 0, 2, 0, 3, 5] generated by the RROS surrogate method matches the one found by the enumeration practice. Table 4.14 provides a summary of the RROS surrogate process.

	Problem	Program	Optimal reliability	CPU time (minutes)	# of Solutions
Step 1	$SP1$,$CR2$ ($e = 0.05 * U$)	RROS surrogate	0.99591	12	4733 near-optimal sol.
Step 2	$SP2$ ($P_0 = 57$)	Microsoft Solver	0.99607		
Step 3	$SP1$,$CR2$ ($e = 0.1 * U$)	RROS surrogate	0.99591	43	8053 near-optimal sol.
Step 4	$SP2$ ($P_0 = 54$)	Microsoft Solver	0.9953		
Control	$CR2$	Enumeration	0.99591	37	4363 feasible sol.

Table 4.14: The RROS surrogate process for problem $CR2$

Chapter 5

Conclusion

This thesis work has proposed an Excel optimization tool, "reliability redundancy optimization solver" (RROS), and a RROS surrogate method for nonlinear integer optimization problems. This application enables users to solve resource allocations problems in a spreadsheet environment with little or no formal OR/MS training.

The RROS is designed for solving a class of separable nonlinear multidimensional problems. This application integrates and implements a hybrid "dynamic programming/depth-first search" (DP/DFS) algorithm, as well as an upper-bound technique, in a Microsoft Visual Basic Application environment. In addition, the RROS can easily generate a feasibility analysis report that lists feasible lattice points with their corresponding constraint resource values. This kind of report is often critical for model interpretation and quantitative reasoning.

This thesis has further introduced a novel RROS surrogate method to solve nonlinear problems with non-separable objectives. With the RROS surrogate method, one of the separable constraints of the original problem serves as the surrogate objective in the surrogate problem. The surrogate problem has the same constraints as the original problem; therefore, both

problems share the same feasible solutions. In addition to providing an approximation of the optimal solution for the original problem, the RROS surrogate method establishes a procedure to determine the optimal for the original problem.

Several benchmark examples, including randomly generated problems, have been discussed in the thesis to demonstrate and evaluate the RROS and the RROS surrogate method. The overall results indicate that the RROS offers accurate solutions in a relatively efficient way.

Future Work

There are many potential extensions to the RROS and the RROS surrogate method. One of the potential extensions could be to solve multi-objective optimization problems. An objective with separable and monotonic function from the original objectives can be selected as the surrogate objective in a relaxed surrogate problem with a single objective. If no such objective exists in the original multi-objective problem, the separable constraint from the original problem can be used as the surrogate objective; therefore, the RROS surrogate method can be applied.

There is also some room for future improvement in the application implementation and deployment. The RROS is written in VBA and packaged as an Add-in file with an .XLA extension. This form is convenient for the Microsoft Excel user, but is not useful to users who run optimizations on other platforms. Building a RROS DLL (dynamic link library) can extend the applicability of the RROS to users who prefer to practice optimization in their own development environments, other than Excel. DLL files, often written in C or Fortran Language, contain executable routines. Faster computation performance, security and ease of distribution are major advantages of DLL. For Microsoft users, a smaller VBA program could be written to link the RROS DLL and Excel program, so that users would experience no difference between using the RROS DLL and the RROS add-in (presented in this thesis). For other optimizers and developers, the RROS DLL is a flexible function routine that they could use with their own work platforms.

Bibliography

S.C. Albright. Premium solver platform for excel. *OR/MS Today*, 28(3):58–63, June 2001.

S.C. Albright. Solvertable, [online]. URL `http://www.columbia.edu/~dj114/6015-st.htm`.

R.E. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

R.E. Bellman and E. Dreyfus. Dynamic programming and reliability of multicomponent devices. *Operationa research*, 16:200–208, Mar-Apr 1958.

R.E. Bellman and E. Dreyfus. *Applied Dynamic Programming*. Princeton University Press, 1962.

G.A. Bohoris. The use of reliability assessment technologies to improve the quality of engineering design. In W. Kuo, editor, *Quality through Engineering Design*, volume 16 of *Advances in Industrial Engineering*, page 245. Bangalore, India, 11-14 January 1993, Elsevier Science Publishers B.V., 1993.

