MULTI RESOURCE AGENT BOTTLENECK GENERALIZED ASSIGNMENT PROBLEM

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ÖZLEM KARABULUT

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
INDUSTRIAL ENGINEERING

MAY 2010

Approval of the thesis:

MULTI RESOURCE AGENT BOTTLENECK GENERALIZED ASSIGNMENT PROBLEM

submitted by **ÖZLEM KARABULUT** in partial fulfillment of the requirement for the degree of **Master of Science in Industrial Engineering Department, Middle East Technical University** by,

Prof. Dr. Canan Özgen
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Nur Evin Özdemirel
Head of Department, **Industrial Engineering** _____

Prof. Dr. Meral Azizoğlu
Supervisor, **Industrial Engineering Dept., METU** _____

Examining Committee Members

Prof. Dr. Gülser Köksal
Industrial Engineering Dept., METU _____

Prof. Dr. Meral Azizoğlu
Industrial Engineering Dept., METU _____

Prof. Dr. Sencer Yeralan
Agricultural and Biological Engineering Dept.,
University of Florida, U.S.A. _____

Asst. Prof. Dr. Sinan Gürel
Industrial Engineering Dept., METU _____

Yavuz Tuna, M.S.
Inova Yazılım ve Danışmanlık _____

Date: **06.05.2010**

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name: Özlem KARABULUT

Signature :

ABSTRACT

MULTI RESOURCE AGENT BOTTLENECK GENERALIZED ASSIGNMENT PROBLEM

Karabulut, Özlem

M.S., Department of Industrial Engineering
Supervisor: Prof. Dr. Meral Azizoğlu

May 2010, 124 pages

In this thesis, we consider the Multi Resource Agent Bottleneck Generalized Assignment Problem. We aim to minimize the maximum load over all agents.

We study the Linear Programming (LP) relaxation of the problem. We use the optimal LP relaxation solutions in our Branch and Bound algorithm while defining lower and upper bounds and branching schemes. We find that our Branch and Bound algorithm returns optimal solutions to the problems with up to 60 jobs when the number of agents is 5, and up to 30 jobs when the number of agents is 10, in less than 20 minutes.

To find approximate solutions, we define a tabu search algorithm and an α approximation algorithm. Our computational results have revealed that these procedures can find high quality solutions to large sized instances very quickly.

Keywords: Bottleneck Generalized Assignment Problem, Multi Periods, Branch and Bound Algorithm, Linear Programming Relaxation

ÖZ

DARBOĞAZ ÇOK KAYNAKLI GENELLEŞTİRİLMİŞ ATAMA PROBLEMİ

Karabulut, Özlem

Yüksek Lisans, Endüstri Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Meral Azizoğlu

Mayıs 2010, 124 sayfa

Bu çalışmada, Darboğaz Çok Kaynaklı Genelleştirilmiş Atama Problemi ele alınmıştır. Amacımız, temsilcilere dönemler üzerinden atanan en büyük toplam iş yükünü enazlamaktır.

Problemin doğrusal programlama gevşetmesini çalıştık. Optimal doğrusal programlama gevşetmesi çözümlerini önerdiğimiz dal-sınır yönteminde alt ve üst sınır ve dallandırma yöntemini belirlemekte kullandık. Dal-sınır yöntemimizin büyüklüğü temsilci sayısı 5 iken 60 işe ve temsilci sayısı 10 iken 30 işe kadar olan problemleri 20 dakikadan daha kısa sürede çözdüğünü gördük.

Yaklaşık çözümler bulmak için, bir tabu arama algoritması ve α yaklaşıklama algoritması geliştirdik. Deneylerimizin sonuçları bu yöntemlerin büyük boyuttaki problemlere kısa sürede yüksek kaliteli çözümler bulduğunu göstermiştir.

Anahtar Kelimeler: Darboğaz Genelleştirilmiş Atama Problemi, Çok Dönemli, Dal-Sınır Yöntemi, Doğrusal Programlama Gevşetmesi

ACKNOWLEDGMENTS

I would like to express my deepest appreciation to my supervisor, Prof. Meral Azizoğlu for not only supervising me throughout the study but also helping and supporting me while choosing my career path with her deep wisdom. Always motivating me with her positive energy, she is much more than a supervisor and will be an ideal figure of academic for me throughout my life.

I would like to thank jury members for their valuable contributions on the thesis.

I owe my deepest gratitude to my family members: Aysel Karabulut, Ertuğrul Karabulut and Mustafa Karabulut for their unconditional support and love. Their patience and understanding made it possible for me to finish this study.

There is not a way to express my appreciation to Sinan Karsu. He was with me in every single step of the study and never ceased supporting me even in the hardest periods. Without him, this work would not have been completed. I would also like to thank his family members, Cordula Karsu, Serdar Karsu and Alisa Candan Karsu for their support and encouragement.

I would like to thank my dear friends Bilgen Katipoğlu for her support and patience; Bilge Çelik and Kerem Demirtaş for their cheerful company, even in the hardest times and Berk Orbay for being a nice neighbor. I would also like to thank Erdem Çolak, Volkan Gümüşkaya, Büşra Atamer and Tülin İnkaya for their company. I am also grateful to my other colleagues for providing me a convivial working environment.

I would like to thank TÜBİTAK for the funding they have provided during my M.S. study.

To my family…

TABLE OF CONTENTS

LIST OF FIGURES

FIGURES

LIST OF TABLES

TABLES

CHAPTER 1

INTRODUCTION

Assignment Problems are motivated and stimulated by the situations where the scarce resources have to be allocated to the activities. The assignment theory is concerned with the optimal allocation of scarce resources to the activities through the development and analysis of mathematical models and techniques.

In service environments, each activity, called opportunity (job, request), requires at most one resource, called agent (server), of limited capacity and availability. In production environments, the opportunities and agents are replaced by jobs (tasks) and machines (workers), respectively.

There are two types of assignment problems, namely total cost and bottleneck based on the objective functions. The total cost type objective functions minimize the sum of the assignment costs whereas the bottleneck type functions minimize the maximum cost over all assignments.

The basic assignment model assumes no capacities on the resources, makes a single assignment to each agent and minimizes the total cost. The basic model and its generalizations have been studied for many decades. Besides their obvious practical importance, the assignment problems appear as subproblems in many well recognized Operations Research problems like the Traveling Salesman Problem, Routing, Scheduling, Location and Layout Problems.

Due to the special structure of its constraint space, the basic assignment problem sets all assignment variables to either zero or one, when the integrality

requirements are relaxed. This follows that the basic model can be solved in polynomial time by Linear Programming (LP) software. Even simpler polynomial time algorithms like Hungarian, Labelling algorithms can be used to solve the problem.

The Generalized Assignment Problem (GAP) is an assignment problem where there are capacities on the agents and multiple job assignments to an agent are allowed.

The GAP has many real-life applications as cited in the literature. Some applications, as cited in Cattrysee and Van Wassenhove (1992), include fixed-charge plant location models in which customer requirements must be satisfied by a single plant, grouping and loading for Flexible Manufacturing Systems (Mazzola, Neebe, and Dunn, 1989), resource scheduling, scheduling of project networks, storage space allocation, designing communications networks with node capacity constraints (Grigoriadis, Tang and Woo, 1974), scheduling payments on accounts where 'lump sum' payments are specified, assigning software development tasks to programmers, assigning jobs to computers in computer networks (Balachandran, 1972), scheduling variable length television commercials into time slots, assigning ships to overhauls (Gross and Pinkus, 1972), routing (Fisher and Jaikumar, 1981). As mentioned by Campbell and Diaby (2002), the GAP is also used as an approximation when allocating cross-trained workers to multiple departments where the benefits of assigning additional personnel to a department are given by a concave function. The p-median problem, the capacity constrained p-median problem and the plant location problem can also be modeled as the GAPs (see Ross and Soland, 1977).

The GAP with the minimum total cost objective is NP-hard in the strong sense, since its feasibility question is so (see Martello and Toth, 1995).

The bottleneck GAP (BGAP) is the GAP that has the minimax objective instead of the minimum-sum objective. The BGAP has many practical applications

especially in the public sector. One noteworthy application is the location of the emergency service facilities.

The BGAP is categorized as task based and agent based problems. The task based BGAP (TBGAP) minimizes the maximum cost over all assignments whereas the agent based BGAP (ABGAP) minimizes the maximum cost over all agents.

The task and agent based BGAP are NP hard in the strong sense based on the fact that GAP is so (Martello and Toth, 1995).

The GAP assumes that there is only one type of resource defined on the agent capacities. Pentico (2007), in his survey of assignment problems, identifies another version of the Generalized Assignment Problem namely, the multiple resource GAP (MRGAP). As the name implies the MRGAP deals with an environment where multiple resources define the agent capacities.

The cited practical applications of the MRGAP include the allocation of databases among the nodes of a distributed computer system (Pirkul, 1986), processor and database location in distributed computer systems (Gavish and Pirkul, 1982 and 1986) and the truck routing problem (Murph, 1986). The vehicle routing problem with multiple resources can also be modeled as the MRGAP. The other potential applications as cited by Gavish and Pirkul (1991) include telecommunication network design, cargo loading on ships, warehouse design and work load planning in job shops.

There are two versions of the MRGAP, namely the minimum-sum MRGAP and the Bottleneck MRGAP. Both problems are NP hard in the strong sense as their single resource versions are strongly NP hard.

In this study we consider the Multi-Resource Agent Bottleneck Generalized Assignment Problem (MRABGAP). To the best of our knowledge, there is no

reported study on the MRABGAP. Our interest on this problem stems from its wide practical applications and lack of any theoretical result.

One practical application that we take our motivation from is faced in a nationally recognized firm in the Heating, Ventilating and Air Conditioning (HVAC) sector. The problem they defined and thereafter we formalized was assigning agents to the opportunities such that the agent assigned to an opportunity will follow the opportunity for multiple periods. The agents have limited time for each period, and the time requirement of an opportunity changes according to the agent it is assigned. It is essential that an opportunity is assigned to a single agent. This is because it would take time to coordinate the agents to follow a single opportunity and the communication between the agents would slow down the process if multiple agents were responsible from an opportunity. On the other hand, due to the capacity limitations, the agents should serve a limited number of opportunities. Moreover, the workload balance between the agents should be regarded from managerial perspective. Even when the capacity of an agent permits to serve all opportunities and this agent serves at the fastest pace, the balanced solution would assign a subset of the opportunities to this agent and the rest to the other slower ones.

In this study we first investigate the properties of the optimal solution and state some rules to detect the infeasibility of the instances. We incorporate those properties to our Linear Programming (LP) relaxations en route to improve its efficiency. Moreover we incorporate some valid cuts that are satisfied by the optimal solution, but not the optimal LP relaxed solution. We then propose a Branch and Bound algorithm that incorporates the optimality properties together with efficient bounding mechanisms. Our lower bounds are of two types: one is simple, but not-as-efficient, and used as a filtering mechanism. The other one is an optimal LP relaxed solution, hard-to-compute, is however efficient in performance.

We could solve test problems with up to 60 jobs for 5 agents and up to 30 jobs for 10 agents and up to 5 periods in reasonable CPU times. We hope our results stimulate future research on the subject.

The rest of the thesis is organized as follows. In Chapter 2, we review the literature on the Generalized Assignment Problems. We also give the related mathematical programming formulations of the problems. In Chapter 3, we present our model and state the properties of the optimal and some feasible solutions. We give the LP relaxation bounds.

We discuss our solution procedures in Chapters 4 and 5. We present our Branch and Bound algorithm in Chapter 4 and discuss some filtering mechanisms we incorporated into the algorithm. In Chapter 5, we discuss heuristic procedures to find approximate solutions: a tabu search algorithm and an α approximation scheme. We report the results of our computational experience in Chapter 6. Chapter 7 concludes the study with our main findings and future research suggestions.

CHAPTER 2

LITERATURE SURVEY

The Generalized Assignment Problem (GAP) is a generalization of the well-known assignment problem (AP). The GAP allows multiple assignments to the agents as long as the capacity restrictions are satisfied. As in the classic AP, the GAP assumes that each job will be assigned to only one agent. Throughout the thesis we use the terms opportunity and job, interchangeably.

In this chapter, we first formulate and review the GAP with total cost objective and then the bottleneck GAP.

2.1. The GAP with Total Cost Objective

In this chapter, we first give the GAP model with total cost objective and then give the related literature review.

2.1.1. The Model

The GAP can be modeled as follows:

$$\min \quad \sum_i \sum_j c_{ij} x_{ij} \qquad (1)$$

$$s.t. \quad \sum_j a_{ij} x_{ij} \le b_i, \quad i \in I, \qquad (2)$$

$$\sum_i x_{ij} = 1, \quad j \in J, \qquad (3)$$

$$x_{ij} = 0 \text{ or } 1, \quad i \in I, \quad j \in J, \qquad (4)$$

where x_{ij} equals 1 if job j is assigned to agent i, 0 otherwise.

c_{ij} is the cost of assigning job j to agent i.

a_{ij} is the agent i's capacity required by job j.

b_i is the available capacity of agent i.

Constraint set (2) ensures that agents are not overloaded and constraint set (3) states that each job can be performed by only one agent. The objective function given in (1) tries to minimize the total cost over all assignments.

2.1.2. Survey on the GAP

We first review the optimization and then the approximation algorithms developed and reported for the GAP.

Optimization Algorithms

The GAP is a widely studied problem in the literature. Many optimization studies are reported most noteworthy of which are due to Ross and Soland (1975), Martello and Toth (1981b), Fisher, Jaikumar and Van Wassenhove (1986), Guignard and Rosenwein (1989a), Wilcox (1989), Jörnsten and Värbrand (1987), Karabakal et al. (1992), Savelsbergh (1997), Cattrysse et al. (1998), Park et al. (1998), Farias and Nemhauser (2001), Nauss (2003), Haddadi et al. (2004), Pigatti et al. (2005), Avella et al. (2008).

For a more thoroughly review of the algorithms, one may refer to the survey paper by Cattrysse and Van Wassenhove (1992).

Ross and Soland (1975) propose a lower bound obtained by deleting the capacity constraints (2), hence solving a classical assignment problem. The assignment based lower bound is then strengthened by adding penalties defined for reassigning jobs from one agent to another that satisfy the capacity constraints.

Ross and Soland (1975) and Fisher, Jaikumar and Van Wassenhove (1986) also show that a lower bound can be obtained by relaxing constraint set (3) by Lagrange multiplier. The relaxed problem is a knapsack problem and solved by setting the Lagrange multipliers to the second smallest c_{ij} values. This lower bound is used in a Branch and Bound (*B&B*) algorithm employing a binary branching strategy based on the remaining agent capacities and the penalty for not assigning a job to the least costly agent. They report computational results for the problems with up to 4000 binary variables.

Martello and Toth (1981b) remove the constraint set (3) from the equivalent maximization model and find single independent knapsacks. Their branching strategy assigns the unassigned jobs and handles the jobs that are assigned more than once. The bound is improved by computing a lower bound on the penalty for satisfying the relaxed constraints. The job with the maximum penalty is chosen as the branching variable in this Branch and Bound algorithm. They report computational results for problems with up to 5 agents and 20 jobs and show the superiority of their results over those of Ross and Soland (1975).

Fisher, Jaikumar and Van Wassenhove (1986) study the Lagrangean relaxation of the problem that dualizes constraint set (3). They set the corresponding Lagrangean multipliers, μ_j, to the second largest c_{ij} value (the maximization version is considered), hence obtain the bound proposed by Ross and Soland. In the first step the jobs are assigned to the agents having $c_{ij} - \mu_j > 0$, i.e., to the maximum profit agents, then an assignment problem is solved for the jobs having $c_{ij} - \mu_j = 0$. In the second step, the multipliers are adjusted by a heuristic procedure so as to assign more jobs. The Branch and Bound algorithm starts when no further improvements are possible. The branching strategy selects the free variable with the largest a_{ij} value. Their computational results for problems with up to 20 jobs and 5 agents show the superiority of the algorithm over those of Ross-Soland (1975) and Martello-Toth (1981b).

Guignard and Rosenwein (1989a) study the procedure proposed by Fisher, Jaikumar and Van Wassenhove (1986). They use a Lagrangean dual ascent procedure which solves a Lagrangean dual at each enumeration node and adds a surrogate constraint to the Lagrangean relaxed model. In the Branch and Bound algorithm, they use a branching scheme that combines depth-first and breadth-first strategies. The branching is based on the jobs with multiple assignments; the job with the maximum of the minimum resource usage is selected to branch on among the ones with multiple assignments. The authors report the satisfactory performance of their algorithm for the problems with up to 500 variables.

Another study on the Lagrangean relaxation of the problem is performed by Wilcox (1989), who relaxes the constraint set (3). The Lagrangean multipliers are adjusted so that the multipliers of the unassigned jobs are higher than those of the multiple assigned jobs. He compares the binary branching and multiple branching strategies and concludes that the multiple branching rule is better. Fixing the variables to zero and one, the size of the problem is reduced. The branching is on the job with the largest number of fixed variables. The problem sizes are up to 5 agents and 40 jobs. When compared to the approach of Fisher, Jaikumar and Van Wassenhove (1986), the algorithm is reported to be faster and results in smaller tree sizes.

Two algorithms are proposed by Jörnsten and Värbrand (1987) for the GAP. The first one strengthens the bound obtained from the Lagrangean relaxation of the constraint set (2), via valid inequalities. When no valid inequality can be generated, a Branch and Bound procedure is used. The second procedure uses a surrogate relaxation of constraint set (2) and valid inequalities. The first procedure is reported to be more efficient. Their test problems have 4 agents and 25 jobs.

Karabakal et al. (1992) propose a more effective multiplier adjustment method than the ones used by Fisher, Jaikumar and Van Wassenhove (1986) and Guignard and Rosenwein (1989a) to solve the Lagrangean relaxation of the

problem. They relax the constraint set (3). The multiplier adjustment method suggested by Fisher, Jaikumar and Van Wassenhove (1986) and then improved by Guignard and Rosenwein (1989a) is improved by a post optimality analysis of the 0-1 knapsack subproblems. This method is embedded in a Branch and Bound algorithm. They use the branching strategy suggested by Bean (1984). A violated constraint (from constraint set 3) with the largest multiplier value is selected. Their computational results show the superiority of the algorithm over that of Martello and Toth (1981a).

Savelsbergh (1997) proposes a branch and price algorithm which uses both column generation and Branch and Bound techniques. The problem is formulated as a set partitioning problem. Branching strategies based on variable fixing are shown to be suitable to the pricing. His computational study includes problems with up to 20 agents and 50 jobs. He compares his results with those of Karabakal et al. (1992) and finds that his algorithm performs better for problems with a relatively small *n/m* ratio (smaller than 5) while the other one performs better for problems that have higher *n/m* ratio. The author concludes that these two algorithms are good complements of each other.

Park et al. (1998) propose a Lagrangean-Dual-Based Branch and Bound Algorithm for the Generalized Multi Assignment Problem, of which the GAP is a special case. For the GAP they compare the performance of their algorithm to those of Guignard and Rosenwein (1989a), Martello and Toth (1990). The authors conclude that their algorithm outpaces the others in terms of problem sizes that can be solved to optimality. They also conclude that as the *n/m* ratio gets bigger, the corresponding GAP becomes harder to solve.

