A LARGE SCALE INTEGER AND COMBINATORIAL OPTIMIZER

By

Qun Chen

A dissertation submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

(Industrial Engineering)

at the

UNIVERSITY OF WISCONSIN – MADISON

2000

Abstract

The topic of this thesis, integer and combinatorial optimization, involves minimizing (or maximizing) a function of many variables, some of which belong to a discrete set, subject to constraints. This area has abundant applications in industry. Integer and combinatorial optimization problems are often difficult to solve due to the large and complex set of alternatives.

The objective of this thesis is to present an effective solution to integer and combinatorial problems by using a traditional reliable branch-and-bound approach as well as a newly developed fast adaptive random search method, namely Nested Partitions. The proposed integer and combinatorial optimizer has two closely related components: FATCOP and NP/GA. FATCOP is a distributed mixed integer program solver written in PVM for Condor's opportunistic environment. It is different from prior parallel branch-and-bound work by implementing a general purpose parallel mixed integer programming algorithm in an opportunistic multiple processor environment, as opposed to a conventional dedicated environment. We show how to make effective use of opportunistic resources while ensuring the program works correctly. The solver also provides performance-enhancing features such as preprocessing, pseudocost branching, cutting plane generation, locking of variables and general purpose primal heuristics. FATCOP performs very well on test problems arising from real applications, and is particularly useful to solve long-running hard mixed integer programming problems.

For many integer and combinatorial optimization problems, application-specific tuning may be required. Users can supply combinatorial approximation algorithms which exploit structure unique to the problem class. These can be run in conjunction with the

system defaults or in place of them. In this thesis, we developed a new hybrid algorithm NP/GA based on Nested Partitions and a Genetic Algorithms. This algorithm retains the global perspective of the Nested partitions method and the local search capacities of the Genetic Algorithm, and can be applied to any integer and combinatorial optimization problems.

We applied our optimizer to product design problems from the marketing area. We started with building NP/GA heuristic algorithms and showed that our algorithms outperform all previous approaches. Following that, we incorporated the heuristics to FAT-COP and solved some reasonable size instances to optimality.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Integer and combinatorial optimization models, also known as discrete optimization models, represent critical decision problems in industry. These applications include operational problems such as production scheduling and machine sequencing, planning problems such as facility location, and design problems such as transportation network design and product design.

Integer and combinatorial optimization models consist of maximizing or minimizing a function of many variables subject to inequality or equality constraints, while some of the variables are restricted to take on integer values. Some models have a linear objective function with linear constraints. Others have either nonlinear objective functions or nonlinear constraints or both. Within mathematical programming, the first class of problems are referred to as linear integer problems (IP or ILP) that require all variables to be integer, or mixed integer problems (MIP or MILP) that require only some of the variables to take integer values. For the sake of generality, we view integer problems as special cases of mixed integer problems. The second class of problems are referred to as mixed integer nonlinear problems (MINLP). We will focus our attention on mixed integer linear problems in this thesis although the proposed optimizer is not restricted to linear problems.

Integer and combinatorial optimization algorithms span a range as wide as the models they address. Among the best known algorithms, we cite branch-and-bound, branch-and-cut, branch-and-price, and heuristic algorithms such as tabu search (TS), simulated annealing (SA), genetic algorithms (GA), and nested partitions (NP). Of these methods, the most popular general purpose method is clearly branch-and-bound. Many commercial available codes implement branch-and-bound algorithms. Theoretically, the branch-and-bound method can always find a provably optimal solution. However in practice many difficult integer and combinatorial problems require lengthy branching and bounding processes even to find *good* solutions. As a result, heuristic algorithms are often used to find *good* solutions without any guarantee that the solutions are optimal. The NP method has been developed recently for global optimization in the context of both deterministic and stochastic problems. The method takes a global perspective and offers a framework to incorporate various kinds of local search techniques. The NP method has been found to be efficient for a variety of problems, such as the traveling salesman problem [61], production scheduling problems [54] and a product design problem [62]. Furthermore, the NP method does not have any requirement on the problem's objective function, hence can be used to solve nonlinear problems.

In this thesis, we develop an integer and combinatorial optimizer that has two related components: FATCOP, and NP/GA. FATCOP is a distributed branch-and-bound based MIP solver developed in an opportunistic multiple processor environment. MIP has been a widely suggested application of parallel computing. Our solver is different from prior parallel branch-and-bound work by implementing a general purpose parallel MIP algorithm in an opportunistic environment, as opposed to the conventional dedicated environment. An example of an opportunistic environment is a large heterogeneous clusters of workstations connected through a fast local network. With the ever decreasing cost of lower-end workstations, such an environments become very common today. In this thesis we show how to develop an efficient and robust MIP solver in an opportunistic environment with the help of Condor, a distributed resource management system.

NP/GA is a hybrid method that combines two heuristic algorithms: NP and GA. The NP method is a randomized optimization method for global optimization. It can be described as an adaptive sampling method that uses partitioning to concentrate the sampling effort in those subsets of a feasible region that are considered the most promising. The GA is also a random search method, which is based on the concept of natural selection. It starts from an initial population and then uses a mixture of reproduction, crossover, and mutation to create new, and hopefully better, populations. Although the GA works with a population of strings rather than a single string, the GA is essentially a sequential heuristic and is not guaranteed to converge to a global optimum. The NP method, on the other hand, maintains a global perspective in every iteration and converges in finite time to a global optimum [64]. The NP/GA algorithm retains the global perspective of the NP and the local search capabilities of the genetic algorithm.

We proposed the following procedure to use the new optimizer for any applicable industrial application. A user starts with building a MIP model for one application and solves the model using FATCOP. If FATCOP can not find a satisfactory solution, the user may develop an NP/GA procedure for this application. The implementation of the NP/GA procedure is problem dependent. We tried to facilitate use by providing a NP/GA library that includes a set of C++ Nested Partitions objects and examples. Users can implement NP/GA procedures on top of the NP/GA library by calling its built-in routines. If some *good* solutions are found, the user can deliver the best solution value to FATCOP and run the MIP solver again. The new solve can be used to measure the quality of the solutions found by the NP/GA procedure and may possibly find better solutions or even prove optimality. The above procedure is illustrated in Figure 1.

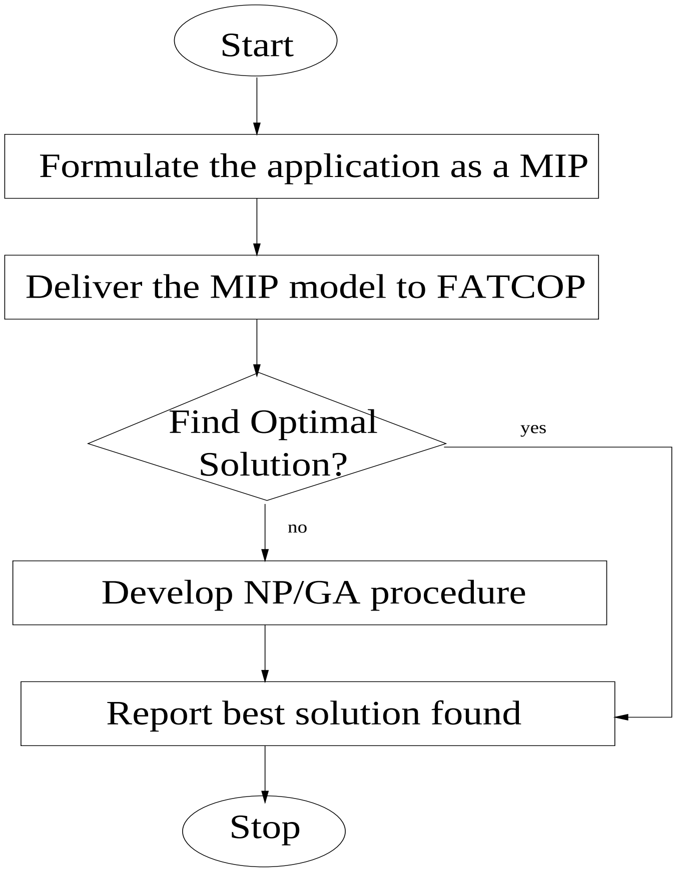

Figure 1: The procedure to apply the proposed optimizer

1.1 The Branch-and-Bound Method

The branch-and-bound method was first described by Land and Doig [46], and further developed by many researchers. We will describe the branch-and-bound algorithms for MIP in Chapter 2. Here we give a brief introduction to the general branch-and-bound method.

The branch-and-bound method can be viewed as an implicit enumeration method to solve optimization problems by enumerating a finite number of solutions in their feasible region. It consists of decomposing the feasible region into a number of subregions. The decomposition often is called *branching operation*. The optimization problem in a subregion is called a subproblem. Each subproblem is associated with a lower bound or/and a upper bound by a *bounding operation*. If a subproblem is not solved directly, a further *branching operation* will be applied to it to get more subproblems. However, a subproblem need not be decomposed if one of the following statements is true.

1. The problem is solved, that is, it is either infeasible or an optimal solution is found.

2. Another subproblem has an optimal value that is not worse than its optimal value.

The branch-and-bound method thus consists of performing branching and bounding operation as well as the elimination tests given above. The process of the method actually builds a tree. The root of the tree is the original problem. The leaves of the tree are the subproblems that have not been decomposed or will be eliminated. When all generated subproblems have been examined, the algorithm stops and the best bound of all evaluated subproblems is the optimal value of the problem.

Branch-and-bound method is often effective in practice, but in the worst case an explicit enumeration may be needed. A further difficulty is that unless there is some special structure, such as all linear constraints, obtaining bounds may be as difficult as solving the original problem.

1.2 The Nested Partitions Method

The Nested partitions method is an adaptive random search method recently proposed by Shi and Olafsson [64]. The method originally was developed for discrete stochastic optimization using simulation, but found to be also efficient for deterministic problems, such as the traveling salesman problem [61], the production scheduling problem [54] and a product design problem [62].

We describe the NP method for combinatorial problems. Note that the method can also be extended to problems with infinite countable or bounded continuous feasible regions. Like the branch-and-bound method, it *partitions* the feasible region into a number of subregions. Instead of evaluating bounds as the branch-and-bound method does, the NP method randomly selects *samples* to estimate the *promising index* for each region. The region with the best promising index is further partitioned while the other regions are aggregated into one region, called the surrounding region. The method then samples in each of the newly partitioned regions as well as the surrounding region to decide which region is best in this iteration. If one of the subregions is found best, this region becomes the most promising region and will be partitioned and sampled in a similar fashion. If the surrounding region has the best promising index, the method *backtracks* to a larger region by some fixed rule. By nested partitioning the most promising region or backtracking to a larger region, the NP method is able to concentrate computational effort in regions that are promising while still taking all feasible solutions into consideration. To estimate the promising index, the method can incorporate local search techniques. Because of this attractive feature, the NP method can be used as an optimization framework to incorporate any existing effective local search methods for a problem [62].

The traversal of promising regions in the NP method is shown to generate a Markov chain. In a deterministic context the global optima are the only absorbing states. As the Markov chain will eventually be absorbed, the method converges to a global optimum. In this thesis, we exploit the Markov structure further to calculate expected time until a global optimum is found under some assumptions.

1.3 Scope of the Thesis

The objective of this thesis is to develop a general and efficient integer and combinatorial optimization tool by using new methodology and computational environments. It is expected that the new optimizer can be an important addition to the existing optimization tools for integer and combinatorial optimization. The major contributions of this thesis are:

1. development of a distributed branch-and-bound algorithm in an opportunistic environments and an MIP solver as its product;
2. development of a hybrid global optimization algorithm NP/GA;
3. refinement of the results to estimate the expected number of iterations until the NP algorithm is absorbed in one global optimum and developed a stopping criteria for the NP algorithm using the new results;
4. investigation of NP/GA and other hybrid algorithms of NP on the product design problems that generate better solutions than previous methods;
5. development of a MIP solution to the product design problems based on FATCOP.

The remainder of the thesis is organized as the follows. In Chapter 2 we describe the design, extension and performance of FATCOP. In Chapter 3 we first review the NP algorithm through examples and investigate the finite time behavior of the NP Markov chain, then present the new hybrid algorithm NP/GA. Chapter 4 discusses a detailed application of FATCOP and NP/GA to the product design problems. This application illustrates the general procedure to apply the optimizer proposed in this thesis. Finally, Chapter 5 contains outlines of our future research.

Chapter 2

FATCOP: Fault Tolerant Condor-PVM Mixed Integer Program Solver

Many industrial engineering applications involve combinatorial optimization, that is, obtaining the optimal solution among a finite set of alternatives. Examples of such problems abound in applications; for example scheduling and location problems, covering and partitioning problems and allocation models. While many of these problems also contain nonlinear relationships amongst the variables, a large number of interesting examples can be effectively modeled using linear relationships along with integer variables. Such problems are typically called mixed integer programs (MIP) [53].

Mixed integer programming (MIP) problems are difficult and commonplace. For many of these hard problems, only small instances can be solved in a reasonable amount of time on sequential computers, resulting in mixed integer programming being a frequently cited application of parallel computing. Most available general-purpose large-scale MIP codes use branch-and-bound to search for an optimal integer solution by solving a sequence of related linear programming (LP) relaxations that allow possible fractional values. In this thesis we discuss a new parallel mixed integer program solver, written in PVM, that runs in the opportunistic computing environment provided by the Condor resource management system.

Parallel branch-and-bound algorithms for MIP have attracted many researchers (see [18, 25, 55] and references therein). Most parallel branch-and-bound programs were developed for large centralized mainframes or supercomputers that are typically very expensive. Users of these facilities usually only have a certain amount of time allotted to them and have to wait their turn to run their jobs. Due to the decreasing cost of lower-end workstations, large heterogeneous clusters of workstations connected through fast local networks are becoming common in work places such as universities and research institutions. In this thesis we shall refer to the former resources as dedicated resources and the later as distributed ownership resources. The principal goal of the research outlined in this thesis is to exploit distributed ownership resources to solve large mixed integer programs. We believe that a distributed branch-and-bound program developed to use these types of resources will become highly applicable in the future.

A parallel virtual machine (PVM) is a programming environment that allows a heterogeneous network of computers to appear as a single concurrent computational resource [23]. It provides a unified framework within which parallel programs for a heterogeneous collection of machines can be developed in an efficient manner. However PVM is not sufficient to develop an efficient parallel branch-and-bound program in a distributed ownership environment. The machines in such an environment are usually dedicated to the exclusive use of individuals. The application programming interface defined by PVM requires that users explicitly select machines on which to run their programs. Therefore, they must have permission to access the selected machines and cannot be expected to know the load on the machines in advance. Furthermore, when a machine is claimed by a PVM program, the required resources in the machine will be "occupied" during the life cycle of the program. This is not a desirable situation when the machine is owned by a person different from the user of the MIP solver.

Condor [20, 48] is a distributed resource management system that can help to overcome these problems. Condor manages large heterogeneous clusters of machines in an attempt to use the idle cycles of some users' machines to satisfy the needs of others who have computing intensive jobs. It was first developed for long running sequential batch jobs. The current version of Condor provides a framework (Condor-PVM) to run parallel programs written in PVM in a distributed ownership environment. In such programs, Condor is used to dynamically construct a PVM out of non-dedicated desktop machines on the network. Condor allows users' programs to run on any machine in the pool of machines managed by Condor, regardless of whether the user submitting the job has an account there or not, and guarantees that heavily loaded machines will not be selected for an application. To protect ownership rights, whenever a machine's owner returns, Condor immediately interrupts any job that is running on that machine, migrating the job to another idle machine. Since resources managed by Condor are competed for by owners and many other Condor users, we refer to such resources as Condor's *opportunistic resources* and the Condor-PVM parallel programming environment as the Condor-PVM *opportunistic environment*.

FATCOP represents a first attempt to develop a general purpose distributed solver for mixed integer programs in Condor's opportunistic environment. It is written in the C++ programming language with calls to PVM library. It is designed to make best use of participating resources managed by Condor while handling resource retreat carefully in order to ensure the eventual and correct completion of a FATCOP job. Key features of FATCOP include:

- parallel implementation under Condor-PVM framework;

- greedy utilization of Condor's opportunistic resources;
- the use of heterogeneous resources;
- the use of multiple LP solvers (CPLEX, OSL and SOPLEX);
- advanced MIP techniques including pseudocost estimation searching, preprocessing, and cutting plane generation;
- the ability to process MPS [51], AMPL [22] and GAMS [7] models;
- object oriented design

We begin in Section 2.1 with a review of the standard MIP algorithm components of FATCOP that are implemented to ensure the branch-and-bound algorithm generates reasonable search trees. Section 2.2 introduces the Condor-PVM parallel programming framework. In section 2.3 we outline the design of the parallel program. We present the parallel implementation of FATCOP in section 2.4 and discuss features and extensions in section 2.5. In section 2.6, a variety of tests are carried out on a representative set of MIP test problems and some real applications.

2.1 Components of Sequential Program

For expositional purposes, we review the basic sequential implementation of a BB MIP solver. A MIP can be stated mathematically as follows:

$$\begin{array}{ll} \min & c^T x \\ \text{s.t.} & Ax \leq b \\ & l \leq x \leq u \\ & x_j \in Z \qquad \forall j \in I \end{array}$$

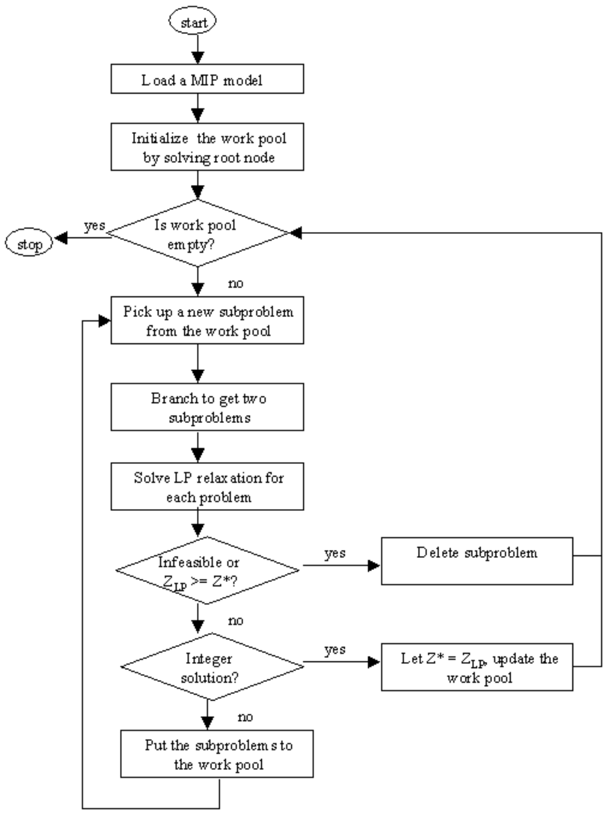

Figure 2: Branch-and-bound algorithm

where Z denotes the integers, A is an $m \times n$ matrix, and I is a set of distinguished indices identifying the integer variables.

Most integer programming textbooks [1, 52] describe the fundamental branch-and-bound algorithm for the above MIP problem. Basically, the method explores a binary tree of subproblems. The key steps of BB are summarized below and the algorithm flowchart can be found as Figure 2.

Algorithm *Branch-and-Bound:*

Step 0. *Initialization:*

Set iteration count $k = 0$ and incumbent (or upper bound of the MIP) $Z^* = \infty$.

Put the initial problem in the *work pool* containing subproblems that are not fathomed yet.

Step 1. *Branching:*

Among the remaining subproblems in the work pool, select one according to some search order. Among the integer-restricted variables that have a noninteger value in the optimal solution for the LP relaxation of this subproblem, choose one according to some branching rule to be the branching variable. Let x_j be this variable and denote by x_j^* its value in the aforementioned solution. Branch from the node for the subproblem to create two new subproblems by adding the respective constraints $x_j \leq \lfloor x_j^* \rfloor$ and $x_j \geq \lfloor x_j^* \rfloor + 1$.

Step 2. *Bounding:*

For each new subproblem, obtain its bound by applying the simplex method (or the dual simplex method when reoptimizing) to its LP relaxation and use the value of Z_{LP} for the resulting optimal solution. The minimum value of such bounds associated with subproblems in the work pool is referred to as lower bound of the MIP.

Step 3. *Fathoming:*

For each new subproblem, apply the three fathoming tests given below, and discard those subproblems that are fathomed by any of the tests.

Test 1: Its bound $Z_{LP} \geq Z^*$.

Test 2: Its LP relaxation has no feasible solutions.

Test 3: The optimal solution for its LP relaxation has integer values for the integer-restricted variables. If this solution is better than the incumbent, it becomes the new incumbent and Test 1 is reapplied to all unfathomed subproblems with the new Z^*.

Step 4. *Optimality test:*

Stop when there are no remaining subproblems. The current incumbent (if any) is optimal. Otherwise, go to step 1.

The above description makes it clear that there are various choices to be made during the course of the algorithm. We hope to find robust strategies that work well on a wide variety of problems. The reminder of this section describes the algorithm components that refines this basic framework and FATCOP's interfaces.

2.1.1 Branching rules

If there are many fractional variables in the solution to the LP relaxation, we must choose one variable to be the branching variable. Because the effectiveness of the branch and bound method strongly depends on how quickly the upper and lower bounds converge, we would like to branch on a variable that will improve these bounds.

Several reasonable criteria exist for selecting branching variables. FATCOP currently provides three variable selection options: pseudocost, maximum integer infeasibility and minimum integer infeasibility. Maximum integer infeasibility will generate two branches that are more likely to differ from the current solution than other branch alternatives. The underlying assumption is that the objective function values are more likely to degrade on both branches. Hence such a branch would represent an influential variable. Minimum integer infeasibility will generate one branch very similar to the current solution and one branch very different from the current solution. The underlying assumption is that the similar branch is where a solution lies and that the different branch will prove uninteresting [1]. Pseudocost attempts to estimate the change rate in objective function value associated with a particular branch. Since the pseudocost method is widely used and known to be efficient [47], we set it as the default branching strategy and briefly describe the method here.

We associate two quantities ϕ_j^- and ϕ_j^+ with each integer variable x_j that attempts to measure the per unit decrease in objective function value if we fix x_j to its rounded down value and rounded up value, respectively. Suppose that $x_j = \lfloor x_j \rfloor + f_j$, with $0 < f_j < 1$. Then by branching on x_j, we will estimate a decrease of $D_j^- = \phi_j^- f_j$ on the down branch and a decrease of $D_j^+ = \phi_j^+(1 - f_j)$ on the up branch . The way to obtain the objective change rate ϕ_j^- and ϕ_j^+ is to use the observed change in objective function value:

$$\phi_j^- = \frac{Z_{LP}^- - Z_{LP}}{f_j} \quad \text{and} \quad \phi_j^+ = \frac{Z_{LP}^+ - Z_{LP}}{1 - f_j}$$

where Z_{LP} is the LP relaxation solution value, and Z_{LP}^+ and Z_{LP}^- are the LP relaxation solution values at up and down branches respectively. Note that the above discussion is for a particular node.

In the course of the solution process, the variable x_j may be branched on many times. The pseudocosts are updated by averaging the values from all x_j branches. The remaining issue is to determine to what values should the pseudocost be initialized. The question of initialization is important, since the branching decisions made at the top of the tree are the most crucial. If similar MIP problems have been solved before, we can use the previous results to initialize pseudocosts for integer variables. If previous results are not available, we can explicitly solve LPs to compute pseudocosts for candidate branching variables which have not been branched yet. Another simple way is to initialize the pseudocost x_j with its corresponding coefficient in the objective function c_i. When taking a branching decision for subproblem P, the algorithm calculates a score s_j for each x_j as suggested in [18]:

$$s_j = \alpha_0 p_j + \alpha_1 D_j^+ + \alpha_2 D_j^-;$$

where p_j is the user specified priority for x_j, $p_j = 0$ if the priority is not specified. α_0, α_1,α_2 are some constants. α_0 usually is very big so that user specified priorities are predominant. Normally the branching candidate variable with highest priority will be selected.

2.1.2 Searching rule

FATCOP provide four options for selecting a node from the remaining nodes:

1. Depth-first: The depth-first rule selects the node that was created most recently. The depth-first rule has the advantage that it needs very small memory and usually finds a feasible solution quickly.

2. Best-first: The best-first rule, as its name suggests, always selects the current node to be the one whose LP solution is the smallest. Since the best-first rule always selects the most attractive subproblem, it will improve the algorithm's convergence speed.

3. Pseudocost-estimation: An estimation for the optimal value of an integer solution for a node can be obtained from its LP solution and its branching candidates' pseudocost:

$$E = Z_{LP} + \sum_{j \in \{\text{all branching candidates}\}} \min\{D_j^+, D_j^-\}$$

The pseudocost-estimation strategy selects the node with the best estimation from the remaining nodes. Like depth-first search, it requires small memory and usually can find good feasible solutions quickly. Therefore, pseudocost-estimation is often used for complex MIP problems when the provable optimal solution is hard to obtain.

4. Deepest-first: This rule selects the node in the bottom of the tree with the expectation that the subtree rooted at this node is likely be fathomed, thus leading to a decrease in the size of the work pool.

We shall discuss FATCOP's default searching rules later in this Chapter.

2.1.3 Preprocessing

Preprocessing refers to a set of simple reformulations performed on a problem instance to enhance the solution process. In linear programming this typically leads to problem size reductions. FATCOP identifies infeasibilities and redundancies, tighten bounds on variables, and improves the coefficients of constraints [60]. At the root node, FATCOP analyzes every row of the constraint matrix. If, after processing, some variables are fixed or some bounds are improved, the process is repeated until no further model reduction occurs. We briefly discuss the techniques here.

Without loss of generality we assume that the inequality currently under consideration is of the form:

$$\sum_j a_j x_j \leq b$$

Define:

l_j	The lower bound of variable x_j;
u_j	The upper bound of variable x_j;
L_{min}	The minimum possible value for left hand side of the constraint ;
L_{max}	The maximum possible value for left hand side of the constraint ;
L^k_{min}	The minimum possible value for left hand side of the constraint without the term $a_k x_k$;

L^k_{max} The maximum possible value for left hand side of the constraint without the term $a_k x_k$;

The following techniques may allow problem reduction:

1. Simple presolving methods:

   - remove empty row or column
   - check infeasible or fixed variable: $l_j > u_j$ or $l_j = u_j$
   - remove singleton row and modify the corresponding bounds.

2. Identification of infeasibility:

$$L_{min} > b$$

3. Identification of redundancy:

$$L_{max} \leq b$$

4. Improvement of bounds: For each variable x_k in the constraint, we have:

$$L^k_{min} + a_k x_k \leq \sum_j a_j x_j \leq b$$

   Then:

$$a_k x_k \leq b - L^k_{min}$$

   If $a_k > 0$,

$$x_k \leq \min\{(b - L^k_{min})/a_k, u_k\}$$

   If $a_k < 0$,

$$x_k \geq \max\{(b - L^k_{min})/a_k, l_k\}$$

   If x_k is an integer constrained variable, the new upper and lower bounds should be rounded down and up.

5. Improvement of coefficients: Define $\delta = b - L^k_{max}$. If x_k is a binary variable and $\delta > 0$, both a_k and b can be reduced by δ. One can easily check the validity by setting x_k to 0 and 1 respectively.

In contrast to LP, preprocessing may reduce the *integrality gap*, i.e., the difference between the optimal solution value and its LP relaxation as well as the size of a MIP problem. For example, for the model *p0548* from MIPLIB [6], an electronically available library of both pure and mixed integer programs arising from real applications, the FATCOP preprocessor can only remove 12 rows, 16 columns, and modify 176 coefficients from the original model that has 176 rows, 548 columns and 1711 non zero coefficients, but pushes the optimal value of the initial LP relaxation from 315.29 up to 3125.92.

2.1.4 Cutting Planes

If the solution to the linear programming relaxation does not satisfy the integrality requirements, instead of generating new subproblems (branching), one may attempt to find an inequality that "cuts off" the relaxed solution. That is, an inequality that is not valid for the relaxed solution, but is valid for all integer solutions. Such an inequality is called a *cutting plane*. Adding cutting planes to the relaxation can result in an improved lower bound for the relaxation, which in turn may mean that the linear subproblem can be fathomed without having to resort to branching.

There are many different classes of cutting planes. FATCOP includes two classes – *knapsack cover inequalities* [14] and *flow cover inequalities* [57]. Knapsack covers and flow covers inequalities are derived from structures that are present in many, but not all, MIP instances. This implies that for some instances, FATCOP will be able to generate useful cutting planes, and for other instances it will not. Both knapsack and flow cover cuts are global valid.

The problem of finding a valid inequality of a particular class that cuts off the relaxed solution is known as the *separation problem.* For both classes of inequalities used in FATCOP, the separation problem is NP-Complete, so a heuristic procedure is used for finding violated inequalities.