M. F. Cardoso, R. L. Salcedo, and S. F. de Azevedo. Non equilibrium simulated annealing: a faster approach to combinatorial minimization. *Industrial Eng'g Chemical Research*, 33: 1908–1918, 1994.

M.W. Carter and C.C. Price. *Operations Research: A Practical Introduction*. CBC Press, 2001.

M.S. Chern. On the computational complexity of relaibility redundancy allocation in a series system. *Operations Research Letters*, 11:309–315, June 1992.

M.S. Chern and R. Jan. Reliability optimization problems with multiple constraints. *IEEE Transactions on Reliability*, R-35(4):431–436, October 1986.

J.W. Chinneck. Practical optimization: A gentle introduction, [online]. 2003. URL `http://www.sce.carleton.ca/faculty/chinneck/po.html`.

David W. Coit and Alice E. Smith. Optimization approaches to the redundancy allocation problem for series-parallel systems. In *Proceedings of the 4th Industrial Engineering Research Conference (IERC)*, 1995.

Eudoxus, [online]. Eudoxus Systems Ltd. URL `http://www.eudoxus.com`, [cited June 1st 2005].

H. Everett. Generalized lagrange multiplier method for solving problems of optimum allocation of resources. *Operations Research*, 11:399–417, 1963.

R. Fourer. Software for optimization: A buyer's guide. *INFORMS Computer Science Technical Section Newsletter*, 17(1), 1996. URL `http://www.ampl.com/buyers1.html`.

D.E. Fyffe, W.W. Hines, and N.K. Lee. System reliability allocation and a computatuinal algorithm. *IEEE Transaction On Reliability*, R-17:64–69, June 1968.

D. Fylstra, L. Lasdon, J. Watson, and A. Waren. Design and use of the microsoft excel solver. *Interface*, 28(5):215–225, 1998.

F. Glover. Surrogate constraint duality in mathematical programming. *Operations Research*, 23(3):434–451, 1975.

H. J. Greenberg and W. P. Pierskalla. Surrogate mathematical programming. *Operations Research*, 18:924–939, 1970.

T.A. Grossman. Spreadsheet add-ins for or/ms software survey. *OR/MS Today*, August 2002.

F.S. Hillier and G.J. Lieberman. *Introduction to Operations Research.* McGraw hill Higher Education, eighth edition, 2005.

J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

Ilog, [online]. ILOG, Inc. URL `http://www.ilog.com`, [cited August 1, 2007].

W.G. Ireson and C.F. Coombs. *Handbook of reliability engineering and management.* McGraw-Hill, 1988.

W. Kuo and V.R. Prasad. An annotated overview of system-reliability optimization. *IEEE Transactions on Reliability*, 49(2):176–187, June 2000.

W. Kuo and M. J. Zuo. *Optimal reliability modeling: principles and applicaitons.* John Wiley & Sons, 2002.

W. Kuo, V.R. Prasad, F.A. Trillman, and C. Hwang. *Optimal reliability design: fundermentals and applications.* The Press Syndicate of the University of Cambridge, 2001.

Lindo, [online]. Lindo System Inc., 2003. URL `http://www.lindo.com`, [cited Nov. 20 2005].

T.M. Magee and F. Glover. Integer programming. In M. Avriel and B. Golany, editors, *Mathematical Programming in Industrial Engineers*, chapter 3, pages 123–261. Marcel Dekker Inc., 1996.

S.V. Majety, M. Dawande, and J. Rajgopal. Optimal reliability allocation with discrete cost-reliability data for components. *Operations Research*, 47(6):699–906, November-December 1999.

Hans .D. Mittelmann. Decision tree for optimization software, [online]. 2007. URL `http://plato.la.asu.edu/topics/problems/nlores.html`, [cited August 1 2007].

K. G. Murty. *Operations Research: Deterministic Optimization Models*. Prentice-Hall, Inc., 1995.

Y. Nakagawa and S. Miyazaki. Surrogate constraints algorithm for reliability optimization problems with two constraints. *IEEE Transaction On Reliability*, R-30:175–180, june 1981.

K.Y.K. Ng and N.G.F. Sancho. A hybrid dynamic programming / depth-first search algorithm, with an application to redundancy allocation. *IEEE Transaction*, 33:1047–1058, 2001.