Cattrysse et al. (1998) discuss an improvement on the standard procedure for generating lifted cover inequalities that yields good upper bounds. Using this improvement they propose two heuristic procedures. They also use some pre processing techniques to reduce the size of the *B&B* tree. These techniques along with the proposed bounds are used in a *B&B* algorithm. They report

computational results for problems with $m \in \{5, 8, 10\}$ and n/m ratio $\in \{3, 4, 5, 6\}$ and show that the Branch and Bound algorithm outperforms the *B&B* algorithm of Martello and Toth (1990) in terms of average CPU time. It is noted that this satisfactory performance is achieved by the cuts and size reduction techniques.

Farias and Nemhauser (2001) discuss a family of inequalities that are valid for the GAP polytope. The proposed inequalities are used in a branch and cut algorithm and computational results indicate a %53 reduction in the number of nodes and a %66 reduction in the CPU time by the proposed inequality. They compare their algorithm with the one proposed by Savelsbergh (1997) and conclude that their proposed algorithm is superior in terms of the computation time and the problem sizes that could be solved to optimality.

Nauss (2003) proposes a Branch and Bound algorithm in which linear programming cuts, feasible solution generators, Lagrangean relaxation and subgradient optimization methods are used. His computational results show that that the algorithm outperforms the Savelsbergh's algorithm especially in terms of the CPU time. Another comparison is done with CPLEX 6.6 on small sized problems and the proposed algorithm is reported to solve the test problems about 3.5 times faster. Nauss mentions that the algorithm reaches good feasible solutions at early stages, hence, could be used as a heuristic when the guarantee of optimality is not essential.

Haddadi et al. (2004) work on a Branch and Bound algorithm using a breadth first approach and selecting the node with the largest upper bound for branching. Lagrangean Relaxation which relaxes constraint set (2) is used. A standard subgradient method is used to solve this Lagrangean dual. At each iteration of the subgradient method, a heuristic is used. The results of their computational study reveal their Branch and Bound algorithm outperforms the Nauss's (2003) algorithm and their Lagrangean heuristic outperforms the tabu search algorithm by Yagiura et al. (1999). The Lagrangean heuristic and the Branch and Bound

algorithm are reported to be better than the tabu search heuristic in terms of speed and both speed and accuracy, respectively. For the tested instances the proposed algorithm is shown to be better than the Branch and Bound algorithm of Nauss (2003).

Pigatti et al. (2005) propose a branch and cut and price algorithm with a stabilization mechanism to improve the convergence of column generation. The authors also propose ellipsoidal cuts which are reported to produce good upper bounds for the branch and cut and price algorithm.

Avella et al. (2008) introduce a cutting plane algorithm and perform a computational study of exact knapsack separation for the GAP. The proposed algorithm is tested on the problems taken from the OR-Library. The solutions for problems with up to 80 agents and 1600 jobs are reported. The authors report that they could solve four 'hard' instances that were reported as unsolved.

Approximation Algorithms

Some noteworthy approximation algorithms on the GAP are due to Martello and Toth (1981a, 1981b), Cattrysse (1990), Cattrysse, Salomon and Van Wassenhove (1994), Osman (1995), Klastorin (1979), Jörnsten and Näsberg (1986), Jörnsten and Värbrand (1991), Trick (1992), Savelsbergh (1997), Hallefjord, Jörnsten and Värbrand (1993), Amini and Racer (1994, 1995), Racer and Amini (1994), Osman (1995), Chu and Beasley (1997), Yagiura et al. (1998, 2006), Higgins (2001), Díaz and Fernández (2001) and Lourenço and Serra (2002). Osman (1995) provides a thorough review of the heuristic methods designed for the GAP.

Martello and Toth (1981b) propose a greedy heuristic that determines the job with the maximum regret and assigns it to the agent that leads to maximum profit. Another version is discussed by the same authors, which assigns the job to the agent for whom the regret is minimum (1981a). The heuristic continues with an

improvement step using a shift procedure. The performance of the heuristic algorithm is tested on problems with up to 20 agents and 200 jobs. The solution is compared with the optimal solution given by the Ross and Soland (1975) algorithm and 0.1% average deviation from the optimal is reported.

Cattrysse (1990) proposes a variable fixing procedure that can be used to reduce the problem sizes. First the Linear Programming (LP) relaxation of the problem is solved and then violated valid inequalities are added to the formulation. The resulting formulation is solved and other valid inequalities are added. This continues until no further valid inequalities can be found. After this procedure, the variables that are found as 1 are fixed to 1 and capacities of the agents are updated accordingly. The resulting problem which is of a smaller size is solved by using a Simulated Annealing (*SA*) procedure. The authors report computational results for problems with up to 10 agents and 60 jobs. The Simulated Annealing algorithm is found to deviate no more than 3.9% from optimality on average. The fixing procedure finds solutions that are 0.72% apart from optimality and reduces the solution times.

Cattrysse (1990) and Cattrysse, Salomon, Van Wassenhove (1994) study a set partitioning heuristic for the GAP. The heuristic proposed in Cattrysse, Salomon, and Van Wassenhove (1994) uses column generation techniques and provides both upper and lower bounds. The authors report an average deviation of 0.13% from optimality.

Osman (1995) imposes a time limit on the execution of the exact algorithms of Martello and Toth (1990) and Fisher and Jaikumar (1981). In this method, depth search is applied first and once a feasible solution is found a tree search is implemented.

The heuristics proposed by Klastorin (1979), Jörnsten and Näsberg (1986) and Jörnsten and Värbrand (1991) use the Lagrangean relaxation and try to obtain feasible solutions based on the results of the relaxation.

Klastorin (1979) proposes a two phased Lagrangean relaxation based heuristics using modified subgradient and Branch and Bound approaches. A computational experience is cited for problems containing up to 12,000 binary variables.

Jörnsten and Näsberg (1986) use Lagrangean decomposition bounds and obtain feasible solutions by modifying them. The violated capacity constraints are handled by interchanges and the resulting feasible solution is improved by reassignments of jobs from one agent to another. Jörnsten and Värbrand (1991) obtain feasible solutions based on the Lagrangean lower bound at every node of the tree search. The results are reported for problems with 4 agents and 25 jobs.

A Linear Programming relaxation based heuristic for the GAP is proposed by Trick (1992). The author first eliminates the assignments for which the job's requirement is greater than the agent's capacity. The LP relaxation is solved and the variables that received value 1 in the relaxed solution are fixed to 1. These jobs are deleted and the agent capacities are updated accordingly. These steps are repeated until no variables are left. This algorithm is followed by an improvement procedure that swaps jobs of two agents or assigns a job to an agent different than the currently assigned one. This heuristic is shown to be consistent when compared to Martello and Toth (1990)'s heuristics. The results are reported for problems with up to 100 agents and 500 jobs. The proposed heuristic is reported to outperform Martello and Toth (1990)'s heuristics.

Savelsbergh (1997) discusses the performance of the truncated tree search algorithms. The first heuristic is based on setting a predefined limit, μ, to the number of nodes and the second one is based on using an optimality tolerance, α such that the nodes having $Z_{LP} \leq (1+\alpha) * Z_{IP}$ are fathomed. He concludes that the proposed heuristics outperform the linear relaxation heuristic of Trick (1992) in terms of solution quality with an acceptable increase in computation time. When α is set to 0.005, a significant increase in computation time is reported.

Hallefjord, Jörnsten and Värbrand (1993) propose an algorithm based on the idea of partitioning the jobs into clusters. The jobs are grouped via a hierarchical cluster analysis. The resulting aggregated GAP is solved to optimality and the optimal aggregated solution is then disaggregated to obtain a feasible solution to the GAP. The authors solve two problems of sizes 4 agents- 25 jobs and 4 agents-1000 jobs.

Amini and Racer (1994) and Racer and Amini (1994) develop a variable-depth search heuristic (VDSH) for the GAP. The proposed algorithm is a two phase algorithm. The first phase generates an initial feasible solution and an LP based lower bound. The second phase is a refinement phase in which a job is assigned to another agent or two jobs assigned to different agents are interchanged while ensuring feasibility. The authors compare the results of their heuristic to that of Martello and Toth (1990) on problem instances with 5-20 agents and 50-200 jobs. They report solutions that are closer to optimality however at an expense of increased solution times.

Osman (1995) studies the implementation of local search descent, hybrid Simulated Annealing / Tabu Search *(SA/TS)* and *TS* methods. He compares the performances with those of the best reported algorithms by Cattrysse (Simulated Annealing) (1990), Cattrysse et al. (Set Partitioning Heuristic) (1994), Fisher et al. (1986) (curtailed *B&B*), and Martello and Toth (1990) (curtailed *B&B*). The computational results with up to 10 agents and 60 jobs reveal that *SA* and *TS* outperform the other heuristics in terms of the solution quality and time. Also the local search descent method is found to be much faster than *SA*/*TS* and *TS* mechanisms and recommended when computation time is a limiting factor.

Amini and Racer (1995) propose a hybrid heuristic that combines greedy heuristic and a refinement phase. Local optimal solutions are avoided by allowing chain moves. They show that the problem instances with up to 200 jobs and 20 agents are solved in 30 seconds on average.

Chu and Beasley (1997) develop a Genetic Algorithm *(GA)*. The performance of the algorithm is compared to those of *SA* heuristic of Cattrysse (1990), the set partitioning heuristic of Cattrysse, Salomon and Van Wassenhove (1994) and the *SA*/*TS* heuristic of Osman (1995). Their computational study shows that the average deviation from optimality is 0.01% and the computational times are compatible with those of other heuristics.

Yagiura et al. (1998) introduce a variable depth search *(VDS)* algorithm with branching search. The algorithm is a more sophisticated version of the *VDS* algorithm developed by the same authors. The authors compare their heuristic with the heuristics by Yagiura et al. (1997), *VDS* by Racer and Amini (1994), tabu search by Laguna et al. (1995), tabu search for the general purpose constraint satisfaction problem by Nonobe and Ibaraki (1998) on the test problems with up to 20 agents and 200 jobs. The solution quality of the proposed algorithm is reported to be better than the existing algorithms in most cases.

Higgins (2001) introduces new versions of the *TS* algorithm. The algorithm applies dynamic oscillation and changes the size of the neighborhood sample as time progresses. The new version is compared with the three existing versions of *TS* for the test problems with up to 50000 jobs and 40 agents. In time limit of 10 minutes, the new version is reported to outperform the others in terms of the solution quality. In order to compare the computation times of the algorithms, the author runs the proposed version for 2 minutes, records the solution, and observes the time for the other versions to reach the quality of this solution. The existing versions are reported to require 1.5-3 times more time than the proposed one for the same solution quality.

Díaz and Fernández (2001) develop a tabu search heuristic. The performance of the algorithm is tested on problems with up to 40 agents and 400 tasks. The proposed algorithm is reported to provide good solutions in competitive computational times with *SA*/*TS* and *TS* heuristics proposed by Osman (1995) and the *GA* proposed by Chu and Beasley (1997).

Lourenço and Serra (2002) apply adaptive search method for the GAP based on GRASP and Ant Colony optimization. Their hybrid approach includes ideas from Max Min Ant Systems (MMAS) and GRASP and combines them with tabu search techniques. The results compare favorably with MMAS and the greedy randomized adaptive heuristics both in terms of time taken and quality of the solution.

Yagiura et al. (2006) introduce a metaheuristic that includes path relinking approach along with an ejection chain approach. The authors test the performance of their algorithm on the instances with up to 20 agents and 200 jobs. They compare their results with those of Alfandari et al.(2002), Díaz and Fernández (2001), Haddadi and Ouzia (2001), Yagiura et al. (2004), Racer and Amini (1994), Laguna et al. (1995), MAX–MIN ant system by Lourenço and Serra (2002), Chu and Beasley (1997) and a mixed integer programming solver CPLEX 6.5. The proposed algorithm is reported to be superior in most instances.

2.2. The Bottleneck GAP

The bottleneck GAP (BGAP) is first mentioned by Francis and White (1974) and is first defined by Mazzola and Neebe (1988). There are two versions of the problem: task based and agent based.

The Task BGAP (TBGAP) minimizes the maximum cost of the assignments. The associated formulation for the Task BGAP is as follows:

$$z = \min\{ \max_{\substack{i\in I \\ j\in J}} \{c_{ij} x_{ij}\}\} \qquad (1)$$

$$s.t. \quad \sum_j a_{ij} x_{ij} \le b_i, \quad i \in I, \qquad (2)$$

$$\sum_i x_{ij} = 1, \quad j \in J, \qquad (3)$$

$$x_{ij} = 0 \text{ or } 1, \quad i \in I, \quad j \in J, \qquad (4)$$

The Task BGAP is shown to be NP hard in the strong sense based on the fact that GAP is so (Martello and Toth, 1995).

Mazzola and Neebe (1988) discuss solution procedures to solve the Task BGAP. Through an appropriate transformation they formulate the problem as a minisum GAP, which can be used to solve the corresponding TBGAP when the number of distinct c_{ij} values is small. The authors also discuss an iterative procedure to solve the Task BGAP and provide an example to illustrate the procedure. No computational results are reported for the procedures.

Mazzola and Neebe (1993) propose an algorithm for the Task BGAP that uses the procedures discussed in Mazzola and Neebe (1988). They use an equivalent formulation of the problem and propose the so called iterative TBGAP algorithm to solve this modified version. The authors find optimal solutions to the instances with up to 5 agents and 20 jobs.

Martello and Toth (1995) discuss the relaxations to the TBGAP. They find bounds by relaxing the resource constraints and applying a surrogate relaxation. They propose an approximate algorithm that finds a feasible solution in less than a given threshold. They define a Branch and Bound algorithm that uses the results of the approximate algorithms. The search strategy used is depth-first search and the branching is done by assigning the selected job to all feasible agents. A simple mechanism is used to fathom the nodes, if the mechanism fails to fathom the node the relaxations are applied in a sequence from weakest to strongest. The authors report computational results of both the exact and approximation algorithms for problems with up to 50 agents and 1000 jobs.

The Agent BGAP (ABGAP) minimizes the maximum of the total costs assigned to each agent. The formulation is as stated below:

$$z = \min\{ \max_{i \in I} \{ \sum_{j \in J} c_{ij} x_{ij} \} \} \qquad (1)$$

$$s.t. \quad \sum_{j} a_{ij} x_{ij} \le b_i, \quad i \in I, \qquad (2)$$

$$\sum_{i} x_{ij} = 1, \quad j \in J, \qquad (3)$$

$$x_{ij} = 0 \text{ or } 1, \quad i \in I, \quad j \in J, \qquad (4)$$

To the best of our knowledge there is no reported study on the Agent Bottleneck Generalized Assignment Problem.

2.3. The Multi Resource GAP (MRGAP)

The MRGAP is a generalization of the GAP with multi resources. The problem is formulated as follows:

Given

- A set of agents I = {1, …, m}
- A set of tasks J = {1,…, n}
- A set of periods T={1,…, s}
- b_{it}: Available capacity of agent *i* in period *t*
- p_{ijt}: Time required by task *j* in period *t* if performed by agent *i*.

Min $\quad z = \sum_{i=1}^{m}\sum_{j=1}^{n} c_{ij} x_{ij}$

Subject to

$$\sum_{i=1}^{m} x_{ij} = 1 \qquad \forall j = 1, ..., n \quad (1)$$

$$\sum_{j=1}^{n} p_{ijt} x_{ij} \le b_{it} \qquad \forall i = 1, ..., m, \quad \forall t = 1, ..., s \quad (2)$$

$$x_{ij} \in \{0, 1\} \qquad \forall i = 1, ..., m \ , \ \forall j = 1, ..., n$$

where

$$x_{ij} = \begin{cases} 1 \text{ if task } j \text{ is assigned to agent } i \\ 0 \text{ otherwise} \end{cases}$$

The assignment decision variables, x_{ij}'s are binary.

Constraint set (1) ensures that the capacities of the agents are not exceeded and constraint set (2) ensures that each task is assigned to one agent.

The studies about the MRGAP are summarized in the following table:

Table 2.1: Literature Review for the MRGAP

Title	Authors	Method	Publication Date
Algorithms for the MRGAP	Gavish and Pirkul	Optimization	Jun-91
Heuristics for the MRGAP	Mazzola and Wilcox	Optimization	Mar-01
A very large-scale neighborhood search algorithm for the MRGAP	Yagiura, Iwasaki, Ibaraki, Glover	Very large-scale neighborhood search	Mar-04
Local search intensified: Very large Scale Variable Neighborhood search for the MRGAP	Minic, Punnen	Very large-scale variable neighborhood search	Apr-09

Gavish and Pirkul (1991) study different Lagrangian relaxations of the problem and develop three heuristics. The first heuristic simply assigns the tasks that would result in high incremental costs if they were not assigned to the least costly agent. The second heuristic uses the solution of the Lagrangian Relaxation by the constraint set (2) as an initial solution and generates a feasible solution by reassigning some tasks.

The third heuristic obtains a feasible solution using the Lagrangian relaxation of constraint set (1). The computational results indicate that for difficult problems, the third heuristic dominates the others and it is used as a bounding scheme in their Branch and Bound algorithm. The bound is calculated when a free variable x_{ij} is fixed to $1\text{-}x_{ij}$*. A combination of three different rules are used in the algorithm. The first one uses a subgradient optimization algorithm to determine the Lagrangean multipliers and new bound. In the second method, the solution for the Lagrangean relaxation is found by using the most recent multiplier set and in the third one sensitivity analysis is used. The branching strategy is based on

selecting a job according to a predefined penalty measure and assigning this job to each agent in increasing order of the penalty values. The optimal solutions to problems with up to 10 agents and 100 jobs are reported.

Mazzola and Wilcox (2001) propose a three phased heuristic that first seeks to construct a feasible solution and then systematically improves the solution. In the first phase, a function based on predefined weight functions is used to calculate a "regret" value for a job. From the unassigned jobs, the job yielding the largest regret is assigned to the agent that minimizes the selection function. In the second phase, jobs from the agents that are overloaded are reassigned to other available agents according to a given priority measure. In the third phase, they propose a solution improving procedure which uses a feasible solution and improves it by using an Integer Programming model that maximizes the improvement of total cost when the jobs are shifted between agents. They also suggest a modification for the heuristic by Gavish and Pirkul (1991) to reduce its computational burden. A hybrid heuristic using this modified heuristic and the three phased heuristic is discussed. The hybrid heuristic is reported to be the most effective of all with an average deviation of less than 3% from the optimal solution for the problems with up to 10 agents, 75 jobs and 4 resource types.

Yagiura et al. (2004) work on a very-large scale neighborhood search algorithm based on tabu search which they call as *TS-CS* (tabu search with chained shift neighborhood). The authors provide computational results for problems with up to 20 agents, 200 jobs and 8 resource types. The optimal solution by alternative *TS* applications are compared with each other and with CPLEX 6.5. They find that *TS-CS* performs better for majority of the problem instances.

Mitrović-Minić and Punnen (2009) develop a very large scale variable neighborhood search *(VNS)* algorithm for the MRGAP. The basic idea is as follows:

- Start with a feasible solution.

- Divide the assignments of the solution into two parts, namely S and S', fix the assignments of the tasks in set S and solve a relatively smaller Integer Program to find the optimal allocation of the tasks of set S'.
- The new solution that includes the optimal assignments for S' set replaces the current feasible solution if it has a better objective function value.