There are about 10 models out of 57 in MIPLIB for which knapsack cuts are useful. We again take *p0548* as an example; the FATCOP code can solve the model in 350 nodes with knapsack cuts applied at each node. However, it is not able to solve the problem to optimality in 100,000 nodes without knapsack cuts.

2.1.5 Primal Heuristics and Variable Locking

A heuristic is a procedure that attempts to generate a feasible integral solution. Feasible solutions are important not only for their own sake, but also as they provide an upper bound on the optimal solution of the problem. With this upper bound, subproblems may be fathomed, and techniques such as reduced cost fixing can be performed.

There are very few general purposes heuristics for mixed integer programs. One simple, yet effective heuristic is known as the *diving* heuristic. In the diving heuristic, some integer variables are fixed and the linear program resolved. The fixing and resolving is iterated until either an integral solution is found or the linear program becomes infeasible.

We have included a diving heuristic in FATCOP. The diving heuristic can be quite time consuming – too time consuming to be performed at every node of the branch and bound tree. We shall discuss the strategy to perform diving heuristics later in this chapter.

FATCOP also incorporates a standard reduced cost fixing procedure [18] that fixes integer variables to their upper or lower bounds by comparing their reduced costs to the gap between a linear programming solution value and the current problem best upper bound.

2.2 Condor-PVM Parallel Programming Environment

Heterogeneous clusters of workstations are becoming an important source of computing resources. Two approaches have been proposed to make effective use of such resources. One approach provides efficient resource management by allowing users to run their jobs on idle machines that belong to somebody else. Condor, developed at University of Wisconsin-Madison, is one such system. It monitors the activity on all participating machines, placing idle machines in the Condor pool. Machines are then allocated from the pool when users send job requests to Condor. Machines enter the pool when they become idle, and leave when they get busy, e.g. the machine owner returns. When an executing machine becomes busy, the job running on this machine is initially suspended in case the executing machine becomes idle again within a short timeout period. If the executing machine remains busy then the job is migrated to another idle workstation in the pool or returned to the job queue. For a job to be restarted after migration to another machine a checkpoint file is generated that allows the exact state of the process to be re-created. This design feature ensures the eventual completion of a job. There are various priority orderings used by Condor for determining which jobs and machines are matched at any given instance. Based on these orderings, running jobs may sometimes be preempted to allow higher priority jobs to run instead. Condor is freely available and has been used in a wide range of production environments for more than ten years.

Another approach to exploit the power of a workstation cluster is from the perspective of parallel programming. Research in this area has developed message passing environments allowing people to solve a single problem in parallel using multiple resources. One of the most widely used message passing environments is PVM that was developed at the Oak Ridge National Laboratory. PVM's design centers around the idea of a *virtual machine*, a very general notion that can encompass a nearly arbitrary collection of computing resources, from desktop workstations to multiprocessors to massively parallel homogeneous supercomputers. The goal of PVM is to make programming for a heterogeneous collection of machines straightforward. PVM provides process control and resource management functions that allow spawning and termination of arbitrary processes and the addition and deletion of hosts at runtime. The PVM system is composed of two parts. The first part is a daemon that resides on all the computers comprising the virtual machine. The second part of the system is the PVM library. It contains user-callable routines for message passing, process spawning, virtual machine modification and task coordination. PVM transparently handles all message routing, data conversion and task scheduling across a network of incompatible computer architectures. A similar message passing environment is MPI [36]. Both systems center around a message-passing model, providing point-to-point as well as collective communication between distributed processes.

The development of resource management systems and message passing environments have been independent of each other for many years. Researchers at the University of Wisconsin have recently developed a parallel programming framework that interfaces Condor and PVM [56]. The reason to select PVM instead of MPI is that the implementation of MPI has no concept of process control, hence cannot handle resource addition and retreat in a opportunistic environment. Figure 3 shows the architecture of Condor-PVM. There are three processes on each machine running a Condor-PVM application:

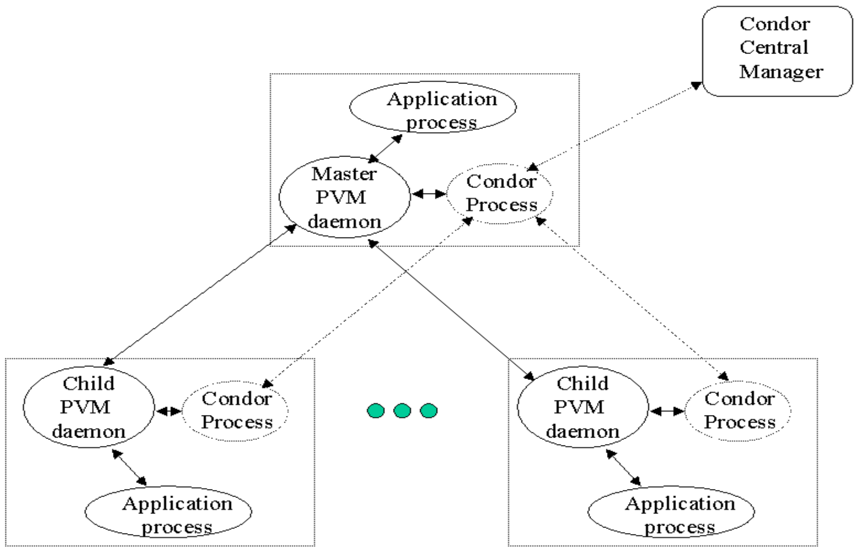

Figure 3: Architecture of Condor-PVM

the PVM daemon, the Condor process and the user application process. The Condor-PVM framework still relies on the PVM primitives for application communication, but provides resource management in the opportunistic environment through Condor. Each PVM daemon has a Condor process associated with it, acting as the *resource manager*. The Condor process interacts with PVM daemon to start tasks, send signals to suspend, resume and kill tasks, and receive process completion information. The Condor process running on the master machine is special. It communicates with Condor processes running on the other machines, keeps information about the status of the machines and forwards resource requests to the Condor central manager. This Condor process is called the *global resource manager*. When a Condor-PVM application asks for a host (we will use host and machine interchangeably in the sequel), the global resource manager communicates with Condor central manager to schedule a new machine. After Condor grants a machine to the application, it starts a Condor process (resource manager) and a PVM daemon on the new machine. If a machine needs to leave the pool, the resource manager will send signals to the PVM daemon to suspend tasks. The master user application is notified of that via normal PVM notification mechanisms.

Compared with a conventional dedicated environment, the Condor-PVM opportunistic environment has the following characteristics:

1. There usually are a large amount of heterogeneous resources available for an application, but in each time instance, the amount of available resources is random, dependent on the status of machines managed by Condor. The resources are competed for by owners and other Condor users.

2. Resources used by an application may disappear during the life cycle of the application.

3. The execution order of components in an application is highly non-deterministic, leading to different solution and execution times.

Therefore a good Condor-PVM application should be tolerant to loss of resources (host suspension and deletion) and dynamically adaptive to the current status of Condor pool in order to make effective use of opportunistic resources.

PVM and Condor-PVM are binary compatible with each other. However there exist some run time differences between PVM and Condor-PVM. The most important difference is the concept of machine class. In a regular PVM application, the configuration of hosts that PVM combines into a virtual machine usually is defined in a file, in which host names have to be explicitly given. Under the Condor-PVM framework, Condor selects the machines on which a job will run, so the dependency on host names must be removed from an application. Instead the applications must use class names. Machines of different architecture attributes belong to different machine classes. Machine classes are numbered 0, 1, etc. and hosts are specified through machine classes. A machine class is specified in the submit-description file submitted to Condor, that specifies the program name, input file name, requirement on machines' architecture, operating system and memory etc.

Another difference is that Condor-PVM has "host suspend" and "host resume" notifications in addition to "host add", "host deletion" and "task exit" notifications that PVM has. When Condor detects activity of a workstation owner, it suspends all Condor processes running there rather than killing them immediately. If the owner remains for less than a pre-specified cut-off time, the suspended processes will resume. To help an application to deal with this situation, Condor-PVM makes some extensions to PVM's notification mechanism.

The last difference is that adding a host is non-blocking in Condor-PVM. When a Condor-PVM application requests a new host be added to the virtual machine, the request is sent to Condor. Condor then attempts to schedule one from the pool of idle machines. This process can take a significant amount of time, for example, if there are no machines available in Condor's pool. Therefore, Condor-PVM handles requests for new hosts asynchronously. The application can start other work immediately after it sends out a request for new host. It then uses the PVM notification mechanism to detect when the "host add" request was satisfied. This feature allows our greedy host request scheme to work well in practice.

Documentation and examples about these differences can be found at

`http://www.cs.wisc.edu/condor/.`

FATCOP was first developed as a PVM application, and modified to exploit Condor-PVM.

2.3 Design of the the Parallel program

FATCOP introduces parallelism when building the branch-and-bound tree. It performs bounding operations on several subproblems simultaneously. This approach may affect the order of subproblems generated during the expansion of the branch-and-bound tree. Hence more or fewer subproblems could be evaluated by the parallel program compared with its sequential version. Such phenomena are known as *search anomalies* and examples are given in later part of this Chapter.

2.3.1 Master-worker paradigm

FATCOP was designed in the master-worker paradigm: One host, called the master manages the work pool, and sends subproblems out to other hosts, called workers, that solve LPs and send the results back to the master. When using a large number of workers, this centralized parallel scheme can become a bottleneck in processing the returned information, thus keeping workers idle for large amounts of time. However this scheme can handle different kinds of resource failure well in Condor's opportunistic environment, thus achieve the best degree of fault tolerance. The basic idea is that the master keeps track of which subproblem has been sent to each worker, and does not actually remove the subproblem out of the work pool. All the subproblems that are sent out are marked as "in progress by worker i". If the master is then informed that a worker has disappeared, it simply unmarks the subproblems assigned to that worker.

2.3.2 A Greedy heuristic to use resources

Another design issue is how to use the opportunistic resources provided by Condor to adapt to changes in the number of available resources. The changes include newly available machines, machine suspension and resumption and machine failure. In a conventional dedicated environment, a parallel application usually is developed for running with a fixed number of processors and the solution process will not be started until the required number of processors are obtained and initialized. In Condor's opportunistic environment, doing so may cause a serious delay. In fact the time to obtain the required number of new hosts from Condor pool can be unbounded. Therefore we implement FATCOP in such a way that the solution process starts as soon as it obtains a single host. The solver then attempts to acquire new hosts as often as possible. At the beginning of the program, FATCOP places a number of requests for new hosts from Condor. Whenever it gets a host, it allocates work to this host then immediately requests a new host. Thus, in each period between when Condor assigns a machine to FATCOP and when the new host request is received by Condor, there is at least one "new host" request from FATCOP waiting to be processed by Condor. This greedy implementation makes it possible for a FATCOP job to collect a significant amount of hosts during its life cycle.

2.3.3 Worker grain size and default searching strategies

A critical problem in master-worker paradigm arises from contention issues at the master. When designing FATCOP, we first let each worker solve two linear programs before reporting back to the master. The master needs to deal with new information coming from the problems the workers have solved, as well as issues related to the addition or deletion of hosts from the virtual machine. When the linear program relaxations are relatively time consuming, this contention is not limiting, but in many cases, the relaxations solve extremely quickly due to advanced basis information.

To alleviate this problem, we defined a new notion of a worker and a task. A task is a subtree for the worker to process, along with a limit on the number of linear programs that can be solved, or a limit on the processing time. The rule to choose the an appropriate grain size at worker is arbitrary. We shall present our numerical results on this issue later in this chapter. The data associated with a task includes what subtree is to be processed along with strategy information for processing this tree, such as the value of the best feasible solution and the node and time limits for processing the task. Note that the subtrees passed to each worker are distinct.

However, a worker now has to store all the MIP information itself, as well as the linear program data. Thus, a worker has a state that consists of all this information (that is passed to it when the worker is initialized), along with any extra information (such as cuts or pseudocosts) that is generated by any task that runs on the worker.

The time limit feature generates new issues, namely how to pass back a partially explored subtree to the master. In order to limit the amount of information passed back, we use depth-first-search as the searching rule in the worker to explore the subtrees, since then a small stack can be passed back to the master encoding the remaining unexplored parts of the subtree. Furthermore, it is also easy to use the small changes to the LP relaxations in such a search mechanism to improve the speed of their solution. Finally, any "local information" that is generated in the subtree is valid and typically is most useful in the subtree at hand. As examples of this last point, we point to the reduced cost fixing and preprocessing techniques that we outline later in this Chapter.

The master solves the first linear programming relaxation. The optimal basis from this relaxation is sent to all workers, so that they may solve all subsequent nodes efficiently.

Furthermore, whenever there are less than a certain number of nodes in the work pool, we switch from a time limit in the worker to an LP solve limit of 1. This allows the work pool to grow rapidly in size. Another important question to be addressed is whether we need to save advanced linear programming basis for every node. The advantage of doing this is that workers can solve root LP of a subtree very faster. However, the amount of information passed back from a worker and stored in the master worker pool is now much bigger. This makes it hard to store many nodes in master's work pools due to the main memory constraint. Therefore, we choose not to store advanced basis information for the nodes and this saving in storage makes it possible to to use best bound as master's default node selection strategy. However when the size of the work pool reaches an upper limit, we switch the node selection strategy to use "deepest-node" in tree, with the expectation that the subtrees rooted at these nodes are likely to be completely explored by the worker, thus leading to a decrease in the size of the work pool.

2.3.4 Global and local information

2.3.4.1 Cutting Planes

Cutting planes provide globally valid information about the problem that is locally generated. Namely, a cutting plane generated at one processor may be used to exclude relaxed solutions occurring at another processor. The question arises of how to distribute the cutting plane information. We have chosen to attach this information to the worker by creating a cut pool on the worker. All newly generated cuts get sent to the master when a task completes, but this information is only sent to new workers, not to existing workers. Thus each worker carries cut information that was generated by the tasks that have run on the worker, but never receives new cuts from the master.

2.3.4.2 Pseudocosts

Pseudocosts pose a challenge to FATCOP in exactly the same way as cutting planes, in that they are globally useful information that is generated locally. As such, we choose to distribute pseudocosts in a manner similar to that for cutting planes. All new pseudocosts get sent to the master when a task completes, but this information is only sent to new workers, not to existing workers.

2.3.4.3 Heuristics

We have included a diving heuristic in FATCOP. The diving heuristic can be quite time consuming – too time consuming to be performed at every node of the branch and bound tree. In FATCOP, since a task is to explore an entire subtree for a specified time limit, this also gives a convenient way to decide from which nodes to perform the diving heuristic. Namely, the diving heuristic is performed starting from the root node of each task. The new integer solution found by the diving heuristic is a piece of globally valid information, thus it is found locally (on the worker) and should be sent back to the master. Preliminary testing revealed that for some instances this strategy for deciding when to perform the heuristic was also too time consuming. Therefore, if the total time spent in carrying out the diving heuristic grows larger than 20% of the total computation time, the diving heuristic is deactivated. Once the time drops to below 10%, the diving heuristic is reactivated.

2.3.4.4 Node preprocessing

It is usually advisable to preprocess the root problem, but it is not clear whether it is beneficial to preprocess every node of the branch-and-bound tree. In a sequential branch-and-bound MIP program, node preprocessing is usually considered too expensive. However, in FATCOP, every worker explores a subtree of problems. The cost of preprocessing is amortized over the subsequent LP solves. Preprocessing may improve the lower bound of this subtree, and increase the chance of pruning the subtree locally; however, the effects of node preprocessing are problem dependent. Therefore, we leave node preprocessing as an option in FATCOP.

The key issue is that the search strategy in FATCOP generates a piece of work whose granularity is sufficiently large for extensive problem reformulations to be effective and not too costly in the overall solution process. All the approaches outlined above are implemented to exploit the locality of the subproblems that are solved as part of a task, and in our implementation are carried out at many more nodes of the search tree than is usual in a sequential code. The benefits and drawbacks of this choice are further explored in a later part of this chapter.

2.4 Implementation of the parallel program

FATCOP consists of two separate programs: the master program and the worker program. The master program runs on the machine from which the job was submitted to Condor. This machine is supposed to be stable for the life of the run, so it is generally the machine owned by the user. The design of FATCOP makes the program tolerant to any type of failures for workers, but if the machine running the master program crashes due to either system reboot or power outage, the program will be terminated. To make FATCOP tolerant even of these failures, the master program writes information about subproblems in the work pool periodically to a log file on the disk. Each time a FATCOP job is started by Condor, it reads in the MIP problem as well as the log file that stores subproblem information. If the log file does not exist, the job starts from the root of the search tree. Otherwise, it is warm started from some point in the search process. The work pool maintained by the master program has copies for all the subproblems that were sent to the workers, so the master program is able to write complete information about the branch-and-bound process to the log file.

The worker program runs on the machines selected by Condor. The number of running worker programs changes over time during the execution of a FATCOP job.

Master and workers communicate through *tasks*. A task from the master to a worker is a subtree for the worker to process, other status and strategic information along with a limit on the number of linear programs that can be solved, or a limit on the processing time. A task from a worker to the master is a set of unsolved nodes as well as pseudocost, cutting planes and possible new integer feasible solutions. A schematic figure showing the basic flow of information and an overview of the implementation of FATCOP is given in Figure 4. The figure shows that the flow of communication between the master and a (prototypical) worker occurs only when the worker is initialized. Tasks run on a worker, get their initial data (MIP problem, cuts and pseudocosts) from the worker they are assigned to, and their specific task information (subtree to work on, incumbent solution) directly from the master. New pseudocost and cut information from the task is saved on the current worker and hence may be used by a new task that runs on this worker. The task also sends solution information (and newly generated cuts and pseudocosts) back to the master so that it can update its work pool, global cut and pseudocost pool, and incumbent solution.

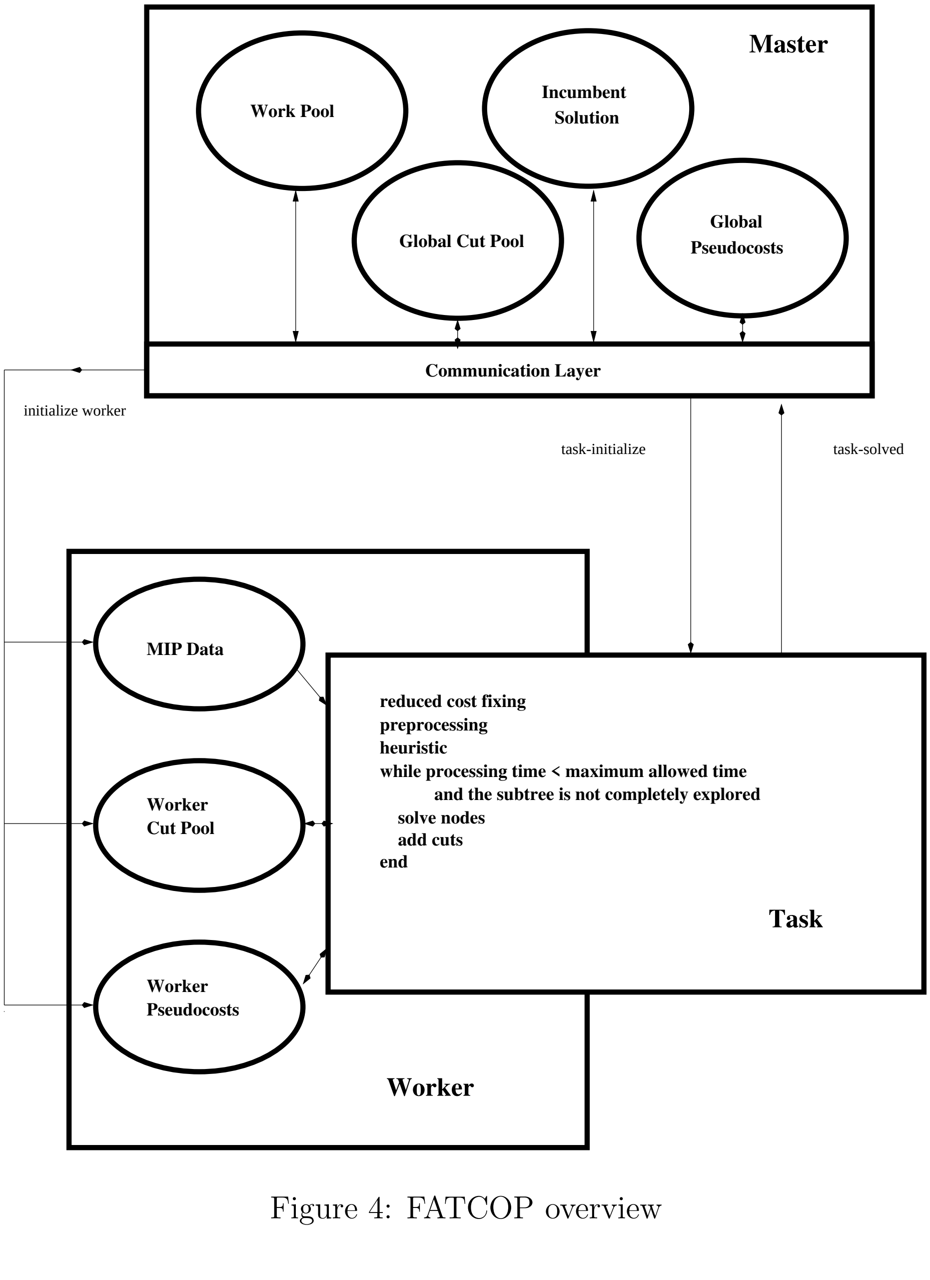

Figure 4: FATCOP overview

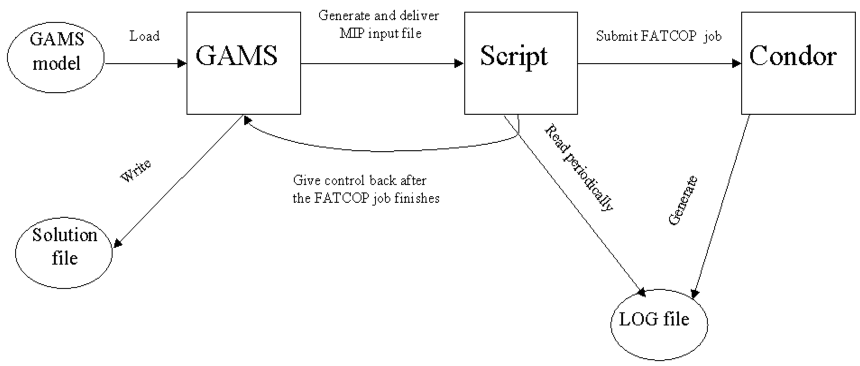

Figure 5: Interactions among Condor, FATCOP and GAMS

2.4.1 The Master Program

FATCOP can take MPS, GAMS and AMPL models as input. The interactions among Condor, FATCOP and GAMS are as follows. A user starts to solve a GAMS model in the usual way from the command line. After GAMS reads in the model, it generates an input file containing a description of the MIP model to be solved. Control is then passed to a PERL script. The script generates a Condor job description file and submits the job to Condor. After submitting the job, the script reads a log file periodically until the submitted job is finished. The log file is generated by Condor and records the status of the finished and executing jobs. After completion, control is returned to GAMS, which then reports the solution to the user. This process is depicted in Figure 5. The process is similar for AMPL and MPS file input.

The master program first solves the LP relaxation of the root problem. If it is infeasible or the solution satisfies the integrality constraints, the master program stops.

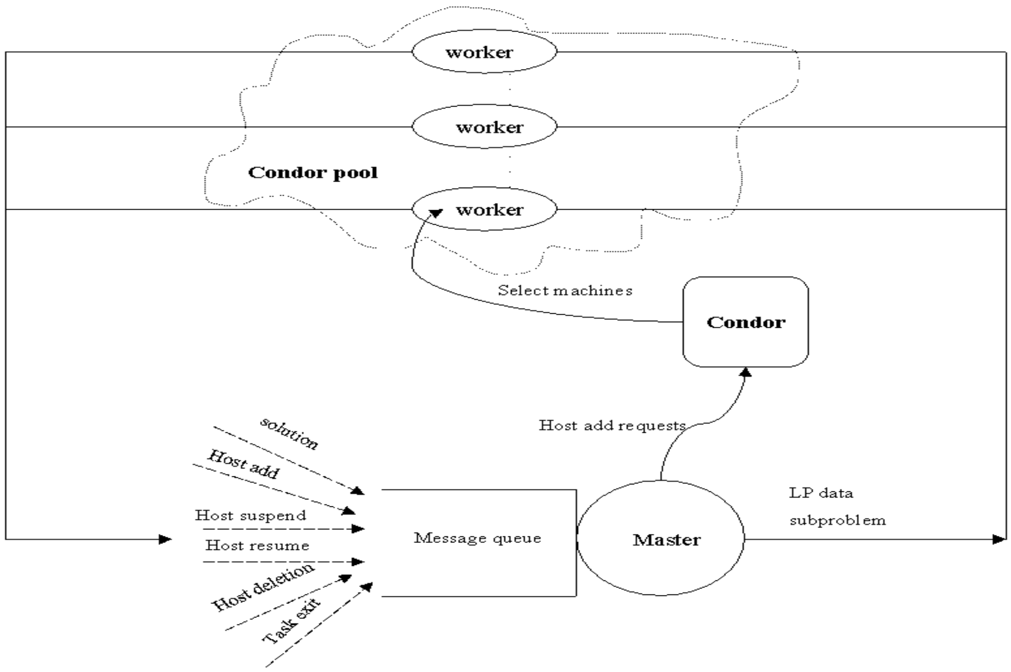

Figure 6: Message passing inside FATCOP

Otherwise, it sends out a number of requests for new hosts, then sits in a loop that repeatedly does message receiving. The master accepts several types of messages from workers. The messages passing within FATCOP are depicted in Figure 6 and are explained further below. After all workers have sent solutions back and the work pool becomes empty, the master program kills all workers and exits itself.

Host Add Message. After the master is notified of getting a new host, it spawns a child process on that host and sends MIP data as well as a subproblem to the new child process. The subproblem is marked in the work pool, but not actually removed from it. Thus the master is capable of recovering from several types of failures. For example, the spawn may fail. Recall that Condor takes the responsibility to find an idle machine and starts a PVM daemon on it. During the time between when the PVM daemon was started and the message received by master program, the owner of the selected machine can possibly reclaim it. If a "host add" message was queued waiting for the master program to process other messages, a failure for spawn becomes more likely.

The master program then sends out another request for a new host if the number of remaining subproblems is at least twice as many as the number of workers. The reason for not always asking for new host is that the overhead associated with spawning processes and initializing new workers is significant. Spawning a new process is not handled asynchronously by Condor-PVM. While a spawn request is processed, the master is blocked. The time to spawn a new process usually takes several seconds. Therefore if the number of subproblems in the work pool drops to a point close to the number of workers, the master will not ask for more hosts. This implementation guarantees that only the top 50% "promising" subproblems considered by the program can be selected for evaluation. Furthermore, when the branch-and-bound algorithm eventually converges, this implementation prevents the program from asking for excess hosts. However, the program must be careful to ensure that when the ratio of number of remaining subproblems to number of hosts becomes bigger than 2, the master restarts requesting hosts.

Solution Message. If a received message contains a solution returned by a worker, the master will permanently remove the corresponding subproblem from the work pool that was marked before. It then updates the work pool using the received results. After that, the master selects one subproblem from the work pool and sends it to the worker that sent the solution message. The subproblem is marked and stays in the work pool for failure recovery. Some worker idle time is generated here, but the above policy typically sends subproblems to workers that exploit the previously generated solution.

Host Suspend Message. This type of messages informs the master that a particular machine has been reclaimed by its owner. If the owner leaves within 10 minutes, the Condor processes running on this machine will resume. We have two choices to deal with this situation. The master program can choose to wait for the solutions from this host or send the subproblem currently being computed in this host to another worker. Choosing to wait may save the overhead involved in solving the subproblem. However the waiting time can be as long as 10 minutes. If the execution time of a FATCOP job is not significantly longer than 10 minutes, waiting for a suspended worker may cause a serious delay for the program. Furthermore, the subproblems selected from the work pool are usually considered "promising". They should be exploited as soon as possible. Therefore, if a "host suspend" message is received, we choose to recover the corresponding subproblems in the work pool right away. This problem then has a chance to be quickly sent to another worker. If the suspended worker resumes later, the master program has to reject the solutions sent by it in order that each subproblem is considered exactly once.

Host Resume Message. After a host resumes, the master sends a new subproblem to it. Note that the master should reject the first solution message from that worker. The resumed worker picks up in the middle of the LP solve process that was frozen when the host was suspended. After the worker finishes its work, it sends the solutions back to the master. Since the associated subproblem had been recovered when the host was suspended, these solutions are redundant, hence should be ignored by the master.

Host Delete/ Task Exit Message. If the master is informed that a host is removed from the parallel virtual machine or a process running on a host is killed, it recovers the corresponding subproblem from the work pool and makes it available to other workers.