M. Obitko. Introduction to genertic algorithm, [online]. 1998. URL `http://cs.felk.cvut.cz/~xobitko/ga/`, [cited June 1st, 2005].

Palisade, [online]. Palisade Corporation. URL `http://www.palisade.com/`, [cited June 1st, 2005].

D. J. Power. A brief history of spreadsheets, [online]. DSSResources.COM, June 2004. URL `http://dssresources.com/history/sshistory.html`.

V. R. Prasad and W. Kuo. Reliability optimization of coherent systems. *IEEE Transactions on Reliability*, 49(3):323–330, 2000.

C.T. Ragsdale. *Spreadsheet Modeling and Decision Analysis, a Practical Introduction to Management Science*. Course Technology, Inc, Cambridge, MA, 1998.

V. Ravi, B. Murty, and P. Reddy. Nonequilibrium simulated annealing-algorithm applied to reliability optimization of complex systems. *IEEE Transaction On Reliability*, 46(2): 233–239, June 1997. Bridge5 example.

S.M. Sait and H. Youssef. *Iterative Computer Algorithms with Applications in Engineering: solving combinatorial optimization problems*. Angela Burgess, 1999. QA402.5.S34 1999.

S. Savage. Weighing the pros and cons of decision technology in spreasheets. *OR/MS Today*, 24(1):1–9, February 1997.

Solver, [online]. FrontLine System Inc. URL `http://www.solver.com`, [cited Aug,2007].

Spreadsheet training and consulting, [online]. System Modelling Ltd. URL `http://www.sysmod.com/spreads.htm#Training`.

H.A. Taha. *Operations Research: an introduction.* Prentice-Hall, Upper Saddle River, NJ, 1997.

F.A. Tillman and J.M. Liittschwager. Integer programming formulation and constrained reliability problems. *Management Science*, 13:887–899, 1967.

F.A. Tillman, C. Hwang, and W. Kuo. Optimization techniques for system reliability with redundancy - a review. *IEEE Transactions on Reliability*, R-26(3):148–153, August 1977. Previous survey for Reliability Engineering.

M. Trick. Integer programming for consultants, [online]. 1996. URL `http://mat.gsia.cmu.edu/mstc/integer/integer.html`.

List of Figures

List of Tables

Appendix A: List of the Spreadsheet Add-in Products

Product Vendor	Commercial	Limitations	Price	Comments
LGO Global Solver Engine V5.0	Frontline Systems, Inc.	NLP/Global-1,000 variables	$995	Solve global optimization problems with general convex and non-convex functions. Uses continuous branch and bound, adaptive random search, gradient-based local search.
OptQuest Solver Engine V5.0	Frontline Systems, Inc.	LP/QP/NLP/NSP-5,000 variables	$995	Uses tabu search, scatter search to solve nonsmooth global optimization problems with mixed-integer and constraint programming features.
XPRESS SOlver Engine V5.0	Frontline Systems, Inc.	LP-200,000 variables, QP-limited by spreadsheet	$6,995	Solve the largest mixed integer problems using dual Simplex, Newton Barrier, Branch and Cut algorithms.
Large-Scale SQP Solver Engine V5.0	Frontline Systems, Inc.	LP/QP/NLP-100,000 variables	4,995	Solve LP, QP, NLP problems using SQP method. Multistart feature for global optimization; handles mixed-integer, constraint programming problems.

Product Vendor	Commercial	Limitations	Price	Comments
Large-Scale GRG Solver Engine V5.0	Frontline Systems, Inc.	NLP-4,000 variables Standard, 12,000 variables Extended	$995-$2,495	Solve nonlinear optimization, mixed-integer, constraint programming problems using Generalized Reduced Gradient method.
Large-Scale LP Solver Engine V5.0	Frontline Systems, Inc.	LP-16,000 variables Standard, 64,000 variables Extended	$995-$2,495	Solve linear, mixed integer, constraint programming problems using dynamic matrix factorization and updating, multiple and partial pricing, preprocessing and probing.
Premium Solver Platform V5.0	Frontline Systems, Inc.	LP/QP-2,000 variables, NLP-500 variables, SP-500 variables	$995	Upgrade features new Interval Global Solver, hybrid Evolutionary Solver, faster GRG Nonlinear and LP/Quadratic Solvers.
Premium Solver V5.0	Frontline Systems, Inc.	LP-1,000 variables, NLP-400 variables, NSP-400 variables	$495	Compatible with Evolutionary, GRG Nonlinear, Simplex Solvers, mixed-integer Branch and Bound, linearity and infeasibility.
What'Best!	LINDO Systems, Inc.	Versions available that are limited only by memory	$195-$4,995	Its powerful enough to handle the toughest models and ideal for providing models to managers or clients.
Evolver	Palisade Corp.		$395-$995	Simplex method for LP and Simple NLP. Faster than other GA solvers to solve NLP
Solvertable	S. Christian Albright		Free	Parametric optimization tool that repeatedly calls Solver and creates table of desired cells for each optimization run.