The most important decision involved is the size of set S, i.e., |S|. For small |S|, searching the neighborhood is almost equivalent to solving the MRGAP itself and for large |S|, the neighborhood is weak. The authors propose nine ways to select |S|. They suggest to start with large |S|, and decrease it gradually, resulting in a very large scale *VNS*. They use the test problems of Yagiura et al. (2004) and compare the performance of their algorithm with *TS* of Yagiura et al. (2004) and with the best solution found by CPLEX in a predefined time limit.

In this study we consider an Agent Bottleneck MRGAP (MRABGAP). To the best of our knowledge, there is no reported study on this problem.

CHAPTER 3

PROBLEM DEFINITION

An Agent Bottleneck MRGAP (MRABGAP) is an Agent Bottleneck Generalized Assignment Problem (ABGAP) with multiple types of resources. We assign the agents to the job opportunities while obeying the agent capacities and assigning each job to exactly one agent. These assignments are valid for a horizon that includes multiple equal-length periods, hence multi resources. That is, in the MRABGAP we seek to find the assignment that balances total agent loads. We achieve load balancing by minimizing the maximum total load over all agents. In our case, the time capacity of the agents is the single physical resource type, but the problem has direct analogy with the multi-resource problem as available capacity changes from period to period for an agent. Unlike the other problem types we do not have any cost concern; we only try to find an optimal balance in the total workloads of agents.

The Generalized Assignment Problem (GAP) is strongly NP-hard, since even its feasibility question is NP-complete (Martello and Toth, 1995). So is any generalization of the GAP. The MRABGAP is a generalization of the GAP with multiple resources and has the same feasible region. This follows that the feasible region of the MRABGAP is strongly NP-complete and the optimality problem is NP-hard in the strong sense.

We next give the mathematical model of the MRABGAP.

3.1. Mathematical Model of the MRABGAP

Parameters

b_{it}: Available capacity of agent i in period t.

p_{ijt}: Time required by task j in period t if performed by agent i.

Decision Variables

The binary variables, x_{ij}s,define the assignments such that

$$x_{ij}= \begin{cases} 1 \text{ if task } j \text{ is assigned to agent } i \\ 0 \text{ otherwise} \end{cases}$$

Constraints

Given

- A set of agents I = {1, …, m}
- A set of tasks J = {1, …, n}
- A set of periods T={1, …, s}

$$\sum_{j=1}^{n} p_{ijt} x_{ij} \le b_{it} \qquad \forall i = 1,...,m, \quad \forall t = 1,...,s \qquad (1)$$

$$\sum_{i=1}^{m} x_{ij} = 1 \qquad \forall\ j = 1,...,n \qquad (2)$$

$$x_{ij} = 0 \text{ or } 1 \qquad \forall i = 1,...,m\ ,\ \forall\ j = 1,...,n \qquad (3)$$

Constraint set (1) ensures that the capacities of the agents are not exceeded and constraint set (2) ensures that each task is assigned to one agent. The assignment restrictions are given by constraint set (3).

Objective Function

The MRABGAP minimizes the maximum of the total loads assigned over all agents. The objective function is as follows:

$$Min \underset{i}{Max}\left\{\sum_{t=1}^{s}\sum_{j=1}^{n} p_{ijt}x_{ij}\right\}$$

We can linearize this objective function by defining a continuous variable, *Z*, which denotes the minimum of the maximum total load over all agents. We add the following constraint set to the formulation:

$$\sum_{j=1}^{n}\sum_{t=1}^{s} p_{ijt}x_{ij} \le Z \quad \forall i = 1,...,m$$

This constraint set defines a bottleneck type objective function that returns the maximum of the total load over all agents.

Below is the linearized formulation of the MRABGAP:

(P)

Min *Z*

$$\sum_{j=1}^{n} p_{ijt}x_{ij} \le b_{it} \qquad \forall i = 1,...,m, \quad \forall t = 1,...,s \quad (1)$$

$$\sum_{i=1}^{m} x_{ij} = 1 \qquad \forall\ j = 1,...,n \quad (2)$$

$$\sum_{j=1}^{n}\sum_{t=1}^{s} p_{ijt}x_{ij} \le Z \quad \forall i = 1,...,m \quad (3)$$

$$x_{ij} = 0 \text{ or } 1 \qquad \forall i = 1,...,m\ ,\ \forall\ j = 1,...,n \quad (4)$$

Size of the model

Number of constraints

Given *m* agents, *n* jobs and *s* periods the number of constraints is *m*s + n+ m*, excluding the set constraints.

Number of decision variables

Binary Variables: Given *m* agents and *n* jobs the number of binary variables (x_{ij}s) is *m*n.*
Continuous Variables: There a single continuous variable (*Z*).

An Example Problem

Consider the following example problem with the parameters provided in Tables 3.1 and 3.2.

Table 3.1: b_{it} Values of the Example Problem

Agent	Period 1	Period 2
1	18	16
2	16	16
3	40	30
4	23	22
5	16	15

Table 3.2: p_{ijt} values of the Example Problem

	Period 1										Period 2									
Agent/Job	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	10	10	7	6	15	12	7	11	7	12	8	10	5	5	14	9	4	12	6	11
2	6	9	11	5	11	14	8	8	11	7	5	11	10	4	11	11	9	9	12	6
3	11	5	12	14	10	14	12	11	8	12	11	4	12	11	10	14	14	10	8	14
4	14	14	11	12	5	8	15	14	12	12	13	14	11	9	5	7	16	14	11	9
5	11	6	14	7	5	8	8	9	9	7	13	6	12	6	4	7	8	9	10	7

A feasible solution to the problem is as follows:

$x_{21}=x_{32}=x_{43}=x_{14}=x_{35}=x_{46}=x_{17}=x_{38}=x_{59}=x_{210}=1$ and all the other x_{ij}s are 0. The bottleneck agent is agent 3 with a total load value of 50. The total loads of agents is demonstrated in Figure 3.1.

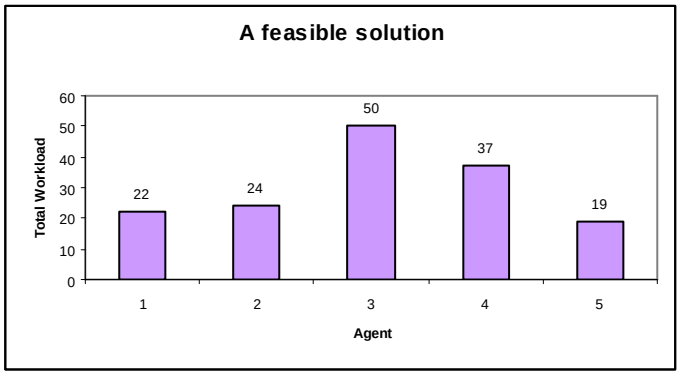

Figure 3.1: The Total Loads of the Agents for a Feasible Solution

An optimal solution to the problem is as follows:

$x_{21}=x_{32}=x_{13}=x_{54}=x_{45}=x_{46}=x_{17}=x_{28}=x_{39}=x_{510}=1$ and all other x_{ij}s are 0. Agent 2 is the bottleneck agent and the optimal solution value, i.e., the total load of agent 2, is 28. Below is a graphical representation of the total loads of the agents.

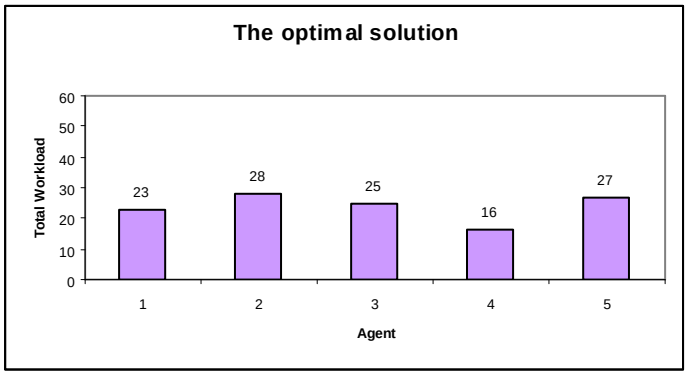

Figure 3.2: The Total Loads of the Agents in the Optimal Solution

3.2. Properties of Solutions

Properties of Feasible Solutions

1. For each agent, total work should be less than or equal to an upper bound (*UB*).

$$\sum_j \sum_t p_{ijt} x_{ij} \le RHS_i \qquad \forall i \qquad (1)$$

$$RHS_i \equiv Min \left\{ UB, \sum_t b_{it} \right\}$$

Note: If $RHS_i = \sum_t b_{it}$ (1) would be redundant.

If $RHS_i = UB$ such that $UB < \sum_t b_{it}$; (1) may be used as a valid cut.

Problem Size Reduction Properties

One can use the following rules before using an optimization algorithm in order to fix some variables to 0 or 1; hence reduce the problem size beforehand.

1.

a. If the time required for agent *i* to perform job *j* in period *t* is greater than the maximum available time agent *i* has in a period over all the periods, then the corresponding variable x_{ij} should be 0.

If $p_{ijt} > \underset{t}{Max} \{b_{it}\}$ then $x_{ij} = 0$ (1.a)

Otherwise the feasibility condition stated by (1) is violated.

b. If the time required for agent *i* to perform job *j* in period *t* is greater than the maximum available time for that period over all the agents, then the corresponding variable, x_{ij} should be 0.

If $p_{ijt} > \underset{i}{Max}\{b_{it}\}$ then x_{ij} =0 (1.b)

Otherwise the feasibility condition stated by constraint set (1) is violated.

c. If the time required for agent *i* to perform job *j* in period *t* is greater than the available time of the agent for that period *t*, then the corresponding variable, x_{ij} should be 0.

If $p_{ijt} > b_{it}$ for any *i, j, t* then x_{ij} =0 (1.c)

Otherwise the feasibility condition stated by (1) is violated.

2. If the total time required for agent *i* to perform job *j* is greater than the total available time of agent *i* over all periods, then the corresponding variable x_{ij} is 0.

If $\sum_t p_{ijt} > \sum_t b_{it}$ for any *i* and *j* then $x_{ij} = 0$ (2)

Otherwise the feasibility condition stated by (1) is violated.

3. If minimum time required for agent *i* to perform any job is greater than the maximum available time of that agent for a period then no jobs are assigned to that agent in a feasible solution.

If $\underset{j,t}{Min}\ p_{ijt} > \underset{t}{Max}\ b_{it}$ for *i* then $\sum_j x_{ij} = 0$ for agent *i* (3)

Otherwise the feasibility condition stated by (1) is violated.

4. If minimum time required for agent *i* to perform any job in period *t* is greater than the available time of the agent in that period then no jobs are assigned to that agent in a feasible solution.

If $\underset{j}{Min}\, p_{ijt} > b_{it}$ for any t then $\sum_j x_{ij} = 0$ for agent i (4)

Otherwise the feasibility condition stated by (1) is violated.

5. The rules 1.a, 1.b, 1.c and 2 can be easily modified for eliminating agents from further considerations as follows:

If $x_{ij} = 0$ holds for all j for an agent i then $\sum_j x_{ij} = 0$ (5)

Otherwise the feasibility condition stated by (1) is violated.

6. If job j cannot be assigned to any agent but agent k then the corresponding variable x_{kj} should be 1 in a feasible solution.

If $x_{ij} = 0$ for all agents but one agent k then $x_{kj} = 1$ (6)

Otherwise constraint set (2) is violated.

Comparison of the Rules

1. Rule 0 and rule 2 are not substitutable, checking one of them does not imply checking the other.

2. Rule 1.c, which already implies 1.a and 1.b, is a stronger rule. 1.a and 1.b are different rules but they are weak rules to be used in practice.

3. Rule 1.c already implies rule 2.

4. Instead of using $\sum_t b_{it}$ it is always possible to use an upper bound (UB) such that $UB < \sum_t b_{it}$ to make the corresponding rules stronger.

Some Rules to Detect Infeasibility

One can use the following rules to detect infeasibility beforehand. One should note that the following checks for infeasibility are one-sided. If the conditions are satisfied then the problem is infeasible. However if the conditions are not satisfied, feasibility is not guaranteed. As the decision version of the problem is NP-complete, there cannot be any rule that detects the infeasibility.

0. If the minimum time needed to perform a job in a period is greater than maximum available time in any period *t*, then the problem is infeasible.

If $\underset{i}{Min}\{p_{ijt}\} > \underset{i}{Max}\{b_{it}\}$ for any *t* and *j* then the problem (P) is infeasible (0)

1. If condition (1.a) holds for all *i* for at least one *j* then job *j* cannot be assigned to any agent, hence (P) is infeasible.

2. If condition (1.b) holds for all *i* for any *j* and *t* then job *j* cannot be assigned to any agent, hence (P) is infeasible.

3. If condition (1.c) holds for all *i* for at least one *j* and *t* then job *j* cannot be assigned to any agent, hence (P) is infeasible.

4. If condition (2.b) holds for all *i* for at least one *j* then the problem is infeasible.

5. If the minimum total time needed to perform all jobs is greater than the total available capacity, then the problem is infeasible.

a. If $\sum_t \sum_j \underset{i}{Min}\{p_{ijt}\} > \sum_t \sum_i b_{it}$ then the problem is infeasible.

The above condition states that even all jobs are performed by their quickest agent in each period the total capacity over all periods is exceeded.

b. If $\sum_j Min_i\left\{\sum_t p_{ijt}\right\} > \sum_t \sum_i b_{it}$ **,** then the problem is infeasible.

The above condition states that even all jobs are performed by their quickest agent over *t* periods; the total capacity over all periods is exceeded.

6. If the minimum total time needed to perform all jobs in any period *t* is greater than the total available capacity for that period, then the problem is infeasible.

If $\sum_j Min_i\{p_{ijt}\} > \sum_i b_{it}$ **,** for any *t* then the problem is infeasible. (6)

Comparison of the Rules

1. Rule 5.b, which implies 5.a., is a stronger cut than rule 5.a.
2. Rule 1.c in problem size reduction part already implies rules 5.a and 5.b.

3.3. Lower Bounds

We develop three lower bounds on the optimal value of the MRABGAP. These are namely Lower Bound 1, Lower Bound 2 and Linear Programming Relaxation Based Lower Bound.

3.3.1. Lower Bound 1 (LB_1)

To find a lower bound on the optimal objective function value, we make the following relaxations:

The capacities of the agents are unlimited (hence remove constraint set (1))
The opportunities can be split between the agents (hence relax the integrality of x_{ij}s)

Moreover we use underestimation for the total load of each opportunity, i.e., hence we relax the time parameters.

The optimal solution to the problem after these relaxations, LB_1, provides a lower bound on the optimal objective function value, Z*.

$$LB_1 = \frac{\sum_j \underset{i}{Min} \left\{ \sum_t p_{ijt} \right\}}{m}$$

Theorem below states this result formally.

Theorem: LB_1 is a lower bound on Z^*.

Proof: Let TAT_j denote the total time job j takes at the optimal solution.

The total load of all jobs at the optimal solution is $\sum_j TAT_j$. We know that $TAT_j \geq \underset{i}{Min}\left\{ \sum_t p_{ijt} \right\}$, implying $\sum_j TAT_j \geq \sum_j \underset{i}{Min}\left\{ \sum_t p_{ijt} \right\}$. $\sum_j \underset{i}{Min}\left\{ \sum_t p_{ijt} \right\}$ is the total load if all the opportunities are performed by their fastest agent, i.e., total load of the opportunity is underestimated. When the integrality constraints on x_{ij} values are relaxed and capacity constraints are removed, the optimal solution divides the load $\sum_j \underset{i}{Min}\left\{ \sum_t p_{ijt} \right\}$ evenly between all agents, and give an objective function value of $\sum_j \underset{i}{Min}\left\{ \sum_t p_{ijt} \right\}/m$. Hence LB_1 is a valid lower bound on Z^* value, as it is an optimal solution to a problem where some parameters are underestimated and some constraints are relaxed.

If the lower bound solution is feasible, i.e., all opportunities can be performed by their fastest agents without violating the capacity constraints and with a bottleneck load of LB_1 units, it is optimal. Moreover if a feasible solution with Z value of LB_1 is achieved then its optimality can be warranted.

To evaluate the partial solution with set S of assigned opportunities and Z_i of load of agent i, we extend LB_1 as follows:

We calculate the minimum total load due to the unassigned jobs as in LB_1 and distribute it to the nonbottleneck agents until they are full up to the bottleneck load. Then, if remains, we distribute the load evenly to all m agents and obtain a lower bound for the optimal solution of the corresponding partial solution.

The lower bound is available by the following expression:

$$LB_1(S) = B_L + Max\left\{0, \frac{\sum_{j \notin S} Min_i\left\{\sum_t p_{ijt}\right\} - ((B_L * m) - \sum_{i \in S}\sum_t p_{ijt} x_{ij}(S))}{m}\right\}$$

where $B_L = Max_i\{Z_i\}$, i.e., the bottleneck load of partial solution S and $x_{ij}(S)$ is the value of the assignment variable in the partial solution.

$\sum_{i \in S}\sum_t p_{ijt} x_{ij}(S)$ is the current total load due to the assignments of the partial solution.

First, all the agents are filled up to the B_L value, assuming a total capacity of B_L*m and hence the unused capacity becomes $B_L*m - \sum_{i \in S}\sum_t p_{ijt} x_{ij}(S)$. If the remaining work is smaller than this value, i.e., $LB_1(S)$ is equal to B_L, the bottleneck will not change. But if it is higher, we distribute the excess load evenly over all agents, raising the bottleneck load to LB_1.

We next discuss further improvements of LB_1.

Strengthened versions of Lower Bound 1

Note that $LB_1(S)$ completely ignores the capacity constraints. We strengthen LB_1 by incorporating the capacity constraints, to some extent. These strengthened versions of $LB_1(S)$ are referred to as version 1 and version 2. The first one strengthens the bound by preventing the assignment of a job to an agent whose

aggregate capacity is not sufficient. The second one performs this check for each individual period and does not allow the assignment of a job to an agent if the available capacity of the agent is not sufficient to serve the job at any period *t*.

Strengthened Version 1

In finding the minimum time of a job, we exclude the agents that do not have enough total available capacity to process that job. As we improve our estimation on total loads while preserving other assumptions, strengthened version 1 dominates $LB_1(S)$. For a partial solution, the capacities are updated by considering the already assigned tasks.

This strengthened version is calculated via the following equation:

$$LB_1(S) = B_L + Max\left\{0, \frac{\sum_{j \notin S} \underset{i \mid \sum_t p_{ijt} \le TAC_i(S)}{Min} \left\{\sum_t p_{ijt}\right\} - ((B_L * m) - \sum_{i \in S}\sum_t p_{ijt} x_{ij}(S))}{m}\right\}$$

where B_L is the bottleneck value of the partial solution and $\sum_{i \in S}\sum_t p_{ijt} x_{ij}(S)$ is the current total load due to the assignments of the partial solution *S*.

$TAC_i(S)$ is the total available capacity of agent *i* in the partial solution.

One can prove easily that this version provides a lower bound for the optimal solution in a similar way as in $LB_1(S)$.