2.4.2 Worker Program

The worker program first receives MIP data, cuts, pseudocosts and strategic information from the master. It then sits in an infinite loop to receive subproblems from the master. For each subproblem, the worker explores it in a depth-first manner. After the subproblem is solved or the resource limit on this worker is reached, it passes back the unexplored parts of the subtree to the master in a compressed form, along with the newly generated pseudocosts and cuts information. The worker program is not responsible for exiting its PVM daemon. It will be killed by the master after the stopping criteria is met.

2.5 Features and extensions

2.5.1 Object oriented design

FATCOP is implemented using Condor-PVM, an extension of the PVM programming environment that allows resources provided by Condor to be treated as a single (parallel) machine. As outlined before, FATCOP utilizes the master-worker computing paradigm. Thus many of the details relating to acquiring and relinquishing resources, as well as communicating with workers are dealt with explicitly using specific PVM and Condor primitives. Many of the features, and several extensions, of the resource management and communication procedures in FATCOP have been incorporated into a new software API, MW [29], that can be used for any master-worker algorithm. Since this abstraction shields all the platform specific details from an application code, FATCOP was designed to use this API, resulting in a much simpler, easier to maintain, object oriented code.

Other benefits also accrue that are pertinent to this work as well. First, MW provides the application (in this case FATCOP) with details of resource utilization that can

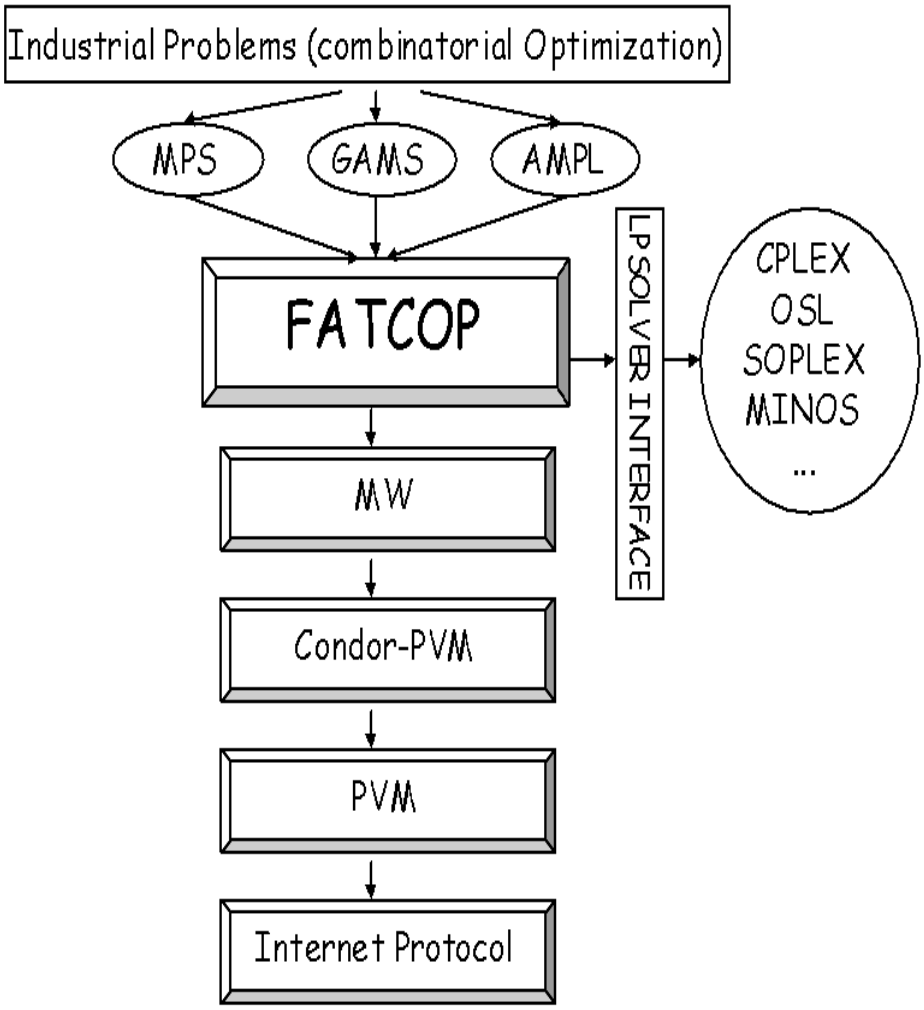

Figure 7: Object Oriented Design of FATCOP

be analyzed to improve efficiency. Secondly, new features of MW immediately become available for use in FATCOP. As an example, a new instantiation of MW that is built upon a communication model that uses disk files (instead of PVM messages) can now be used by FATCOP without any change to the FATCOP source code. Since this instantiation also uses standard Condor jobs instead of PVM tasks for the workers, facilities such as worker checkpointing that are unavailable in the PVM environment also become usable in the file environment. (Condor provides a checkpointing mechanism whereby jobs are frozen, vacated from the machine, and migrated to another idle machine and restarted.) Also, other potential instantiations of MW utilizing MPI or NEXUS for communication or Globus for resource management are immediately available to FATCOP. Thirdly, FATCOP can also drive new developments to MW, such as the requirement for a broadcast mechanism in MW to allow dispersion of new cuts to all workers. Such an extension would undoubtedly benefit other MW applications such as those outlined in [29].

SOPLEX is an effective linear programming code, but commercial codes such as CPLEX [40], OSL [39], and XPRESS [15] significantly outperform SOPLEX for solving the LP problem relaxations. In many cases, several copies of these solvers are available to a user of FATCOP and so we design the code to allow a variety of LP solvers to be used interchangeably. We used a general LP solver interface *LPSOLVER* that was developed at Argonne National Labs. The FATCOP code only needs to interact with LPSOLVER while the implementation of each concrete LP solver is invisible to FATCOP.

We depict the object oriented design of FATCOP in Figure 7. There are five layers of the software system. PVM sits upon network TCP and UDP protocol. Condor-PVM is an extension to PVM. MW is an abstraction of master-worker algorithm in Condor-PVM parallel programming environment. As an application of MW, FATCOP contains the major MIP algorithms. Three industrial standard interfaces are available, namely MPS, GAMS and AMPL, through which applications from different industries can be modeled. FATCOP interacts with the abstract LPSOLVER interface, under which all LP solvers are implemented.

2.5.2 Heterogeneity

The MW framework is built upon the model of requesting more resources as soon as resources are delivered. In order to increase the amount of resources available to the code, we exploited the ability of MW to run in a heterogeneous environment. In this way, the code garnered computational resources from a variety of machines including Sun SPARC machines running Solaris, INTEL machines running Solaris, and INTEL machines running Linux. While INTEL machines running NT are in the Condor pool, currently the MW framework is unavailable on this platform. To effect usage of workers on different architectures, all we needed to do was:

1. Compile each worker program for the specific architectures that it will run on.

2. Generate a new "job description file" for FATCOP 2.0 that details the computational resources that are feasible to use.

Since the source code for the solver SOPLEX [65] is available, compiling the worker code on several platforms is straightforward. The benefits of this increase in number of workers is shown in the end of this Chapter.

We allow FATCOP to use several LP solvers. At any given time, some of the workers may be using CPLEX, while others are using OSL and still others are using SOPLEX. The LP interface deals carefully with issues such as how many copies of CPLEX are allowed to run concurrently (for example if a network license is available), what machines are licensed for XPRESS, and what architectures can OSL be run upon. If none of the faster solvers are available, SOPLEX is used as the default solver.

2.5.3 User defined heuristics

In practice many problem specific heuristics are effective for finding near-optimal solutions quickly. Marrying the branch-and-bound algorithm with such heuristics can help both heuristic procedures and a branch-and-bound algorithm. For example, heuristics may identify good integer feasible solutions in the early stage of the branch-and-bound process, decreasing overall solution time. On the other hand, the quality of solutions found by heuristic procedures may be measured by the (lower-bounding) branch-and-bound algorithm. FATCOP can use problem specific knowledge to increase its performance. Based on interfaces defined by FATCOP, users can write their own programs to round an integer infeasible solution, improve an integer feasible solution and perform operations such as identifying good solutions or adding problem specific cutting planes at the root node. These user defined programs are dynamically linked to the solver at run time and can be invoked by turning on appropriate solver options.

We demonstrate a user defined heuristics using the following set cover problem.

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & Ex \geq e \\ & x_j = \ 0 \text{ or } 1 \end{aligned}$$

where $E = (e_{ij}$ is an m by n matrix whose entries e_{ij} are 0 or 1, $C = (c_j), j = (1, 2, \ldots, n)$ is a cost row with positive components, and e is an m vector of $1's$. If the $Ex \geq e$ constraints are replaced by the equalities: $Ex = e$, the integer program is referred to as a set partition problem.

We implemented a standard rounding heuristic for set covering problem given in many text books [59] and applied it to the problem with $m = 300, n = 600$. We called the heuristic routine every 10 iterations after an LP relaxation was solved in the first run, then solve the problem again with the rounding heuristic off. We show the progress of the two runs in Figure 8 and Figure 9. In the figures, the horizontal axis is the number of nodes FATCOP used, and the vertical axis is the current best solution value found. The upper line represents the change of upper bound, and the lower line represents the change of lower bound. Note the objective value of this problem is integer, so optimality should be proved when the difference of the lower bound and upper bound is less than 1.

From the figures we can see that with the rounding heuristics, FATCOP found integer feasible solution in the first iteration and the optimal solution was found after 800 iterations. It took about 3300 iterations to prove the optimality. However, without the rounding heuristics, FATCOP could not find a feasible solution until the 2600th iteration and it took about 4400 iterations to prove the optimality.

2.6 Numerical Results

A major design goal of FATCOP is fault tolerance, that is, solving MIP problems correctly using opportunistic resources. Another design goal is to make FATCOP adaptive to changes in available resources provided by Condor in order to achieve maximum possible parallelism. Therefore the principal measures we use when evaluating FATCOP are *correctness* of solutions, and *adaptability* to changes in resources. *Execution time* is another important performance measure. Due to the asynchronous nature of the algorithm, the nondeterminism of running times of various components of the algorithm, and the nondeterminism of the communication times between processors, the order in which the

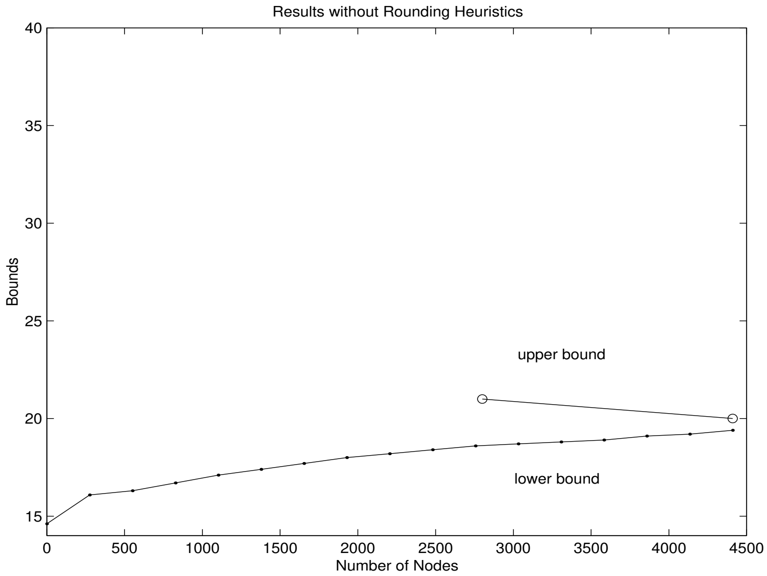

Figure 8: Without Rounding Heuristics

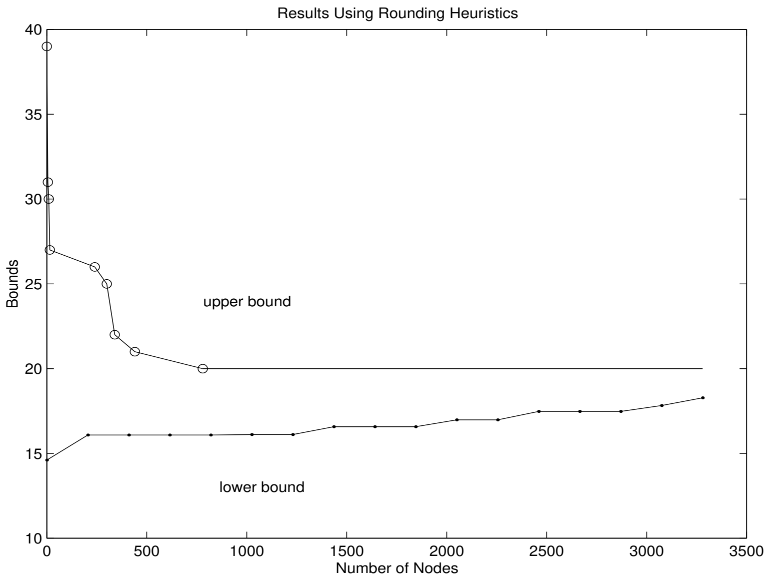

Figure 9: With Rounding Heuristics

nodes are searched and the number of nodes searched can vary significantly when solving the same problem instance. Other researchers have noticed the stochastic behavior of asynchronous parallel branch and bound implementations [18]. Running asynchronous algorithms in the dynamic, heterogeneous, environment provided by Condor only increases this variance. As such, for all the computational experiments, each instance was run a number of times in an effort to reduce this variance so that meaningful conclusions can be drawn from the results.

In this section we first show how FATCOP uses as many resources as it is able to capture, and demonstrate how reliable it is to failures in its environment. Following that, we assess different strategies for FATCOP and set up appropriate default strategies based on the experiments. We conclude this section with numerical results on a variety of test problems taken from the MIPLIB set and some real applications.

2.6.1 Resource Utilization

In Wisconsin's Condor pool there are more than 100 machines in our desired architecture class. Such large amounts of resources make it possible to solve MIP problems with fairly large search trees. However the available resources provided by Condor change as the status of participating machines change. Figure 10 demonstrates how FATCOP is able to adapt to Condor's dynamic environment. We submitted a FATCOP job in the early morning. Each time a machine was added or suspended, the program asked Condor for the number of idle machines in our desired machine class. We plot the number of machines used by the FATCOP job and the number of machines available to the job in Figure 10. In the figure, time goes along the horizontal axis, and the number of machines is on the vertical axis. The solid line is the number of working machines and dotted line is

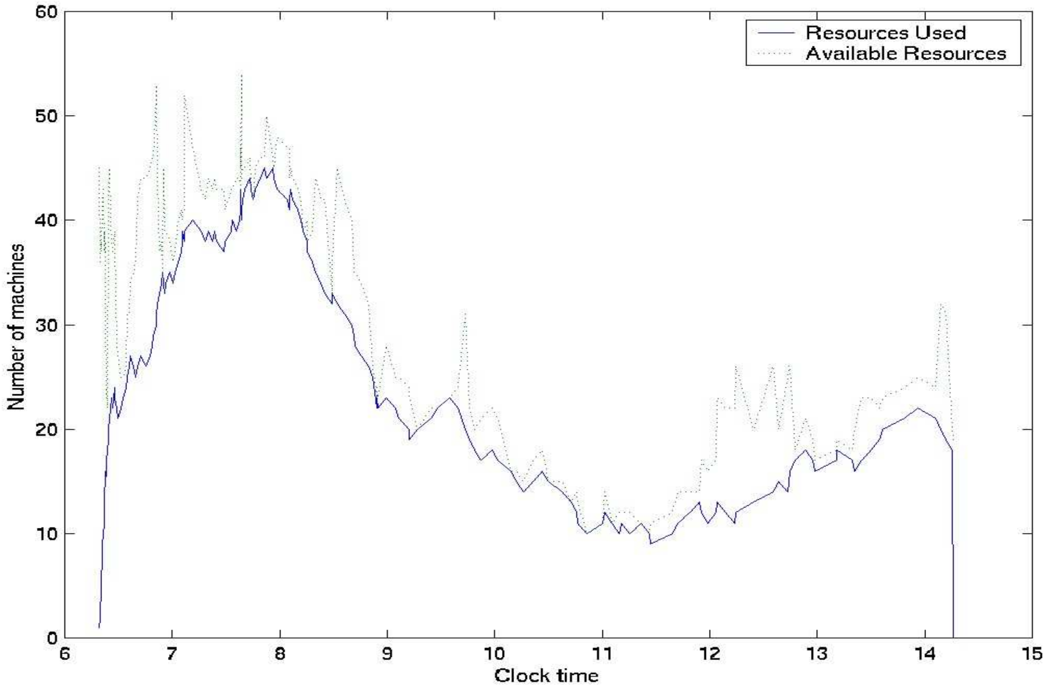

Figure 10: Resource utilization for one run of FATCOP

the number of available machines that includes idle machines and working machines used by our FATCOP job. At the start, there were some idle machines in Condor pool. The job quickly harnessed about 20 machines and eventually collected more than 40 machines with a speed of roughly one new resource every minute. At 8 a.m. it became difficult to acquire new machines and machines were steadily lost during the next four hours. There were some newly available resources at 8:30 and 10:00 (see the peaks of the dotted lines), but they became unavailable again quickly, either reclaimed by owners or scheduled to other Condor users with higher priority. At noon, another group of machines became available and stayed idle for a relatively long time. The FATCOP job acquired some of these additional machines during that time. In general, the number of idle machines in Condor pool had been kept at a very low level during life cycle of the FATCOP job

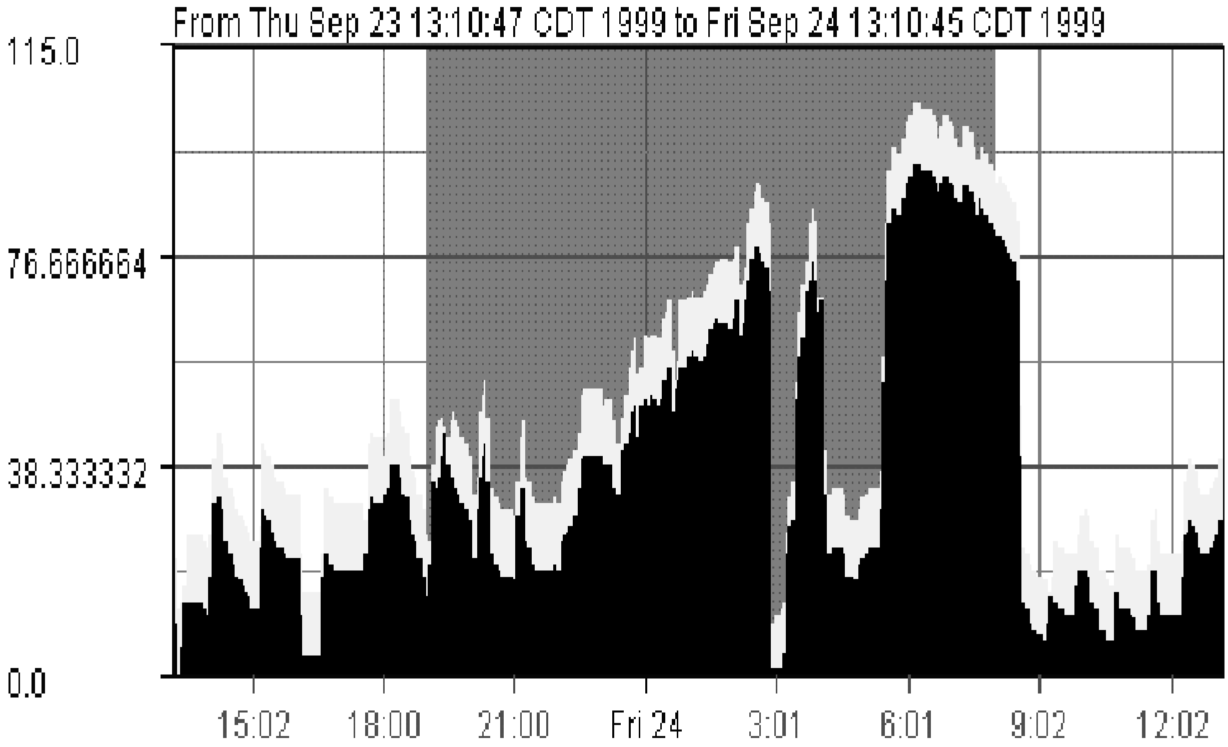

Figure 11: Daily log for a FATCOP job

except during the start-up phase. When the number of idle machines stayed high for some time, FATCOP was able to quickly increase the size of its virtual machine. We believe these observations exhibit that FATCOP can utilize opportunistic resources very well.

We show a FATCOP daily log in Figure 11. The darkly shaded area in the foreground is the number of machines used and the lightly shaded area is the number of outstanding resource requests to Condor from this FATCOP job. During the entire day, the number of outstanding requests was always about 10, so Condor would consider assigning machines to the job whenever there were idle machines in Condor's pool. At night, this job was able to use up to 85 machines. Note that the Computer Sciences Department at the University of Wisconsin reboots all instructional machines at 3 a.m. every day. This job

Run	Starting time	Duration	$\bar{P}$	Number of suspensions
1	07:50	13.5 hrs	32	145
2	12:01	14.9 hrs	29	181
3	16:55	11.1 hrs	40	140
4	21:00	10.1 hrs	49	118

Table 1: Average number of machines and suspensions for 4 FATCOP runs

lost almost all its machines at that time, but it quickly got back the machines after the reboot.

To get more insight about utilization of opportunistic resources by FATCOP, we define the average number of machines used by a FATCOP job $\bar{P}$ as:

$$\bar{P} = \frac{\sum_{k=1}^{P_{max}} k\tau_k}{T}, \tag{1}$$

where τ_k is the total time when the FATCOP job has k workers, T is the total execution time for the job, P_{max} is the number of available machines in the Condor's pool. We ran 4 replications of a MIP problem. The starting time of these runs is distributed over a day. In Table 1 we record the average number of machines the FATCOP job was able to use and number of machines suspended during each run. The first value shows how much parallelism the FATCOP job can achieve and the second value indicates how much additional work had to be done. In general the number of machines used by FATCOP is quite satisfactory. At run 4, this value is as high as 49 implying that on average FATCOP used close to 50% of the total machines in our desired class. However, the values vary greatly due to the different status of the Condor pool during different runs. In working hours it is hard to acquire machines because many of them are used by owners. After working hours and during the weekend, only other Condor users are our major competitors. As expected FATCOP lost machines frequently during the daytime.

Table 2: Effect of node preprocessing, data averaged over 3 replications

Name	Node preprocessing			No node preprocessing		
	Nodes	Time	$\bar{P}$	Nodes	Time	$\bar{P}$
cap6000	119232	6530.2	30	129720	3317.0	30
egout	11	11.3	2	26	12.3	2
gen	7	14.2	3	19	195.5	4
l152lav	4867	222.6	17	6018	475.1	25
p0548	215	16.1	2	222	20.0	2
p2756	2447	928.5	19	3044	1058.4	36
vpm2	1070217	654.5	17	1897992	940.1	40

However during the runs at night FATCOP also lost many machines. It is not surprising to see this, because the more machines FATCOP was using, the more likely it would lose some of them to other Condor users.

2.6.2 Assessing FATCOP strategies

2.6.2.1 Assessing node preprocessing

It is well known that lifted knapsack covers, flow covers and diving heuristics are effective in solving MIP problems [14, 57, 52]. However, the reported overall benefits of node preprocessing are less clear due to the amount of computing time they may take. A key issue is that node preprocessing is too expensive to carry out at every node. Since our tasks now correspond to subtrees of the brand-and-bound tree, it makes sense in this setting to experiment with preprocessing just at the root nodes of these subtrees. In this section we report results for experiments that ran a number of MIP problems with node preprocessing turned off and on, while all other advanced features (cutting planes, diving heuristics, reduced cost fixing and root preprocessing) were turned on.

The algorithmic parameters that were used are as stated above. Each instance was replicated three times. We report the number of nodes, the wall clock time and the

Table 3: Effect of varying worker grain size: results for vpm2

Grain size	$\bar{E}$	Nodes	Time	$\bar{P}$
2	0.16	809945	1335.8	45
100	0.64	1479350	743.9	25
200	0.61	1938241	1053.2	29

average number of processors $\bar{P}$ used with and without node preprocessing in Table 2.

As expected, all the test problems were solved in less nodes with node preprocessing, since the subtrees were pruned more effectively in the branch-and-bound process. An interesting observation is that it took longer to solve cap6000 even though the search tree is smaller with node preprocessing. In fact, node preprocessing combined with local reduced cost fixing worked very effectively on this problem. After the first integer feasible solution was found, preprocessing and reduced cost fixing usually can fix more than half of the binary variables at the root node of a subtree. But the problem is that cap6000 has a very large LP relaxation. The cost to reload the preprocessed LP model into the LP solver is significant compared with task grain size. This observation suggests a better implementation for modifying a formulation in a LP solver is necessary. However, based on this limited experimentation, FATCOP uses node preprocessing by default.

2.6.2.2 Grain size and master contention

A potential drawback of a master-worker program is the master *bottleneck* problem. When using a large number of processors, the master can become a bottleneck in processing the returned information, thus keeping workers idle for large amounts of time. In FATCOP , we deal with this problem by allowing each worker to solve a subtree in a fixed amount of time. The rule to choose an appropriate grain size at worker is arbitrary. In this section we show the results for FATCOP on vpm2 by varying worker grain size.

We ran FATCOP on vpm2 with worker grain size 2, 100 and 200 seconds respectively,

Table 4: Effect of search strategy: comparison of the parallel solver and sequential solver for tree size

Problem Name	sequential solver	parallel solver
air04	3,606	3,666
air05	18,512	14,775
l1521av	4,046	4,702
pp08acuts	3,469,870	5,001,600
vpm2	626,358	1,088,824

under the proviso that at least one LP relaxation is completed. In each case, we ran three replications employing all advanced features. The results are reported in Table 3. For each test instance, we report average worker efficiency $\bar{E}$, number of nodes, execution time, and average number of processors $\bar{P}$. The average worker efficiency, $\bar{E}$, was computed as the ratio of the total time workers spent performing tasks to the total time the workers were available to perform tasks. A grain size of two seconds had a very low worker utilization. Each worker finishes its work quickly, resulting in a large amount of result messages queued at the master. The node utilization corresponding to grain size of 100 seconds is satisfactory. Increasing grain size does not improve node utilization further. As stated in [9], all Condor-PVM programs risk losing the results of their work if a worker is suspended or deleted from the virtual machine. Taking this into consideration, we prefer a smaller worker grain size so that only small amounts of computation are lost when a worker disappears from the virtual machine. We have found that a grain size of around 100 seconds strikes a good balance between contention and loss of computation and is appropriate for the default.

Table 5: Effect of using heterogeneous machines: results for 10teams

Machine architecture	Nodes	Time	$\bar{P}$
SUN4	16910	763.5	24
X86	20364	1290.5	16
SUN4 and X86	23840	636.7	38

2.6.2.3 Search strategy effects

The FATCOP parallel program uses different default search strategies in the master and workers. Best bound searching rule is known to be the best rule to improve the branch-and bound algorithms convergence speed. It is set as the default searching rule in the master. However, in order to limit the amount of information passed from workers to the master, we use depth-first-search as the searching rule in the workers. As a result, the FATCOP parallel solver might need to explore a larger tree to solve a problem compared with its sequential counterpart that uses best bound as its default searching rule. We ran the FATCOP parallel and sequential solvers on a set of models from MIPLIB, and report the tree size in Table 4. For all the test problems, except *air05* that found good solution quickly displaying a strong search anomaly [18], the parallel solver evaluated more nodes than the sequential solver.

2.6.2.4 Heterogeneity

In this section we show how FATCOP exploits heterogeneous resources, including both heterogeneous machines and LP solvers. We ran the problem 10teams on a pool of Sun SPARC machines running Solaris (SUN4), a pool of INTEL machines running Solaris (X86), and a pool of both types of machine. Note that the worker executables are different on these different architectures. Each instance was replicated three times and we report the results in Table 5. Clearly, FATCOP was able to get more workers when

Table 6: Effect of using heterogeneous LP solvers: results for air04

LP solver	Nodes	Time	$\bar{P}$
SOPLEX only	3623	19125.0	43
SOPLEX and CPLEX	3661	6626.2	16

requesting machines from two architecture classes.

We also ran some experiments to show the effects of heterogeneous LP solvers. We solved the problem air04 with SOPLEX only, and both SOPLEX and CPLEX. We limited the maximum number of CPLEX copies to 10 in the latter case. Results are shown in Table 6. The problem air04 has very large LP relaxations, so the worker running SOPLEX usually can only solve one LP in the specified grain size (120 seconds), while a worker running CPLEX is able to evaluate a number of nodes in the depth first fashion outlined previously. We notice from Table 6 that using CPLEX and SOPLEX the problem was solved three times faster using less machines compared with using SOPLEX only.

2.6.3 Raw performance

Based on the experiments outlined above, we set appropriate choices of the parameters of our algorithm. In this subsection, we attempt to show that FATCOP works well on a variety of test problems from MIPLIB. Our test set is from MIPLIB. The selected problems have relatively large search trees, so that some parallelism can be exploited.