Appendix B: Input Data for the Randomly Generated Problems

	Constraint1		Constraint2		Constraint3		Constraint4	
Test	α	RHS	β	RHS	γ	RHS	δ	RHS
R1	(8.72 ,6.89 ,9.99 ,9.97 ,9.65 ,7.68 ,7.19 ,7.99 ,8.72 ,7.13 ,9.95 ,9.39 ,8.87 ,6.48 ,9.76)	171	(4.06 ,1.52 ,3.93 ,1.56 ,1.55 ,3.4 ,1.42 ,4.73 ,1.22 ,3.37 ,2.7 ,1.21 ,3.29 ,3.72 ,2.91)	89	(13.98 ,14.76 ,19.81 ,11.84 ,16.72 ,11.32 ,14.91 ,17.57 ,17.83 ,14.16 ,18.38 ,12.51 ,11.02 ,15.1 ,18.63)	305	(21.96 ,34.6 ,26.34 ,23.13 ,23.44 ,25.05 ,31.2 ,35.19 ,31.01 ,21.42 ,27.5 ,26.57 ,37.01 ,35.76 ,34.54)	450
R2	(8.72 ,6.89 ,9.99 ,9.97 ,9.65 ,7.68 ,7.19 ,7.99 ,8.72 ,7.13 ,9.95 ,9.39 ,8.87 ,6.48 ,9.76)	192.68	(4.06 ,1.52 ,3.93 ,1.56 ,1.55 ,3.4 ,1.42 ,4.73 ,1.22 ,3.37 ,2.7 ,1.21 ,3.29 ,3.72 ,2.91)	100.55	(13.98 ,14.76 ,19.81 ,11.84 ,16.72 ,11.32 ,14.91 ,17.57 ,17.83 ,14.16 ,18.38 ,12.51 ,11.02 ,15.1 ,18.63)	342.9	(21.96 ,34.6 ,26.34 ,23.13 ,23.44 ,25.05 ,31.2 ,35.19 ,31.01 ,21.42 ,27.5 ,26.57 ,37.01 ,35.76 ,34.54)	652.18
R3	(7 ,8 ,10 ,6 ,8 ,7 ,8 ,10 ,9 ,7 ,8 ,6 ,8 ,8 ,9 ,8 ,6 ,7 ,7 ,7)	204.82	(3 ,1 ,3 ,5 ,2 ,5 ,1 ,5 ,2 ,1 ,3 ,1 ,2 ,3 ,2 ,4 ,5 ,2 ,2 ,3)	120.6	(12 ,16 ,14 ,13 ,15 ,17 ,14 ,14 ,11 ,11 ,17 ,18 ,11 ,16 ,20 ,16 ,11 ,14 ,16 ,14)	385.7	(35 ,38 ,29 ,40 ,34 ,40 ,22 ,23 ,40 ,29 ,40 ,32 ,26 ,24 ,40 ,28 ,24 ,21 ,26 ,30)	825.93
R4	(7 ,8 ,10 ,6 ,8 ,7 ,8 ,10 ,9 ,7 ,8 ,6 ,8 ,8 ,9 ,8 ,6 ,7 ,7 ,7)	616	(3 ,1 ,3 ,5 ,2 ,5 ,1 ,5 ,2 ,1 ,3 ,1 ,2 ,3 ,2 ,4 ,5 ,2 ,2 ,3)	453.39	(12 ,16 ,14 ,13 ,15 ,17 ,14 ,14 ,11 ,11 ,17 ,18 ,11 ,16 ,20 ,16 ,11 ,14 ,16 ,14)	1450	(35 ,38 ,29 ,40 ,34 ,40 ,22 ,23 ,40 ,29 ,40 ,32 ,26 ,24 ,40 ,28 ,24 ,21 ,26 ,30)	3105