Strengthened Version 2

The second strengthened version of $LB_1(S)$ dominates the strengthened version 1. It differs from version 1 in the way the capacity constraints are incorporated. The second version excludes the agents that do not have available capacity in any

period *t*. For a partial solution, strengthened version 2 is calculated via the following equation:

$$LB_1(S) = B_L + Max\left\{0, \frac{\sum_{j \notin S} \underset{i|p_{ijt} \le cap_{it}(S)}{Min} \forall t \left\{\sum_t p_{ijt}\right\} - ((B_L * m) - \sum_{i \in S}\sum_t p_{ijt} x_{ij}(S))}{m}\right\}$$

where B_L is the bottleneck value of the partial solution at hand, $\sum_{i \in S}\sum_t p_{ijt} x_{ij}(S)$ is the current total load due to the assignments of the partial solution and $cap_{it}(S)$ is the available capacity of agent *i* for period *t* in the partial solution.

Numerical Example

We illustrate our lower bounds on an example problem with 5 agents, 10 jobs and 2 periods. The problem data are given in Tables 3.3, 3.4 and 3.5.

Table 3.3: b_{it} Values of the Example Problem

Agent	Period 1	Period 2
1	18	16
2	16	16
3	23	23
4	23	22
5	16	15

Table 3.4: p_{ijt} Values of the Example Problem

	Period 1										Period 2									
Agent/Job	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	10	10	7	6	15	12	7	11	7	12	8	10	5	5	14	9	4	12	6	11
2	6	9	11	5	11	14	8	8	11	7	5	11	10	4	11	11	9	9	12	6
3	11	5	12	14	10	14	12	11	8	12	11	4	12	11	10	14	14	10	8	14
4	14	14	11	12	5	8	15	14	12	12	13	14	11	9	5	7	16	14	11	9
5	11	6	14	7	5	8	8	9	9	7	13	6	12	6	4	7	8	9	10	7

Table 3.5: $\sum_t p_{ijt}$ Values of the Example Problem

Agent/Job	1	2	3	4	5	6	7	8	9	10
1	18	20	12	11	29	21	11	23	13	23
2	11	20	21	9	22	15	17	17	23	13
3	22	9	24	25	20	28	26	21	16	26
4	27	28	22	21	10	15	31	28	23	21
5	24	12	26	13	9	15	16	18	19	14

The minimum total time that jobs require and the corresponding agents are given in Table 3.6.

Table 3.6: Minimum Total Loads and Corresponding Agents of the Example Problem

Job	Min Total Load	Min Total Load Agent
1	11	2
2	9	3
3	12	1
4	9	2
5	9	5
6	15	2,4,5
7	11	1
8	17	2
9	13	1
10	13	2

At the root node, when all the jobs are unassigned the lower bound is calculated as follows:

LB_1 = (11+9+12+……+13+13)/5 = 23.8

Suppose that we have a partial solution *S*, where $x_{16}(S) = x_{22}(S) = x_{35}(S) = x_{43}(S) = x_{54}(S) = 1$. Jobs 1, 7, 8, 9 and 10, are unassigned and the lower bound on the remaining work will be found based on these jobs. Figure 3.3 gives the loads of the agents given the partial solution *S*. Note that the bottleneck value of the partial solution, B_L is 22 and the total load over all agents is 96.

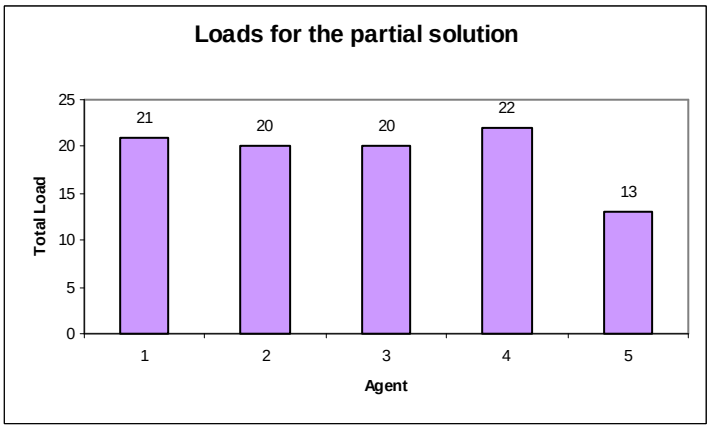

Figure 3.3: Total Agent Loads in a Partial Solution of the Example Problem

$LB_1(S)$ is calculated as follows:

$LB_1(S)$ = 22+ [(11+11+17+13+13)-((22*5)-(21+20+20+22+13))]/5

$LB_1(S)$ = 32.2

Figure 3.4 shows the estimated loads of the agents by lower bound 1.

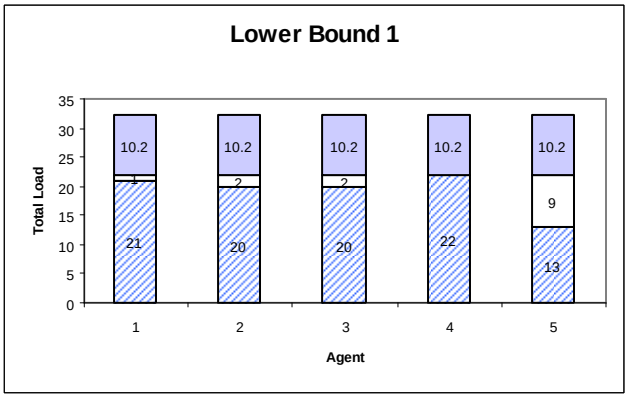

Figure 3.4: Lower Bound 1 Calculation for the Example Problem

The first strengthened version uses the total available capacity of the agents, tabulated below.

Table 3.7: Available Capacities of the Agents for the Example Problem

Agent	Available Capacity (S)
1	13
2	12
3	26
4	23
5	18

Based on these capacities we estimate our minimum total load. Agent 2 is the minimum load agent for the unassigned jobs 8 and 10; however the agent does not have enough capacity to serve them. Hence, the minimum total load estimations for jobs 8 and 10 are updated. The results are given in the Table 3.8. The changes are shown in bold.

Table 3.8: Minimum Total Load Estimation Updates for the Example Problem -Version 1

$LB_1(S)$			Strengthened version 1		
Job	Min. Total Load	Min. Total Load Agent	Job	Min Total Load	Min Total Load Agent
1	11	2	1	11	2
7	11	1	7	11	1
8	17	2	8	**18**	**5**
9	13	1	9	13	1
10	13	2	10	**14**	**5**

The new lower bound is calculated as follows:

$LB_1(S)$ = 22+ [(11+11+**18**+13+**14**)-((22*5)-(21+20+20+22+13))]/5

$LB_1(S)$ = 32.6

The second strengthened version uses the available capacity information of the agents for each period in the partial solution, which is given in Table 3.9.

Table 3.9: Available Capacities of Agents at each Period for the Example Problem

	$b_{it}(S)$	
Agent	**Period 1**	**Period 2**
1	6	7
2	7	5
3	13	13
4	12	11
5	9	9

Based on these available capacities we update our minimum total load estimations. The minimum total loads for jobs 7 and 9 are determined by the load of agent 1. However, note that agent 1 does not have sufficient capacity at period 1 to serve these jobs. Hence, agent 1 is not considered while estimating the minimum load for these jobs. Agents 5 and 3 determine the minimum total load estimations for jobs 7 and 9, respectively. The results are given in Table 3.10; the changes are shown in bold.

Table 3.10: Minimum Total Load Estimation Updates for the Example Problem -Version 2

Strengthened version 2		
Job	**Min Total Load**	**Min Total Load Agent**
1	11	2
7	**16**	**5**
8	18	5
9	**16**	**3**
10	14	5

The new lower bound is calculated as follows:

$LB_1(S)$ = 22+ [(11+**16**+18+**16**+14)-((22*5)-(21+20+20+22+13))]/5

$LB_1(S)$ = 34.2

Our pilot runs revealed that the strengthened version 2 does not increase the computational time significantly while providing the tightest bound value. Hence, we use it in the optimization algorithm as a bounding scheme and hereafter call it LB_1.

3.3.2. Lower Bound 2 (LB_2)

LB_1 assigns all opportunities to their minimum time agents. The minimum time agent for many opportunities may correspond to the same agent as the capacities of the agents are not considered in defining the assignments. LB_2 recognizes this fact and defines an upper bound on the number of opportunities to be assigned to each agent for a feasible solution. We let this upper bound be n_i for agent i and in defining the total load we select a maximum of n_i opportunities from agent i. We find n_i using the following procedure:

a. Find an upper bound on the objective function value, Z_{UB}.

b. Sort the $\sum_t p_{ijt}$ values from minimum to maximum for an agent i, such that $\sum_t p_{ijt} \leq \sum_t p_{ij+1t}$ for all j.

c. Find n_i such that, $\sum_{j=1}^{n_i} \sum_t p_{ijt} \leq Min\{Z_{UB}, \sum_t b_{it}\}$ and $\sum_{j=1}^{n_i+1} \sum_t p_{ijt} > Min\{Z_{UB}, \sum_t b_{it}\}$

As the minimum possible durations are considered, n_i is an upper bound on the number of opportunities that can be processed by agent i.

After n_is are found, we look for the number of opportunities that are assigned to agent i in $LB_1(S)$ computations. We let this number be r_i. Formally,

r_i = number of j's such that $\underset{i|p_{ijt} \leq cap_{it}\ \forall t}{Min} \left\{ \sum_t p_{ijt} \right\} = \sum_t p_{ijt}$.

If $\sum_i Max\{0, r_i\text{-}n_i\}$ is too big then LB_1 becomes a poor estimation. We set a threshold k and revise LB_1 if $\sum_i Max\{0, r_i\text{-}n_i\} > k$. In doing so, we assign $(\sum_i Max\{0, r_i\text{-}n_i\} - k)$ jobs to their second minimum agents. To ensure the validity of the lower bound the selected second minimum values are the ones that cause a minimum increase in the total load value. Having found a tighter lower bound on the total remaining workload, the lower bound is calculated like LB_1.

If the lower bound solution is feasible, i.e., all opportunities can be performed by their fastest or second fastest agents without violating the capacity constraints and with a bottleneck load of LB_2 units, it is optimal. Moreover if a feasible solution with Z value of LB_2 is achieved then its optimality can be warranted.

Numerical Example

We illustrate LB_2 computations on our previous example instance. Note that from now on we use the strengthened version 2 as LB_1.

When no upper bound is available, n_i values can be calculated according to $\sum_t b_{it}$ or $\sum_t b_{it}\ (S)$ for partial solution S, as follows:

For agent 1 the total processing times are 18, 20, 12, 11, 29, 21, 11, 23, 13 and 23. In the partial solution, jobs 2, 3, 4, 5 and 6 are assigned. The processing times of the unassigned jobs are 18, 11, 23, 13, 23 and the total available capacity of agent 1, i.e. $\sum_t b_{it}\ (S)$, is 13.

We now order the processing times of the unassigned jobs by agent 1.

11, 13, 18, 23, 23

$11 \leq 13$ and $11+13>13$, hence n_1 is 1.

The other n_i values are calculated accordingly and given together with the r_i values in the following table.

Table 3.11: n_i and r_i Values of the Example Problem

Agent(*i*)	n_i	r_i
1	1	0
2	1	1
3	1	1
4	1	0
5	2	1

$\sum_i Max\{0, r_i\text{-}n_i\}$=1, hence 1 job that causes minimum increase in the total load will be reassigned. The increase in the total load value when jobs are assigned to their second minimum load agents is calculated for each job. Note that while finding the second minimums we select from the agents that have sufficient capacity every period.

Table 3.12: Difference between Minimum and 2nd Minimum Loads

Job	Min Total Load	Sec. Min. Total Load	Difference
1	11	22	11
7	16	26	10
8	18	21	3
9	16	23	7
10	14	21	7

The job resulting in minimum increase is reassigned to its second minimum load agent. The minimum difference occurs when job 8 is reassigned. The lower bound for the minimum total load is now 78, hence 3 units more than LB_1. We calculate the LB_2 value as follows:

LB_2 = 22+ [(11+16+**21**+16+14)-((22*5)-(21+20+20+22+13))]/5

LB_2 = 34.8

3.3.3. Linear Programming Relaxation Based Bound (LB_3)

LB_3 is found by simply relaxing the integrality constraints on x_{ij} values and letting $0 \le x_{ij} \le 1$ for all i and j. LB_3 dominates LB_1 and LB_2 as it considers the capacity constraints and uses exact time values, unlike LB_1 and LB_2.

We propose a strengthened version of LB_3 that uses the upper bound on the maximum number of opportunities that can be assigned to an agent and incorporates it as a cut on the pure LP relaxation. The upper bound on the number of opportunities that can be assigned to an agent is found as in LB_2. This is then introduced to the LP relaxation by adding the following constraint set:

$$\sum_j x_{ij} \le n_i \qquad \forall i$$

where n_i is the maximum number of jobs that can be assigned to agent i.

We hereafter refer to this strengthened version as LB_3. We calculate LB_3 simply after using LB_1 and LB_2 for node elimination. If the node cannot be eliminated using LB_3, this bound is used in determining the node to branch on.

The LB_3 value calculated at the root node for the previous example is 24.9. Recall that the LB_1 value at the root node was 23.8, which indicates that LB_3 is stronger than LB_1.

CHAPTER 4

BRANCH AND BOUND ALGORITHM

Our preliminary runs on the Linear Programming (LP) relaxation of the problem have revealed that the average number of fractional variables at the optimal LP relaxation solution is relatively low. Hence we use LP relaxation in the Branch and Bound algorithm to define our branching scheme. We solve LP relaxation at every branch of the algorithm; use the results to select the variable to branch on and the optimal solution value of the relaxation as a lower bound.

4.1. Selection Strategy and the Branching Scheme:

At a node we find the job corresponding to the highest fractional variable of the LP Relaxation and select this job to branch on. Then the selected job is assigned in turn to each agent. For a selected job, *j*, the following *m* subproblems (nodes) are created:

Subproblem 1: $x_{1j}=1$

Subproblem 2: $x_{2j}=1$

Subproblem 3: $x_{3j}=1$

Subproblem 4: $x_{4j}=1$

.

.

Subproblem m: $x_{mj}=1$

The associated tree is shown in Figure 4.1.

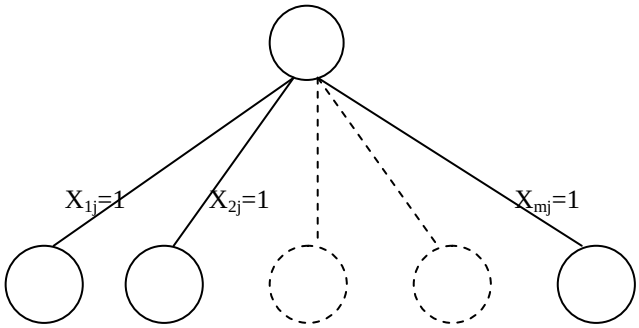

Figure 4.1: The Branching Scheme

We evaluate each of the *m* nodes using the elimination mechanisms discussed next.

4.2. Elimination Strategies

We fathom a node representing the assignment of opportunity *j* to agent *i* if one of the following conditions holds:

1. The node cannot lead to a feasible solution.

1.1. Given the assigned jobs of the partial solution, if the available capacity of agent *i* is not sufficient to serve job *j* for any period *t*, constraint set (1) is violated, the subproblem cannot lead to a feasible solution. Hence the associated node should be fathomed. Formally,

If $C_{it}(S) + p_{ijt} > b_{it}$, then x_{ij} cannot be set to 1. (1.1)

where $C_{it}(S)$ = load of agent *i* in period *t*.
Otherwise constraint set (1) is violated. ($x_{ij}(S)=0$).

1.2. This mechanism finds an upper bound on the number of jobs that each agent can serve. Assume the total available capacities of the agents are used to calculate the upper bound on the total number of jobs that all the agents can

serve. If this upper bound is less than the number of jobs, then constraint set (3) is surely violated, hence no feasible solution exists for the subproblem. Assume an upper bound on the objective function value, $Z_{UB,}$ is used for calculating the upper bound. If the upper bound on total number of jobs all the agents can serve is less than the number of jobs, then the subproblem cannot lead to a solution better than the incumbent solution. Hence, we fathom the node. Formally,

a. Find an upper bound on the objective function value, Z_{UB}.

b. Sort the $\sum_t p_{ijt}$ values from minimum to maximum for an agent i, such that $\sum_t p_{ijt} \leq \sum_t p_{ij+1t}$ for all j.

c. Find n_i such that,

$$\sum_{j=1}^{n_i} \sum_t p_{ijt} \leq Min\{Z_{UB}, \sum_t b_{it}\} \text{ and } \sum_{j=1}^{n_i+1} \sum_t p_{ijt} > Min\{Z_{UB}, \sum_t b_{it}\}$$

As the minimum possible durations are considered, n_i is an upper bound on the number of opportunities that can be served by agent i.

If $\sum_i n_i < n$ then fathom the node (1.2.)

If this mechanism cannot fathom the node, i.e., $\sum_i n_i \geq n$, then this information is introduced to the LP relaxation as a cut to strengthen its performance (see Section 3.3.3). In doing so, the following constraint set is added to the LP relaxation of the subproblem:

$\sum_j x_{ij} \leq n_i$, $\forall i$.

2. The node cannot lead to an optimal solution.

2.1. We check whether the best objective value will be exceeded if the corresponding branch is used. At each branch we assign a job j to an agent i.

After that assignment if the total load of the agent becomes no less than the upper bound (*UB*), we guarantee that if this variable is set to 1, the objective function value cannot be less than *UB*. Hence, the node is fathomed.

If $Z_i(S) + \sum_t p_{ijt} \geq UB$, then the assignment of job *j* to agent *i* raises the load of agent *i* above *UB*. Hence such an assignment cannot lead to a unique optimal solution. (x_{ij} *(S)*=0).

3. The optimal solution emanating from that node can be found easily.

3.1. On a branch that assigns a job *j* to an agent *i*, we first determine the set of unassigned jobs that can be assigned to agent *i*. If it is possible to assign all unassigned jobs to agent *i* without violating feasibility and still having total load for agent *i* no bigger than the bottleneck load of partial solution, *Z(S)*, then the optimality is guaranteed with an optimal value of *Z(S)*. We fathom the node by optimality. If all unassigned jobs are feasibly assigned to agent *i* but the bottleneck value is exceeded, we obtain an integer feasible solution; hence can use the load of agent *i* as an upper bound. Formally,

a. Determine S_i, set of jobs that can be assigned to agent *i* such that $S_i = \{j \mid p_{ijt} < b_{it}(S)$ for all $t\}$

b. If all the unassigned jobs are included in S_i and $\sum_t \sum_{j \in S_i} p_{ijt} + Z_i \leq \underset{k}{Max}\{Z_k\}$ then fathom the node as the optimal solution is found.

c. If all the unassigned jobs are included in S_i and $Z_{UB} > \sum_t \sum_{j \in S_i} p_{ijt} + Z_i > \underset{k}{Max}\{Z_k\}$ then update the upper bound such that $Z_{UB} = \sum_t \sum_{j \in S_i} p_{ijt} + Z_i$ *(S)*.

Our preliminary runs have revealed that this mechanism does not effect the computation time significantly. This is mostly due to the Branch and Bound algorithm's power in finding close optimal solutions quickly, thereby decreasing the effect of the mechanism.