In Table 7, for each test problem, we report the number of nodes, solution time, average worker efficiency $\bar{E}$, and average number of processors $\bar{P}$, averaged over the five replications that were carried out. For each of these statistics, we also report the minimum and maximum values over the five replications.

Figure 12 shows, for one particular trial and instance, the number of participating processors. Figure 13 shows, for the same trial and instance, the instantaneous worker

Table 7: Performance of FATCOP: min (max) refer to the minimum (maximum) over five replications of the average number of processors (nodes, time) used

Instance	Statistic	$\bar{E}$	$\bar{P}$	Nodes	Time
10teams	average	64	44	9340	677
	[min max]	[49 79]	[34 49]	[8779 9655]	[550 754]
air04	average	84	82	3666	2639
	[min max]	[79 89]	[68 91]	[3604 4019]	[2308 3033]
air05	average	45	69	14755	1515
	[min max]	[41 54]	[57 79]	[9979 17419]	[1353 2549]
danoint	average	88	61	686680	60586
	[min max]	[71 95]	[53 66]	[630954 708513]	[59514 60586]
fiber	average	64	23	9340	125
	[min max]	[56 69]	[19 29]	[8779 9655]	[108 143]
gesa2	average	60	53	7965014	2982
	[min max]	[50 66]	[44 61]	[7013876 8243657]	[2768 3044]
gesa2_o	average	91	78	2739772	1818
	[min max]	[82 94]	[74 88]	[2206782 4031245]	[1642 2219]
l152lav	average	51	16	4702	206
	[min max]	[43 58]	[11 20]	[3985 6381]	[118 317]
modglob	average	52	3	358	27
	[min max]	[42 58]	[3 3]	[21 953]	[21 53]
p2756	average	51	14	2145	995
	[min max]	[44 62]	[8 21]	[1936 3115]	[866 1216]
pk1	average	74	55	3047981	2800
	[min max]	[66 79]	[37 70]	[3018755 4148176]	[2111 3567]
pp08aCUTS	average	67	54	4213412	2038
	[min max]	[55 70]	[47 61]	[3785673 4648207]	[1500 2353]
qiu	average	61	23	9687	303
	[min max]	[49 71]	[18 27]	[6249 14115]	[266 347]
rout	average	91	94	4510670	42274
	[min max]	[89 94]	[78 101]	[4249369 4600843]	[37697 45326]
vpm2	average	73	17	1088824	633
	[min max]	[65 79]	[13 21]	[974832 1344618]	[453 701]

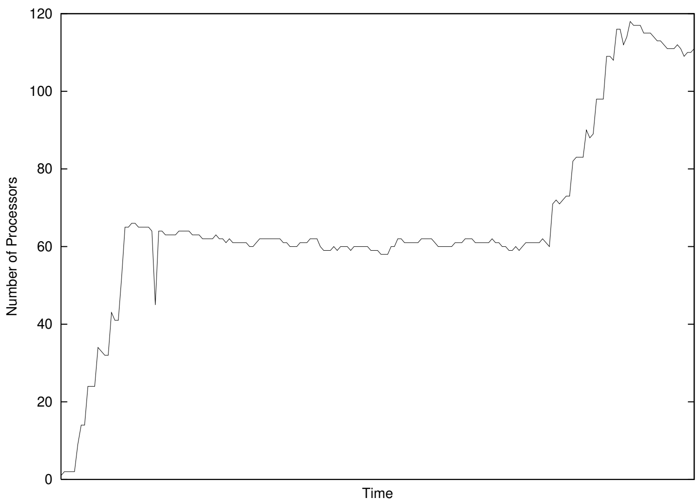

Figure 12: Average number of processors participating in solving gesa2_o

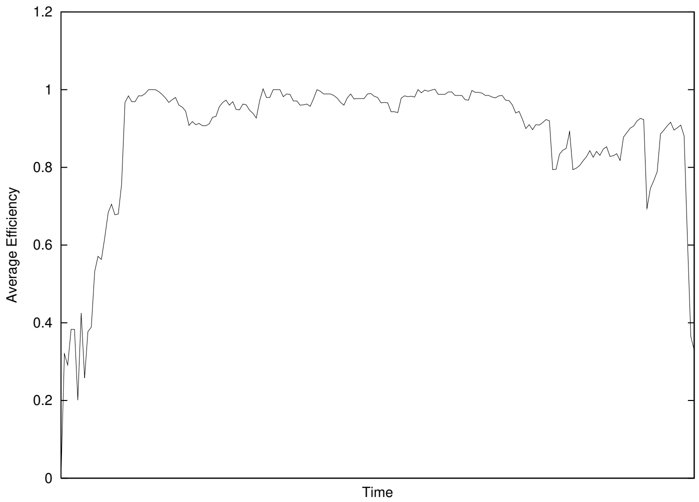

Figure 13: Average worker efficiency during solution of gesa2_o

Name	#rows	#columns	#nonzeros	#integers	application area
mcsched	2107	1747	8088	1731	Conference Scheduling
nsa	1297	388	4204	36	National Security Agency
sp97ic	1088	1662	53245	1662	Dutch Railway
sp98ar	4149	5478	211207	5478	Dutch Railway
sp98ic	2124	2508	122518	2508	Dutch Railway
t0415	1518	7254	48867	7254	Set Covering

Table 8: Summary of the test problems from real applications

efficiency, measured as $\sum_{k=1}^{n_t} l_k^t/n_t$, where n_t is the number of processors participating at time t and l_k^t is the *load average* of the processor k at time t. The load average, computed using the UNIX command `uptime`, and number of participating processors were sampled at 30 second intervals during the run.

The efficiency of a run may be less than "ideal" (1.0) due to

- Contention – The workers are idle during the time they send the results of their task to the master until they receive the next task. If the master needs to respond to many requests, workers may idle for long periods waiting for new work, thus reducing efficiency.

- Starvation – There are not enough active tasks in the work pool for all the participating workers.

- Inaccuracy of measurements – The load average reported by the UNIX operating system is computed as the average number of processing jobs during the last minute, so even though a processor is working on a task, the reported load average may be less than 1.0.

The last experiment is to test FATCOP on some MIP models from real applications. Problem size and application area for the test problems are summarized in Table 8. All

Name	Execution time	Tree size	$\bar{P}$	Solution Gap
mcsched	1.3 hrs	1,331,176	41	0%
nsa	31.8 hrs	255,854,364	111	0%
sp97ic	2.1	822,588	35	3%
sp98ar	1.1 hrs	147,597	33	3%
sp98ic	0.7 hrs	130,600	23	3%
t0415	30.6hrs	531,707	96	0%

Table 9: Results for real applications obtained by FATCOP

Name	Solver	5%	3%	1%	0%
mcsched	FATCOP	1.2	1.2	1.2	1.3
	CPLEX	-	-	-	-
nsa	FATCOP	18.6	24.5	29.2	31.8
	CPLEX	-	-	-	-
sp97ic	FATCOP	0.7	2.1	-	-
	CPLEX	3.2	-	-	-
sp98ar	FATCOP	0.4	1.1	-	-
	CPLEX	2.4	3.8	-	-
sp98ic	FATCOP	0.3	0.7	-	-
	CPLEX	1.4	1.4	-	-
t0415	FATCOP	8.5	13.4	13.4	30.6
	CPLEX	-	-	-	-

Table 10: Comparison of FATCOP and CPLEX for the real applications. Time unit is hour. Both solvers have a time limit of 48 hours. “-” stands for no results

these test problems are large scale hard mixed integer programs.

We tried to solve these problems using FATCOP with a time limit 48 hours. Of the six problems, three of them were solved to optimality, and the others were solved to 3% gap (the relative difference of the lower bound and upper bound of the branch and bound algorithm). We report execution time, tree size, average number of machines used and relative gap in Table 9.

We also compared FATCOP with the CPLEX MIP solver on these test problems. We let both solvers run for 48 hours (can be less than 48 hours if a problem is solved to optimality before the time limit) on each problem. FATCOP outperformed CPLEX for the problems *mcsched*, *nsa*, *sp97ic* and *t0415* in terms of solution quality. Both FATCOP and CPLEX solved *sp98ar* and *sp98ic* to 3% gap in the given time, but FATCOP used less time than CPLEX to achieve the gap. We report gap and solution time for FATCOP and CPLEX in Table 10.

In one run of the *nsa* problem, FATCOP used 122 machines of three architectures, 26 of which were from a remote condor pool at University of New Mexico, 11 of which were from a condor pool at National Center for Supercomputing Applications. Average number of machine used is 111, wall clock time is 31.8 hours, total worker CPU time is 3014 hours and total worker suspension time is 40 hours.

Chapter 3

NP/GA: A New Hybrid Optimization Algorithm

In the previous Chapter, we presented a branch-and-bound solution to applications that involves discrete variables and can be modeled using mixed integer programs. However, in many realistic problems, the size of the branch-and-bound tree is enormous, and requires huge amounts of computing resources to solve the underlying program. Furthermore, due to bounding and fathoming, the shape of the tree is generally very irregular and cannot be determined a-priori. As such, it is important to develop effective heuristics for these problems. These heuristics must generate solutions fast with cost close to the best, or optimal, cost.

There are a number of general-purpose heuristic techniques that have proved useful in industrial problems. Among them the most important class of techniques are referred to as local search procedures. A procedure based on local search usually attempts to find a solution better than the current one through a search in the neighborhood of the current solution. Two solutions are neighborhood if one can obtained through a defined modification of the other. Neighborhood search procedures that are currently popular are simulated annealing [19], tabu search [26] and genetic algorithms [27]. Simulated annealing is a search procedure that mimics the physical process of annealing in an attempt to escape local optima. In simulated annealing moves to worse solutions are allowed. The reason for allowing these moves is to give the procedure the opportunity to move away from a local minimum and find a better solutions later on. The probability for a non-improving move is lower in later iterations of the search process. Tabu search is in many ways similar to simulated annealing. The procedure also moves from one solution to another, with the next solution being possibly worse than the preceding solution. The basic difference between tabu search and simulated annealing lies in the mechanism used for approving candidate moves. In tabu search the mechanism is not probabilistic but rather of a deterministic nature. Genetic algorithm is a search method based on the concept of natural selection. It starts from an initial population and then uses a mixture of reproduction, crossover, and mutation to create new populations. Unlike simulated annealing and tabu search, a genetic algorithm works with a population of solutions in order to get improvement.

All of above algorithms are sequential in the sense that they move iteratively between single solutions or sets of solutions. However, in some applications it may be desirable to maintain a more global perspective, that is, to consider the entire solution space in each iteration. In this thesis we propose a new optimization algorithm (NP/GA) to address the difficult combinatorial problems. The new method combines two optimization algorithms: the Nested Partitions (NP) method and the GA.

The NP method is a randomized optimization method that has recently been developed for global optimization [64]. This method has been found to be promising for difficult combinatorial optimization problems. The NP method may be described as an adaptive sampling method that uses partitioning to concentrate the sampling effort in those subsets of the feasible region that are considered the most promising. It combines global search through global sampling of the feasible region, and local search that is used to guide where the search should be concentrated. The method is flexible in that it can be applied to a variety of optimization problems, and it can also be combined with other methods. Although the GA method works with a population of strings rather than a single string, it is essentially a sequential heuristic and is not guaranteed to converge to a global optimum. The NP method, on the other hand, maintains a global perspective in every iteration and converges in finite time to a global optimum [64]. The hybrid algorithm proposed in this thesis retains the benefits of both methods.

In this Chapter we first describe the NP method: its algorithm, components and convergence results. We then study the convergency speed of the NP method through investigating the underlying Markov chain of the NP algorithm. Following that we present the new hybrid algorithm NP/GA and show that NP/GA has faster convergency speed than pure NP method through numerical results.

3.1 The Nested Partitions method

In this section we describe the NP method in the context of deterministic optimization for combinatorial problems. We want to solve the following problem:

$$\min_{\theta \in \Theta} f(\theta). \tag{2}$$

where $f : \Theta \to \mathbf{R}$ is an objective function, Θ is a finite discrete feasible region. The NP algorithm partitions the feasible region into several subregions and samples each subregion in a random manner. It then determines which region is the most promising based on sampling information. This most promising region is partitioned further and the other regions are aggregated into one subregion called the surrounding region. Each of these subregions is then randomly sampled. The most promising region hence receives more computational effort while the surrounding region is still considered. The algorithm

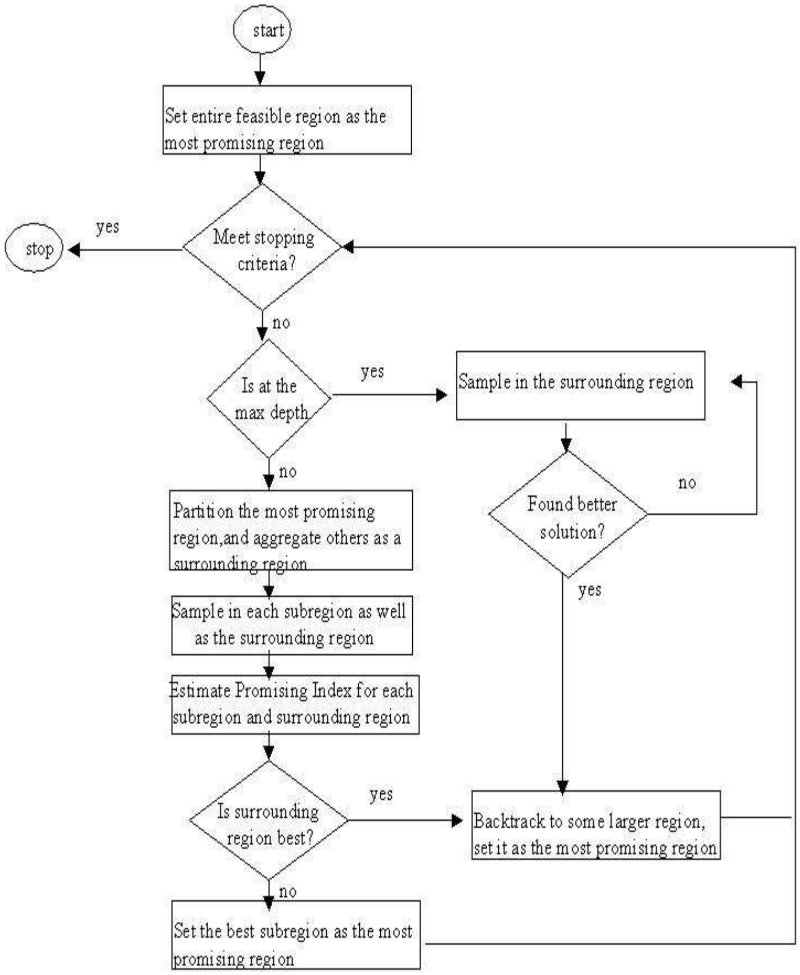

Figure 14: Flowchart of NP algorithm

proceeds in this manner and can either identify a smaller most promising region or backtrack to larger region dependent on the sampling results.

3.1.1 Algorithm

We define the following notations first, then describe a generic NP algorithm. The flowchart of the this algorithm is given in Figure 14.

$$\Sigma \;=\; \{\sigma \subseteq \Theta | \sigma \text{ is a region constructed using a selected partitioning scheme } \}$$

$$\sigma_k \;=\; \text{the most promising region in the k-th iteration} \sigma_k \in \Sigma$$

σ_k^i = the i-th subregion of $\sigma_k, \sigma_k^i \in \Sigma$

d_k = the depth of the nested partitions in the k-th iteration

d_* = the maximum depth

θ_i^j = the i-th sample point in the j-th subregion

M = number of subregions to be partitioned

N = sample size

Z^* = the best solution found so far

$\hat{P}$ = Promising index function,$\hat{P} : \Sigma \rightarrow \mathbf{R}$

Algorithm *Generic Nested Partitions:*

step 0. *Initialization:*

Let $k = 0$, $\sigma_k = \Theta$, $Z^* = infinity$

step 1. *Partition:*

Check stopping condition. If the stopping criteria is met, the algorithms stops. Otherwise, if $d_k = d_*$, go to step 7, else partition σ_k into σ_k^1, σ_k^2, $\ldots$, σ_k^M and aggregate $\sigma_k^{M+1} = \Theta \backslash \sigma_k$

step 2. *Sampling:*

For each region $\sigma_k^j, j = 1, 2, \ldots, M+1$, randomly pick N points: $\theta_1^j, \theta_2^j, ..., \theta_N^j$. Calculate $f(\theta_1^j), f(\theta_2^j), ..., f(\theta_N^j), j = 1, 2, \ldots, M+1$. Then compute the promising index by

$$\hat{P}(\sigma_k^j) = \min_{i=1,2,\ldots,N} f(\theta_i^j), j = 1, 2, ..., M+1, \tag{3}$$

step 3. *Calculating the Index of the Promising Region:*

Update Z^* if necessary.

$$Z_k = \min_{j=1,\ldots,M+1} \hat{P}(\sigma_k^j) \tag{4}$$

If $Z_k < Z^*$, let $Z^* = Z_k$. Then calculate the index of the most promising region

$$\hat{j}_k = arg \min_{j=1,...,M+1} \hat{P}(\sigma_k^j) \tag{5}$$

step 4. *Identify a smaller promising region:*

If $\hat{j}_k = M + 1$, go to step 5. Otherwise, let $\sigma_{k+1} = \sigma_k^{\hat{j}_k}$ and $d_{k+1} = d_k + 1$

step 5. *Backtracking:*

Let $\sigma_{k+1} = \sigma_k$ and $d_{k+1} = d_k - 1$

step 6. *Increase counter:*

Let $k = k + 1$, go to step 1

step 7. *Sample in the maximum depth:*

The maximum depth region only contains a single point. Let f^* denotes its objective value. Randomly pick N points in Θ: $\theta_1, \theta_2, ..., \theta_N$. Calculate $f(\theta_1), f(\theta_2), ..., f(\theta_N)$. Then compute the minimum sample values.

$$f_{min} = \min_{i=1,2,...,N} f(\theta_i), \tag{6}$$

If $f_{min} < f^*$, let $\sigma_{k+1} = \sigma_k$ and $d_{k+1} = d_k - 1$, then increase iteration counter $k = k + 1$, go to step 1. Otherwise, repeat this step.

3.1.2 Components of the NP algorithm

The algorithm described above has four main steps: *partition*, *sampling*, *calculation of promising index*, and *backtracking*. Each of these steps can be implemented in a generic fashion, but can also be adapted to take advantage of any special structure of a given problem. In the subsequent sections we illustrate the main steps through examples in hope that better understanding about the NP algorithm can be accomplished.

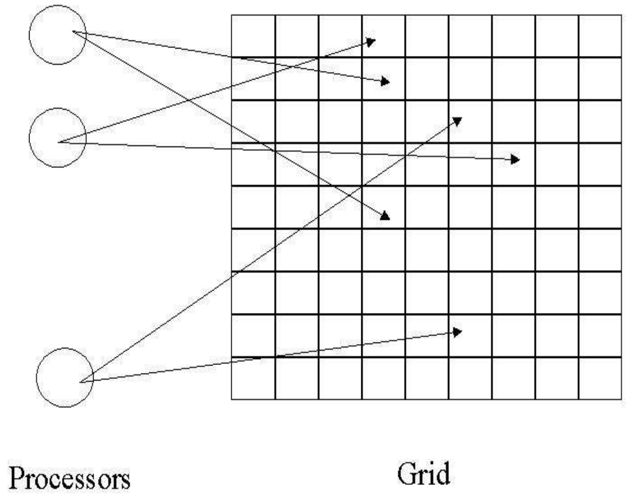

Figure 15: MPE problem

3.1.2.1 Partition

The partitioning strategy chosen is very important for the convergence of the algorithm. If the partitioning is such that most of the good solutions to the problem tend to be clustered together in subregions, it is likely that the algorithm quickly concentrates the search in these subsets of the feasible region. It should be noted that for problems with finite discrete solution space such good partitioning always exists, but may not be easy to identify.

We present two examples that illustrate how partition is accomplished. The first example has finite discrete feasible region, while the second has bounded continuous feasible region.

Example 3.1: Partitions for Minimum Perimeter Equi-partition problem Minimum Perimeter Equi-partition problem(MPE) is a graph partitioning problem that, when restricted to rectangular grids, can be stated as follows: given a rectangular grid of dimensions M*N and a number of processors P, where P divides MN, find the partition of the grid that minimizes the total perimeter induced subject to the constraint that each processor is assigned the same number of grid cells. Geometrically, the problem may be thought of as partitioning the grid into P equi-area regions(each of area $A := MN/P$) of minimum total perimeter. The problem is illustrated in Figure 3.1.2.1. An Integer Programming formulation is given in [12].

$$min \sum_{i,i'=1}^{M} \sum_{j,j'=1}^{N} \sum_{p,p'=1,p\neq p'}^{P} x_{iji'j'} x_{ij}^{p} x_{i'j'}^{p'} \tag{7}$$

s.t.:

$$\sum_{i=1}^{M} \sum_{j=1}^{N} x_{ij}^{p} = \frac{MN}{P} \tag{8}$$

$$\sum_{p=1}^{P} x_{ij}^{p} = 1 \tag{9}$$

$$x_{ij}^{p} \in B = 0,1 \tag{10}$$

where

$$c_{iji'j'} = \begin{cases} 1 \quad if \quad |i-i'| = 1 \quad and \quad j = j' \\ 1 \quad if \quad |j-j'| = 1 \quad and \quad i = i' \\ 0 \qquad\qquad else \end{cases} \tag{11}$$

This formulation has an objective function that is the sum of quadratic terms. The NP method however doesn't have any requirement on the objective function. This important feature distinguishes the NP method from gradient based searches. The objective function is described by 3-dimensional variables:

$$x_{ij}^{p} \qquad i = 1,2,...,M; \qquad j = 1,2,...,N; \qquad p = 1,2,...P$$

We can represent these variables in a 2-dimensional $P \times MN$ matrix:

$$R = \begin{pmatrix} x_{11}^1 & x_{12}^1 & \dots & x_{1N}^1 & x_{21}^1 & x_{22}^1 & \dots & x_{2N}^1 & \dots & x_{M1}^1 & x_{M2}^1 & \dots & x_{MN}^1 \\ x_{11}^2 & x_{12}^2 & \dots & x_{1N}^2 & x_{21}^2 & x_{22}^2 & \dots & x_{2N}^2 & \dots & x_{M1}^2 & x_{M2}^2 & \dots & x_{MN}^2 \\ \dots \\ x_{11}^P & x_{12}^P & \dots & x_{1N}^P & x_{21}^P & x_{22}^P & \dots & x_{2N}^P & \dots & x_{M1}^P & x_{M2}^P & \dots & x_{MN}^P \end{pmatrix}$$

R is subject to two constraints: (1) the summation of each row is equal to MN/P due to equation 8; (2) the summation of each column is equal to 1 due to equation 9. Note that there is only one non-zero element in each column of R. We only need to know the position of non-zero element in each column to represent a feasible solution. Thus a 1-dimensional array Y of size MN subject to constraint 8 is able to represent a feasible solution.

$$Y = (y_{11}\ y_{12}\ \dots y_{1N},\ y_{21}\ y_{22}\ \dots y_{2N},\ y_{M1}\ y_{12}\ \dots y_{MN})$$

where each element in the array can take one of the values $\{1, 2, ..., P\}$. If $M = N = P = 3$ (usually referred to as MPE(3,3,3)), array (1 2 3 2 3 1 3 1 2) represents the following feasible solution:

1	2	3
2	3	1
3	1	2

We can formally define the feasible region for MPE(M,N,P) problem as follow.

$$\Theta = \{\theta = (y_1, y_2, \dots, y_{MN}) | \theta \text{ is a permutation of } \{\overbrace{1\ 1\ ...\ 1}^{MN/P}\ ...\ \overbrace{P\ P\ ...\ P}^{MN/P}\}\}(1) \qquad (12)$$

We consider the following approach to partition Θ. Given values of M, N and P, first we divide Θ into P subregions by fixing y_1 to be one of $(1, 2, \dots, P)$. Then we further partition each subregion into P smaller subregions by fixing y_2 to be one of $(1, 2, \dots, P)$.

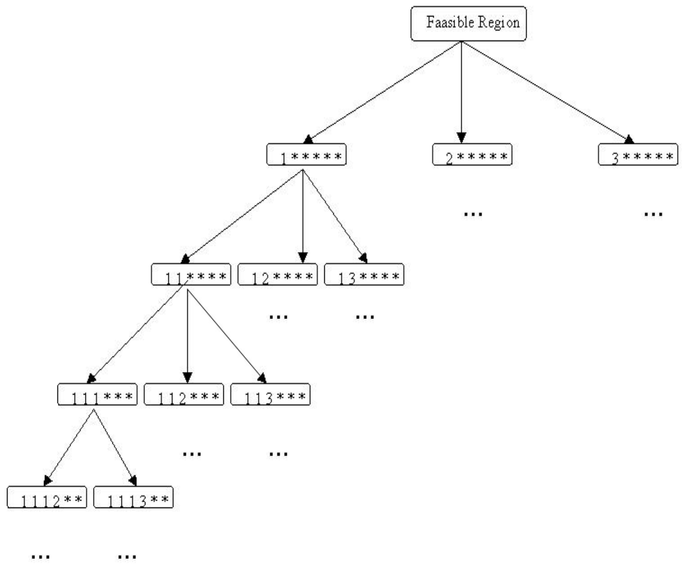

Figure 16: Partition for MPE problem

This procedure can be repeated until the maximum depth is reached, that is, when all cells have been assigned. Note that not all regions have P subregions because of constraint 8. Take MPE(3,3,3) as an example. If we partition a region with first three cells being assigned processor 1, we only can partition this region into two subregions, that is fixing y_4 to 2 and 3 respectively. Let (1 1 1 $******$) represent the region to be partitioned, where "*" means no processor is assigned to that cell. Then (1 1 1 1 $*****$) cannot be its subregion, because only $\frac{3\times 3}{3} = 3$ cells can be assigned processor 1. See Figure 16 for this partition.

Example 3.2: Partitions for Weights Recovering Problem in Analytical Hierarchy Process The Analytical Hierarchy Process (AHP) [58] is one of the most influential methods in multiple criteria decision making. It involves the following optimization problem to recover weights from a given preference matrix $A = (a_{ij})$.

$$min \sum_{i=1}^{n} \sum_{j=1}^{n} (a_{ij} - w_i/w_j)^2 \tag{13}$$

subject to:

$$\sum_{i=1}^{n} w_i \leq 1, w_i > 0 \tag{14}$$

This constrained nonlinear programming problem has simple constraint but a non-convex objective function. Existing nonlinear programing solvers such as MINOS5 can not guarantee an optimal solution. The NP method can be applied to this problem. A super region of this problem is as follow.

$$\Theta = \{(w_1, w_2, ..., w_n) | \sum_{i=1}^{n} w_i \leq 1, w_i > 0\} \tag{15}$$

Θ is an n-dimensional simplex with vertices

$$v_0^0 = (0, 0, \ldots, 0), \qquad v_1^0 = (1, 0, \ldots, 0), \ldots v_n^0 = (0, 0, \ldots, 1)$$

To partition Θ, we define the center of Θ as:

$$v_c^0 = \frac{1}{n+1} \sum_{i=0}^{n} v_i^0 \tag{16}$$

Let the simplex $S_k^1, k = 0, 1, 2, ..., n$ have the following vertices

$$v_0^1 = v_0^0, \qquad v_1^1 = v_1^0, \ldots v_k^1 = v_c^0, \ldots, v_n^1 = v_n^0$$

Then $S_k^1, k = 1, 2, ..., n$ constitute n+1 subregions of Θ. S_k^1 then can be further partitioned by the same manner until sufficiently small subregions are obtained. See Figure 17 for a 2-dimensional example.

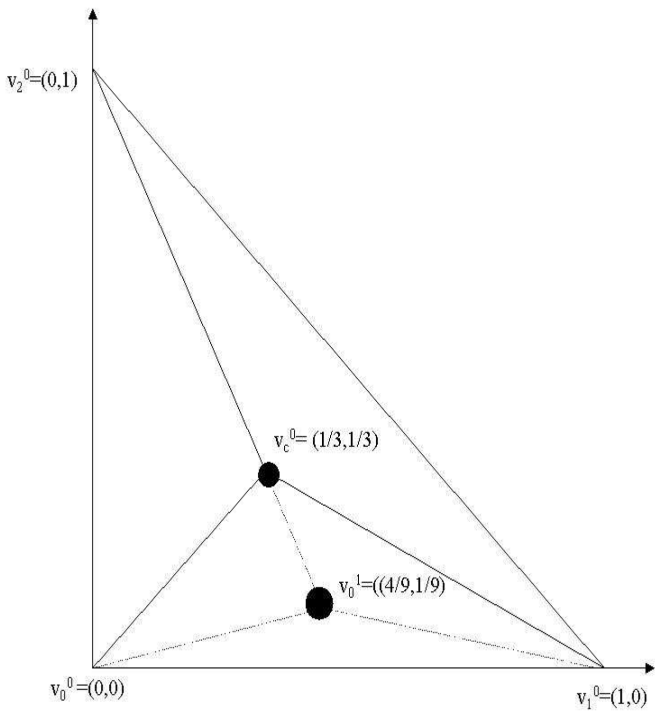

Figure 17: Partition for Weight Recovering problem

3.1.2.2 Sampling

Random sampling can be done in almost any fashion. The only condition is that each solution in a given sampling region should be selected with a positive probability. This condition is essential for the proof of convergency. Uniform sampling can always be chosen as the generic option. However, it may often be worthwhile to incorporate special structure into the sampling procedure. The goal of the random sampling is to select good solutions with a higher probability than poor solutions.