R5	(9.24 ,8.04 ,8.66 ,7.29 ,8.05 ,6.31 ,9.98 ,8.37 ,8.7 ,7.18 ,9.04 ,7.27 ,9.3 ,9.65 ,7.69 ,7.31 ,7.81 ,8.86 ,7.39 ,6.91 ,9.44 ,8.97 ,6.64 ,9.1 ,7.91)	273.00	(2.8 ,2.94 ,4.68 ,3.43 ,1.96 ,1.61 ,3.62 ,4.74 ,2.76 ,2.57 ,1.42 ,1.29 ,4.12 ,2.31 ,3.41 ,3.4 ,1.85 ,1.22 ,4.47 ,1.61 ,3.7 ,3.23 ,1.65 ,1.96 ,2.75)	153.00	(12.39 ,18.39 ,19.15 ,17.57 ,16.42 ,15.18 ,19.74 ,16.77 ,19.85 ,12.09 ,14.85 ,13.57 ,18.43 ,17.71 ,15.84 ,17.69 ,18.64 ,12.81 ,11.43 ,18.96 ,12.5 ,17.95 ,17.46 ,16.58 ,12.71)	538	(34.72 ,36.89 ,36.36 ,36.72 ,33.3 ,26.4 ,26.59 ,38.47 ,27.73 ,22.22 ,37.11 ,31.73 ,30.52 ,30.44 ,21.8 ,28.82 ,32.87 ,28.81 ,34.96 ,28.24 ,38.74 ,33.87 ,29.24 ,37.32 ,32.97)	1060
R6	(6.06 ,7.3 ,9.4 ,7.44 ,8.69 ,7.86 ,8.03 ,6.68 ,7.5 ,6.21 ,8.09 ,7.18 ,7.75 ,7.05 ,8.87 ,9.87 ,7.46 ,7.31 ,7.12 ,6.67 ,7.02 ,8.94 ,6.08 ,6.61 ,9.14 ,9.6 ,8.41 ,7.06 ,9.05 ,6.59)	462.46	(3.72 ,2.08 ,2.87 ,3.81 ,3.69 ,1.48 ,4.74 ,2.53 ,3.86 ,4.77 ,3.06 ,3.13 ,3.64 ,4.21 ,3.22 ,2.22 ,4.63 ,4.68 ,1.35 ,2.08 ,4.43 ,4.21 ,2.88 ,3.86 ,3.46 ,3.81 ,2.29 ,1.37 ,2.17 ,1.02)	314.69	(11.74 ,19.15 ,18.75 ,15.32 ,11.75 ,14.4 ,11.89 ,19.17 ,18.28 ,15.46 ,18.73 ,15.78 ,18.43 ,16.45 ,16.64 ,17.34 ,10.74 ,14.6 ,13.69 ,15.07 ,16.78 ,13.12 ,13.87 ,17.61 ,14.15 ,18.62 ,14.84 ,14.6 ,19.13 ,19.41)	951.28	(23.09 ,37.14 ,22.82 ,28.21 ,30.96 ,38.12 ,38.12 ,34.32 ,37.27 ,35.4 ,24.99 ,28.12 ,27.18 ,37.83 ,26.16 ,25.22 ,29.82 ,27 ,29.59 ,22 ,32.4 ,28.83 ,23.51 ,25.2 ,35.92 ,22.17 ,37.56 ,21.21 ,30.41 ,24.01)	1769.4
R7	(6.44 ,9.61 ,6.82 ,6.3 ,7.17 ,8.76 ,8.42 ,6.71 ,6.13 ,6.07 ,7.44 ,9.47 ,8.56 ,7.78 ,9.54 ,8.09 ,9.14 ,6.53 ,6.35 ,8.84 ,7.56 ,8.13 ,8.27 ,9.21 ,8.49 ,7.1 ,9.94 ,6.69 ,9.34 ,7.48 ,9.24 ,8.71 ,9.83 ,9.24 ,8.14 ,8.31 ,9.53 ,8.9 ,8.69 ,7.76)	432.20	(1.77 ,1.11 ,4.86 ,2.35 ,2.94 ,3.51 ,4.22 ,1.93 ,2.15 ,2.38 ,3.06 ,3.87 ,3.17 ,1.24 ,4.5 ,1.03 ,1.5 ,1.41 ,3.13 ,1.92 ,1.5 ,3.65 ,1.69 ,2.59 ,3.11 ,4.55 ,2.9 ,4.76 ,3.71 ,3.76 ,1.34 ,1.18 ,4.3 ,3.49 ,3.21 ,4.92 ,2 ,1.54 ,3.06 ,4.43)	249.75	(16.73 ,18.79 ,16.73 ,13.77 ,19.62 ,19.42 ,14.43 ,12.36 ,14.86 ,16.04 ,15.59 ,14.43 ,18.39 ,17.62 ,16.89 ,12.72 ,12.49 ,17.61 ,16.76 ,14.32 ,11.05 ,16.62 ,11.34 ,13.97 ,15.48 ,11.92 ,17.95 ,15.76 ,13.98 ,17.83 ,13.8 ,12.19 ,14.92 ,14.63 ,15.77 ,19.26 ,18.12 ,13.01 ,17.56 ,12.97)	821.78	(26.3 ,23.95 ,36.3 ,34.92 ,32.02 ,24.95 ,28.79 ,34.8 ,27.08 ,27.66 ,29.39 ,38.57 ,24.67 ,28.42 ,34.78 ,34.28 ,26.33 ,36.22 ,22.61 ,35.39 ,27.31 ,36.13 ,24.1 ,23.92 ,30.56 ,30.27 ,32.7 ,22.36 ,22.04 ,24 ,37.88 ,29.68 ,25.61 ,34.84 ,23.04 ,30.84 ,21.04 ,38.76 ,32.85 ,36.88)	1400
R8	(8.83 ,8.71 ,7.12 ,8.02 ,8.12 ,7.13 ,8.02 ,7.24 ,9.38 ,8.71 ,6.62 ,9.61 ,9.97 ,6.83 ,9.02 ,9.01 ,7.66 ,6.61 ,6.36 ,6.24 ,6.68 ,7.91 ,6.45 ,8.22 ,6.67 ,8.92 ,9.86 ,7.24 ,9.68 ,6.46 ,6.96 ,8.69 ,8.12 ,6.52 ,7.44 ,8.33 ,9.83 ,9.18 ,9.5 ,7.46 ,6.73 ,8.8 ,8.12 ,6.91 ,6.9 ,7.96 ,6.19 ,7.44 ,6.3 ,8.29)	498.78	(3.82 ,3.93 ,3.77 ,2.76 ,1.44 ,4.06 ,1.67 ,4.57 ,4.34 ,2.28 ,3.6 ,1.72 ,3.26 ,2.77 ,2.12 ,2.31 ,4.58 ,4.53 ,3.14 ,1.38 ,2.2 ,2.71 ,3.25 ,2.12 ,1.14 ,4.41 ,3.14 ,1.41 ,4.68 ,2.95 ,1.57 ,1.69 ,2.32 ,4.28 ,1.11 ,2.42 ,2.42 ,2.14 ,3.98 ,4.15 ,4.9 ,1.59 ,2.2 ,3.67 ,1.32 ,4.54 ,1.62 ,2.55 ,3.74 ,3.84)	317.07	(11.43 ,13.56 ,16.94 ,13.6 ,15.61 ,14.79 ,19.75 ,11.86 ,13.9 ,19.16 ,18.06 ,15.61 ,13.53 ,15.35 ,17.16 ,17.14 ,15.81 ,15.21 ,12.49 ,11.12 ,13.65 ,17.52 ,14.03 ,19.61 ,19.51 ,19.8 ,19.48 ,16.03 ,13.09 ,11.61 ,11.94 ,18.85 ,14.24 ,13.49 ,14.03 ,11.19 ,11.08 ,14 ,12.2 ,12.27 ,17.04 ,17.79 ,19.41 ,11.36 ,19.79 ,13.23 ,13.12 ,16.48 ,12.25 ,18.59)	985.47	(26.69 ,28.68 ,38.12 ,37.44 ,27.82 ,30.48 ,28.86 ,39.41 ,36.2 ,38.83 ,28.94 ,32.42 ,39.17 ,29.89 ,32.13 ,28.52 ,23.66 ,22.29 ,23.79 ,25.58 ,28.99 ,30.11 ,22.16 ,37.43 ,26.07 ,23.19 ,22.82 ,22.62 ,25.38 ,34.62 ,22.44 ,30.22 ,22.51 ,22.96 ,26.62 ,28.97 ,26.75 ,33.91 ,29.5 ,27.98 ,25.88 ,26.41 ,25.57 ,38.93 ,39.1 ,38.21 ,26 ,22.13 ,39.56 ,28.71)	1937.8