Agent *i* is removed from all future considerations if one of the following conditions hold:

1. There cannot be any feasible assignment to agent *i*.

An agent is removed from all future considerations emanating from node *S*, if it cannot process any unassigned job. We first check whether $\sum_t b_{it}(S) < \underset{j}{Min}\left\{\sum_t p_{ijt}\right\}$, i.e., the total remaining capacity of agent *i* is not sufficient to process even the minimum total requirement over all unassigned jobs. If this condition is passed, we make a more precise check and test whether $b_{it}(S) < \underset{j}{Min}\left\{\sum_t p_{ijt}\right\}$, i.e., the remaining capacity of agent *i* for any period *t* is not sufficient to process even the minimum time required in that period. Formally we check,

If $\sum_t b_{it}(S) < \underset{j}{Min}\left\{\sum_t p_{ijt}\right\}$, then remove agent *i*.

If , $b_{it}(S) < \underset{j}{Min}\left\{\sum_t p_{ijt}\right\}$ for any period *t*, remove agent *i*.

2. The optimal assignment to agent *i* is available.

On a branch that assigns a job *j* to an agent *i*, we first determine the set of unassigned jobs that can be assigned to agent *i*. Even when all the jobs in this set are assigned to agent *i* the total load of agent is lower than the bottleneck load of the partial solution, we assign the jobs in this set to the agent and fix the

corresponding variables to 1, hence reduce the problem size. Since we have already assigned all possible (feasible) jobs to agent *i*, we remove this agent from all further considerations. Formally,

a. Determine S_i, set of jobs that can be assigned to agent *i* such that
$S_i = \{j \mid p_{ijt} < b_{it}(S)$ for all $t\}$

b. If $\sum_t \sum_{j \in S_i} p_{ijt} + Z_i \leq \underset{k}{Max}\{Z_k\}$ then assign the jobs in S_i to agent *i*, and remove agent *i* from further considerations.

For each unfathomed node, we first calculate LB_1. If LB_1 does not eliminate we calculate LB_2. If LB_2 cannot eliminate the node as well, we calculate the LP Relaxation bound, LB_3. We select the node having the smallest LB_3 value for further branching. If LB_3 gives a feasible solution then we fathom the node. We update the upper bound value if the solution is lower than the best solution value at hand. If all nodes are fathomed at any level then we backtrack to the previous level.

We next discuss an algorithm that we use to find an upper bound on the optimal solution of the problem.

Upper Bounds

In this section we discuss a heuristic procedure used for obtaining upper bounds. The upper bounds are either used as an initial solution in our Branch and Bound algorithm or as an approximate solution. Our upper bound is based on the LP relaxation; hence the algorithm runs in polynomial time. Recall that the decision version of our problem is NP-complete, which implies any heuristic procedure, hence ours, cannot guarantee feasibility. However, our preliminary runs have proved that this simple heuristic is very effective in finding feasible solutions.

Our heuristic has two phases. The first phase constructs an initial solution, if possible. This initial solution is obtained by modifying LP relaxation solution. If we can find a feasible solution better than the incumbent then the second phase that tries to improve the initial solution is performed. The improvement phase uses two steps. In the first step, one job from the bottleneck agent is reassigned to another agent and in the second one; two jobs are interchanged between the bottleneck agent and a non bottleneck agent. The change that results with the maximum improvement in the objective function value is performed. The phase continues until a predefined limit for the non improving moves or for the total number of moves is reached. The detailed explanation of these phases is provided in Section 5.1.

Example

We now demonstrate the branching scheme on an example problem with 5 agents and 10 jobs.

Suppose that we branch from node *n*. In the corresponding LP Relaxation solution the job with the highest fractional assignment is job 5. The best solution value at hand, *UB*, is 110. We first create 5 branches as shown in Figure 4.2.

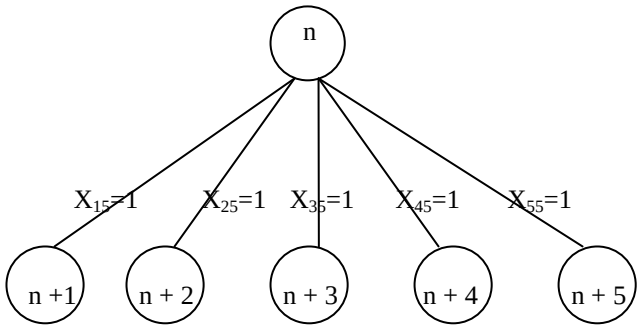

Figure 4.2: Branching Scheme for the Example Problem-1

For each branch generated, we apply the following procedure:

1. Check rules 1.1 and 2.1. If the node can be fathomed go to the next branch at the same level. If not, continue.

2. Calculate LB_1 and check whether $LB_1 > UB\text{-}1$. The objective function value, $Min\, Max_i \left\{ \sum_{t=1}^{u} \sum_{j=1}^{n} p_{ijt} x_{ij} \right\}$, is integer as we assume integer p_{ijt} values. Hence if $LB_1 > UB\text{-}1$ the node can be fathomed unless we are after alternative optimal solutions. If the node is fathomed, go to the next branch and start with step 1. Else, continue.

3. Check elimination rule 1.2. If the node is fathomed by this rule, go to the next branch and start with step 1. Else, continue.

4. Calculate LB_2 if $\sum_i Max\{0, r_i\text{-}n_i\} > k$, where k is a predefined threshold parameter. (We take k as 1 in our runs). Check whether $LB_2 > UB\text{-}1$. If the node is fathomed, go to the next branch and start with step 1. If not, continue.

5. Finally calculate LB_3. Check whether $LB_3 > UB\text{-}1$. If the node is fathomed, go to the next branch and start with step 1. If not, check whether the solution is feasible. If the solution is feasible update the upper bound, i.e., set $UB = LB_3$ and fathom the node. Else, keep the lower bound and the job with the most fractional variable in memory; go to the next branch and repeat this procedure.

After this procedure is applied to all nodes of the same level, we may end up with two outcomes. Either all the nodes are fathomed or we have at least one unfathomed node. If we have unfathomed nodes, the one with the lowest LB_3 value is selected to branch on. Since we keep the job with the highest fractional

assignment variable, we branch the selected node based on this job and open 5 (*m*) branches and repeat the above procedure for these new branches. If all the nodes at the same level are fathomed, we backtrack to the nearest level and find the open node with the lowest LB_3 value to branch on.

Suppose that after applying the procedure for all the branches we end up with the following situation:

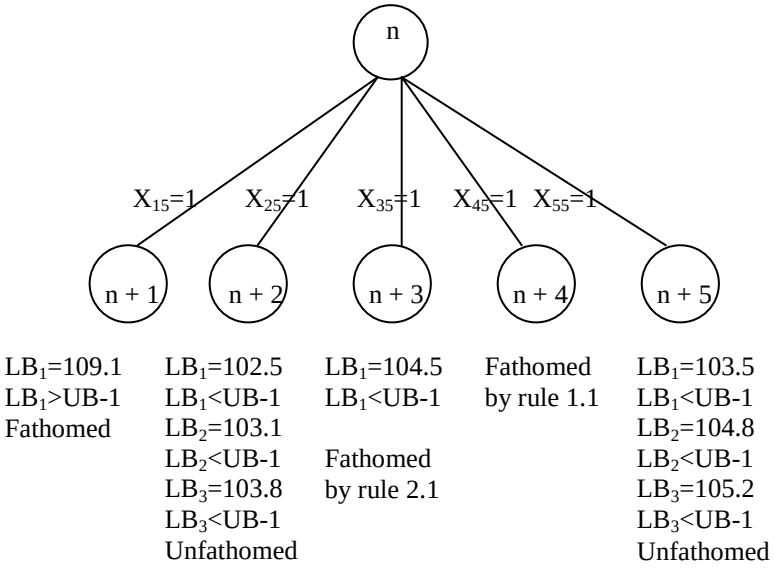

Figure 4.3: Branching Scheme for the Example Problem-2

As seen from the above figure, there are two unfathomed nodes, node *n*+2, and *n*+5 with respective LB_3 values of 103.8 and 105.2. Node *n*+2 has smaller lower bound; hence it is selected to branch on. Suppose that the job with the highest assignment value is 9 at the LP relaxation solution. We create the next branches as shown in Figure 4.4 below.

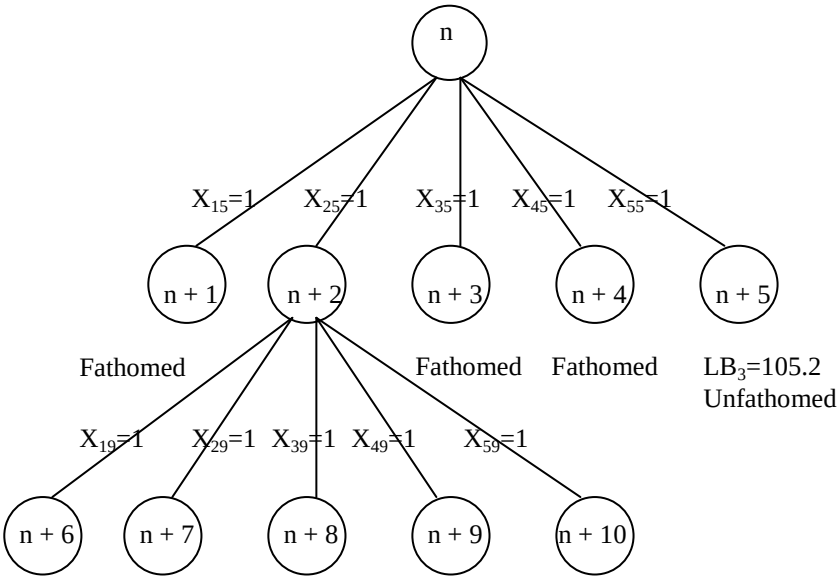

Figure 4.4: Branching Scheme for the Example Problem-3

We repeat the procedure for the generated branches.

Figure 4.5 shows the flowchart of the algorithm for the root node and Figure 4.6 summarizes the procedure for the intermediate nodes.

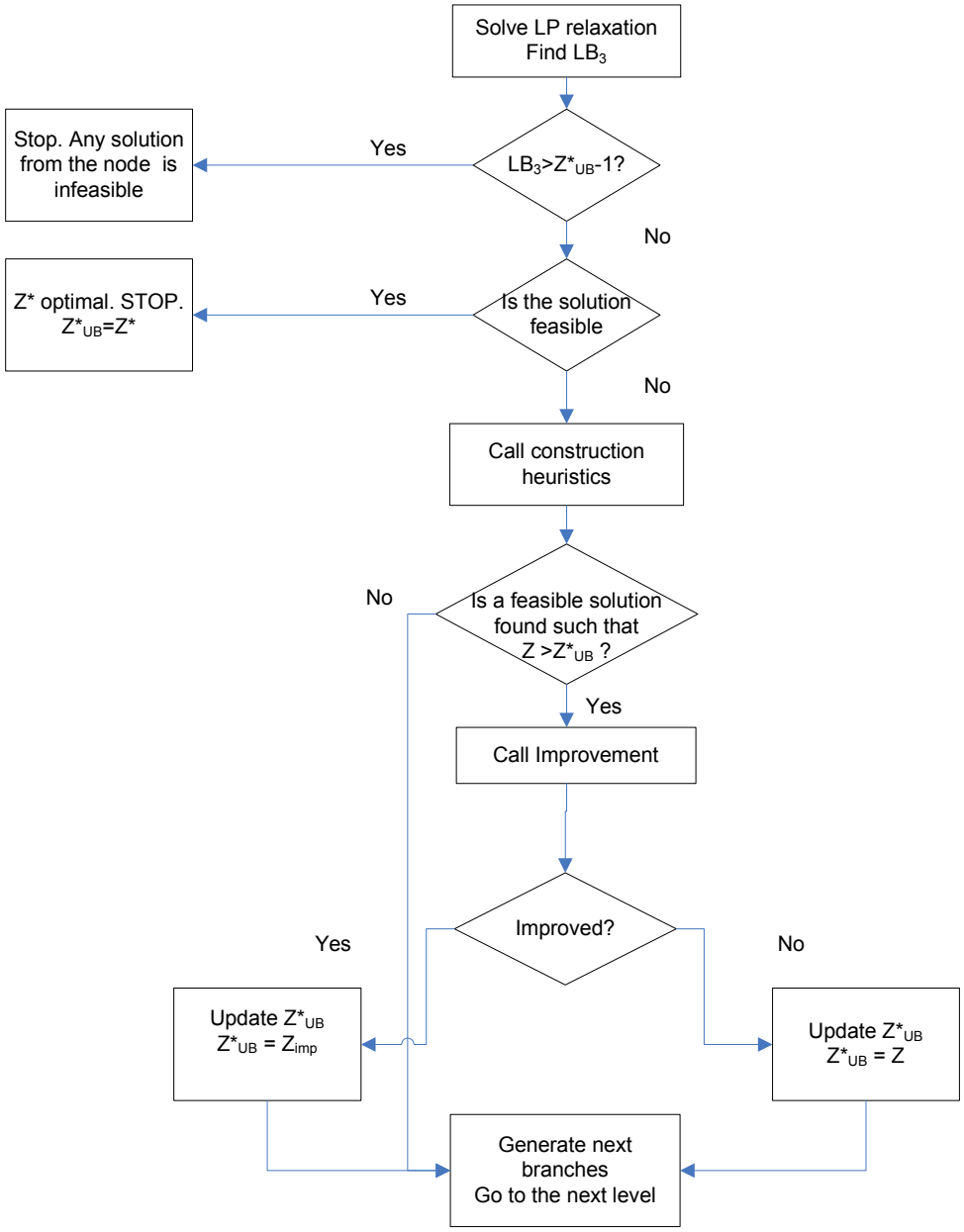

Figure 4.5: Flowchart of B&B Algorithm for the Root Node

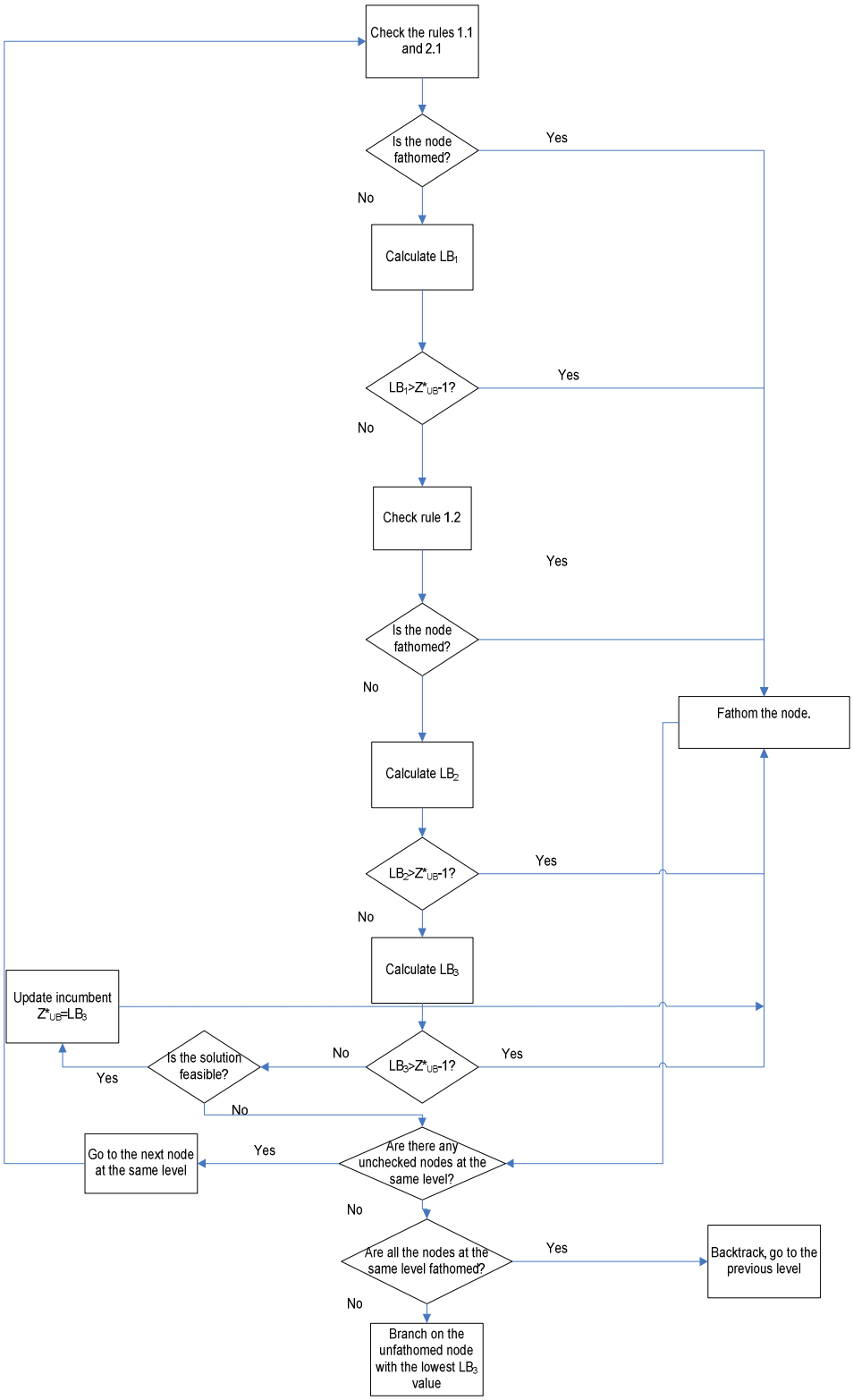

Figure 4.6: Flowchart of B&B Algorithm for Intermediate Nodes

Alternate Branching Scheme

We propose an alternative branching scheme that creates two nodes from each parent node. We branch from the highest fractional variable of the optimal LP relaxation solution. For the chosen variable, x_{ij}, we generate the following subproblems:

Subproblem 1: x_{ij} =0
Subproblem 2: x_{ij} =1

The associated tree is demonstrated in Figure 4.7.

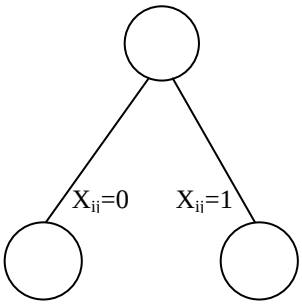

Figure 4.7: Alternate Branching Scheme

The selection strategy for the alternate branching scheme is as follows:
When the two branches are created at a node, continue with the one that fixes the variable to 1. If that node is fathomed continue with the one that sets the variable to 0.

Our preliminary runs have revealed that the first branching strategy outperforms the second one in terms of average CPU time. Hence we used the first branching scheme in our main runs. The detailed comparison of the two branching schemes will be discussed later in preliminary experiments section of Chapter 6.

CHAPTER 5

HEURISTIC PROCEDURES

In this chapter we discuss the heuristics we propose for the Multi Resource Agent Bottleneck Generalized Assignment Problem (MRABGAP). First we study a tabu search algorithm and then we discuss a Branch and Bound (*B&B*) based heuristics that uses an α approximation scheme.

5.1. Tabu Search

In this section, we discuss our tabu search algorithm. For the detailed information on tabu search techniques, one may refer to the book of Glover and Laguna (1997).

We first discuss our neighborhood structure.

Two neighborhoods are used in the algorithm, namely N1 and N2. Since we have a bottleneck type objective function, we construct the neighborhood of a solution around its bottleneck agent(s).

The basic move of N1 is taking a task already assigned to the bottleneck agent and reassigning it to another agent. This basic move makes two decisions: which task to select and whom to assign. N1 is defined as follows:
Assign every task of the bottleneck to every other agent as long as the assignment is feasible, i.e., the newly assigned agent has available capacity.