Example 3.3: Random Sampling for MPE problem Recall in the MPE(M,N,P) problem, a feasible solution can be represented by an array of size MN, while a region can be represented by an array of size L that is less than MN. Take MPE(3,3,3) as an example. Array (1 1 1 2 2 2 3 3 3) is a feasible solution, while (1 1 1 2) is a region with the first four elements fixed and the last five elements undetermined. To random sample in a region σ with the first L elements of each point in this region being fixed, we create a feasible point in σ first, then permute the last $MN - L$ elements to get a sample point. Suppose we want to sample in region (1 1 1 2). We first create a feasible point:

$$(1112 \quad \underline{22333}),$$

then permute the last five elements (2 2 3 3 3) to get sample points.

Example 3.4: Random Sampling for the Weight Recovering Problem in AHP

Random sampling for weight recovering problem involves generating random vectors inside of a simplex. The *pdf* of a random vector x uniformly distributed inside an n-dimensional simplex

$$S_n = \{(x_1, x_2, ..., x_n) | \sum_{i=1}^{n} x_i \leq 1, x_i \geq 0\} \tag{17}$$

is

$$f_X(x) = \begin{cases} n! & if \quad x \in S_n \\ 0 & otherwise \end{cases} \tag{18}$$

The density is a particular case of Dirichlet density

$$f_X(x) = \frac{\Gamma(v_1 + ... + v_{n+1})}{\Gamma(v_1)...\Gamma(v_{n+1})} x_1^{v_1 - 1} ... x_n^{v_n - 1} (1 - x_1 - ... - x_n)^{v_{n+1} - 1} \tag{19}$$

with parameters $v_1, v_2, ..., v_n, v_{n+1}$. Equation 17 equals 18 for $v_1 = v_2 = ... = v_{n+1} = 1$. We shall denote the density 17 D(1,...,1,1). It is also known that if $Y_i, i = 1, 2, \ldots, n+1$ are independent variables distributed according to gamma distribution with parameter v_i and 1, that is,

$$f_{Y_i}(y) = \begin{cases} \frac{y^{v_i - 1} e^{-y}}{\Gamma(v_i)} & if \quad y > 0, v_i > 0 \\ 0 & otherwise \end{cases} \tag{20}$$

The the random vector

$$X = (X_1, ..., X_n) = (\frac{Y_1}{\sum_{i=1}^{n+1} Y_i}, ..., \frac{Y_n}{\sum_{i=1}^{n+1} Y_i}) \tag{21}$$

is distributed $D(v_1, v_2, ..., v_n, v_{n+1})$. If $v_1 = v_2 = ... = v_{n+1} = 1$, $f_{Y_i}(y)$ is distributed exp(1). The algorithm for generating random variables from D(1,...,1,1) can be written as follows:

Algorithm *Uniform Sampling inside a n-dimensional simplex :*

step 1. Generate $n + 1$ random variables $(Y_1, Y_2, \ldots, Y_n, Y_{n+1})$ from exp(1).

step 2. Apply formula 21 and deliver X as a vector distributed from D(1,...,1,1).

3.1.2.3 Promising Index

The promising index of a region is calculated by using the samples obtained in this region. It provides a measure on the "quality" of a region. The generic NP algorithm given in the beginning of this Chapter uses the minimum sampling values as the estimate of promising index. It is obvious that one can average the sampling values to estimate the promising index. The NP method in fact offers a great deal of flexibility in this step. The only requirement imposed on a promising index is that it agrees with the original performance function on regions of maximum depth, i.e., on singletons. Due to such a simple requirement, many heuristics can be used in this step to construct a promising index.

Example 3.5: Promising index for MPE problem For the MPE(M,N,P) problem, suppose we get a random sample θ in region σ where the first $L-th$ elements are fixed. To estimate the promising index of σ, we perform tabu search for a fixed number of iterations using θ as starting point. We define the neighborhood of θ as the set of feasible solutions

Figure 18: One iteration of tabu search to obtain better promising index for MPE problem

obtained by swapping two adjacent cells. The key issue is that the new solution should be also in σ, hence we let the swapping operations do not involve the fixed elements in σ. For the MPE(3,3,3) problem, suppose we get a sample $\hat{\theta} = (1\ 1\ 1\ 2\ 3\ 2\ 3\ 2\ 3)$ in region $\hat{\sigma} = (1\ 1\ 1\ 2)$. The objective value of $\hat{\theta}$ is 10. We keep the first four elements fixed and perform tabu search for 1 iteration. The resulting feasible solution in region $\hat{\sigma}$ is ($\hat{\theta}' = 1\ 1\ 1\ 2\ 2\ 2\ 3\ 3\ 3$) with objective value 6. Therefore the promising index of $\hat{\sigma}$ is 6. See 18 for the process of tabu search.

Example 3.5: Promising index for Weights Recovering Problem in AHP We also can incorporate local search to estimate a promising index for Weights Recovering problem in AHP. After we obtain a sample θ in region σ, we use θ as initial point and perform a local search using steepest decent method. Again we must be careful to keep the new solutions inside region σ. We let the steepest decent algorithm stop as soon as the new solution is out of σ or a pre-defined number of iterations is reached.

3.1.2.4 Backtracking

If the surrounding region has the best promising index, the generic NP algorithm backtracks. This can be done in several manners. The NP algorithm given in the beginning of this Chapter backtracks to the superregion of the current most promising region. An alternative way can be backtracking all the way to the top, i.e., to the entire feasible region, or or to any region that is between superregion and the entire feasible region. We sometimes consider backtracking to a region containing the best solution found in the surrounding region. In this case we can also retreat more than one depth.

The choice of backtracking scheme is important to developing efficient NP applications. The way of moving one depth back to the superregion of the current most promising region requires much more transactions to escape from a suboptimal solution. On the contrary, backtracking all the way back to the initial feasible region leaves the current region in just one transaction, but dismisses the previous sampling effort immediately. In our experience, for many problems it is efficient to backtrack to a superregion of the maximum depth region which contains the best solution found in the current iteration.

One potential drawback to the generic NP algorithm is that it may get stuck for considerable time at good but suboptimal solutions at the maximum depth. In other words if σ is a singleton region that corresponds to a solution that is better than most of the other solutions, it may take many iterations before the NP algorithm backtracks from this region. More computational effort might be needed in the surrounding region to quickly escape this suboptimal solution. Another approach to escape suboptimal solution at maximum depth is making use of history information. The generic NP algorithm only store the information about the current most promising region and the best found solution. Those good but not optimal solutions may be used to come up with better solutions and help the algorithm to backtrack quickly when the algorithm reaches the maximum depth. We propose an approach to use history information in the hybrid algorithm of NP and genetic algorithm in the next Chapter. There is not any general scheme to use history information in the NP algorithm yet, and this issue deserves more research.

3.1.3 Convergence

The NP algorithm for problems with finite discrete solution space converges to a global optimum in finite time. Due to the randomness of sampling, the sequence of the most promising region in each iteration is a stochastic process. Since the most promising region in any iteration only depends on which region is the most promising in the previous iteration, the sequence of the most promising region is a Markov chain. The optimal solutions, as singleton regions in the Markov chain are the only absorbing states. All other states are transient, hence can be visited a finite number of times. Since the number of states is also finite, the absorbing time must be finite too. The Markov structure can also be used to derive results about the expected number of iterations until the global optimum is found. The results can be used as a basis for stopping criteria.

3.2 Finite Time Behavior and Stopping Criteria of the NP method

In the last section we showed that the NP Markov chain will be absorbed at a global optimum in finite time. It is of much more interest to answer questions such as:

How quickly does the algorithm converge to a global optimum?

The answer to this question can be used as a basis to decide when the NP algorithm should stop. In this section we propose an approach to estimate expectation and variance of *algorithm length*, the number of iterations until the NP Markov chain is absorbed. We first summarize the notations used in this section.

notations

Θ : The initial feasible region.

Σ_1 : The regions containing at least one optimal solution(not including Θ) .

Σ_2 : The regions which do not contain any optimal solution.

Π : The set of all feasible regions. $\Pi = \Theta \cup \Sigma_1 \cup \Sigma_2$.

d^* : Maximum depth.

M : Number of partitions for each region

p_0 : The probability moving in the correct direction(a move from $\eta_1 \in \Sigma_1$ to $\eta_2 \in \Sigma_1$, or from $\eta_1 \in \Sigma_2$ to Θ.

p_1 : The probability moving from $\eta_1 \in \Sigma_1$ to Θ. $p_1 < 1 - p_0$.

X : The number of iterations until the NP algorithm is absorbed at a global optimum, referred to as *algorithm length*.

3.2.1 Estimate expectation and variance of *algorithm length*

Assume that:

A1: The problem only has a single global optimum.

A2: The NP algorithm backtracks all the way to the initial feasible region.

A3: p_0 is the probability of the NP algorithm moving in the correct direction, i.e., backtracking if the optimum is not in the current most promising region and selecting the correct subregion if it is in the current region. p_0 is referred to as success probability in the rest of this thesis.

A4: The probability of moving in a specific wrong direction is the same in all directions, and hence is given by $(1-p_0)/M$. Recall M is the number of subregions being partitioned.

Based on above assumptions, we state the following theorem.

Theorem 1 *The expected number of iterations until the NP algorithm considers the global optimum to be the most promising region is given by the following equation:*

$$E(X) = \frac{1 - p_0 p_1 - p_0^{d^*}(1 - p_0 - p_1 + p_0^2)}{p_0^{d^*-1}(1 - p_0)} \tag{22}$$

proof: Assume that the NP algorithm starts from the initial feasible region Θ, then the algorithm will move in either a correct or wrong direction in the next iteration. Define $E_k = E(X|\text{from } \eta \in \Sigma_1 \text{and } d(\eta) = k)$, $k = 1, 2, \ldots, d_*$, and $V = E(X|\text{from } \eta \in \Sigma_2)$, then conditioning on the first move we have:

$$E(X) = p_0 E_1 + (1 - p_0)V + 1 \tag{23}$$

If the NP algorithm starts in a region $\eta \in \Sigma_2$, it will move back to the initial region Θ with probability p_0 and move to one of subregions (still in Σ_2) with probability $1 - p_0$. Therefore, the expected number of iterations the algorithm stays in Σ_2 starting from a region $\eta \in \Sigma_2$ is $1/p_0$. Hence we have the following equation:

$$V = E(X) + \frac{1}{p_0} \tag{24}$$

If the NP algorithm starts in a region $\eta \in \Sigma_1$ and the depth of η $d(\eta) = k, 1 \leq k \leq d^*$, then one of three events will occur, backtracking to the the initial region Θ with probability p_1, moving one depth further to a region in Σ_1 with probability p_0 (i.e. a correct move), and moving one depth further to a region in Σ_2 with probability $1-p_0-p_1$. Recall we define p_1 as the probability of moving from $\eta \in \Sigma_1$ to Θ, $p_1 < 1 - p_0$. Hence we have the following conditional expectation equation:

$$E_k = p_0 E_{k+1} + p_1 E(X) + (1 - p_0 - p_1)V + 1 \qquad k = 1, 2, \ldots, d^* - 1 \text{ and } E_{d^*} = 1 \quad (25)$$

By induction from equation 25 we have E_1 as follow:

$$E_1 = \frac{p_1 E(X) + (1 - p_0 - p_1)V + 1}{1 - p_0} + p_0^{d^*-1}(1 - \frac{p_1 E(X) + (1 - p_0 - p_1)V + 1}{1 - p_0}) \quad (26)$$

Solving equations 23, 24 and 26 we can get equation 22. The proof is completed.

Still under the same assumptions, we state the following theorem for variance of *algorithm length.*

Theorem 2 *The variance of number of iterations until the NP algorithm considers the global optimum to be the most promising region can be obtained by solving a system of* $d^* + 1$ *equations*

proof: We use similar analysis as above to compute $E(X^2)$ first, then give an expression for $Var(X)$. Define $E_k^{(2)} = E(X^2|\text{from } \eta \in \Sigma_1 \text{and } d(\eta) = k)$, $k = 1, 2, \ldots, d_*$, and $V^{(2)} = E(X^2|\text{from } \eta \in \Sigma_2)$, then by conditioning on the first move we have:

$$E[(X-1)^2] = p_0 E_1^{(2)} + (1 - p_0)V^{(2)} \quad (27)$$

i.e.

$$E(X^2) = p_0 E_1^{(2)} + (1 - p_0)V^{(2)} + 2E(X) - 1 \quad (28)$$

By the same analysis as used to develop equation 24 and 26, we have the following two equations about $V^{(2)}$ and $E_k^{(2)}$:

$$V^{(2)} = E(X^2) + \frac{1}{p_0}(2V - 1) \tag{29}$$

$$E_k^{(2)} = p_0 E_(k{+}1)^{(2)} + p_1 E(X^2) + (1-p_0-p_1)V^{(2)} + 2E_k - 1 \qquad k = 1, 2, \ldots, d^*-1 \text{ and } E_{d^*}^{(2)} = 1 \tag{30}$$

There are $d^* + 1$ unknowns $E(X^2)$, $V^{(2)}$ and $E_k^{(2)}$ $k = 1, 2, \ldots, d^* - 1$, and $d^* + 1$ equations in 28, 29 and 30 . An unique solution can be found for $E(X^2)$ for some proper values of p_0, p_1 and d^*. The variance of the sample length $Var(X)$ can then be computed by the following formula:

$$Var(X) = E(X^2) - [E(X)]^2 \tag{31}$$

The proof is completed.

3.2.2 Stopping Criteria

Using the results obtained in the last section, we can construct a confidence interval for *algorithm length*. The following equation gives a 95% confidence interval for X (assume X has normal distribution):

$$CI(95\%) = [E(X) - 1.98\sqrt{Var[X]}, E(X) + 1.98\sqrt{Var[X]}] \tag{32}$$

We then can define the upper bound of the confidence interval

$$I_{max} = E(X) + 1.98\sqrt{Var[X]}; \tag{33}$$

as a good estimate of number of iterations for an NP application.

Success Probability (p_0)	Expectation	standard deviation	upper bound of 95% C.I.
0.5	3991	3983	11876
0.6	662	653	1955
0.7	157	149	452
0.8	50	42	133
0.9	21	11	40
0.99	11	0	11

Table 11: Data for algorithm length: $M = d^* = 10$

Success Probability (p_0)	Expectation	standard deviation	upper bound of 95% C.I.
0.5	4141873	4141854	12342743
0.6	111909	111891	333452
0.7	5819	5801	17303
0.8	520	503	1516
0.9	79	63	202
0.99	23	4	30

Table 12: Data for algorithm length: $M = d^* = 20$

Success Probability (p_0)	Expectation	standard deviation	upper bound of 95% C.I.
0.5	4.259176e+09	4.259176e+09	1.269234e+10
0.6	18621220	18621192	55491178
0.7	207817	207789	619238
0.8	4936	4910	14657
0.9	246	222	684
0.99	36	10	55

Table 13: Data for *algorithm length*: $M = d^* = 30$

Based on assumption **A4**, we can let $p_1 = (1 - p_0)/M$. We calculate the expectation, standard deviation and upper bound of 95% confidence interval of *algorithm length* by taking different input values of M, d^* and p_0. The results are shown in Table 11 - 13. We observed from the tables that:

- Variance of the NP algorithm length is very big. That explains why we need to construct a confidence interval to decide when to terminate the algorithm.

- For most problems, if the success probability p_0 is less than 0.5, the NP algorithm seems not very efficient. On the other hand, for a large range of success probability, the NP algorithm is very efficient. For example, when $M = d^* = 20$ and $p_0 = 0.7$, I_{max} equals to 17303. If we assume 10 samples from each region, then the expected total number of samples is $17303 \times 10 = 173030$. We can think this application corresponds to a problem with $20^{20} \approx 10^{26}$ feasible solutions. Hence, only one out of 10^{21} total points are expected to be evaluated to find a global optimum. If 0.01 seconds is needed to evaluate the objective function, the algorithm will stop in half an hour.

So far we have not explained how to calculate p_0. If we know where the true optimum is we can estimate p_0 with $\tilde{p_0}$ using the following formula:

$$\tilde{p_0} \approx \frac{1}{n}\sum_{i=1}^{n} z_i, \tag{34}$$

where n is the total number of iterations, z_i is a binary integer. $z_i = 1$ if the algorithm moves to a region containing at least one of global optima in the i-th iteration, and zero otherwise.

If the optima are not known, we can estimate p_0 from the run time behavior of the NP algorithm. We say that the algorithm changes direction if in the last iteration it moved to a subregion and in the current iteration it backtracks, or if in the last iteration it backtracked and in the current iteration it moves to a subregion. In $n-th$ iteration, we use the following formula to estimate p_0:

$$\tilde{p_0} \approx 1 - \frac{c_n}{n} \tag{35}$$

where c_n is the number of times the algorithm changed direction in first n iterations.

Note that the results for *algorithm length* are obtained under the assumptions that the NP algorithm backtracks to the initial feasible region. In many NP applications, we use other backtracking rules such as retreating to a superregion of the maximum depth region which contains the best solution found in the current iteration. In our experience, these rules usually work better. Therefore, I_{max} can be viewed as a "conservative" stopping criteria for NP applications.

3.2.3 Hybrid NP/GA Algorithm

We will combine the NP method with the GA as follows. First recall that when applying GA each solution is assumed to have a given fitness value, and the GA search starts with an initial population and imitates natural selection to improve the fitness of the population with the overall goal of finding the fittest solution. Therefore, we let the promising index of a region be the fitness value of the fittest solution in the region and we use GA search to estimate the promising index, that is, to seek the fittest solution in each region. The random sampling step then becomes equivalent to obtaining an initial population for the GA search and the final population is used for the promising index value. The partitioning and backtracking steps remain unchanged. This is a natural combination of the two methods and the resulting hybrid algorithm retains the global perspective of the NP method and the powerful local search capabilities of the GA.

We now describe the *hybrid NP/GA algorithm* in detail. In the k-th iteration we assume that there is a subregion $\sigma(k) \subseteq \Theta$ of the feasible region that is considered to be the *fittest*. Initially we assume that nothing is known about the fittest region so $\sigma(0) = \Theta$. The first step is to *partition* the fittest region into M disjoint subregions and aggregate the surrounding region $\Theta \setminus \sigma(k)$ into one region. Hence, in each iteration we consider a partitioning of the entire feasible region. The second step is to use some randomized method to obtain an initial population of solutions or chromosomes from each region. This should be done in such a way that each chromosome has a positive probability of being selected in the initial population. The third step is to apply the GA search to each initial population individually. This search should be constrained to stay within the region where the initial population was obtained. The forth step is to calculate a summary statistic of the final population in each region and use that as a measure of the *overall fitness* of the region. This summary statistic is usually the performance of the fittest chromosome in the final population. The fifth and final step is to determine the fittest region for the next iteration. If one of the subregions is estimated to have the best overall fitness, this region becomes the fittest region in the next iteration. The new fittest region is thus nested within the last. On the other hand, if the surrounding region is found to have the best overall fitness, the algorithm *backtracks* to a larger region that contains the fittest chromosome in the surrounding region. The partitioning continues until singleton regions are obtained and no further partitioning is possible.

A potential drawback to the pure NP method is that it may get stuck for considerable time at good but suboptimal solutions at maximum depth. In other words if $\sigma(k)$ is a singleton region that corresponds to a solution that is better than most but not all of the other solutions, it may take many iterations before the NP algorithm backtracks from this region. In the hybrid NP/GA algorithm we avoid this problem as follows. We note that once maximum depth is reached the surrounding region contains almost all of the feasible points. Furthermore, in the first iteration the entire feasible region is sampled with equal intensity. In this step, after applying GA search to each subregion we can therefore take the best chromosome from each region and form a set of M high quality chromosomes that, since they all come from different regions, are assured to have diverse genetic material. These chromosomes can then be reproduced into the initial population whenever the surrounding region of a maximum depth region is sampled. This helps the algorithm to backtrack from suboptimal maximum depth regions.

3.2.3.1 NP/GA algorithm

To state the hybrid NP/GA algorithm more precisely we need a few definitions. First we need to define the space of subregions on which the algorithm moves.

Definition 1 *A region constructed using the partitioning scheme described above is called a* valid region *given the fixed partition. The collection of all valid regions forms a space of subsets, which we denote by* Σ. *Singleton regions are of special interest and we let* $\Sigma_0 \subset \Sigma$ *denote the collection of all such valid regions.*

Thus the hybrid NP/GA algorithm moves from one element on Σ to another, eventually aiming at locating a singleton element, that is, an element in Σ_0 that contains a global optimum. To keep track of the status of the algorithm it is convenient to define the regions by their depth in the partitioning tree and by how they are obtained by partitioning regions of one less depth.

Definition 2 *We call the singleton regions in* Σ_0 *regions of* maximum depth, *and more generally, talk about the* depth *of any region. The depth function,* $d : \Sigma \to \mathbf{R}$, *is defined* *iteratively in the obvious manner, with* Θ *having depth zero and so forth. We assume that the maximum depth is uniform, that is,* $d^* = d(\sigma)$ *for all* $\sigma \in \Sigma_0$.

Definition 3 *If a valid region* $\sigma \in \Sigma$ *is formed by partitioning a valid region* $\eta \in \Sigma$*, then* σ *is called a* subregion *of region* η*, and region* η *is called a* superregion *of region* $\sigma \in \Sigma$.

Using the notation and definitions above the hybrid NP/GA algorithm is given below.

Algorithm *NP/GA*

Step 0 **Initialization.** Set $k = 0$ and $\sigma(k) = \Theta$.

Step 1 **Partitioning.** If $d(\sigma(k)) \neq d^*$, that is, $\sigma(k) \notin \Sigma_0$, partition the fittest region, $\sigma(k)$, into $M_{\sigma(k)}$ subregions $\sigma_1(k), ..., \sigma_{M_{\sigma(k)}}(k)$. If $d(\sigma(k)) = d^*$ then let $M_{\sigma(k)} = 1$ and $\sigma_1(k) = \sigma(k)$.

If $d(\sigma(k)) \neq 0$, that is, $\sigma(k) \neq \Theta$, aggregate the surrounding region $\Theta \setminus \sigma(k)$ into one region $\sigma_{M_{\sigma(k)}+1}(k)$.

Step 2 **Initial Population.** If $d(\sigma(k)) \neq d^*$ use a randomized method to obtain an initial population of N_j strings from each of the regions $\sigma_j(k),\ j = 1, 2, ..., M_{\sigma(k)} + 1$,

$$POP_I^j = \left[\theta_I^{j1}, \theta_I^{j2}, ..., \theta_I^{jN_j}\right], \quad j = 1, 2, ..., M_{\sigma(k)} + 1. \tag{36}$$

If $d(\sigma(k)) = d^*$ then obtain a population $POP_{I_0}^j$ of $N_j - M$ strings as above and let the initial population be $POP_I^j = POP_{I_0}^j \cup POP_0$, where POP_0 is the diverse high quality population found in the first iteration (see Step 6). Note that we implicitly assume that $N_j > M$.

Step 3 **GA Search.** Apply the GA to each initial population POP_I^j individually, obtaining a final population for each region $\sigma_j(k),\ j = 1, 2, ..., M_{\sigma(k)} + 1$,

$$POP_F^j = \left[\theta_F^{j1}, \theta_F^{j2}, ..., \theta_F^{jN_j}\right], \quad j = 1, 2, ..., M_{\sigma(k)} + 1. \tag{37}$$

If $k = 0$ then go to Step 6, otherwise go to Step 4.

Step 4 **Overall Fitness.** Estimate the *overall fitness* of the region by the performance of the fittest chromosome in the final population. That is, the overall fitness of each region is estimated by

$$\hat{F}(\sigma_j) = \max_{i \in \{1,2,...,N_j\}} f(\theta_F^{ji}), \quad j = 1, 2, ..., M_{\sigma(k)} + 1. \tag{38}$$

Step 5 **Update Fittest Region.** Calculate the index of the region with the best overall fitness,

$$\hat{j}_k \in \arg \max_{j \in \{1,2,..,M_{\sigma(k)}+1\}} \hat{F}(\sigma_j). \tag{39}$$

If more than one region is equally fit, the tie can be broken arbitrarily, except at maximum depth where ties are broken by staying in the current fittest region. If this index corresponds to a region that is a subregion of $\sigma(k)$, then let this be the fittest region in the next iteration. That is $\sigma(k+1) = \sigma_{\hat{j}_k}(k)$ if $\hat{j}_k \leq M_{\sigma(k)}$. If the index corresponds to the surrounding region, backtrack to a region $\eta \in \Sigma$ that is defined by containing the fittest chromosome in the surrounding region and being of depth Δ less than the current most promising region. Note that this fittest chromosome θ_{fit} is known as the argument where the minimum (38) is realized for $j = M_{\sigma(k)} + 1$. In other words, $\sigma(k+1) = \eta$ where

$$d(\eta) = d(\sigma(k)) - \Delta, \tag{40}$$

and

$$\theta_{fit} \in \eta. \tag{41}$$

This uniquely defines the next most promising region.

Let $k = k + 1$. Go back to Step 1.

Step 6 **Initial Diverse Population.** Let $\theta^{j}_{\hat{i}_j}$ be the fittest chromosome from the j-th region

$$\hat{i}_j = \arg \max_{i \in \{1,2,...,N_j\}} \theta_F^{ji}, \tag{42}$$

for $j = 1, 2, ..., M$. Let POP_0 be the set of the fittest chromosome from each region

$$POP_0 = \left[\theta_F^{1\hat{i}_1}, ..., \theta_F^{M\hat{i}_M}\right]. \tag{43}$$

Go back to Step 4.

By Step 1 of the algorithm partitioning is performed if the current most promising region is not of maximum depth. In practice we usually stop partitioning $\delta > 0$ depth levels above the maximum depth, that is, partition if and only if $d(\sigma(k)) < d^* - \delta$. The reason for this is that close to the maximum depth the number of solutions in each subregion is small and can be explored efficiently. We also note that due to its iterative nature, some chromosomes may be generated more than once in Step 2 of the algorithm.

3.2.3.2 Global Convergence

We have presented global convergence for the pure NP algorithm in the last section. We state the convergency results for the NP/GA algorithm in the following therom.

Theorem 3 *The hybrid NP/GA algorithm converges to a global optimum in finite time. That is, let $\mathcal{S}$ denote the set of strings that are globally optimal. Then with probability one there exists $\theta \in \mathcal{S}$ and $k^* < \infty$ such that $\sigma(k) = \{\theta\}$, for all $k \geq k^*$.*

Proof: Due to the random method of obtaining initial populations, and the randomness involved in the GA search, the sequence of the fittest regions $\{\sigma(k)\}_{k=0}^{\infty}$ is a stochastic process. Furthermore, since the fittest region in any iteration only depends on which region was the fittest in the last iteration, $\{\sigma(k)\}_{k=0}^{\infty}$ is a Markov chain. It is also clear that for any $\theta \in \mathcal{S}$, if $\sigma(k_1) = \{\theta\} \in \Sigma_0$ for some $k_1 > 0$, then since $\sigma(k_1) = \{\theta\} \in \Sigma_0 \in \mathcal{S}$ is a global optimum and ties at the maximum depth are broken by favoring the current fittest region, $\sigma(k) = \{\theta\}$ for all $k \geq k_1$. Hence $\{\theta\} \in \Sigma_0$ is an absorbing state. Furthermore, if $\theta \in \Theta \setminus \mathcal{S}$ and $\sigma(k) = \{\theta\}$, then in the next iteration there is a positive probability that the final population in the surrounding region contains a string from $\mathcal{S}$ and the algorithm backtracks. Therefore, θ is not absorbing. Hence, the strings in $\mathcal{S}$ correspond to the only absorbing states of the Markov chain and the chain will eventually be absorbed in one of these states with probability one.

Finally, since all the non-absorbing states are transient each can only be visited a finite number of times; and since there is a finite number of transient states the absorbing time must be finite. Therefore, with probability one there exists $\theta \in \mathcal{S}$ and $k^* < \infty$ such that $\sigma(k) = \{\theta\}$, for all $k \geq k^*$.