R9	(7.21 ,8.48 ,7.41 ,8.26 ,8.96 ,7.6 ,6.98 ,9.61 ,7.23 ,7.47 ,7.33 ,8.64 ,7.17 ,6.48 ,6.06 ,7.9 ,9.71 ,9.27 ,8.78 ,9.19 ,7.63 ,6.44 ,7.45 ,9.26 ,6.81 ,8.34 ,7.83 ,7.07 ,9.8 ,7.48 ,8.74 ,7.93 ,8.91 ,9.02 ,9.02 ,9.36 ,8.61 ,8.4 ,8.44 ,6.21 ,6.79 ,9.34 ,7.67 ,6.2 ,7.04 ,6.87 ,8.75 ,6.65 ,7.85 ,6.14 ,7.3 ,9.21 ,8.18 ,9.08 ,9.83 ,7.56 ,7.79 ,8.81 ,6.5 ,6.29)	571.97	(1.68 ,4.87 ,2.14 ,4.64 ,3.33 ,4.9 ,2.77 ,1.29 ,4.39 ,2.56 ,1.33 ,3.65 ,3.15 ,4.43 ,3.07 ,4.01 ,4.43 ,3.88 ,1.82 ,3.88 ,3.14 ,3.17 ,1.68 ,1.9 ,1.02 ,2.76 ,3.94 ,3.95 ,3.58 ,3.46 ,1.74 ,1.87 ,2.32 ,4.38 ,2.49 ,4.44 ,2.84 ,4.2 ,3.24 ,3.53 ,2.07 ,2.29 ,3.68 ,4.26 ,3.27 ,3.76 ,2.2 ,2.69 ,4.63 ,1.65 ,1.62 ,2.18 ,2.78 ,2.17 ,4.07 ,2 ,1.72 ,4.6 ,3.61 ,3.19)	365.27	(11.26 ,14.47 ,14.92 ,19.15 ,18.71 ,14.06 ,17.49 ,14.42 ,17.14 ,19.31 ,18.74 ,19.69 ,11.91 ,15.74 ,14.06 ,14 ,11.89 ,16.16 ,17.5 ,14.11 ,11.81 ,17 ,13.38 ,15.24 ,17.84 ,12.34 ,15.11 ,12.64 ,16.61 ,12.68 ,18.35 ,14.58 ,12.68 ,15.05 ,13.93 ,12.39 ,17.6 ,17.09 ,17.52 ,19.5 ,16 ,19.12 ,19.59 ,19.33 ,16.83 ,16.08 ,11.85 ,16.89 ,13.41 ,15.04 ,18.43 ,19.74 ,13.17 ,18.37 ,17.4 ,13.85 ,13.3 ,16.73 ,12.93 ,13.82)	1128.3	(29.59 ,29.79 ,27.82 ,29.04 ,26.5 ,29.7 ,21.97 ,30.39 ,36.35 ,24.4 ,35.21 ,33.24 ,28.46 ,32.3 ,21.05 ,27.08 ,36.18 ,32.68 ,27.34 ,32.98 ,34.82 ,33.72 ,26.44 ,23.75 ,30.79 ,26.24 ,29.12 ,24.63 ,23.36 ,34.41 ,34.12 ,36.31 ,36.19 ,38.21 ,31.95 ,32.15 ,32.23 ,39.24 ,24.06 ,36.57 ,36.71 ,28.58 ,25.05 ,36.04 ,29.84 ,24.88 ,30.17 ,28.4 ,27.34 ,29.74 ,28.12 ,24.02 ,25.2 ,28.82 ,21.74 ,21.16 ,39.58 ,39.22 ,25.21 ,32.89)	2164