The neighborhood size is a variable that changes based on the number of tasks at the current bottleneck agent/s. It is upper bounded by *n*(m-1)* as there can be at most *m* bottleneck agents and each agent will be a candidate for an exchange over *m-1* possible locations. A tighter upper bound is: $\sum_k n_{bk} * (m-1)$ where n_{bk} is the number of tasks assigned to the bottleneck agent *k*.

Basic move of N2 is based on choosing two tasks, one from the bottleneck agent and the other from another agent and exchanging their agents. In this basic move we have to make decisions about which tasks should be selected for exchange. N2 is defined as follows: Exchange all job pairs between bottleneck and non bottleneck agents, as long as the exchange results with a feasible solution.

The neighborhood size is upper bounded by $\binom{n}{2}$. This bound is loose, since the jobs that are already assigned to the same agent cannot be exchanged. A tighter upper bound can be as follows: $\sum_k n_{bk} * (n - n_{bk} - \sum_{l<k} n_{bl})$ where n_{bk} and n_{bl} are the number of jobs assigned to the bottleneck agents *k* and *l*, respectively.

For each neighborhood savings are kept in memory, resulting in a memory requirement of *O (n*(m-1) + n*(n-1)/2)*, that is *O* (n^2*+n*m)*.

As N2 does not change the number of jobs, some solutions become unreachable from the current solution. This brings dependency on the initial solution. However N1 avoids this dependency. On the other hand, if we cannot find a feasible move in N1 due to the capacity restrictions, we use N2 with the hope of improving the current solution.

In N1, the tabu attributes can be defined as the bottleneck agent and the job that is taken from this bottleneck agent most recently.

In N2, tabu attributes are the jobs that are swapped most recently.

Upper bound on the tabu tenure of N1 is a variable since the bottleneck agent changes as we take a job from that agent. *(m-1)*(n-1)* is an upper bound, however, not a tight one.

Upper bound on the tabu tenure of N2 is also a variable as the number of possible and feasible swaps changes. A theoretical upper bound is $\binom{n}{2}$.

Due to the dynamic nature of the neighborhood size, tabu tenures are determined empirically. Three levels for tabu tenure are used which are 10, 50 and 100. Tenure sizes of 10 and 100 are selected to see the effect of relatively small and large tenure values on the performance of the algorithm and 50 is selected as an intermediate value.

Aspiration criterion is chosen as the best solution. Tabu statuses of the moves that improve the best solution found so far are overridden.

In neighborhood 1, solutions involving most recent *TabuTenure* changes will be classified as Tabu. Accordingly, if job *j* is taken from agent *i* in an iteration, it will not be reassigned to that agent for *TabuTenure* iterations unless aspiration criterion is satisfied.

In neighborhood 2, solutions involving most recent *TabuTenure* swaps are defined as Tabu. Accordingly, if jobs *j* and *k* are swapped, they will not be reswapped for *TabuTenure* iterations unless aspiration criterion is satisfied.

The solution attributes that have changed recently are recorded in the recency based memory. For neighborhood 1, a matrix called Tabu is used for this purpose. The structure of the memory is represented in Figure 5.1. The row headers correspond to the agent indices and the column headers correspond to the job indices.

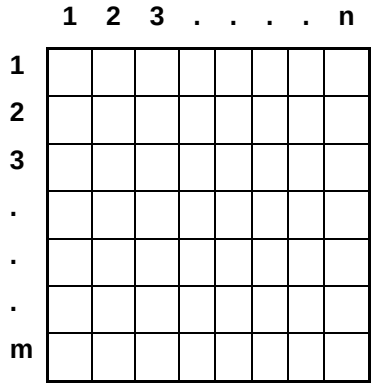

Figure 5.1: Schematic Representation of Recency Based Memory for N1

The recency based memory requirement is *O (m*n)* for N1.

For neighborhood 2, a matrix called Tabu2 is used for this purpose. The structure of the memory is represented in Figure 5.2.

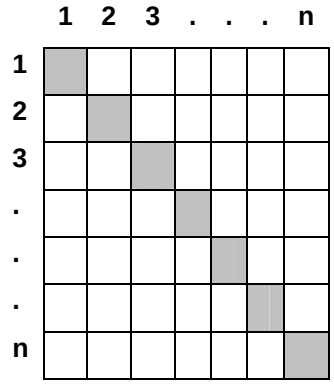

Figure 5.2: Schematic Representation of Recency Based Memory for N2

Note that, only the upper triangle of this matrix is used for recency based memory since we are dealing with swaps. The recency based memory requirement is *O*(*n*n*) for N2.

Our algorithm finds all feasible neighbors of a solution and selects the most improving one using the recency based memory and aspiration criterion. The search is intensified by considering all neighborhoods and the selecting the move with the steepest descent.

We now discuss the termination condition of the algorithm.

Termination criterion is set to the number of nonimproving moves and the total number of iterations. When the number of nonimproving moves equals to a predetermined limit called *nonimplimit* or the total number of iterations equals to the limit *maxiter,* the algorithm terminates.

We try 50, 150 and 250 for *nonimplimit* in our experiments. Our preliminary runs revealed that this limit is affecting the performance of the algorithm significantly; the higher the limit the better is the performance. Hence a value of 250 is used for the *nonimplimit* in the final algorithm. The limit on the total number of iterations, *maxiter*, is set to 1000.

The Algorithm

Our tabu search algorithm starts with a feasible solution. Two neighborhoods are defined for this feasible solution. The savings corresponding to these moves are calculated and kept in memory. To avoid the infeasible moves we assign a saving value of -2000000. Then the non-tabu move with the maximum saving value or the tabu move that satisfies the aspiration criterion is selected and we move from the current solution to that solution. That is, we move in the direction of steepest descent. If the new solution is better than the best solution at hand, the best solution value is updated. The algorithm stops when the number of nonimproving moves or the number of iterations reach their predetermined limits. Below a stepwise description of the algorithm is given.

STEP 1. Initialization

As the decision problem of the MRABGAP is NP-complete it is not possible to find a polynomial algorithm that guarantees a feasible solution. The LP relaxation based construction algorithm that is used in the Branch and Bound algorithm is used to find a solution, hopefully a feasible one. This algorithm runs in

polynomial time, hence returns an initial solution quickly. However the resulting solution is not necessarily feasible. We use this solution as an initial solution in our *TS* algorithm. Note that even when an infeasible solution is given to the *TS* algorithm there is still a chance that the *TS* algorithm returns a feasible one.

Subroutine for the construction procedure

1. Solve LP Relaxation of the Problem
2. ***For j=1,….,n (for all the jobs)***

    Find the highest assignment value in the LP relaxation, x_{ij} for job *j*

    Set x_{ij} =1

End For

STEP 2. Neighborhood Search

STEP 2.1. If the termination criterion (*nonimp>nonimplimit* or *iter>maxiter*) is reached, stop.

S**TEP 2.2.** Generate all neighbors of the current solution, using the following subroutine.

Subroutine for the neighborhood search procedure

For b=1,….,numofbot (for all the bottleneck agents of the current solution)

    Calculate savings for each move of N1

    Find the maximum saving in N1 for bottleneck agent *b*

    Calculate savings for each move of N2

    Find the maximum saving in N2 for bottleneck agent *b*

End For

The above subroutine finds the savings values for all moves in N1 and N2. If the current solution has more than one bottleneck agent these neighborhoods are generated considering each bottleneck.

The subroutines that calculate the savings from the moves are stated below.

Subroutine to calculate the savings from the move (*i*,*j*) in N1

For i=1,….,m
For j=1,….,n
$X_{ne}=X_{now}$
If (*j* is not assigned to the bottleneck agent considered***)Then***
Savings1 (i ,j) = -2000000
Else
If (If *i* is the bottleneck agent considered***)Then***
Savings1(i,j) = -2000000
Else
If(The move is not feasible)***Then***
Savings1(i,j) =-2000000
Else
In X_{ne} assign job *j* to agent *i*
Update the loads of the agents for X_{ne}
Find the bottleneck value for X_{ne}
Savings1(i,j)=(Bottleneck valueof X_{now} -Bottleneck value of X_{ne})
End if
End if
End If

This subroutine finds the savings of the shift move. Note that the moves involving the jobs of the non-bottleneck agents, the moves that assign a job to the current

bottleneck agent and infeasible moves are given small saving values. Hence, these moves are avoided.

Subroutine to calculate the savings from the move (j, k) in N2

For j=1,….,n

For k=1,….,n

$X_{ne}=X_{now}$

If (j is not assigned to the bottleneck agent considered***)Then***

Savings2 (j , k) = -2000000

Else

If (If k is assigned to the same bottleneck agent***) Then***

Savings2 (j , k) = -2000000

Else

If(The move is not feasible)***Then***

Savings2 (j ,k) = -2000000

Else

In X_{ne} swap jobs j and k

Update the loads of the agents for X_{ne}

Find the bottleneck value for X_{ne}

Savings2(j,k)=(Bottleneck value of X_{now}- Bottleneck value of X_{ne})

End if

End if

End If

Again, in this subroutine only the feasible moves that take a job from the bottleneck agent and swap it with a job of another agent are considered, the others are avoided by giving a very small saving value.

At the end of these two subroutines, the move is selected based on the steepest descent rule and aspiration criterion. Note that, if the saving of the selected move is -2000000, which may happen when all the moves are infeasible in the neighborhood, then we allow an infeasible move, update the current solution

accordingly and continue the search, but in such a case we do not update X_{best}. Although never happened in the computational experiments such a case is more likely to occur towards the end of the algorithm. This strategy may help the diversification of the algorithm.

STEP 2.3. Making the move

The current solution is updated by making the selected move and updating the used capacities (loads) of the agents, finding the new bottleneck agent(s) and the bottleneck (objective function) value. If the objective function value is better than the best solution at hand, X_{best} is updated. Below is the subroutine of the move.

Subroutine for making the selected move

If (The selected move is from N1) ***Then***
Update the corresponding recency based memory
Make the selected move
Update X_{now} using the attributes of the selected move
Find the bottleneck(s) of X_{now} and the new objective function value
End if

If (The selected move is from N2) **Then**
Update the corresponding recency based memory
Make the selected move
Update X_{now} using the attributes of the selected move
Find the bottleneck(s) of X_{now} and the new objective function value
End if

If (saving from the selected move ≤ 0) ***Then***
nonimp=nonimp+1
Else
lastimp=iter
End if

If (The selected move is not infeasible & Objective function value of X_{now}< Objective function value of X_{best}) ***Then***

$X_{best}=X_{now}$

End if

Below is the overall statement of the *TS* algorithm.

Statement of the *TS* Algorithm

Use ***Construction*** to find an initial (feasible) solution

Set X_{now} as the solution of the construction

$X_{best}=X_{now}$

nonimp=0

Repeat

Compute *saving*s for all moves in N1and N2

Find the move with max{*saving*}from N1(X_{now}) and N2(X_{now})

Make the selected move using the 'move' subroutine

Until *nonimp*>*nonimplimit* or *iter*>*maxiter*

Fine Tuning the Tabu Search Heuristic

Based on the results of our factorial design, we fine-tuned our algorithm. Our decisions are the levels of *Tabutenure* and *nonimplimit*.

Our preliminary runs have revealed that the effect of *Tabutenure* on % deviation of tabu solution from the optimal solution is insignificant, i.e., our algorithm seems to be robust to changes in *Tabutenure*. We report the results of the algorithm with a *Tabutenure* value of 50. Moreover the preliminary tests have showed that the performance of the algorithm is sensitive to the stopping condition parameter, *nonimplimit*. As *nonimplimit* increases, the quality of the solutions increases, at an expense of the increase in the running time of the

algorithm. Based on the relative importance of these two measures one can determine the *nonimplimit*, i.e., if we have time restrictions a low *nonimplimit* value like 150 can be used, but if the main concern is the solution quality, one can increase the *nonimplimit*. With further analysis an upper bound for *nonimplimit*, after which the solution quality does not significantly change can also be determined. In our experiments, we use a *nonimplimit* value of 250 as it has returned high quality solutions without causing a significant increase in the solution times.

5.2. Branch and Bound Based Heuristics - α Approximation Scheme

The quality of the linear programming relaxation bound and the fact that good feasible solutions are found at earlier steps suggest that the Branch and Bound based algorithms may be used as efficient approximation algorithms.

In this study, we develop an α % approximation scheme. The algorithm executes like our Branch and Bound algorithm, however uses a different rule to check the promise of the node (inflates lower bound).

The α approximation scheme defines an optimality tolerance, α and fathoms a node if $(1+ \alpha)*Z_{LB} > Z_{UB}$. With this scheme one can guarantee that the solution found is within the α*100 percent of the optimal objective function value; however it requires exponential time as the original Branch and Bound algorithm with $\alpha = 0$.

CHAPTER 6

COMPUTATIONAL RESULTS

In this chapter we aim to test the performance of our algorithms together with bounding and reduction mechanisms. We present our data generation scheme, state our performance measures and discuss the results of our preliminary and main computational experiment.

6.1. Generating Test Problems

We generate the processing times for each agent for the first period from a discrete uniform distribution between a and b. We use the following three sets for a and b:

Set I. a=5, b=25

p_{ij1}~ U [5, 25]

Set II. a=15, b=25

p_{ij1}~ U [15, 25]

Set III. a=25, b=35

p_{ij1}~ U [25, 35]

Set I represents cases where variance (range) of the distribution is relatively high. Set II is used to see the effect of a decrease in the range of the distribution while keeping the expected value nearly the same. Set III stands for the instances where the range of the distribution is low while the expected value is high.

We hereafter refer to these sets as *S1*, *S2* and *S3*.

For each set, we generate p_{ijt} and b_{it} values using the procedure reported in the OR library for the GAP. The stepwise description of the procedure is given below.

Step 0. Generate $p_{ij1} \sim$ U [*a*, *b*] for defined set (Set I, Set II, Set III)

Step 1. Set $b_{i1} = c\sum_{j \in J} p_{ij1} / m$, where *c* is a predefined factor

Step 2. Set $p_{ijt} = 3p_{ij1} / 4 + \gamma_{ijt} p_{ij1} / 2$ for each $t \geq 2$, where γ_{ijt} are random numbers from [0,1].

Step 3. Set $b_{it} = c\sum_{j \in J} p_{ijt} / m$ for each $t \geq 2$

While generating the capacity of an agent *i* in period *t*, b_{it}, we calculate a load for that agent as if all jobs are performed by that agent. Then we assume distributing this load to all agents evenly and set the capacity equal to this distributed load multiplied by a factor. Note that, as the p_{ijt} values change from agent to agent and period to period, the corresponding b_{it} values change accordingly.

The choice of the factor *c* is important, setting it too low results in many infeasible instances. In steps 2 and 4, we use the following two sets for *c* values.

Set I. $c = 1.0$
Set II. $c = 1.2$

We hereafter refer to these sets as *C1* and *C2*. Sets *C1* and *C2* represent low and high capacity instances, respectively. We also tried for $c = 0.8$ in our preliminary experiments. However, we observed that the majority of the instances were infeasible. When $c = 1.0$, we could obtain feasible solutions for almost all of the instances.

Three sets of processing times and two sets of capacities together yield six combinations. For each combination, we generate instances starting with $n = 10$,

m = 5, and s = 2 and increasing them in increments of 10, 5 and 1 respectively. For each set combination, m, n, and s values we generate 10 problem instances.

6.2. Performance Measures

In this section, we discuss the performance measures we used to evaluate the efficiency of our Branch and Bound (*B&B*) algorithm, Heuristics and Lower Bounds.

To evaluate our Branch and Bound algorithm we used the following performance measures:

1. CPU time in seconds (average, maximum)
2. Number of nodes generated (average, maximum)
3. Number of nodes until the optimal solution is found (average, maximum)

We set a termination limit of 20 minutes for our mathematical model and Branch and Bound algorithm.

We use the following performance measures for our heuristics.

1. CPU time in seconds (average, maximum)
2. Percent deviation from the optimal (or best known) solution
3. Number of times optimal solution is reached

For lower bound we only report the percent deviations from the optimal solutions. The optimal solutions are found by CPLEX 10.1. CPLEX is run for 1200 seconds. The same termination limit is put to our Branch and Bound algorithm as well. All experimentations are done in Pentium IV 2.8 GHz, 1 GB RAM. All algorithms are coded with Microsoft Visual C++ 2005.

6.3. Preliminary Experiments

We design a preliminary experiment to evaluate the effects of the lower and upper bounds, branching schemes on the performance of our Branch and Bound algorithm. We define different versions of the Branch and Bound algorithm using different choices of these mechanisms. Table 6.1 shows these versions and the corresponding abbreviations.

Table 6.1: Branch and Bound Algorithm Versions

Lower Bounds (*LB*)	
LB_0	With LB_1, LB_2 and LB_3
$LB_{1\text{-}2}$	Without LB_1 and LB_2
LB_2	Without LB_2
LB_3	Without LB_3
Branching Strategy (*BS*)	
BS_1	Normal Branching Scheme
BS_2	Alternate Branching Scheme
Upper Bounds (*UB*)	
UB_0	Without using the upper bound heuristics
UB_1	With using the upper bound heuristics

We use 10 instances with parameter settings *S2C1* and *S3C1*, each with 5 agents, 30 jobs and 3 periods. We now present the results of our preliminary runs.

Effects of the Lower Bounds

We use three versions of the Branch and Bound algorithm to see the effect of the lower bounds. The first one uses only LB_3, but not LB_1 and LB_2, whereas the second one uses LB_1 and LB_3, but not LB_2. We do not construct a *B&B* algorithm that uses LB_2 but not LB_1 as LB_2 uses the information from LB_1 calculations. Finally, we try to see the impact of using LB_3 by generating a *B&B* algorithm that uses LB_2 in the branch selecting scheme instead of LB_3, i.e., the node with the lowest LB_2 value is selected to branch on. In this version LB_3 is only used to determine the job to branch on.

The results of the preliminary runs are shown in Tables 6.2 and 6.3. Note that we set a limit on the number of nodes. If the algorithm cannot find the optimal solution after 399996 nodes, the best solution on hand is reported with the corresponding solution time.

Table 6.2: Preliminary Run Results for the Effects of *LB*s- *S2C1*

		LB_0	LB_{1-2}	LB_2	LB_3
CPU time	Average	9.40	9.89	9.74	130.63
	Maximum	29.03	29.47	30.59	215.51
# of nodes	Average	9462	9736	9501	257416
	Maximum	36239	37455	36254	399866
Node of optimality	Average	4175	4276	4177	270136
	Maximum	26402	27223	26417	399996

*In LB_3 3 instances could not be solved to optimality within the node limit, hence the results are underestimates.

Table 6.3: Preliminary Run Results for the Effects of *LB*s- *S3C1*

		LB_0	LB_{1-2}	LB_2	LB_3
CPU time	Average	4.00	4.39	4.23	143.57
	Maximum	14.28	14.92	15.69	216.41
# of nodes	Average	3638	3797	3640	281866
	Maximum	13131	13850	13144	399996
Node of optimality	Average	136	144.6	136	265548
	Maximum	1017	1012	1017	399719

*In LB_3 only 2 instances could be solved to optimality within the node limit, hence the results are underestimates.