3.2.3.3 Convergency Speed

In this section we compare convergency speed of the pure NP algorithm and the NP/GA algorithm, and test the stopping criteria proposed in this Chapter. We select the product design problems [30, 31, 32] as our test problems. We shall discuss this class of problems in more detail in the next Chapter. We select three small problems (P1, P2 and P3), the optimal solutions of which can be found through exhaustive search.

Both the pure NP algorithm and the NP/GA algorithm use $N = 20$ sample points from each region We let both algorithms backtrack to the initial region because the stopping criteria is developed for this backtracking rule. Since the optima for the test problems are known, we can use equation 34 to estimate p_0. For each of algorithms we ran for 10 iterations. Table 14 - 16 show the estimated success probability $\tilde{p_0}$, the observed number of iterations X,and 95% confidence interval of X under this estimated

Iteration	NP-pure			NP/GA		
	$\tilde{p_0}$	X	95% C.I.	$\tilde{p_0}$	X	95% C.I.
1	0.074	162	(5,19277392)	1.000	5	(5,5)
2	0.089	269	(5,6520873)	1.000	5	(5,5)
3	0.058	119	(5,80467612)	1.000	5	(5,5)
4	0.077	168	(5,15768159)	1.000	5	(5,5)
5	1.000	5	(5,5)	0.875	7	(5,17)
6	1.000	5	(5,5)	1.000	5	(5,5)
7	0.039	542	(5,872542648)	1.000	5	(5,5)
8	0.065	614	(5,42676216)	0.875	7	(5,17)
9	0.037	3551	(5,1218469132)	1.000	5	(5, 5)
10	0.106	112	(5,2247070)	1.000	5	(5,5)

Table 14: Stopping criteria results for product design problem P1.

probability. If the lower bound of the confidence interval is negative, we set it to d^* since it is the minimum number of iterations required for the NP applications to find an optimum solution. We compare the estimate of success probability $\tilde{p_0}$ for the pure NP and the NP/GA algorithms in Table 17.

Our numerical results show that in each iteration both algorithms were absorbed to the optimal region before the stopping criteria applied , and each of the observed value of X falls into its 95% confidence interval. The observed value of X is much smaller than I_{max}, the upper bound of 95% confidence interval of X. This can be explained by the large variance of X. In order to get more tight confidence interval for X, we might need to consider p_0 as a function of depth d and refine the results we obtained in this Chapter. This issue is left as our future research topic.

From Table 17 we observed that NP/GA has much larger mean and lower variance for estimated success probability. The results indicate that the local search capacity of GA increases the chance for the NP algorithm to move in the correct direction.

Iteration	NP-pure			NP/GA		
	$\tilde{p_0}$	X	95% C.I.	$\tilde{p_0}$	X	95% C.I.
1	0.706	16	(6,89)	1.000	6	(6,6)
2	0.015	1997	(6, 1.78e+13)	0.889	8	(6,21)
3	0.022	696	(6, 1.21e+12)	0.800	9	(6,41)
4	0.048	788	(6, 5.23e+09)	0.889	8	(6,21)
5	0.031	260	(6, 1.01e+11)	1.000	6	(6,6)
6	0.189	52	(6, 426824)	1.000	6	(6,6)
7	0.224	75	(6, 135468)	1.000	6	(6,6)
8	0.021	1502	(6,1.68e+12)	1.000	6	(6,6)
9	0.156	134	(6, 1574478)	1.000	6	(6,6)
10	0.542	23	(6,433)	0.733	14	(6,71)

Table 15: Stopping criteria results for product design problem P2.

Iteration	NP-pure			NP/GA		
	$\tilde{p_0}$	X	95% C.I.	$\tilde{p_0}$	X	95% C.I.
1	0.667	20	(7,195)	0.588	16	(7,465)
2	1.000	7	(7, 7)	1.000	7	(7,7)
3	0.094	993	(7, 7542537361)	0.442	94	(7,3570)
4	0.070	1188	(7, 5.45e+09)	1.000	7	(7,7)
5	0.043	2112	(7, 2.66e+11)	0.548	30	(7,762)
6	0.117	299	(7, 96565818)	1.000	7	(7,7)
7	0.737	18	(7, 99	0.650	19	(7,234)
8	0.392	50	(7,8650)	0.632	18	(7,284)
9	0.041	2298	(7, 3.91e+11)	0.545	21	(7,792)
10	0.041	1307	(7,3.91e+11)	0.733	14	(7,104)

Table 16: Stopping criteria results for product design problem P3.

Algorithm	P1		P2		P3	
	Avg of $\tilde{p_0}$	Std of $\tilde{p_0}$	Avg of $\tilde{p_0}$	Std of $\tilde{p_0}$	Avg of $\tilde{p_0}$	Std of $\tilde{p_0}$
Generic NP	0.255	0.394	0.195	0.242	0.320	0.357
NP/GA	0.975	0.050	0.931	0.094	0.714	0.201

Table 17: Comparison of NP-pure and NP/GA: estimation of p_0

Chapter 4

Application to Product Design Problem

Product design optimization problems have received considerable attention in the literature. These problems may be divided into single product design problems where the objective is to design the attributes of a single product [43, 3], and product line design where multiple products are offered simultaneously [31, 44, 16, 50]. The product design problem involves determining the levels of the attributes of a new or redesigned product in such a way that it maximizes a given objective function. We assume that the part-worths preferences of individual customers, or market segment, have been elicited for each level of every attribute, for example using conjoint or hybrid conjoint analysis [66, 35]. These part-worths preferences are assumed to be independent, but by a simple modification of the objective function dependencies can also be accounted for in the new framework. Furthermore, we assume that all product designs found by combining different levels of each attribute are technologically and economically feasible, although the optimization framework here can be easily adapted to handle infeasible designs. We also assume that a customer will choose the offered product if its utility is higher than that of a competing status quo product, which may be different for each customer. This problem is usually referred to as the share-of-choices problem. On the other hand, if the objective is to maximize the total utility of the customers or the firm's marginal return, the problem is called the buyers' welfare problem, or seller's welfare problem, respectively.

The product share-of-choices problem is very difficult to solve, especially as the product complexity increases and more attributes are introduced. In fact, it belongs to the class of NP-hard problems, for which efficient exact solution procedures do not exist [43]. This implies that for realistically sized problems exact solution methodologies might not be feasible. Several heuristic method have therefore been suggested in the literature. These methods are generally sufficiently fast to be applicable in practice, but being heuristics they do not guarantee optimality and the product designs found by these methods may sometimes be substantially suboptimal. In this Chapter we introduce a new optimization framework (include the NP/GA algorithm) for product design and present empirical results for numerical examples that are significantly larger than those previously reported in the literature. These numerical examples indicate that the new methodology performs well relative to earlier methods with respect to both speed and solution quality, especially for very large optimization problems, which correspond to complex product designs. We also introduce the mixed integer solution for single product design problems. We incorporated the NP/GA algorithm and other problem specific heuristics into the FATCOP's branch-and-bound framework and solved some reasonable size problems to optimality.

The remainder of this chapter is organized as follows. In Section 4.1 we state the mixed integer formulation of the product design problems. We review the previous solution methodologies in Section 4.2. In section 4.3, we introduce the new optimization framework and show how to incorporate earlier heuristics into the methodology. To illustrate the new framework, in Section 4.4 we analyze a simple design problem in detail. In Section 4.5 numerical results are presented to demonstrate the performance of the new method and a similar approach is applied to product line design problem. We illustrate how users can supply problem specific heuristics to FATCOP and report some successfully solved instances in section 4.6.

4.1 Mixed integer programming formulation

Mathematically, the single product design problem can be stated as follows. A product has K attributes of interest, each of which can be set to L_k levels, $k = 1, 2, .., K$. When the level of each attribute has been determined a complete product profile has been found. If only some of the attributes have been determined, we call that a partial product profile. There are N customers, or market segments, and each has given utilities for every level of each attribute, and the problem is to select levels for each attribute to satisfy the producers objective. Each customer $i \in \{1, 2, ..., N\}$ is assumed to have a known utility u_{ikl} associated with level l of the k-th attribute, $1 \leq k \leq K$, $1 \leq l \leq L_k$. Given these utilities, the objective may be to maximize market share, profit, or customer utility. In this thesis we will concentrate on the problem where the objective is to maximize market share, namely the *share-of-choices* problem [43]. The other two objectives, although also important, are not relevant to the treatment here as the single product design problem is easily solved for those objectives [10]. We consider a status-quo brand for each buyer. Let l_k^* denote the level of attribute k that appears in the product profile of the status-quo brand for buyer i. Let $c_{ikl} = u_{ikl} - u_{ikl_k^*}$ denote the part worth of level l of attribute k relative to the part worth of level l_k^* of attribute k for consumer i. A buyer selects a product profile over status-quo only if its relative part worth utility is strictly positive.

Let each consumer's interval-scaled part-worths be normalized to sum to 1. It follows that each product profile has part-worths utility no larger than 1 and a relative part-worths utility that lies between -1 and 1. To formulate the share-of-choices problem we define the decision variables x_{kl},

$$x_{kl} = \begin{cases} 1, & \text{if attribute k is set to level l,} \\ 0, & otherwise. \end{cases}$$

and auxiliary variable z_i,

$$z_i = \begin{cases} 0, & \text{if customer i accepts the selected product,} \\ 1, & otherwise. \end{cases}$$

where $i = 1, 2, \ldots, N$.

Then we can formulate the problem as the following MIP:

$$\min \sum_i z_i \tag{44}$$

subject to:

$$\sum_{l=1}^{L_k} x_{kl} = 1, \;\; k = 1, 2, ..., K \tag{45}$$

$$\sum_{k=1}^{K} \sum_{l=1}^{L_k} c_{ikl} \cdot x_{kl} + K \cdot z_i \geq 0, \;\; i = 1, 2, ..., N \tag{46}$$

Constraint 45 requires that only one level of an attribute be associated with a product. Constraint 46 restricts z_i to be 1 only if the product profile provides customer i no higher utility than the status quo. The objective 44 is to minimize the number of customers who reject the new product profile.

The product line design problem differs from the single product design problem by selecting a line of products profiles. The objective then is to maximize the number of consumers who will buy *at least* one product profile in the selected product line composition. A MIP formulation for this problem is given in [44].

It will be convenient to write a product profile θ in terms of its K attributes as $\theta = [l_1 \; l_2 \; \ldots \; l_K]$, $1 \leq l_k \leq L_k$, indicating which level is chosen for attribute k, $1 \leq k \leq K$.

The share-of-choices problem (both single product design and product line design) has been shown to be NP-hard [43], and is therefore usually solved using some heuristic method [31, 16].

4.2 Previous Solution Methods

Several heuristic solution methodologies have been proposed for the share-of-choices problem as well as other product design problems. The greedy search (GS) heuristic and dynamic programming (DP) heuristics are applied in [43] for the share-of-choices problem, and Balakrishnan and Jacob [3] apply a genetic algorithm (GA) approach for this problem. Divide-and-Conquer (DC) is applied in [30]. Finally, in [50], a Beam Search (BS) heuristic is proposed for the product line selection problem.

Variants of the GS heuristic have been used for the buyers' and seller's problem [31] and for the share-of choices problem [43]. In this thesis we define this heuristic as follows. For each attribute k, the overall relative utilities for all the customers is calculated, that is

$$u_{kl} = \sum_{i=1}^{M} u_{ikl} - u_{ik\tilde{l}_k},$$

where $\tilde{l}_k$ is the level of the k-th attribute for the status-quo product. The GS heuristic then selects the level l_k^* for the k-th attribute that maximizes the relative utilities. Thus the level of each attribute is determined independently of all other attributes. A detailed algorithm for the GS heuristic is given in Appendix A.

The DP heuristic has been used for a variety of product design problems [42, 43, 44]. This heuristic treats attributes as stages in a dynamic program and the levels as states. Thus similarly to the GS heuristic, in each iteration the DP heuristic sets one attribute to its 'best' level and does not consider all the possible interactions between attributes.

For example, at the first stage the best level of the first attribute is determined for every level of the second attribute. This generates L_2 partial product profiles. At the next stage, L_3 partial profiles are generated by, for each level of the third attribute, finding the best combination of levels for the first and second attribute from the previous stage. This continues until all the attributes have been considered. At this point L_K partial product profiles have been identified, and we can select the best one from this set. A detailed algorithm for the DP heuristic is given in Appendix B.

The DC heuristic was proposed for single product design and sequential product lines design [33]. Variants of the DC were later employed by SIMOPT, a conjoint analysis oriented, product positioning model [35]. One version of the DC method can be implemented as follows. Suppose there are K attributes. The algorithm starts with $T(\leq K)$ attributes and finds their best combination through complete enumeration, conditioned on randomly fixed levels for the remaining K-T attributes. The next T attributes are then conditionally optimized. The algorithm continues in this manner until all attributes have been completed. It then starts over based on current levels of all attributes, and continue until there is no change in the objective function.

In [3] a GA approach is proposed for the single product design problem. The GA is a random search method based on the concept of natural selection. It starts from an initial population and then uses a mixture of reproduction, crossover, and mutation to create new, and hopefully better, populations. The GA usually works with an encoded feasible region, where each feasible point is represented as a string, which is commonly referred to as a chromosome. Each chromosome consists of a number of smaller strings called genes. The reproduction involves selecting the best, or fittest, chromosomes from the current population, the crossover involves generating new chromosomes from this selection, and finally mutation is used to increase variety by making small changes to the genes of a given chromosome. The GA was originally proposed by Holland [38] and has been found to be very effective for a variety of combinatorial optimization problems [27]. A detailed algorithm for the GA method is given in Appendix C.

While many algorithms for single product design were proposed, there are only a few algorithms for product line design. Kohli and Sukumar [43] developed DP based heuristics for three standard formulations: *share-of-choice*, *buyer's welfare*, and *seller's return*. Nair, Thakur, and Wen [50] presented Beam Search (BS) based heuristics for these problems in 1995. They showed that beam search approaches perform better than the DP based heuristics in all important performance measures.

Beam search originated from the Artificial Intelligence area. Simply put, it is a breadth-first search with no backtracking. However at any level of the process, the breadth is limited to the most promising b nodes, where b is called beam width. BS heuristics for product line problem begin with K part-worths matrices, and systematically prune unpromising attributes levels by iteratively combining pairs of work matrices, and then selecting the b most promising combinations of levels. Thus we will now have at most half the number of matrices we started with. The algorithm continue this procedure until there only one work matrix remaining. We, then, have b product profiles for the products in b product lines. For the second product in the product line, we reduce the original data set to a smaller one by removing rows corresponding to customers who decide to buy the first product. Similarly, we can design the remaining $b-1$ product lines.

4.3 Description of the Optimization Framework

All of the product design problems mentioned above, including the buyers' welfare problem, seller's welfare problem, and the share-of-choices problem for designing both single products and product lines, can be formulated as an optimization problem where the goal is to maximize an objective function, $f : \Theta \to \mathbf{R}$, over a finite feasible region Θ containing all the possible product profiles, that is,

$$\max_{\theta \in \Theta} f(\theta). \tag{47}$$

Here we describe a NP framework for solving any such problem that is capable of incorporating all of the previously suggested heuristic approaches, including greedy search, dynamic programming, and genetic algorithms. As stated before, we focus on the share-of-choices problem. However, this framework can be extended to the product line problem and arbitrary objective functions that incorporate interaction effects such as cannibalization.

4.3.1 A Generic NP Algorithm for Product Design

We now describe a generic implementation of the NP algorithm for the product design problems in detail. Some of this material has appeared in Chapter 3, where the NP method is given.

Partitioning: At each iteration the current most promising region is partitioned into L_{k^*} subregions by fixing the k^*-th attribute to one of its L_{k^*} possible levels. This implies that the maximum depth is $d^* = K$. The sequence in which the attributes are selected to be fixed is arbitrary, but may affect the performance of the algorithm. Intuitively the 'most critical' attributes should be fixed first, so if the attributes can be ranked in order of importance this may be incorporated into the partitioning strategy. Here the 'most critical' means the attribute with the largest effect on the performance function. Recall that we can write a product profile $\theta \in \Theta$ in terms of its K attributes as $\theta = [l_1 \ l_2 \ \ldots \ l_K]$, and a region $\sigma \in \Sigma$ of depth $d(\sigma)$ is therefore determined by $\sigma = \{\theta = [l_1 \ l_2 \ \ldots \ l_K] \in \Theta : l_i = l_i^\sigma, \ i = 1, 2, ..., d(\sigma)\}$, where l_i is a variable used to denote the level of the i-th attribute in the product profile θ, and l_i^σ is a fixed value for this variable for all the attributes that are fixed in the region, $1 \leq i \leq K$.

In general, the partitioning step involves partitioning a region $\sigma \in \Sigma$ into $L_{d(\sigma)+1}$ subregions,

$$\sigma_j = \left\{\theta = [l_1, l_2, \ldots, l_K] \in \Theta : l_k = l_k^\sigma, \ k = 1, 2, ..., d(\sigma), l_{d(\sigma)+1} = j\right\}, \ j = 1, 2, ..., L_{d(\sigma)+1},$$

and this can be done using the following algorithm.

Algorithm *Partitioning*

Step 1 Attributes that are fixed in σ are fixed to the same levels for each of the subregions $\{\sigma_j\}_{j=1}^{L_{d(\sigma)+1}}$. That is, let $l_k = l_k^\sigma$, for $k = 1, 2, ..., d(\sigma)$.

Step 2 Fix the next attribute of each subregion to the corresponding level

$$l_{d(\sigma)+1} = j, \ j = 1, 2, ..., L_{d(\sigma)+1}, \tag{48}$$

We now turn our attention to developing an efficient sampling procedure for Step 2 of the NP algorithm.

Random Sampling: The set of $N_{\sigma_j(k)}$ sample product profiles from a subregion $\sigma_j \in \Sigma$ can be selected by *uniform* or *weighted* sampling of the region. We note that σ_j is defined by $d(\sigma_j)$ of the K attributes being fixed to levels $\left[l_1^{\sigma_j} \ l_2^{\sigma_j} \ ... \ l_{d(\sigma_j)}^{\sigma_j}\right]$, that is, by a partial product profile

$$\sigma_j = \left\{\theta = [l_1 \ l_2 \ ... \ l_K] \in \Theta : l_k = l_k^{\sigma_j}, \ k = 1, 2, ..., d(\sigma_j)\right\}.$$

Hence, the sampling procedure should select levels for the remaining $K - d(\sigma_j)$ attributes. By uniform sampling we mean that each of the L_k possible levels of the k-th attribute has equal probability of being selected. A procedure for generating a uniform sample product profile $\theta^s = [l_1^s \; l_2^s \; ... \; l_K^s]$ from a region $\sigma_j \in \Sigma \setminus \Sigma_0$ is therefore given by the algorithm below. The output of this algorithm is a single product profile θ^s, so it should be repeated $N_{\sigma_j(k)}$ times.

Algorithm *Uniform Sampling*

Step 0 For the attributes that are fixed by the partitioning let $l_i^s = l_i^{\sigma_j}$, for all $i \leq d(\sigma_j)$. Set $k = d(\sigma_j) + 1$.

Step 1 Generate a uniform random variable u from $\mathcal{U}(0,1)$.

Step 2 If $\frac{i-1}{L_k} \leq u < \frac{i}{L_k}$ then set the k-th attribute to level i, $i = 1, 2, ..., L_k$, $l_k^s = i$.

Step 3 If $k = K$ stop, otherwise let $k = k + 1$ and go back to Step 1.

Here $\mathcal{U}(0,1)$ denotes the uniform distribution on the unit interval.

In Section 3.2 and Section 3.3 below, heuristic methods are used to derive weighted sampling strategies that bias the sampling towards good product profiles.

Estimating the promising index: Given the sample profile in k-th iteration, we estimate the promising index function $I : \Sigma \rightarrow \mathbf{R}$, for each region. Here we use the following estimate

$$\hat{I}(\sigma_j(k)) = f\left(H_{\sigma_j(k)}\left(\theta^j\right)\right) \tag{49}$$

where for each $\sigma \in \Sigma$, $H_\sigma : \Theta^{N_\sigma} \rightarrow \sigma$ is a function that transforms the set of sample profiles into a single profile representing the region. In its simplest form it could for example simply be the best profile, that is,

$$H_{\sigma^j(k)}\left(\theta^j\right) = \arg\max\left\{f\left(\theta^{j1}\right), f\left(\theta^{j2}\right), ..., f\left(\theta^{jN^{\sigma_j(k)}}\right)\right\}, \tag{50}$$

for all $j = 1, 2, ..., M$. However, as will become clear later, the function H, and thus the estimation of the promising index can also be used to incorporate local search heuristics into the NP framework.

Backtracking: If backtracking is selected in Step 4 of the k-th iteration of the NP algorithm, the algorithm moves to a larger region containing the current most promising region. Several options exist, but here we backtrack to a region $\eta \in \Sigma$ that is defined by containing the best product profile in the surrounding region and being of depth Δ less than the current most promising region. Note that this best product profile θ^{best} is known as the argument where the minimum (49) is realized for $j = M$. In other words, $\sigma(k+1) = \eta$ where $d(\eta) = d(\sigma(k)) - \Delta$, and $\theta^{best} \in \eta$. This uniquely defines the next most promising region.

4.3.2 The NP/GS Algorithm

The greedy search (GS) heuristic can be incorporated into the NP framework by using it to bias the sampling distribution used in Step 2 of the NP algorithm. Recall that according to the greedy heuristic, each attribute k is set to the level l_k^* that maximizes the overall relative utility

$$l_k^* = \arg \max_{l=1,\dots,L_k} u_{kl} = \sum_{i=1}^{M} \left(u_{ikl} - u_{ik\tilde{l}_k} \right). \tag{51}$$

Thus a GS sampling algorithm could select the level l_k^* that satisfies equation (51) with a given probability w_1, and select any other level with uniform probability $\frac{1-w_1}{L_k-1}$. This ensures that all product profiles are selected with positive probability, and product profiles that are 'close' to the profile generated by pure GS are selected with higher probability. This procedure is stated exactly in Appendix A. We will refer to the NP algorithm that uses GS Sampling to randomly sample from each region as the NP/GS algorithm.

4.3.3 The NP/DP Algorithm

As with the GS heuristic the dynamic programming (DP) heuristic can be incorporated in the sampling step. As before, the objective is to bias the sampling distribution towards product profiles that are heuristically good. In the GS Sampling procedure described in the last section, each *attribute* of a sample product profile was selected either greedily or uniformly, that is, the attribute selection was randomized. Here, on the other hand, we use a different approach: each sample *product profile* is selected according to a randomized DP heuristic with a given probability, and uniformly with a given probability. Thus the randomization is applied to entire product profiles rather than individual attributes. Furthermore, if the profile is to be selected using the DP heuristic then it is randomized by selecting a random order in which the DP heuristic fixed the attributes. In general, different orders of attributes leads to different product profiles. This procedure selects any product profile with a positive probability, but selects profiles found by applying the DP heuristic to some order of the attributes with a higher probability. The details of this procedure are described in Appendix B. We will refer to the NP algorithm that uses DP Sampling to randomly sample from each region as the NP/DP algorithm.

4.3.4 The NP/GA Algorithm

We describe the NP/GA algorithm for the product design problems as follows. In Step 3 of the NP algorithm, once an initial population POP_I^j has been obtained from a subregion σ_j, then the GA search proceeds as follows. First the $\frac{N_j}{2}$ fittest product profiles are selected for reproduction from the population. These profiles will also survive intact in the next population. Secondly, pairs of product profiles are selected randomly from the reproduced profiles. Each attribute, that is not fixed in σ_j, of the selected pair has a given probability of participating in a crossover, that is, being swapped with the corresponding attribute of the other product profile. Finally, each profile may be selected with a given probability as a candidate for mutation. If a product profile is mutated, an attribute from that profile is selected at random and assigned a random level. As before, only attributes that are not fixed in σ_j may be selected to be mutated. The algorithm in Appendix C gives the exact procedure for the GA search in a region $\sigma_j \in \Sigma$.

4.3.5 The NP/GA/GS Algorithm

In this section we have described three methods of incorporating well known and effective heuristics into the NP framework. In addition to the resulting NP/GS, NP/DP, and NP/GA algorithms, it is also possible to incorporate more than one heuristic at a time into the NP optimization framework. This results in the NP/GA/GS and NP/GA/DP algorithms. The latter of these was found to have too much overhead to be efficient, however, promising numerical results for the NP/GA/GS algorithm are reported in the next section. This algorithm uses the GS Sampling algorithm for the random sampling in Step 2 of the generic NP algorithm, and the GA Search algorithm to estimate the promising index in Step 3 of the generic NP algorithm.

4.4 An Illustrative Example

To illustrate the NP optimization framework for product design, consider a small problem with $K = 3$ attributes, $L_1 = 3$, and $L_2 = L_3 = 2$. For some $N = 3$ individuals, let the part-worths data matrices $U(k)$ be given as follows, $k = 1, 2, 3$:

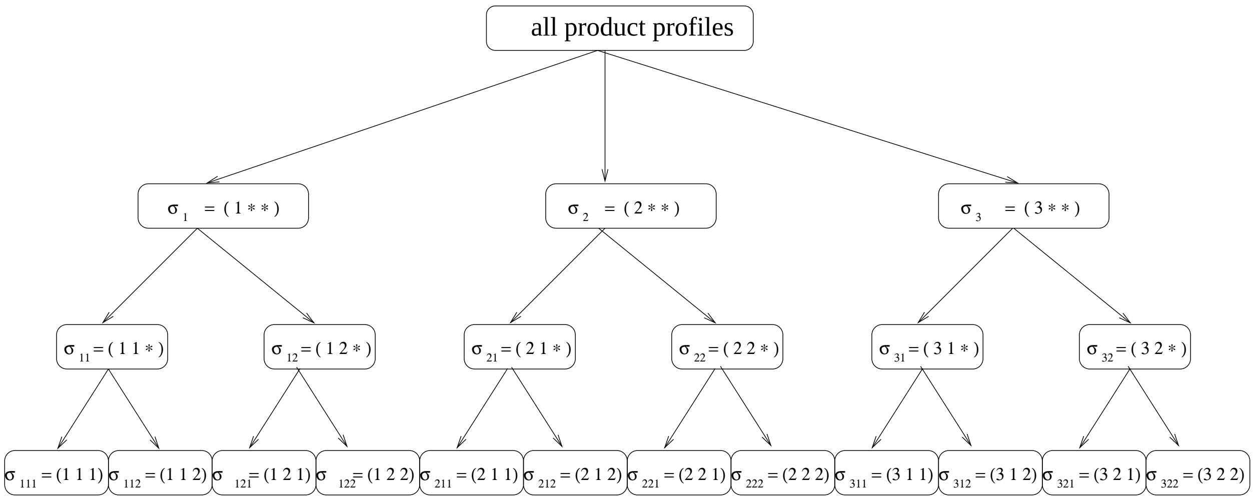

Figure 19: Partitioning tree for a simple product design example.

$$U(1) = \begin{bmatrix} 1 & 1 & 3 \\ 1 & 4 & 1 \\ 2 & 2 & 0 \end{bmatrix}, \qquad U(2) = \begin{bmatrix} 3 & 0 \\ 1 & 2 \\ 2 & 2 \end{bmatrix}, \qquad U(3) = \begin{bmatrix} 1 & 1 \\ 1 & 4 \\ 3 & 1 \end{bmatrix}.$$

Suppose the status-quo is represented by the values of level 1 for each attribute. Then subtracting the first column from each column of $U(k)$ gives relative parts-worth data matrices $\tilde{U}(k), k = 1, 2, 3$:

$$\tilde{U}(1) = \begin{bmatrix} 0 & 0 & 2 \\ 0 & 3 & 0 \\ 0 & 0 & -2 \end{bmatrix}, \qquad \tilde{U}(2) = \begin{bmatrix} 0 & -3 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}, \qquad \tilde{U}(3) = \begin{bmatrix} 0 & 0 \\ 0 & 3 \\ 0 & -2 \end{bmatrix}.$$

This allows us to calculate the performance of each design, that is, the expected number of customers selecting the product, according to the MIP model given in the beginning of this Chapter. We refer to this product design problem as the 3-Attribute example.

4.4.1 A Generic NP Algorithm for the 3-Attribute Example

Partitioning: The partitioning step can be demonstrated in 4.4. Here we assume attribute 1 is the first attribute to be determined, followed by attribute 2, and finally attribute 3. Changing the order would lead to a different partitioning, which in general affects the performance of the method. This example has 22 (including the entire feasible region) valid regions. The region containing all product profiles $\Theta \in \Sigma$ has three subregions σ_1, σ_2, and σ_3; and each of these subregions has two subregions of its own. The maximum depth is $d^* = 3$ and there are 12 regions of maximum depth, that is $|\Sigma_0| = 12$. Notice that these singleton regions define a complete product profile, whereas every region $\eta \in \Sigma$ such that $0 < d(\eta) < d^*$ defines a partial product profile. Also note that the maximum depth will always be equal to the number of attributes. The best solution value is initialized as $f_{best} = 0$.