As can be observed from Tables 6.2 and 6.3, the CPU times and number of nodes in the *B&B* tree slightly increase when both LB_1, LB_2 and LB_2 are not used. This indicates that using these elimination mechanisms does not increase the computational time while decreasing the number of nodes. Hence we use these mechanisms in our main experiment.

It can be observed from the tables, the CPU times and the number of nodes increase considerably when the LP relaxation is not used (in version LB_3). This proves the strength of the LP relaxation as a bounding scheme.

Based on these preliminary results we use all three lower bounds in our main runs.

Branching Strategy Selection

After LB_0 (the setting with all three lower bounds) is chosen, we compare the branching schemes. Tables 6.4 and 6.5 demonstrate the average and maximum CPU times, number of nodes in *B&B* tree and node of optimality.

Table 6.4: Preliminary Run Results for Branching Strategies - *S2C1*

		BS_1	BS_2
CPU time	Average	9.40	116.41
	Maximum	29.03	1042.50
# of nodes	Average	9462	44303
	Maximum	36239	390231
Node of optimality	Average	4174	4481
	Maximum	26402	18689

Table 6.5: Preliminary Run Results for Branching Strategies - *S3C1*

		BS_1	BS_2
CPU time	Average	4.00	6.284
	Maximum	14.28	26.61
# of nodes	Average	3638	2665
	Maximum	13131	10756
Node of optimality	Average	136	890
	Maximum	1017	6952

As can be observed from Table 6.4 the alternate branching strategy results in larger sized trees; hence larger CPU times. The increase in CPU time is also observed from Table 6.5; however it is observed that number of nodes is smaller in the alternate branching strategy. This is due to the fact that our branching strategy opens *m* branches at a level while alternate branching strategy opens only two. A similar situation occurs for the node of optimality results. Hence, it is seen from Table 6.4 that the maximum number of nodes for the BS_1 is larger than that of the BS_2 although on the average BS_1 has better node of optimality results.

Note from the tables that BS_1 outperforms BS_2 in terms of CPU times, number of nodes and node of optimality. Based on these results we perform our main experiments using BS_1.

Effect of the Upper Bound

We conduct experiments on the same instances to see the effect of our upper bounds on the performance of the Branch and Bound algorithm. We use LB_0 and BS_1 combination based on the results of the previous experiments. Tables 6.6 and 6.7 report the average and maximum CPU times, number of nodes in the *B&B* tree and node of optimality for two versions of the *B&B* algorithm.

Table 6.6: Preliminary Run Results for Effect of Upper Bound - *S2C1*

		UB_0	UB_1
CPU time	Average	13.704	9.459
	Maximum	37.47	29.11
# of nodes	Average	13971	9462
	Maximum	35679	36239
Node of optimality	Average	8930	4175
	Maximum	28350	26402

Table 6.7: Preliminary Run Results for Effect of Upper Bound - *S3C1*

		UB_0	UB_1
CPU time	Average	5.951	3.957
	Maximum	16.17	13.86
# of nodes	Average	5368	3638
	Maximum	13725	13131
Node of optimality	Average	2025	136
	Maximum	5852	1017

It can be observed from the tables that using upper bounds results in lower CPU times and fewer nodes on the average. Moreover, the node where the optimal solution is reached decreases remarkably when the upper bounds are used. Hence we use upper bounds in our main experiment.

6.4. Main Experiment

In our main experiment, we use the best mechanisms found by our preliminary experiment.

We first study the performance of the LP relaxation lower bound, LB_3, which is the objective function value of the optimal LP relaxed problem obtained at the root node. We compute the deviation form the optimal solution as

$$\%Dev = \left(\frac{OPT - LB}{OPT} \right) \times 100 \text{ where}$$

OPT = Optimal solution value

LB = Optimal solution value of the LP Relaxed Problem

In Tables 6.8, 6.9 and 6.10 we report the average and maximum values of the deviations of the lower bound from the optimal solution, the number of fractional variables in the optimal LP relaxation solution, the number of jobs that are not assigned to a unique agent, average number of agents that such jobs are assigned. The tables show the results for s=2, s=3 and s=5, respectively.

As can be observed from the tables, the lower bounds behave consistent for almost all problem instances. Almost all average deviations are below 10% and almost all maximum deviations are below 12%, except for the setting m=10 and n=20 where a significant increase in the deviation is observed. For such instances the average deviation and the maximum deviations are about 17% and 23%, respectively.

This satisfactory performance of the lower bounds can be explained by very few fractional variables produced by the optimal LP relaxation. It can also be observed from the tables that the number of fractional variables is very low. It is also observed that, for fixed m as the number of jobs, n, increases the power of the lower bound increases.

Table 6.8: Lower Bound 3 Deviations for *s*=2

s=2										
C1										
			# of fractional variables		# of jobs split to multiple agents		# of agents with split jobs		Dev. (%)	
	m	*n*	Avg.	Max.	Avg.	Max.	Avg.	Max.	Avg.	Max.
S1	5	20	7.5	8	3.5	4	2.20	3	6.66	9.00
		30	7.5	8	3.5	4	2.23	3	4.16	4.96
		40	7.8	8	3.8	4	2.07	2.33	2.41	2.85
		50	7.8	8	3.8	4	2.07	2.33	1.64	2.95
		60	8	8	4	4	2.00	2	1.51	2.20
	10	20	16.4	17	7.4	8	2.23	2.29	17.34	19.43
		30	17.2	18	8.2	9	2.11	2.29	9.50	11.61
S2	5	20	7.80	8	3.80	4	2.07	2.33	3.17	4.66
		30	7.44	8	3.44	4	2.22	3.00	2.39	4.91
		40	7.70	8	3.70	4	2.10	2.33	1.45	2.38
		50	8.00	8	4.00	4	2.00	2.00	1.10	1.70
		60	7.90	8	3.90	4	2.03	2.33	0.97	1.76
	10	20	17.00	18	8.00	9	2.14	2.50	7.06	11.78
		30	17.00	18	8.00	9	2.14	2.50	5.40	11.98
S3	5	20	7.70	8	3.70	4	2.13	3.00	2.58	4.88
		30	7.70	8	3.70	4	2.10	2.33	1.95	4.33
		40	7.90	8	3.90	4	2.03	2.33	1.17	2.03
		50	7.90	8	3.90	4	2.03	2.33	0.97	2.17
		60	8.00	8	4.00	4	2.00	2.00	0.71	1.28
	10	20	16.60	18	7.60	9	2.20	2.50	4.85	7.54
		30	16.80	17	7.80	8	2.16	2.29	3.89	7.03
C2										
S1	5	20	7.5	8	3.5	4	2.20	3	6.66	9.00
		30	7.5	8	3.5	4	2.23	3	4.16	4.96
		40	7.8	8	3.8	4	2.07	2.33	2.41	2.85
		50	7.8	8	3.8	4	2.07	2.33	1.64	2.95
		60	8	8	4	4	2.00	2	1.51	2.20
	10	20	16.4	17	7.4	8	2.23	2.29	17.34	19.43
		30	17.2	18	8.2	9	2.11	2.29	9.50	11.61
S2	5	20	7.80	8	3.80	4	2.07	2.33	3.17	4.66
		30	7.44	8	3.44	4	2.22	3.00	2.39	4.91
		40	7.70	8	3.70	4	2.10	2.33	1.45	2.38
		50	8.00	8	4.00	4	2.00	2.00	1.03	1.70
		60	7.90	8	3.90	4	2.03	2.33	0.97	1.76
	10	20	17.00	18	8.00	9	2.14	2.50	7.06	11.78
		30	17.00	18	8.00	9	2.14	2.50	5.40	11.98
S3	5	20	7.70	8	3.70	4	2.13	3.00	2.58	4.88
		30	7.70	8	3.70	4	2.10	2.33	1.95	4.33
		40	7.90	8	3.90	4	2.03	2.33	1.17	2.03
		50	7.90	8	3.90	4	2.03	2.33	0.97	2.17
		60	8.00	8	4.00	4	2.00	2.00	0.71	1.28
	10	20	16.60	18	7.60	9	2.20	2.50	4.85	7.54
		30	16.80	17	7.80	8	2.16	2.29	3.89	7.03

Table 6.9: Lower Bound 3 Deviations for *s*=3

s =3										
C1										
			# of fractional variables		# of jobs split to multiple agents		# of agents with split jobs		% dev	
	m	*n*	Avg.	Max.	Avg.	Max.	Avg.	Max.	Avg.	Max.
S1	5	20	7.60	8	3.60	4	2.13	2.33	7.14	9.34
		30	7.80	8	3.80	4	2.07	2.33	3.83	4.93
		40	7.80	8	3.80	4	2.07	2.33	2.34	3.26
		50	7.70	8	3.70	4	2.10	2.33	1.82	2.62
		60	7.80	8	3.80	4	2.07	2.33	1.35	1.75
	10	20	16.80	18	7.80	9	2.18	2.50	16.81	23.17
		30	17.10	18	8.10	9	2.12	2.29	10.15	12.60
S2	5	20	7.80	8	3.80	4	2.07	2.33	3.62	4.88
		30	7.90	8	3.90	4	2.03	2.33	2.36	4.17
		40	7.90	8	3.90	4	2.03	2.33	1.57	2.44
		50	7.80	8	3.80	4	2.07	2.33	1.42	2.10
		60	8.00	8	4.00	4	2.00	2.00	0.77	1.11
	10	20	16.80	18	7.80	9	2.17	2.50	6.64	10.95
		30	17.10	18	8.10	9	2.12	2.29	4.86	8.22
S3	5	20	7.70	8	3.70	4	2.10	2.33	2.39	3.44
		30	7.90	8	3.90	4	2.03	2.33	1.61	2.75
		40	7.90	8	3.90	4	2.03	2.33	1.27	4.14
		50	7.90	8	3.90	4	2.03	2.33	1.23	2.01
		60	7.90	8	3.90	4	2.03	2.33	0.67	1.00
	10	20	16.90	19	7.80	9	2.19	2.50	5.04	5.84
		30	17.70	18	8.70	9	2.04	2.29	3.76	5.67
C2										
S1	5	20	7.60	8	3.60	4	2.13	2.33	7.14	9.34
		30	7.80	8	3.80	4	2.07	2.33	3.83	4.93
		40	7.80	8	3.80	4	2.07	2.33	2.34	3.26
		50	7.70	8	3.70	4	2.10	2.33	1.82	2.62
		60	7.80	8	3.80	4	2.07	2.33	1.35	1.75
	10	20	16.80	18	7.80	9	2.18	2.50	16.81	23.17
		30	17.10	18	8.10	9	2.12	2.29	10.15	12.60
S2	5	20	7.80	8	3.80	4	2.07	2.33	3.62	4.88
		30	7.90	8	3.90	4	2.03	2.33	2.36	4.17
		40	7.90	8	3.90	4	2.03	2.33	1.54	2.44
		50	7.80	8	3.80	4	2.07	2.33	1.42	2.10
		60	8.00	8	4.00	4	2.00	2.00	0.77	1.11
	10	20	16.80	18	7.80	9	2.17	2.50	6.64	10.95
		30	17.10	18	8.10	9	2.12	2.29	4.86	8.22
S3	5	20	7.70	8	3.70	4	2.10	2.33	2.36	3.44
		30	7.90	8	3.90	4	2.03	2.33	1.61	2.75
		40	7.90	8	3.90	4	2.03	2.33	1.27	4.14
		50	7.90	8	3.90	4	2.03	2.33	1.23	2.01
		60	7.90	8	3.90	4	2.03	2.33	0.67	1.00
	10	20	16.70	18	7.70	9	2.19	2.50	4.99	5.84
		30	17.70	18	8.70	9.00	2.04	2.29	3.76	5.67

Table 6.10: Lower Bound 3 Deviations for *s*=5

s=5										
C1										
			# of fractional variables		# of jobs split to multiple agents		# of agents with split jobs		% dev	
	m	*n*	Avg.	Max.	Avg.	Max.	Avg.	Max.	Avg.	Max.
S1	5	20	7.50	8	3.50	4	2.17	2.33	7.09	8.45
		30	7.40	8	3.40	4	2.23	3.00	4.01	6.50
		40	7.90	8	3.90	4	2.03	2.33	2.34	3.24
		50	7.90	8	3.90	4	2.03	2.33	1.82	2.44
	10	20	16.40	18	7.40	9	2.25	2.80	16.09	20.95
		30	16.60	18	7.60	9	2.22	2.80	10.05	12.09
S2	5	20	8.00	8	4.00	4	2.00	2.00	3.33	6.53
		30	7.90	8	3.90	4	2.03	2.33	2.33	4.34
		40	7.90	8	3.90	4	2.03	2.33	1.38	2.70
		50	8.00	8	4.00	4	2.00	2.00	1.13	1.71
	10	20	16.90	18	7.80	9	2.19	2.50	6.18	9.01
		30	17.50	18	8.50	9	2.07	2.29	5.32	8.47
S3	5	20	8.20	10	4.00	5	2.07	2.33	2.62	4.93
		30	8.00	8	4.00	4	2.00	2.00	1.89	4.00
		40	8.00	8	4.00	4	2.00	2.00	1.09	2.19
		50	7.80	8	3.80	4	2.07	2.33	0.97	1.39
	10	20	17.70	20	8.10	9	2.20	2.50	5.00	7.97
		30	17.20	18	8.20	9	2.11	2.29	4.02	5.79
C2										
S1	5	20	7.50	8	3.50	4	2.17	2.33	7.09	8.45
		30	7.40	8	3.40	4	2.23	3.00	4.01	6.50
		40	7.90	8	3.90	4	2.03	2.33	2.34	3.24
		50	7.90	8	3.90	4	2.03	2.33	1.82	2.44
	10	20	16.40	18	7.40	9	2.25	2.80	16.09	20.95
		30	16.60	18	7.60	9	2.22	2.80	10.05	12.09
S2	5	20	8.00	8	4.00	4	2.00	2.00	3.33	6.53
		30	7.90	8	3.90	4	2.03	2.33	2.33	4.34
		40	7.90	8	3.90	4	2.03	2.33	1.38	2.70
		50	8.00	8	4.00	4	2.00	2.00	1.13	1.71
	10	20	16.80	18	7.80	9	2.17	2.50	6.18	9.01
		30	17.50	18	8.50	9	2.07	2.29	5.32	8.47
S3	5	20	7.80	8	3.80	4	2.07	2.33	2.51	4.93
		30	8.00	8	4.00	4	2.00	2.00	1.88	4.00
		40	8.00	8	4.00	4	2.00	2.00	1.09	2.19
		50	7.80	8	3.80	4	2.07	2.33	0.95	1.39
	10	20	16.60	18	7.60	9	2.20	2.50	4.72	7.97
		30	17.20	18	8.20	9	2.11	2.29	4.02	5.79

On the other hand, increasing the number of agents, *m*, has a negative affect on the power of the lower bound. This is expected since as *m* gets bigger, a job is split between more agents, hence the number of fractional variables increase which in turn reduces the performance of the lower bound. Hence we can conclude that the higher the *n/m* ratio, the better the performance of the LP relaxation.

We could not observe any notable difference in the lower bound performances between different *s* values.

It is observed from the tables that the power of the lower bound is affected by the processing times. The lower bound is weaker for set *S1* where the variance of the processing times is relatively high and the mean is low and the best performances are observed for set *S3* where the variance is low and the mean is high.

We now discuss the performance of our Branch and Bound algorithm. We perform our main experiment using BS_1, UB_1 and all the three lower bounds LB_1, LB_2 and LB_3.

The size of the problem is basically determined by the number of agents, *m*, and the number of jobs, *n*. This is because, the number of decision variables increase as these parameters increase. Hence, we expect an increase in the complexity of the problem with an increase in the problem size parameters.

We choose two values of *m*; *m*= 5 and 10. *n* values are between 20 and 60 (for *s*=5, up to 50) in increments of 10. We use three values of *s*, which are 2, 3 and 5. For all six combinations we report the average and maximum number of nodes, node of optimality and CPU times. We also report the number of instances that can be solved to optimality in our termination limit of 1200 seconds. The average and maximum CPU times of CPLEX and the number of instances that can be solved to optimality by CPLEX within 1200 seconds are also reported. For a fair comparison only the instances that can be solved by both CPLEX and *B&B*

algorithms are used while computing the CPU times. The number of such instances is also reported.

Tables 6.11, 6.12 and 6.13 show the results for s=2, s=3 and s=5, respectively.

As can be observed from Tables 6.11, 6.12 and 6.13, when the number of jobs increases, the CPU times and in turn the number of nodes of the *B&B* algorithm increases. There are a few exceptions, one of which is due to *S1C2*, s=2, m=5. For n=50 the average CPU time is 150.22 seconds whereas it is 56.92 seconds for n=60. However, note here that, one out of ten instances could not be solved when n=60, hence the average is calculated over 9 instances while it is calculated over 10 instances when n=50. This increase in the number of nodes and CPU time is mainly due to the fact that the depth of the *B&B* tree increases as n increases. The effect of n on the problem complexity can also be observed from the increase in CPU times of CPLEX with an increase in n for fixed m and s.

The effect of the number of agents, m, is more notable on the CPU times and the number of nodes of the *B&B* algorithm. As m increases, the number of nodes and in turn the CPU times increase. This is true for all the problem combinations. For example, for *S1C1*, n=30 and s=2 when m is increased from 5 to 10; the average CPU time increases from 10.03 to 232.58 seconds, i.e., increases more than 20 times. This is an expected behavior due to our branching strategy. As the number of agents increase, the number of branches increases at each level, resulting in a bigger tree size, hence higher CPU time. The effect of m can also be observed from the results of CPLEX: either the CPU time increases or the number of instances that can be solved to optimality within 20 minutes decreases as m increases for fixed n and s. It is also observed from the three tables that an increase in the number of agents has a more substantial effect on the solution time of the *B&B* algorithm than that of CPLEX especially for the sets *S2C1* and *S3C1*.