Random Sampling: The generic NP algorithm starts from the initial most promising region, Θ (the entire region), and must determine which of its three subregions: $\sigma_1 = \{[l_1\ l_2\ l_3] : l_1 = 1, 1 \leq l_2 \leq 2, 1 \leq l_3 \leq 2\}$, $\sigma_2 = \{[l_1\ l_2\ l_3] : l_1 = 2, 1 \leq l_2 \leq 2, 1 \leq l_3 \leq 2\}$, and $\sigma_3 = \{[l_1\ l_2\ l_3] : l_1 = 3, 1 \leq l_2 \leq 2, 1 \leq l_3 \leq 2\}$, will become the most promising region in the next iteration. To determine this region, each of these subregions is sampled by selecting the values of l_2 and l_3. This involves two steps. In the first step we generate a uniform random variable $u \in (0, 1)$. If $u < \frac{1}{2}$ then we set $l_2 = 1$, and if $\frac{1}{2} \leq u < 1$ then we set $l_2 = 2$. In the second step we generate another uniform random variable $v \in (0, 1)$. If $v < \frac{1}{2}$ then we set $l_3 = 1$, if $\frac{1}{2} \leq v < 1$ then we set $l_3 = 2$. For example, when we sample in σ_1, if $u < \frac{1}{2}$ in the first step and $\frac{1}{2} \leq v < 1$ in the second step, then the sample product profile generated is $\theta^1 = [l_1^1\ l_2^1\ l_3^1] = [1\ 1\ 2]$. This procedure is identical for each of the three regions. Suppose the other two samples from σ_2 and σ_3

are $\theta^2 = [l_1^2\ l_2^2\ l_3^2] = [2\ 1\ 1]$ and $\theta^3 = [l_1^3\ l_2^3\ l_3^3] = [3\ 2\ 2]$, respectively.

It should be noted that though in this thesis we assume that all product design profiles constructed by combining different levels of each attribute are feasible, the NP framework can be easily adapted to handle infeasible designs. This can be done through the sampling procedure by only selecting feasible levels of each attribute.

Estimating the Promising Index: After we obtain one sample product profile from each region currently under consideration, the next step is to estimate the promising index of each region using the sample product profiles. Usually the number of samples in each region should be larger than one. However, for ease of exposition, in this example we use a single sample in each region and its objective value as estimated promising index:$\hat{I}(\sigma_1) = f(\theta^1) = f([1\ 1\ 2]) = 1$, $\hat{I}(\sigma_2) = f(\theta^2) = f([2\ 1\ 1]) = 1$, $\hat{I}(\sigma_3) = f(\theta^3) = f([3\ 2\ 2]) = 1$. By breaking the tie arbitrarily the algorithm moves to region $\sigma_1 = \{[l_1\ l_2\ l_3] : l_1 = 1, 1 \leq l_2 \leq 2, 1 \leq l_3 \leq 2\}$ and we update the best solution value to $f_{best} = 1$.

Backtracking: As σ_1 is now the most promising region, the algorithm then samples the subregions of σ_1,

$$\sigma_{11} = \{[l_1\ l_2\ l_3] : l_1 = 1, l_2 = 1, 1 \leq l_3 \leq 2\}$$

and

$$\sigma_{12} = \{[l_1\ l_2\ l_3] : l_1 = 1, l_2 = 2, 1 \leq l_3 \leq 2\},$$

as well as the surrounding region $\Theta \setminus \sigma_1$. Suppose the surrounding region $\Theta \setminus \sigma_1$ has the best estimation of promising index, the algorithm backtracks to a larger region. There are many ways to backtrack, but here we assume the algorithm backtracks to a region containing the best product profile found in this iteration and with depth of $\lfloor \frac{d^*}{2} \rfloor + 1$, where $\lfloor x \rfloor$ denotes the largest integer smaller than or equal to x, for any $x \in \mathbf{R}$. Therefore, the most promising region will be set to σ_{31}, and the best solution $f_{best} = 2$ is updated. It can be shown by enumeration that 2 customers is the optimal solution for this data set. The algorithm will continue in this manner until some predefined stopping criteria is met.

We notice from this example that, unlike the dynamic programming or beam search heuristics that only search the part of the feasible region considered promising, the NP algorithm takes the entire feasible region, that is all possible product designs, into consideration while focusing the computational effort on the promising part. Thus, as illustrated by this example, the algorithm may in the first iteration move to a region that does not contain an optimal product profile, but can then backtrack to a region containing the optimal product profile in the next iteration, since feasible product profiles in the surrounding region still have a chance of being selected and evaluated.

4.4.2 The NP/GS Algorithm for the 3-Attribute Example

The NP/GS algorithm differs from the generic NP algorithm in the random sampling step. For example, when sampling in the region

$$\sigma_3 = \{[l_1\ l_2\ l_3] : l_1 = 3, 1 \leq l_2 \leq 2, 1 \leq l_3 \leq 2\},$$

we first generate a uniform random variable u_0 from $\mathcal{U}(0,1)$. If $u_0 > w_1$, we generate another uniform random variable u_1 from $\mathcal{U}(0,1)$. If $u_1 < \frac{1}{2}$ then $l_2 = 1$, and if $\frac{1}{2} \leq u_1 < 1$ then $l_2 = 2$. Otherwise, we select a level for attribute 2 using the greedy heuristic. Since column one of $\tilde{U}(2)$ has the larger column summation, l_2 is set to one. This process is repeated to set a level for attribute 3. In σ_3, there are four feasible product profiles. Each of them has equal probability 0.25 being selected by uniform random sampling. It is easy to compute that using greedy heuristic sampling the optimal product profile

$\theta^* = [3\ 1\ 2]$ has probability $(0.5 + 0.5w_1)^2$ being selected. If we let w_1 be equal to 0.9 (as we did in the numerical experiments reported below), then the probability of selecting this optimal product profile is 0.9025. Therefore, the greedy heuristic sampling can help the NP algorithm to move in a correct direction, hence increasing the speed of convergence. Note that here $w_1 \in (0, 1)$ is a control parameter and if $w_1 \to 0$ then this algorithm is pure uniform sampling, and if $w_1 \to 1$ then this algorithm is simply the non-randomized greedy heuristic.

4.4.3 The NP/DP Algorithm for the 3-Attribute Example

Alternatively, the NP/DP algorithm may incorporate the dynamic programming (DP) heuristic in the sampling step. Lets again take the sampling in σ_3 as an example. As with greedy heuristics sampling, the algorithm first generates a uniform random variable u_0 from $\mathcal{U}(0, 1)$. If $u_0 > w_2$, it generates a uniform random sample as described in the last section. Otherwise, it uses the DP heuristics with a randomized order to obtain a sample. It is easy to verify that DP applied in this region is able to identify the optimal product profile, $\theta^* = [3\ 1\ 2]$, regardless of the order of the attributes. Therefore, similar to NP/GS, NP/DP will select the optimal product profile with probability $(0.5+0.5w_2)^2$. If w_2 is set to 0.9, this probability is 0.9025, which is much larger than the probability to select the optimal product profile by uniform random sampling. As before $w_2 \in (0, 1)$ is a control parameter and if $w_2 \to 0$ then this algorithm is pure uniform sampling, and if $w_2 \to 1$ then this algorithm is a randomized DP heuristic. In practice, since the DP heuristic is computationally more intensive than the greedy heuristic, we usually set w_2 to a small value, for example $w_2 = 0.1$ as we did in our numerical experiments reported below.

4.4.4 The NP/GA Algorithm for the 3-Attribute Example

The NP/GA algorithm applies GA search in the step of estimating promising index of a region. Again we illustrate how GA can be applied when evaluating the region σ_3. Instead of obtaining just one sample, we now sample two points since GA requires at least two points in its initial population. Suppose we apply uniform sampling and obtain two sample points: $\theta^1 = [3\ 1\ 1]$ and $\theta^2 = [3\ 2\ 2]$. The GA is then applied to a population consisting of these two points. Readers can verify that the optimal product profile [3 1 2] is obtained immediately after the first iteration using the GA operators defined in Appendix C. Therefore, in this example, with GA search the NP algorithm directly moves to the right region with probability one.

4.4.5 The NP/GA/GS Algorithm for the 3-Attribute Example

The NP/GA/GS algorithm also applies GA search in the step of estimating promising index of a region, but use greedy heuristics to select the initial population. As we described before, if we set $w_1 = 0.9$ and use a two point initial population, the optimal product profile $\theta^* = [3\ 1\ 2]$ will have probability of more than 0.9025 to be selected in the initial population.

4.5 Numerical Results

We have introduced the NP optimization framework and several variants that incorporate existing heuristics. In this section we evaluate each of these variants empirically, and compare them with the pure GS, pure DP heuristic, and pure GA search. The primary numerical results are for seven different problems, five small to moderately sized problems, and two large problems. Due to the size of most of these problems it is not possible to solve them exactly within a reasonable amount of time and the true optimum is therefore unknown. All the problems have $N = 400$ customers and for simplicity we let all the attributes have the same number of levels, that is, $L_k = L$ for all $k = 1, 2, .., K$. The part-worths preferences for each level of each attribute and the status quo prices are generated uniformly for each customer. The details about how the simulated data sets can be generated are described in [50]. All of the NP algorithms use $N = 20$ sample points from each region, and our numerical experience indicates that the performance of the algorithms is fairly insensitive to this number. If backtracking is chosen in the k-th iteration then the algorithm backtracks $\lfloor \frac{d(\sigma(k))}{2} \rfloor$ steps, or to a depth that is half of the current depth.

Our first set of experiments compares the performance of using the pure heuristic methods of GS, DP, GA, and DC versus incorporating them into the NP optimization framework. For the NP/GS algorithm, that uses the GS sampling algorithm, we let $w_1 = 0.9$ be the control parameter that determines the probability of an attribute being set uniformly or greedily. Our numerical experiments indicate that this parameter should be selected fairly large, for example at least $w_1 \geq 0.5$. For the NP/DP algorithm our results indicate a smaller control parameter is warranted, and in the experiments reported here we use $w_2 = 0.1$, which implies that on the average 10% of the sample points were obtained using the DP heuristic and 90% using uniform sampling. For the NP/GA algorithm, which incorporates the GA search algorithm to estimate the promising index, a total of ten GA search steps were used in each region. Both the NP/GA and the pure GA had a 20% mutation rate. Note it is possible to incorporate GS sampling into pure GA to select initial population, resulting a new hybrid algorithm GA/GS. Our numerical results show that GA/GS is able to find better product profiles more quickly than pure GA in most cases. However, the long-run performance of pure GA is slightly better than

$K = L$	STAT	GS	DP	GA	DC	NP-Pure	NP/GS	NP/DP	NP/GA	NP/GA /GS
5	AVG.	219	228	238	233	238	238	238	238	238
	S.E.	0.0	3.7	0.0	2.8	0.0	0.0	0.0	0.0	0.0
	MAX.	219	235	238	235	238	238	238	238	238
6	AVG.	237	234	239	234	239	239	239	239	239
	S.E.	0.0	3.1	0.0	4.5	0.0	0.0	0.0	0.0	0.0
	MAX.	237	237	239	238	239	239	239	239	239
7	AVG.	225	231	241	229	239	241	241	241	241
	S.E.	0.0	3.2	0.0	2.5	1.1	0.0	0.0	0.0	0.0
	MAX.	225	236	241	233	241	241	241	241	241
8	AVG.	228	234	240	227	236	239	239	240	240
	S.E.	0.0	6.0	0.0	3.6	2.0	0.4	0.8	0.0	0.0
	MAX.	228	239	240	233	240	240	240	240	240
9	AVG.	223	235	245	235	240	244	245	247	247
	S.E.	0.0	4.6	1.5	4.5	3.7	2.3	1.6	0.0	0.0
	MAX.	223	241	247	240	245	247	247	247	247
10	AVG.	241	241	250	239	241	248	249	252	253
	S.E.	0.0	4.8	1.3	5.5	4.6	3.5	2.6	1.0	0.0
	MAX.	241	248	251	246	248	252	252	253	253
20	AVG.	237	247	261	246	248	256	258	266	269
	S.E.	0.0	5.2	2.8	9.3	5.0	4.4	4.2	3.1	2.7
	MAX.	237	259	263	257	258	264	265	269	272

Table 18: Comparison of all the algorithms

that of GA/GS. This is intuitively reasonable since the initial population selected by GS sampling usually consists of better quality but more homogeneous product profiles than randomly selected population. Therefore, the GA algorithm with this kind of population seed tends to find good product profiles quickly, but converges prematurely.

The performance of each algorithm after a fixed CPU time is given in Table 18 for ten replications of each algorithm, except for the GS algorithm, which is completely deterministic and thus requires only one replication. These results indicate that for all of the GS, DP, and GA method, it is beneficial to incorporate a heuristic into the NP framework. The resulting NP/GS, NP/DP, and NP/GA algorithms perform no worse than their pure counterpart heuristics in all cases. The overall percentage improvement is showed in Table 19. For example, for the problem $K = L = 10$ the hybrid NP/GA/GS algorithm has an average improvement of 4.6%. This implies that out of 400 potential customers, an average of 18 more customers are anticipated to purchase the offered product if the design found by the NP/GA/GS algorithm is used rather than the design found by the pure GS algorithm. Furthermore, for all randomized algorithms, Table 18 shows that the corresponding NP algorithm has lower standard deviation than the heuristic alone. This indicates that the product profile quality obtained using the NP framework is more predictable.

In Table 18, results from implementation of pure NP are also provided. In the pure NP algorithm, we use only uniform sampling scheme to select each product profile. The data indicates that the pure NP performs well compared to GS and DP, but any hybrid algorithm, such as NP/GS can produce more effective results.

All of the results above are based on a fixed CPU time comparison, that is, each algorithm ran for the same given length of time. It is, however, also of interest to consider how each algorithm evolves over time. In Figure 20, the pure GA, NP/GA, and

Problem		Performance Improvement			
K	L	over GS	over DP	over GA	over DC
5	5	8.7%	4.4%	0.0%	2.2%
6	6	0.8%	2.1%	0.0%	2.1%
7	7	7.1%	4.3%	0.0%	5.2%
8	8	5.3%	2.6%	0.0%	5.7%
9	9	10.8%	5.1%	0.8%	5.1%
10	10	4.6%	4.6%	0.8%	5.4%
20	20	13.5%	8.9%	3.1%	9.4%

Table 19: Comparison of NP/GA/GS to heuristics without the NP framework.

NP/GA/GS are compared for a large problem that has 50 attributes, with 20 levels of each attribute. For each algorithm, the left hand graph shows the replication that gave the best results plotted against the CPU time used, the right hand graph contains the same graph for the worst replication. These results indicate several noteworthy features. We see that for this example the NP/GA/GS algorithm dominates the NP/GA algorithm for any CPU time used. This implies that the NP/GA/GS algorithm is always preferable. The pure GA search produces near optimal solutions sooner than the NP/GA and NP/GA/GS algorithms, but seems to get stuck at a local optima and have difficulty improving the solution. Thus in the long run, both the NP/GA and NP/GA/GS algorithms outperform pure GA with a considerable margin.

The same comparison is made for the GS, NP/GS, and NP/GA/GS algorithms in Figure 21. For this example the NP/GA/GS algorithm does not dominate the NP/GS algorithm. The NP/GS algorithm appears to be faster and reports better solutions for the first few hundred seconds. For this time it also outperforms the pure GS, and in fact it has a considerable better solution quality from the first solution reported. Therefore, it is preferable to use the NP optimization framework rather than one of the pure heuristics.

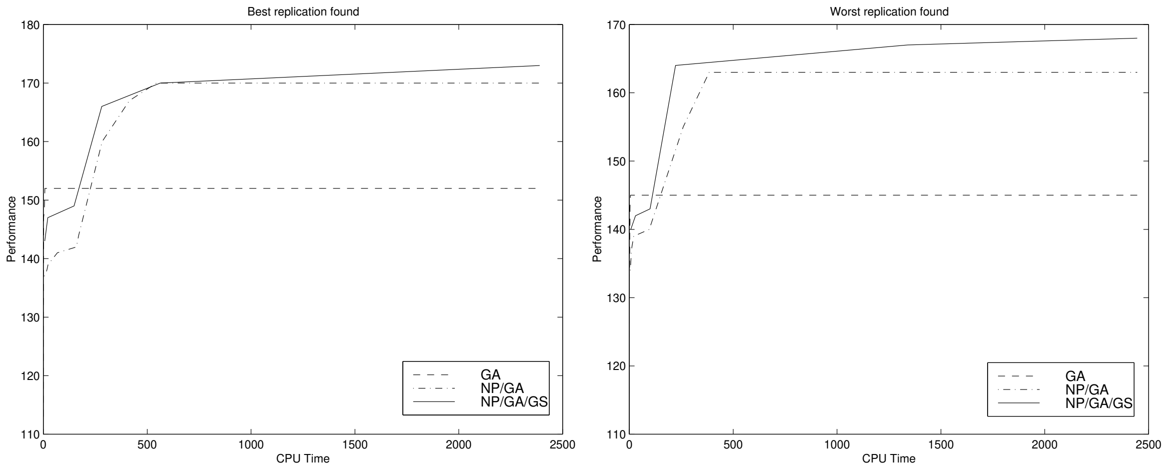

Figure 20: Performance of the GA, NP/GA, and NP/GA/GS algorithms as a function of time for the 50 attribute, 20 level problem.

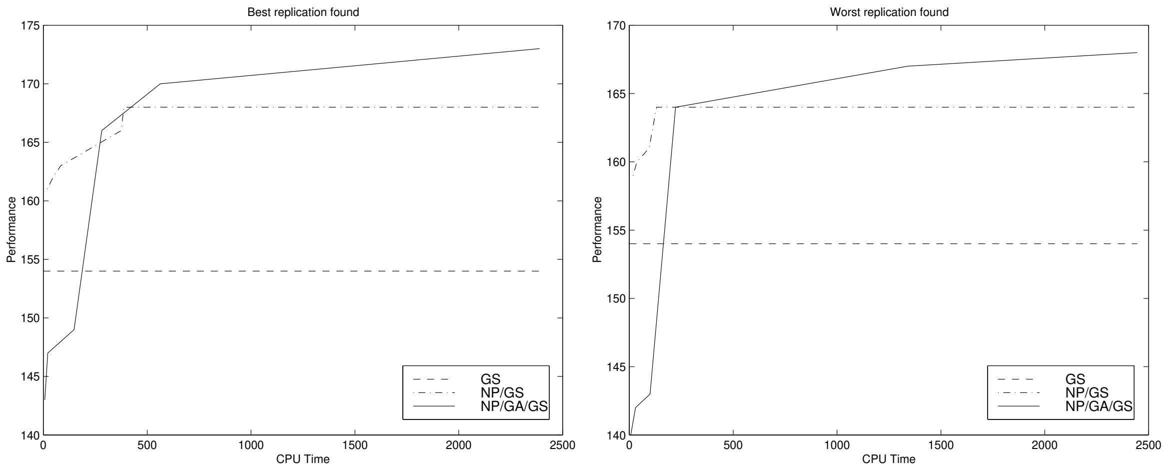

Figure 21: Performance of the GS, NP/GS, and NP/GA/GS algorithms as a function of time for the 50 attribute, 20 level problem.

Problem		Algorithm						
K	L	GS	NP/GS	DP	NP/DP	GA	NP/GA	NP/GA/GS
5	5	217	237	235	237	237	237	237
10	10	236	249	240	251	253	255	255
20	20	238	255	250	258	263	268	270

Table 20: Average performance for beta distributed customer preference

To evaluate the robustness of the above results we repeat several of the comparisons for part-worths customer preferences that follow a beta distribution rather than a uniform distribution. In particular, we use a beta distribution with parameters $\beta_1 = \beta_2 = 2$, which results in a symmetric concave function. Instead of repeating the experiments for all the parameter settings, we restricted ourselves to $K \in \{5, 10, 20\}$ and $L = K$. The results can be found in Table 20 which indicates that the average performance benefits of the NP framework is insensitive to the distribution of the part-worths preferences.

Finally, to gain some understanding into why the hybrid algorithms with heuristics incorporated into the NP framework perform well, and in particular, why such algorithms are not as prone to get stuck at a local optimum, we consider a few of the actual product profiles found by the various algorithms. The results are reported in Table 21 for $K = L = 10$ and in 22 for $K = L = 20$. Each table gives the actual product profile, objective function value of that profile, that is the number of customers purchasing the offered product, and the algorithm that found the given product profile. Indeed, the best profile, which is found by NP/GA algorithms, has about the half of the same attribute levels as the profiles found by the DP, NP/DP, and GA algorithms. For the larger problem, however, the situation is quite different. The best profile, found by the NP/GA algorithm, has only four out of twenty attributes in common with the second best profile found by the pure GA algorithm. Similarly, the profile found by the NP/DP algorithm has only

Product profile	Objective	Algorithm
4 9 6 4 2 8 3 6 10 1	241	GS
6 8 6 3 10 2 2 6 2 1	250	NP/GS
4 3 6 4 2 2 3 6 10 6	248	DP
2 6 6 10 2 2 9 6 10 6	251	NP/DP
6 3 6 7 10 2 9 6 10 1	251	GA
6 8 6 10 10 2 9 6 2 1	253	NP/GA

Table 21: Product profiles for the $K = L = 10$ problem

Product profile	Objective	Algorithm
09 05 14 06 16 15 18 20 15 15 06 10 05 09 03 08 16 11 12 17	237	GS
09 04 14 06 16 15 17 20 15 15 06 20 05 11 03 08 10 11 12 02	263	NP/GS
18 05 17 19 19 17 12 12 13 04 15 18 07 05 03 10 12 12 14 17	254	DP
16 04 20 06 16 14 07 12 13 12 14 10 17 19 17 16 04 12 11 19	264	NP/DP
09 16 15 06 16 15 07 20 03 01 16 13 18 13 17 16 20 13 08 17	266	GA
14 05 20 09 12 14 07 08 13 15 06 10 14 13 03 16 10 11 12 17	271	NP/GA

Table 22: Product profiles for the $K = L = 20$ problem

one attribute in common with the profile found by the DP algorithm. This illustrates the benefits of using global search when seeking the optimal product profile, since it is capable of finding profiles that are very different from those found by a local search heuristic, which iteratively move from one profile to another similar profile. The result is an optimization framework that appears to be more effective for product design than any of the previously proposed methods.

4.5.1 Product Line Design

The NP framework can also be applied to more important product line design problems. We have shown that NP/GA/GS works best for single product design, so we choose it as the base algorithm for product line design. For the sake of simplicity, we call the

Problem		M=2			M=3			M=4		
K	L	NP-Line	BS	δf	NP-Line	BS	δf	NP-Line	BS	δf
5	5	306	306	0.0%	341	336	1.5%	357	353	1.1%
6	6	312	310	0.7%	347	343	1.2%	365	360	1.4%
7	7	318	309	2.9%	350	344	1.7%	367	361	1.7%
8	8	321	308	4.2%	352	341	3.2%	369	361	2.2%
9	9	330	316	4.4%	365	355	2.8%	381	372	2.4%
10	10	328	319	2.8%	360	354	1.7%	378	370	2.2%
20	20	341	323	5.6%	375	363	3.3%	393	385	2.1%

Table 23: Comparison of NP-Line and BS for product line problems

NP/GA/GS based product line design algorithm the NP-Line algorithm. (Nair, Thakur, and Wen 1995) have demonstrated that Beam Search (BS) dominated DP for product line design using 435 simulated problems and a real data set. Thus, we compare NP only with BS for product line design.

We use the same problem data sets described in the last section. For each dataset, we set number of products in the product line to 2, 3 and 4, resulting in 21 test problems. We ran 10 replications for each problem. The average performance is summarized in Table 23.

By adopting a similar approach as is used in (Nair, Thakur, and Wen 1995) we can obtain the second product in a production line by reducing the original data set to a smaller data set and then applying NP-Line algorithm. Our numerical results demonstrate that NP-Line ties with BS in the smallest problem, but dominates BS heuristics in all other cases.

4.6 Mixed Integer Programming Solution

The mixed integer programming model of the single product design problem is difficult to solve. In this section we incorporate several performance tuning heuristics into the FATCOP branch-and-bound framework and provide provable optimal solutions for some reasonable size product design problems. All these heuristics run through the standard interfaces defined in FATCOP and dynamically linked to the solver at run time.

4.6.1 User defined heuristics

4.6.1.1 Upper bound through NP/GA

At the root node of each problem, we first run the NP/GA algorithm on the product design problem. The best solution value found by the NP/GA algorithm is then delivered to the FATCOP solver and set as the upper bound.

4.6.1.2 Priority Branching through Greedy Heuristics

There are two classes of variables in the mixed integer programming formulation given in the beginning of this Chapter: x_{kl} and z_i. We set higher priority for the class of decision variables x_{kl}. Among the decision variables we use a greedy heuristic to set up priorities. For level l of attribute k, we sum the part-worths across the customers to get s_{kl}. The priority for x_{kl} is given based on the value of s_{kl}. The decision variable with the highest s value is assigned the highest priority.

4.6.1.3 Reformulation

The left hand of Constraint 46 has two items: $\sum_{k=1}^{K}\sum_{l=1}^{L_k} c_{ikl} \cdot x_{kl}$ and $K \cdot z_i$. The first item is the overall utility of customer i for a given product profile. If it is negative, z_i is forced to be 1 to make the constraint satisfied. Therefore we require that the coefficient of z_i should be big enough to make the constraint satisfied. Since a relative part-worths utility is always bigger than -1, it is sufficient to set z_i's coefficient to K. However, for many problems, z_i's coefficient is over set. In fact, the minimum possible value of z_i's coefficient (C_i) is given by the following formula:

$$C_i = -\sum_{k=1}^{K} \min_l c_{ikl}$$

Consider a small problem with $N = 1$, $K = 2$, $L_1 = 3$, and $L_2 = 2$. let the part-worths data matrices $U(k)$ be given as follows, $k = 1, 2$:

$$U(1) = \begin{bmatrix} 0.3 & 0.2 & 0.5 \end{bmatrix}, \qquad U(2) = \begin{bmatrix} 0.6 & 0.4 \end{bmatrix},$$

Suppose the status-quo is represented by the values of level 1 for each attribute. Then subtracting the first column from each column of $U(k)$ gives relative parts-worth data matrices $\tilde{U}(k), k = 1, 2, 3$:

$$\tilde{U}(1) = \begin{bmatrix} 0.0 & -0.1 & 0.2 \end{bmatrix}, \qquad \tilde{U}(2) = \begin{bmatrix} 0.0 & -0.2 \end{bmatrix},$$

We can write constraint 46 for this problem as:

$$(0.0)x_{11} + (-0.1)x_{12} + (0.2)x_{13} + (0.0)x_{21} + (-0.2)x_{22} + 2 * z_1 \geq 0$$

we have $c_1 = -(\min\{0, 0, -0.1, 0.2\} + \min\{0, 0, -0.2\}) = 0.3$. Therefore we can change the coefficient of z_1 to 0.3. We can use the same idea to reformulate a subproblem. For example, for the subproblem with x_{11} fixed to 1, we have $c_1 = -(0.0 * 1 + \min\{0, 0, -0.2\}) = 0.2$, so we can further change the the coefficient of z_1 to 0.2. Note that the smaller the coefficient of z_1 is, the tighter the formulation is. We shall show through numerical results that the new formulations have bigger LP relaxation values.

Node	LP Relaxation	
	without reformulation	with reformulation
0	1.58	1.73
1	1.73	4.12
2	2.85	4.63
3	1.74	5.43
4	2.23	4.86
5	1.74	6.13
6	2.87	5.30
7	1.74	5.06
8	2.97	5.14
9	3.22	7.82

Table 24: Effects of Reformulation for the problem $N = 50$, $K = L = 5$

4.6.2 Numerical Results

We have introduced several heuristics for single product design problems. In this section we evaluate each of these heuristics empirically, and compare the FATCOP with CPLEX MIP solver. We also report results for a commercial-size problem. The single product design problem is formulated as a GAMS model given in Appendix D.

4.6.2.1 Effects of Heuristics

The selected test problem has sample size $N = 50$, and $K = L = 5$. The FATCOP sequential solver is used in this section. We first assess the effect of reformulation. Each node in the branch-and-bound tree was solved twice, one with reformulation and the other without reformulation. We report the the LP relaxations of the first 10 nodes in Table 24. The results demonstrate that reformulation is able to increase the LP relaxation, hence provides better lower bound.

We then evaluate effects of the heuristics by adding one heuristic each time. We test on the same problem and report the tree sizes in Table 25. The numerical results show

	without heuristics	NP/GA	NP/GA and Prio. Bran.	NP/GA, Prio. Bran. and reformulation
Tree size	3431	3095	3001	913

Table 25: Effects of the proposed Heuristics for the problem $N = 50$, $K = L = 5$

that each heuristic is able to enhance the performance of the FATCOP solver. Of them, reformulation has most significant impact on the solver's performance.

4.6.2.2 Raw results and comparison to CPLEX

The primary numerical results in this section are for two sets of randomly generated problems. One set of problems has N fixed at 50, and the other set of problems has $K = 12, L = 3$. We turn on and off the heuristics for the FATCOP sequential solver, and run CPLEX with default settings on the test problems for comparison. In Table 26 and Table 27 , for each test instance, we report the size of the search tree for each solver. For all the test instances, FATCOP with user defined heuristics outperforms pure FATCOP solver and CPLEX MIP solver. A product design problem with 12 attributes and 3 levels in average is typically thought as a large commercial-size problem. FATCOP solved the problem with $K = 12, L = 3, N = 100$ in less than 10 minutes CPU time.

However, a larger sample size is usually preferred in practice. Our last experiment is to run FATCOP parallel solver on a problem with $K = 12, L = 3, N = 200$. We use best-bound as our primary searching strategy, and as described in Chapter 2, all workers explore a subtree in a depth first fashion. Knapsack cuts can be generated to this problem, but our numerical experience showed that the cuts are not useful for solving the problem. It is more beneficial to spend the cut generation time on branching and bounding. Therefore we turned off all cutting planes. We also turned off diving

Problem K	L	FATCOP (without heuristics)	FATCOP (with heuristics)	CPLEX
5	5	3,431	913	3,233
6	6	33,715	11,180	27,207
7	7	305,340	46,395	239,973

Table 26: Tree size for the problems with $N = 50$

Problem N	FATCOP (without heuristics)	FATCOP (with heuristics)	CPLEX
50	100,382	29,442	86,435
100	420,119	57,109	292,426

Table 27: Tree size for the problems with $K = 12, L = 3$

heuristics. NP/GA performed in the root node normally could find an optimal or near-optimal solution. All the user defined heuristics described in this section are turned on. The FATCOP job lasted 10.1 hours, evaluated 18,115,883 nodes and utilized 75 machines in average. The solution found by NP/GA was proved to be the optimal solution.

In practice, decision makers often just want to see a good solution in a short time and may only have a single machine available. It is also possible to quickly measure quality of the solutions found by the NP/GA algorithm on a single machine using the FATCOP sequential solver. We ran the FATCOP sequential solver on the same problem. It proved the solution found by the NP/GA algorithm was at most 20% away from the optimal solution in 45 minutes. We noticed that user defined heuristics (particularly reformulation) helped to increase the lower bound quickly. On the other hand, CPLEX could not solve effectively on this problem. We compared the FATCOP sequential solver and CPLEX for this problem and depicted the bound changes in Figure 22.

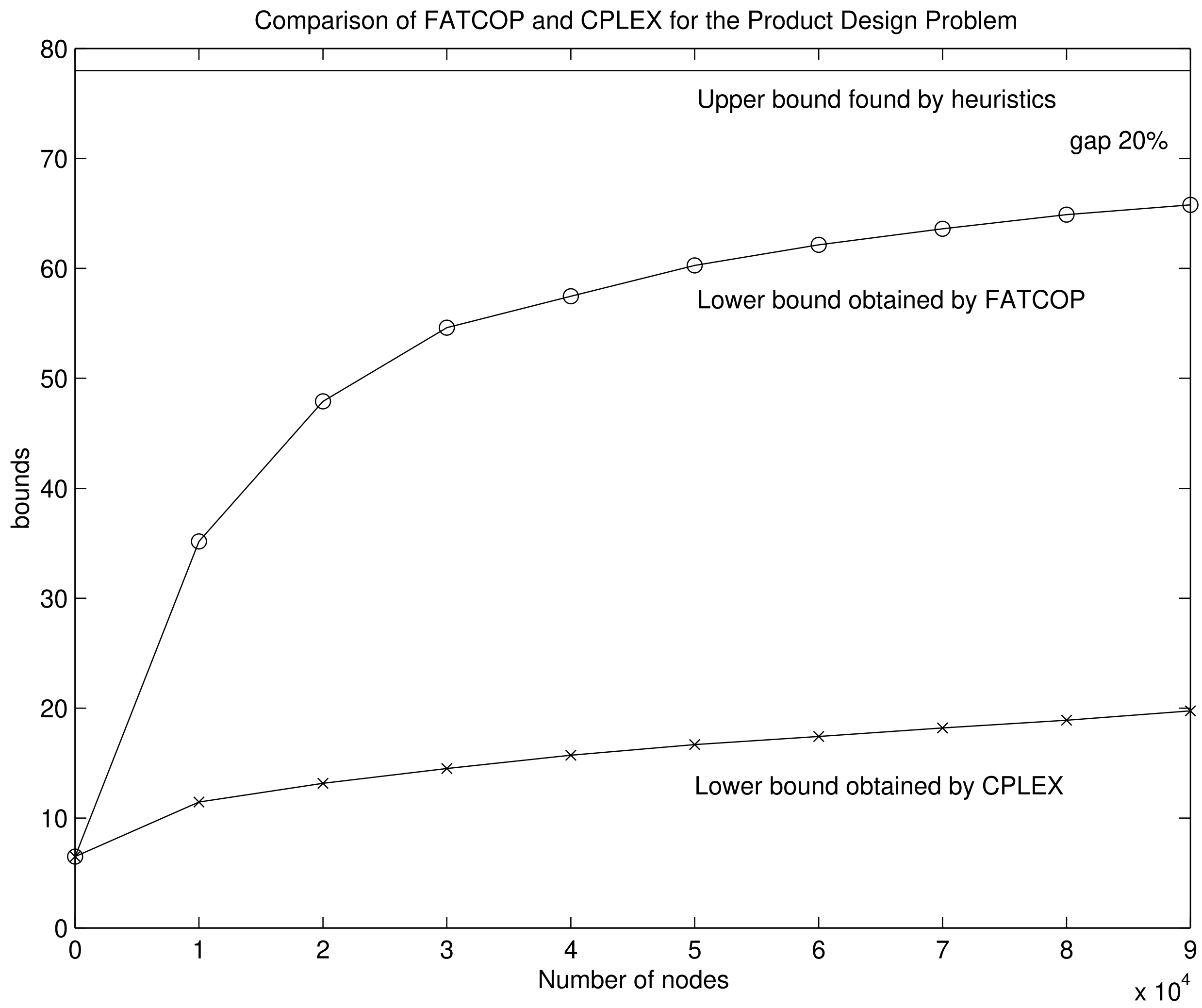

Figure 22: Comparison of FATCOP and CPLEX for the problem with $K = 12, L = 3, N = 200$

Chapter 5

Conclusion and Future Research

In this thesis we provide an integer and combinatorial optimization optimizer that has two components: FATCOP and NP/GA. FATCOP is a parallel branch-and-bound implementation for MIP using distributed privately owned workstations. The solver is designed in the master-worker paradigm to deal with different types of failures in an opportunistic environment with the help of Condor, a resource management system. To harness the available computing power as much as possible, FATCOP uses a greedy strategy to acquire machines. FATCOP is built upon Condor-PVM and SOPLEX, both of which are freely available. The results reported in this thesis show that FATCOP is both an effective MIP solver for a variety of test problems arising in the literature, and an efficient user of opportunistic resources.

NP/GA is a new optimization algorithm that combines a recently proposed global optimization method called the Nested Partitions (NP) method and Genetic Algorithms (GA) in a novel way. The resulting algorithm retains the benefits of both methods, that is, the global perspective and convergence of the NP method and the powerful local search capabilities of the GA. This new optimization algorithm was shown empirically to be capable of outperforming pure GA search for a difficult product design problem. This is especially true for very large problems where the global perspective of the NP method is of particular importance.

As an application of our proposed optimizer, this thesis presented a new optimization framework for constructing product profiles directly from part-worths data obtained from market research. In particular we have focused on the share-of-choices problem, that is, to maximize the number of customers that are anticipated to purchase the offered product. The new methodology, the Nested Partitions (NP) method, consists of a global search phase that involves partitioning and sampling, and a local search phase of estimating a promising index used to guide the partitioning. The method is capable of incorporating known heuristics to speed its convergence. In particular, any construction heuristic, that is a heuristic that builds a single product profile, can be incorporated into the sampling step. This was illustrated using the greedy heuristic and dynamic programming heuristic. On the other hand, any improvement heuristic, that is a heuristic that starts with a complete product profile or set of profiles and finds better product profiles, can be incorporated by using it to estimate the promising index. This was demonstrated using the genetic algorithm. Numerical examples were used to compare the new optimization framework with existing heuristics, and the results indicated that the new method is able to produce higher quality product profiles. Furthermore, these performance improvements were found to increase with increased problems size and we presented results for examples that are considerably larger than those previously reported in the literature. These results indicate that the new NP optimization framework is an important addition to the product design and development process, and will be particularly useful for designing complex products that have a large number of important attributes. We also provided a mixed integer programming solution to the single product design problems. By incorporating some heuristics to the FATCOP solver, we successfully solved some reasonable size product design problems.

There are numerous future research directions that may be pursued, both regarding further development of the FATCOP, and further application of the NP method. We outline these directions as follows.

1. Further experiments with FATCOP will be made to investigate how well the ideas presented scale with an increased number of available resources. Also, we intend to investigate the use of different cutting planes, as well as further exploitation of the local nature of information when performing a task.

2. In this thesis we have solved some reasonable size single product design problems through incorporating problem-specific heuristics into FATCOP. There are other heuristics specific to product design problems. For instance, it is natural to replace the default binary branching by GUB branching [53] in order to have a more balanced and small search tree. The default node selection rule (best-bound) only utilizes the LP relaxation (lower bound) information. For product design problems, we have developed a set of sampling schemes (uniform, GS, DP) in each subregion (or node). Therefore, FATCOP could use both the lower bound information and sample information that provides upper bounds to select the next node to evaluate. In particular, we can use some convex combination of the lower bound and upper bounds to estimate a node. Further experiments include testing some big single product design problems and more difficult product line design problems.

3. In this thesis we have concentrated on designing a single product or product line with the objective of maximizing market share. Several other objectives, such as total buyers' welfare or seller's marginal return, should also be considered. Finally, we plan to apply the new methodology to real data obtained from industry.

4. The results for *algorithm length* presented in Chapter 3 are based on the assumption that success probability p_0 does not dependent on the current most promising

region, or the depth of the region, i.e., the transition of the algorithm may be described as *iid* Bernoulli random variables. Further research includes studying the results for *algorithm length* under the assumption that p_0 is dependent on current most promising region.

5. The NP method is highly compatible with parallel computer structures. A parallel NP algorithm is developed to solve Traveling Salesman problem [61]. The algorithm has each processor concentrate on one part of the feasible region, and chooses the number of processors to coincide with number of partitions M. This algorithm is not suitable for NP applications with small number of partitions, since full utilization of available processors is not achieved. We shall investigate this issue and develop other parallel NP algorithms for both deterministic and opportunistic environments.

Bibliography

[1] M. Avriel and B. Golany. *Mathematical Programming for Industrial Engineers*. Marcel Dekker, 1996.

[2] M. Avriel and B. Golany. Triangulation in Decision Support Systems: Algorithms for Product Design. *Decision Support Systems* **14**, 313-327 (1995).

[3] P.V. Balakrishnan and V.S. Jacob. Genetic algorithms for product design. *Management Science* **42**, 1105-1117 (1996).

[4] E. Balas and C.H. Martin. Report on the session on branch and bound/implicit enumeration, in discrete optimization. *Annals of Discrete Optimization*, 5, 1979.

[5] M. Benichou and J.M. Gauthier. Experiments in mixed-integer linear programming. *Management Science*, 20(5):736–773, 1974.

[6] R. E. Bixby, S. Ceria, C. M. McZeal, and M.W.P. Savelsbergh. MIPLIB 3.0. http://www.caam.rice.edu/b̃ixby/miplib/miplib.html.

[7] A. Brook , D. Kendrick and A. meeraus. *GAMS: Auser's Guide*. The scientific Press, South San Francisco, CA, 1988.

[8] S. Ceria, C. Cordier, H. Marchand and L. A. Wolsey. Cutting planes for integer programs with general integer variables. *Mathematical Programming*, 81(2):201-214, 1998.

[9] Q. Chen and M. C. Ferris. FATCOP: A fault tolerant Condor-PVM mixed integer program solver. Mathematical Programming Technical Report 99-05, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin, 1999.

[10] Q. Chen, S. Ólafsson, and L. Shi, "Remarks on 'Genetic Algorithms for Product Design," submitted to *Management Science* (1998).

[11] R. Cheng and M. Gen. Parallel machine scheduling problems using memetic algorithms. *Computers & Industrial Engineering* **33**, 761-764 (1997).

[12] I.T.Christou and R.R. Meyer. "Optimal equi-partition of rectangular domains for parallel computation," *Technical Report95-04, Computer Science, UW-Madison* 1995.

[13] Condor Group, UW-Madison. *Condor Version 6.0 Manual*. 1998.

[14] H. Crowder, E. L. Johnson, and M. W. Padberg. Solving large scale zero-one linear programming problems. *Operations Research*, 31:803–834, 1983.

[15] Dash Associates, Blisworth House, Blisworth, Northants, UK. *XPRESS-MP User Guide*. http://www.dashopt.com/.

[16] G. Dobson and S. Kalish. "Heuristics for Pricing and Positioning a Product Line Using Conjoint and Cost Data," *Management Science* **39**: 160-175 (1993).

[17] D.L.Eager, M. Ferris and M.K. Vernon. "Optimal Regional Caching for On-Demand Data Delivery," *Technical Report, Computer Science, UW-Madison* 1998.

[18] J. Eckstein. Parallel Branch-and-bound Algorithm for General Mixed Integer Programming on the CM-5. *SIAM J. Optimization*, 4(4):794–814, 1994.

[19] H.L. Emile Aarts and J. Korst. *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*. Wiley, Chichester, 1989.

[20] D.H.Epema, M.Livny et al. A wordwide flock of condors: load sharing among workstation clusters. *Journal on Future Generations of Computer System*,1996.

[21] J. Forrest et al. Practical solution of large scale mixed integer programming problems with UMPIRE. *Annals of Discrete Optimization*, 5, 1979.

[22] R.Fourer, D.M. Gay and B. W. Kernighan *AMPL: A Modeling Language For Mathematical Programming*. Scientific Press, 1993.

[23] A. Geist, A. Beguelin et.al. *PVM: Parallel Virtual Machine - A user's Guide and Tutorial for Networked Parallel Computing*. The MIP Press, Cambride, Massachusetts, 1994.

[24] M. Gen, G.S. Wasserman and A.E. Smith. Special issue on genetic algorithms and industrial engineering. *Computers & Industrial Engineering* **30** (1996).

[25] B. Gendron and T. G. Crainic. Parallel Branch-and-Bound algorithms: survey and systhesis. *Operations Research*, 42(6):1042–1060, 1994.

[26] F. Glover. Tabu search: A tutorial. *Interfaces*, 20(4):74-94, 1990.

[27] D.E. Goldberg. *Genetic algorithms in search, optimization, and machine learning.* Addison-Wesley, Reading, MA (1989).

[28] D. Gong, M. Gen, G. Yamazaki and W. Xu. Hybrid evolutionary method for capacitated location-allocation problem. *Computers & Industrial Engineering* **33**, 577-580 (1997).

[29] J.-P. Goux, J. Linderoth, and M. Yoder. Metacomputing and the master-worker paradigm. Technical report, Argonne National Laboratory, 1999.

[30] P.E. Green and A.M. Krieger. Recent contribution to optimal product positioning and buyer segmentation. *European Journal of Operations Research* **41**, 127-141 (1989).

[31] P.E. Green and A.M. Krieger. Models and Heuristics for Product Line Selection. *Marketing Science* **4**: 1-19 (1985).

[32] P.E. Green and A.M. Krieger. Recent Contributions to Optimal Product Positioning and Buyer Segmentation. *European Journal of Operations Research* **41**, 127-141 (1989).

[33] P.E. Green and A.M. Krieger. A simple heuristic for selecting 'good'products in conjoint analysis. *J. Advances in Management Science* **5**, R. Schultz (ed.), JAI Press, Greenwich, CT (1987).

[34] P.E. Green and A.M. Krieger. An application of a product positioning model to pharmaceutical products. *Marketing Science* **11**: 117-132 (1992).

[35] P.E. Green and V. Srinivasan. Conjoint analysis in consumer research: new developments and directions. *Journal of Marketing* **54**, 3-19 (1990).

[36] W. Gropp, E. Lusk, A. Skjellum. *Using MPI : portable parallel programming with the message-passing interface*. The MIT Press, Cambridge, Massachusetts, 1994.

[37] Y.C. Ho, R.S. Sreenivas and P. Vakili. Ordinal optimization of DEDS. *Discrete Event Dynamic Systems: Theory and Applications* **2**, 61-88 (1992).

[38] J.H. Holland. *Adaptation in natural and artificial systems*. The University of Michigan Press, Ann Arbor, MI (1975).

[39] M. S. Hung, W. O. Rom, and A. D. Warren. *Handbook for IBM OSL*. Boyd and Fraser, Danvers, Massachusetts, 1994.

[40] ILOG CPLEX Division, Incline Village, Nevada. *CPLEX Optimizer*. http://www.cplex.com/.

[41] S. Kekre and K. Srinivasan. Broad Product Line: A Necessity to Achieve Success? *Management Science* **36**: 1216-1231 (1990).

[42] R. Kohli and R. Khrisnamurti. A heuristic approach to product design. *Management Science* **33**, 1523-1533 (1987).

[43] R. Kohli and R. Khrisnamurti. Optimal product design using conjoint analysis: computational complexity and algorithms. *European Journal of Operations Research* **40**, 186-195 (1989).

[44] R. Kohli and R. Sukumar. Heuristics for Product-Line Design using Conjoint Analysis. *Management Science* **35**: 1464-1478 (1990).

[45] V. Kumar and A. Gupta. Analyzing scalability of parallel algorithms and architectures. *Tech. Report TR 91-18, Department of Computer Science, University of Minnesota*, 1991.

[46] A. Land and A. Doig. An automatic method of solving discrete programming problems. *Econometrika* **28(3)**, 497-520 (1960).

[47] J. Linderoth and M.W.P. Savelsbergh. A Computational Study of Search Strategies for Mixed Integer Programming. *Report LEC-97-12, Georgia Institute of Technology*, 1997.

[48] M.J.Litzkow, M.Livny et al. Condor - A hunter of idle workstations *in Proceedings of the 8th International Conference on Distributed Computing Systems, Washington, District of Columbia, IEEE Computer Society Press*, 108-111, 1988.

[49] T. Murata, H. Ishibuchi and H. Tanaka. Genetic algorithms for flowshop scheduling problems. *Computers & Industrial Engineering* **30**, 1061-1072 (1996).

[50] S.K. Nair, L.S. Thakur, and K. Wen. Near Optimal Solutions for Product Line Design and Selection: Beam Search Heuristics. *Management Science* **41**: 767-785 (1995).

[51] J. L. Nazareth. *Computer Solution of Linear Programs* Oxford University Press, 1987.

[52] G.L. Nemhauser, M.W.P. Savelsbergh, G.S. Sigismondi. MINTO, a Mixed Integer Optimizer. *Oper. Res. Letters*, 15:47–58, 1994.

[53] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization.* Wiley, New York, NY, 1988.

[54] S. Ólafsson and L. Shi. A method for scheduling in parallel manufacturing systems with flexible resources. To appear in *IIE Transactions* (1998).

[55] E.A. Pruul and G.L. Nemhauser. Branch-and-Bound and parallel computation: a historical note. *Oper. Res. Letters*, 7:65-69, 1988.

[56] J. Pruyne and M. Livny. Interfacing Condor and PVM to harness the cycles of workstation clusters. *Journal on Future Generations of Computer Systems*, 12, 1996.

[57] T. J. Van Roy and L. A. Wolsey. Solving mixed integer 0-1 programs by automatic reformulation. *Operations Research*, 35:45–57, 1987.

[58] T. L. Saaty. *The Analytical Hierarchy Process.* RWS Publications (1980).

[59] H. M. Salkin and K. Mathur. *Foundations of Integer Programming.* North-Holland (1989).

[60] M.W.P. Savelsbergh. Preprocessing and Probing for Mixed Integer Programming Problems. *ORSA J. on Computing*, 6: 445–454, 1994.

[61] L. Shi, S. Ólafsson and N. Sun. New parallel randomized algorithms for the traveling salesman problem. To appear in *Computers & Operations Research* (1997).

[62] L. Shi and S. Ólafsson, and Q. Chen, "An optimization framework for product design," submitted to *Management Science* (1998).

[63] L. Shi and S. Ólafsson, and Q. Chen. A New Hybrid Optimization Algorithm. *Computers & Industrial Engineering* **36**, 409-426 (1999).

[64] L. Shi and S. Ólafsson. Nested partitions method for global optimization. *Operations Research* **48** (3) (2000).

[65] R. Wunderling. SOPLEX: User documentation. http://www.zib.de/Optimization/Software/Soplex/.

[66] F. Zufryden. A conjoint-measurement-based approach to optimal new product design and market segmentation. In *Analytical approaches to product and market planning*, A.D. Shocker (Ed.), Marketing Science Institute, Cambridge, MA (1977).

Appendix A

GS Sampling

Step 0 For the attributes that are fixed by the partitioning, let

$$l_i^s = l_i^{\sigma_j}, \tag{52}$$

for all $i \leq d(\sigma_j)$.

Compute the overall relative utilities,

$$u_{kl} = \sum_{i=1}^{M} \left(u_{ikl} - u_{ik\tilde{l}_k} \right), \tag{53}$$

for each attribute $k = 1, 2, ..., K$, and each level $l = 1, 2, ..., L_k$ of that attribute.

Note that here $\tilde{l}_k$ is the level of the k-th attribute for the status-quo product.

Set $k = d(\sigma_j) + 1$.

Step 1 Generate a uniform random variable u_0 from $\mathcal{U}(0, 1)$.

Step 2 If $u_0 > w_1$ then go to Step 3, otherwise go to Step 5.

Step 3 Generate a uniform random variable u_1 from $\mathcal{U}(0, 1)$.

Step 4 If $\frac{l-1}{L_k} \leq u_1 < \frac{l}{L_k}$ then set the k-th attribute to level l, $l = 1, 2, ..., L_k$,

$$l_k^s = l. \tag{54}$$

Go to Step 6.

Step 5 Select the level l_k^* of the k-th attribute that satisfies

$$l_k^* = \arg \max_{l=1,2,...,L_k} u_{kl}, \tag{55}$$

and set the k-th attribute to this level,

$$l_k^s = l_k^*. \tag{56}$$

Continue to Step 6.

Step 6 If $k = K$ stop, otherwise let $k = k + 1$ and go back to Step 1.

Appendix B

DP Sampling

Step 0 For the attributes that are fixed by the partitioning, let the sample profile θ^s have the same levels,

$$l_i^s = l_i^{\sigma_j}, \tag{57}$$

for all $i \leq d(\sigma_j)$. Thus a partial profile that fixes $d(\sigma_j)$ attributes is determined by the partitioning.

Compute the overall relative utilities,

$$u'_{kl} = \sum_{i=1}^{M} \left(u_{ikl} - u_{ik\tilde{l}_k} \right), \tag{58}$$

for each attribute $k = 1, 2, ..., K$, and each level $l = 1, 2, ..., L_k$ of that attribute.

Set $k = d(\sigma_j) + 1$.

Step 1 Generate a uniform random variable u_0 from $\mathcal{U}(0, 1)$.

Step 2 If $u_0 > w_2$ then go to Step 3, otherwise go to Step 5.

Obtain a uniform sample:

Step 3 Generate a uniform random variable u_1 from $\mathcal{U}(0, 1)$.

Step 4 If $\frac{l-1}{L_k} \leq u_1 < \frac{l}{L_k}$ then set the k-th attribute to level l, $l = 1, 2, ..., L_k$,

$$l_k^s = l. \tag{59}$$

If $k = K$ stop, otherwise let $k = k + 1$ and go back to Step 3.

Obtain a randomized DP sample:

Step 5 We need to maintain a set of partial profiles for each attribute that has not been determined. Initialize this as the singleton set that contains only the partial profile that is fixed by the partitioning,

$$P_{K-1} = \{\theta^s\}\,, \tag{60}$$

with θ^s as in Step 0 above.

Randomize the order of the remaining $n - d(\sigma_j)$ attributes

$$l^s_{i_1}, l^s_{i_2}, ..., l^s_{i_{n-d(\sigma_j)}}, \tag{61}$$

where $\{i_1, i_2, ..., i_{n-d(\sigma_j)}\}$ is a permutation of $\{d(\sigma_j)+1, d(\sigma_j)+2, ..., n\}$.

Step 6 Given P_{k-1}, the set of last stages partial profiles, and the possible levels of the current attribute $l^s_{i_{k-d(\sigma_j)}}$, identify the k-th stage set of L_k partial profiles,

$$P_k = \{p_{1k}\ p_{2k}\ ...\ p_{L_k k}\}\,. \tag{62}$$

This set is obtained by, for each level l of the $i_{k-d(\sigma_j)}$-th attribute finding the partial profile $p_{l'(k-1)}$ in P_{k-1} that maximizes the overall relative utilities. The new partial profile p_{lk} is then constructed from $p_{l'(k-1)}$ by also letting the $i_{k-d(\sigma_j)}$-th attribute be set to the l'-th level.

Step 7 If $k = K$ let the sample product profile be

$$\theta^s = p_{l^*K}, \tag{63}$$

where

$$l^* = \arg \max_{l=1,2,...,L_K} f\left(p_{lK}\right), \tag{64}$$

and stop. Otherwise let $k = k+1$ and go back to Step 6.

The details of how to implement Step 6 of the algorithm have appeared in [43], and are omitted here for brevity.

Appendix C

GA Search

Step 0 **Initialization.** Let $POP_0 = \theta_j$, where θ_j is defined in step 2 of the NP algorithm given in Chapter 3.

Let $k = 0$.

Step 1 **Reproduction.** Sort the product profiles in the current population $POP_k = \left[\theta_k^{[1]}\ \theta_k^{[2]}\ ...\theta_k^{N[j]}\right]$ according to their fitness

$$f\left(\theta_k^{[1]}\right) \geq f\left(\theta_k^{[2]}\right) \geq ... \geq f\left(\theta_k^{[N_j]}\right). \tag{65}$$

Add the fittest product profiles to the new population,

$$POP_{k+1} = \left[\theta_k^{[1]}\ \theta_k^{[2]}\ ...\theta_k^{\left[\frac{N_j}{2}\right]}\right]. \tag{66}$$

Step 2 **Crossover.** Select two random indices $1 \leq i, j \leq \frac{N_j}{2}$, $i \neq j$. Select a random attribute k where $d(\sigma_j) < k \leq K$, and create two new product profiles by replacing the k-th attribute of $\theta_k^{[i]}$ with the k-th attribute of $\theta_k^{[j]}$, and vice versa. Add these new product profiles to the new population POP_{k+1}. Repeat until $\frac{N_j}{2}$ new product profiles have been created.

Step 3 **Mutation.** Select a random index $1 \leq i \leq N_j$. Select a random attribute k where $d(\sigma_j) < k \leq K$, and mutate $\theta_k^{[i]}$ by replacing its k-th attribute with a random level. Repeat.

Step 4 Let $k = k+1$. If $k > k_{max}$, the total number iterations, go to Step 5. Otherwise, go back to Step 1.

Step 5 Finally, use the best product profile found using the GA search

$$H_{\sigma_j(k)}\left(\theta_j\right) = \theta_k^{[1]}. \tag{67}$$

to estimate the promising index in Step 3 of the NP algorithm.

Appendix D

GAMS model for MIP formulation of single product design problem

```
$title product design, share of choice problem

$setglobal nAttr 12
$setglobal nLev  3
$setglobal nCust 50

set at /1*%nAttr%/;
set le /1*%nLev%/;
set cu /1*%nCust%/;

alias(at,a);
alias(le,l);
alias(cu,c);
```

```
alias(pl,p);


****** read utility matrix uniformly ******

parameter util(cu,at,le)/
$include util.inc
/;


****** varaibles ******
binary variable x(at,le);
binary variable z(cu);
variable obj;


****** equations ******
equation  objective,
sinLev(at),
cons1(cu);

objective.. obj =e= sum(c,z(c));

sinLev(a).. sum(l, x(a,l)) =e= 1;
```

```
cons1(c).. sum((a,l),x(a,l)*util(c,a,l)) + %nAttr%*z(c) =g= 0;



model prodDesign /all/;

prodDesign.optcr = 0.0;

prodDesign.optca = 0.9999;

prodDesign.reslim=1000000;

prodDesign.nodlim=100000000;


prodDesign.optfile =  1;


prodDesign.prioropt =  1;

x.prior(a,l) =  2;

z.prior(c) =  1;



solve prodDesign using mip minimizing obj;
```