Table 6.11: *B&B* and CPLEX Results for *s*=2

s =2													
C1													
			Number of nodes in *B&B* Tree		Node of Optimality		*B&B* Solution Time (CPU seconds)		BB*	CPLEX Solution Time (CPU seconds)		CPX*	B*
	m	*n*	Avg.	Max.	Avg.	Max.	Avg.	Max.		Avg.	Max.		
S1	5	20	2328	9215	1112	7445	1.34	5.09	10	0.59	4.03	10	10
		30	12088	38860	8358	37785	9.83	36.34	10	0.77	1.41	10	10
		40	26241	57200	11213	32674	28.08	65.55	10	1.30	4.25	10	10
		50	104524	334110	89317	315451	123.68	323.81	9	1.63	4.17	10	9
		60	45342	175675	7104	29891	58.51	245.31	9	2.12	4.92	10	9
	10	20	38053	101274	15112	92145	26.38	67.13	10	3.76	24.05	10	10
		30	107890	255160	41755	194967	241.09	750.98	7	31.50	64.30	10	7
S2	5	20	752	4007	75	601	0.48	2.44	10	0.17	0.66	10	10
		30	801	5003	247	1897	0.97	6.17	10	40.88	296.78	8	8
		40	1207	3660	400	1504	2.02	5.76	10	7.57	24.02	10	10
		50	22272	74170	17270	71632	56.00	176.38	9	156.11	829.88	9	8
		60	9886	38185	7839	37729	19.14	68.17	7	121.15	533.92	6	5
	10	20	42545	389596	709	3807	43.42	403.11	10	0.34	2.00	10	10
		30	25091	145227	24213	145220	60.30	351.52	6	23.22	73.00	10	5
S3	5	20	374	1578	68	596	0.24	0.92	10	0.12	0.38	10	10
		30	112	421	31	273	0.18	0.53	10	0.22	0.70	10	10
		40	2143	7850	1557	7678	3.57	12.52	10	3.55	11.91	9	9
		50	4784	15199	1773	11727	9.57	36.72	10	36.25	303.42	9	9
		60	12545	37365	9073	37323	26.09	72.33	9	58.86	177.45	7	6
	10	20	945	6044	45	216	0.96	5.92	10	0.27	1.33	10	10
		30	37150	243203	33880	243199	66.84	451.20	9	1.02	4.14	9	9
C2													
S1	5	20	2333	9250	1118	7480	1.77	5.03	10	0.55	3.67	10	10
		30	12099	38855	8357	37780	10.03	35.14	10	1.10	3.70	10	10
		40	26249	57185	11223	32769	27.70	66.80	10	1.47	4.19	10	10
		50	120666	335270	105811	316631	159.22	587.66	10	3.00	9.86	10	10
		60	45191	175800	6959	29891	56.92	234.89	9	3.23	13.27	10	9
	10	20	38180	100573	14773	88586	26.53	65.03	10	3.44	19.75	10	10
		30	107883	254360	41656	194277	232.58	779.91	7	48.18	114.63	10	7
S2	5	20	819	4275	85	685	0.64	2.69	10	0.31	1.00	10	10
		30	716	5035	219	1897	0.97	6.86	10	54.62	364.45	9	9
		40	1186	3615	386	1489	2.07	6.11	10	51.36	346.50	10	10
		50	61560	376195	2546	12088	122.55	755.19	10	93.93	351.48	7	7
		60	9866	38145	7831	37689	19.15	67.26	7	90.13	389.72	6	5
	10	20	5068	31267	977	4760	3.78	22.13	9	2.31	14.48	10	9
		30	49548	211919	15050	46255	73.30	318.13	7	221.68	983.64	5	5
S3	5	20	405	1695	86	751	0.24	0.91	10	2.58	23.33	10	10
		30	134	425	38	313	0.20	0.51	10	28.80	229.17	9	9
		40	2286	7875	1765	7703	3.66	11.88	10	9.05	42.14	8	8
		50	4771	15265	1784	11708	9.17	34.26	10	79.65	655.06	9	9
		60	12688	37295	9195	37253	26.22	70.89	8	132.76	436.11	7	6
	10	20	637	3122	30	250	2.54	17.09	10	0.20	0.50	10	10
		30	17405	38608	11983	28358	41.91	118.69	9	21.19	66.25	6	6

Table 6.12: *B&B* and CPLEX Results for *s*=3

s =3													
C1													
			Number of nodes in *B&B* Tree		Node of Optimality		*B&B* Solution Time (CPU seconds)			CPLEX Solution Time (CPU seconds)			
	m	*n*	Avg.	Max.	Avg.	Max.	Avg.	Max.	BB*	Avg.	Max.	CPX*	B*
S1	5	20	897	2570	175	1048	0.48	1.33	10	0.33	0.97	10	10
		30	10834	36820	4787	12577	8.68	25.52	10	0.69	1.42	10	10
		40	46535	117920	25559	101697	46.00	109.67	10	2.35	10.80	10	10
		50	65883	188740	27542	113462	70.25	229.51	10	1.86	5.74	10	10
		60	120276	264255	71515	234349	231.14	781.63	8	1.92	2.61	10	8
	10	20	45183	131672	22607	121344	31.32	93.09	10	1.65	4.38	10	10
		30	144407	262970	34080	152264	201.66	347.03	6	211.87	566.64	9	6
S2	5	20	569	1520	67	567	0.37	0.95	10	0.14	0.34	10	10
		30	9462	36239	4175	26402	9.54	29.09	10	61.22	321.42	10	10
		40	6648	18940	2118	18284	9.66	27.27	10	146.85	967.22	9	9
		50	71204	269425	66052	253053	153.23	553.69	9	89.16	270.16	5	5
		60	135379	277260	79390	257407	310.91	689.89	7	35.32	117.23	10	7
	10	20	24149	160291	8074	27505	21.73	144.69	9	0.26	1.20	10	9
		30	46618	210336	4125	14875	82.11	377.02	5	3.68	9.52	9	5
S3	5	20	400	2554	44	360	0.30	1.80	10	0.09	0.16	10	10
		30	3638	13131	136	1017	3.95	13.38	10	0.34	0.73	10	10
		40	16446	36911	7621	20244	24.71	69.41	10	55.90	325.86	9	9
		50	4190	17716	3799	17080	10.66	44.36	10	45.66	204.59	5	5
		60	13959	34338	3459	22695	36.91	96.01	9	62.68	217.51	9	7
	10	20	20749	38232	10037	38228	19.47	36.36	9	0.82	1.34	10	9
		30	13469	69758	5647	22854	22.31	112.36	6	4.71	27.45	10	6
C2													
S1	5	20	899	2570	176	1048	0.48	1.28	10	0.34	0.88	10	10
		30	10834	36820	4786	12577	8.29	22.83	10	0.88	2.58	10	10
		40	46527	117925	25546	101697	46.01	110.06	10	2.22	6.66	10	10
		50	65879	188740	27542	113472	72.09	246.99	10	1.74	3.47	10	10
		60	120011	264160	71262	234269	186.56	455.25	8	2.69	7.14	10	8
	10	20	46496	135087	23625	124704	31.52	94.26	10	2.17	5.06	10	10
		30	146320	268700	34506	152401	203.27	351.19	6	188.11	565.80	9	6
S2	5	20	602	1660	70	608	0.39	1.13	10	1.11	7.08	10	10
		30	9706	37775	4300	27583	9.32	28.36	10	175.35	944.69	10	10
		40	5576	18975	2387	18319	7.38	25.08	10	54.52	191.42	8	8
		50	71150	269440	66000	253078	160.58	579.42	9	200.28	556.95	5	5
		60	134756	278165	79674	258232	315.14	672.25	7	151.72	630.91	9	7
	10	20	26899	165915	7607	30085	22.99	142.34	9	1.10	2.58	10	9
		30	4181	9219	3497	9211	5.64	11.83	5	135.47	934.98	5	4
S3	5	20	396	2805	47	449	0.25	1.59	10	0.41	1.27	10	10
		30	3766	13560	165	1217	4.01	13.67	10	9.93	60.22	10	10
		40	16966	38735	7869	19629	26.78	77.56	10	195.90	1074.30	9	9
		50	4277	18260	3866	17592	10.74	44.94	10	115.55	477.39	5	5
		60	14163	34410	4007	23701	37.75	102.47	9	133.28	399.76	8	8
	10	20	16109	43519	5966	23166	14.97	46.52	10	1.01	3.13	10	10
		30	23467	78640	10931	28516	35	116	5	101.18	248.78	5	4

Table 6.13: *B&B* and CPLEX Results for *s*=5

s =5													
C1													
			Number of nodes in *B&B* Tree		Node of Optimality		*B&B* Solution Time (CPU seconds)			CPLEX Solution Time (CPU seconds)			
	m	*n*	Avg.	Max.	Avg.	Max.	Avg.	Max.	BB*	Avg.	Max.	CPX*	B*
S1	5	20	1495	4435	288	2824	0.82	2.19	10	0.28	0.64	10	10
		30	15946	62640	9120	52078	14.79	56.74	10	1.51	4.58	10	10
		40	86122	323210	13641	109465	89.68	370.77	10	2.69	9.30	10	10
		50	108594	344820	39672	106514	120.06	362.11	9	2.83	7.36	10	9
	10	20	26848	81051	11628	79285	19.96	61.08	9	1.77	5.11	10	9
		30	175324	336179	37604	136766	210.64	413.95	5	219.08	473.08	10	5
S2	5	20	394	1414	111	780	0.33	1.19	10	0.17	0.48	10	10
		30	3987	13171	1322	6738	4.65	15.83	10	72.22	554.38	8	8
		40	43625	363214	27035	236265	66.97	556.72	10	7.72	25.03	9	9
		50	32238	108345	7835	28120	61.47	200	8	32.36	104.09	7	6
	10	20	16795	82062	4198	37072	15.56	72.88	10	0.27	0.89	10	10
		30	41102	153948	39040	153949	67.53	255.09	8	3.93	17.92	10	8
S3	5	20	621	3876	66	393	0.59	3.78	10	0.14	0.27	10	10
		30	8925	73083	717	5546	11.85	100.75	10	9.66	84.63	10	10
		40	8040	42452	5010	41747	12.65	63.42	10	7.09	53.02	9	9
		50	62025	352777	53270	322194	153.30	902.30	10	62.98	217.55	7	7
	10	20	29080	212432	5197	17513	32.22	239.17	9	0.55	1.14	10	9
		30	35649	208492	27427	136008	73.05	462.48	9	1.14	4.33	10	9
C2													
S1	5	20	1496	4455	286	2804	0.82	2.28	10	0.31	0.66	10	10
		30	15951	62660	9124	52093	15.25	57.95	10	1.89	6.92	10	10
		40	86135	323275	13634	109505	92.67	385.94	10	4.67	23.05	10	10
		50	108591	344750	39668	106484	117.02	354.16	9	5.18	17.31	10	9
	10	20	26900	79099	11554	78208	18.61	57.59	9	2.64	7.91	10	9
		30	166348	304100	32485	110957	204.06	364.73	5	260.62	452.83	10	5
S2	5	20	558	2130	249	2127	0.41	1.31	10	0.96	7.02	10	10
		30	4206	14140	1388	6992	4.52	16.13	10	71.80	551.06	8	8
		40	46179	386295	29543	258804	69.20	577.83	10	21.97	79.45	9	9
		50	32156	107680	7816	28315	59.50	194.39	8	52.83	141.98	7	6
	10	20	41440	243933	26603	243933	40.51	261.44	10	7.71	44.97	10	10
		30	15293	39341	11824	39334	20.90	51.44	9	185.77	490.41	4	4
S3	5	20	212	1330	5	36	0.18	0.94	10	0.25	0.78	10	10
		30	12389	82035	5923	41231	13.16	87.88	10	40.74	315.72	8	8
		40	7386	43045	4590	42324	11.44	64.67	10	41.90	353.49	10	10
		50	12241	33015	7131	26524	26.18	72.17	9	25.88	62.88	6	6
	10	20	18204	123897	13719	118224	18.24	121.00	10	0.42	1.36	10	10
		30	22888	100974	17247	100971	40.71	149.58	8	309.30	1,045.06	7	7

We do not observe a remarkable effect of *s* in our experiments. This is because the value of *s* does not affect the number of decision variables of the problem. The number of periods, *s*, affects the feasible region of the problem and the feasible region does not have a certain effect on the solution speed. The increase in *s* may help our *B&B* algorithm in that more potential assignments can be eliminated using feasibility checks. On the other hand an increase in *s* also means an increase in the total load of the agents when the other parameters are fixed. Such an increase may lead to an increase in the number of alternative optimal solutions, which makes the verification of optimality more difficult, since there are many promising branches. To illustrate, for set *S1C1* *m*=5, *n*=30 when *s* increases from 2 to 3 the number of nodes, node of optimality and CPU times decrease and when *s* increases from 3 to 5, all these three performance measures increase. The same results hold for CPLEX.

We observe from the three tables that the parameter settings affect the problem complexity considerably. This is mainly due to the change in the power of the LP relaxation lower bound.

For set *S1* where the variance of the processing times is relatively high and the mean is low, CPLEX seems to be more effective. On the other hand for the sets *S2* and *S3* our Branch and Bound algorithm is compatible with or better than CPLEX in terms of both CPU times and number of instances that can be solved to optimality within 20 minutes for most instances. For example for set *S2C2*, *m*=10, *n*=30 and *s*=5, the average CPU time is 20.9 seconds and 9 instances are solved by the *B&B* algorithm whereas the average CPU time is 185.77 seconds and 4 instances are solved by the CPLEX. The *B&B* finds the optimal solution about 9 times faster than CPLEX and solves more instances to optimality within the same time limit. There are some exceptions to this situation. However in most of these instances although the average solution time of the *B&B* algorithm is higher than CPLEX, the number of instances it could solve to optimality is bigger. One such exception occurs for *S3C1*, *m*=5, *n*=50 and *s*=5. The average CPU times are 153.3 and 62.98 seconds whereas the numbers of instances solved to optimality are 10

and 7 for B&B and CPLEX, respectively. Moreover in some instances we observe the effect of a single dominating instance on the average solution time of the *B&B* algorithm. To illustrate, for *S2C1* m=5, n=40, s=5 the average solution time for the *B&B* algorithm is 66.97 seconds with a single instance having a solution time of 556.72 seconds. The *B&B* algorithm solves all 10 instances to optimality in the time limit. However CPLEX could not solve one instance to optimality, hence the average is calculated over 9 solved instances. When the dominating instance is excluded, the average solution time decreases to 5.75 seconds and becomes smaller than the average CPU time by CPLEX which is 7.72 seconds.

We also investigate the effect of the capacity factor, c, and could not observe any consistent behavior. The average solution times of both *B&B* and CPLEX increase or decrease when c is increased from 1.0 to 1.2. The c value changes the feasible region of the problem, however a change in the feasible region may not have consistent effect on the performances; the optimal solution may change or may stay the same. However we observe that our *B&B* algorithm is more insensitive to the changes in the c value. This is most probably due to the fact that the negative effect of the change of the optimal solution and the feasible region is balanced by the enhanced performance of our improvement heuristic. Recall that the heuristic switches a job or interchanges jobs between agents. The increase in the capacities increases the number of feasible improving moves; hence the improvement heuristic returns better solutions. One such noteworthy result is observed for *S2* m=10, n=30 and s=5. When c increases from 1.0 to 1.2 the average solution time of the *B&B* algorithm decreases from 67.53 to 20.9 seconds while the number of instances solved to optimality increases from 8 to 9. On the other hand the average solution time of CPLEX increases from 3.93 to 185.77 seconds while the number of instances solved to optimality decreases from 8 to 4.

In Tables 6.14 and 6.15, we summarize the results of our Branch and Bound algorithm by reporting the averages of number of nodes, node of optimality and CPU times for low and high capacities, respectively. Recall that a time limit of

1200 seconds is set and the instances that cannot be solved to optimality within this time limit are not considered while computing the averages.

Table 6.14: Branch and Bound Algorithm Results for *C1*

C1								
			S1		*S2*		*S3*	
m	*n*	*s*	Avg. # of nodes	Avg. CPU time	Avg. # of nodes in *B&B* tree	Avg. CPU time	Avg. # of nodes in *B&B* tree	Avg. CPU time
5	20	2	2328	1.34	752	0.48	374	0.24
	30		12088	9.83	801	0.97	112	0.18
	40		26241	28.08	1207	2.02	2143	3.57
	50		104524	123.68	22272	56.00	4784	9.57
	60		45342	58.51	9886	19.14	12545	26.09
10	20		38053	26.38	42545	43.42	945	0.96
	30		107890	241.09	25091	60.30	37150	66.84
5	20	3	897	0.48	569	0.37	400	0.30
	30		10834	8.68	9462	9.54	3638	3.95
	40		46535	46.00	6648	9.66	16446	24.71
	50		65883	70.25	71204	153.23	4190	10.66
	60		120276	231.14	135379	310.91	13959	36.91
10	20		45183	31.32	24149	21.73	20749	19.47
	30		144407	201.66	46618	82.11	13469	22.31
5	20	5	1495	0.82	394	0.33	621	0.59
	30		15946	14.79	3987	4.65	8925	11.85
	40		86122	89.68	43625	66.97	8040	12.65
	50		108594	120.06	32238	61.47	62025	153.30
10	20		26848	19.96	16795	15.56	29080	32.22
	30		175324	210.64	41102	67.53	35649	73.05

Table 6.15: Branch and Bound Algorithm Results for *C2*

C2								
			S1		*S2*		*S3*	
m	*n*	*s*	Avg. # of nodes	Avg. CPU time	Avg. # of nodes in *B&B* tree	Avg. CPU time	Avg. # of nodes in *B&B* tree	Avg. CPU time
5	20	2	2333	1.77	819	0.64	405	0.24
	30		12099	10.03	716	0.97	134	0.20
	40		26249	27.70	1186	2.07	2286	3.66
	50		120666	159.22	61560	122.55	4771	9.17
	60		45191	56.92	9866	19.15	12688	26.22
10	20		38180	26.53	5068	3.78	637	2.54
	30		107883	232.58	49548	73.30	17405	41.91
5	20	3	899	0.48	602	0.39	396	0.25
	30		10834	8.29	9706	9.32	3766	4.01
	40		46527	46.01	5576	7.38	16966	26.78
	50		65879	72.09	71150	160.58	4277	10.74
	60		120011	186.56	134756	315.14	14163	37.75
10	20		46496	31.52	26899	22.99	16109	14.97
	30		146320	203.27	2338	3.37	15315	23.74
5	20	5	1496	0.82	558	0.41	212	0.18
	30		15951	15.25	4206	4.52	12389	13.16
	40		86135	92.67	46179	69.20	7386	11.44
	50		108591	117.02	32156	59.50	12241	26.18
10	20		26900	18.61	41440	40.51	18204	18.24
	30		166348	204.06	15293	20.90	22888	40.71

We next discuss the performance of our heuristic procedures. We evaluate the performances over the solution times and solution quality. For solution quality, we use the percentage deviation from the optimal solution.

$$\%Dev = \left(\frac{Heuristic\ Solution - OPT}{OPT} \right) \times 100$$

We first discuss the results of the tabu search algorithm. In our experiments, we set the *Tabutenure* to 50 and take *nonimplimit* and *maxiter* as 250 and 1000, respectively.

Table 6.16 reports the results of the tabu search algorithm for *s*=5. The results of the algorithm for s=2 and *s*=3 are given in Appendix A, in Tables A.1- A.2 and A.3-A.4, respectively.

As can be observed from the tables the deviations and the CPU times of the construction phase and the entire algorithm including the construction phase are very small. The maximum average construction time is 0.05 seconds and the maximum time construction takes over all instances is 0.27 seconds. As expected, for some instances the CPU times increase slightly as the number of agents or the number of jobs increase. Moreover the construction algorithm behaves consistently well over all instances in terms of CPU time.

When the CPU time of the entire tabu search algorithm is investigated, it is observed that the maximum average time is 0.28 seconds and the worst CPU time is 0.30 seconds for sets *S1C1* and *S1C2* when *m*=5, *n*=60 and *s*=2 (See Tables A.1 and A.2, Appendix A). The effects of *m* and *n* are clear in the solution times of the tabu search algorithm. As expected, the increase in these parameters, results in a slight increase in the solution time for almost all instances. This is because increase in *m* and *n* results in an increase in the problem size, thereby in the neighborhood size.

Table 6.16: Tabu Search Results for *s*=5

s=5								
C1								
			Tabu time		Construction time		% dev	
	m	*n*	Avg.	Max.	Avg.	Max.	Avg.	Max.
S1	5	20	0.04	0.05	0.03	0.11	0.00	0.00 (10)
		30	0.09	0.09	0.02	0.05	0.49	1.26 (5)
		40	0.16	0.17	0.02	0.03	0.42	1.13 (5)
		50	0.25	0.27	0.03	0.03	0.22	1.18 (6)
	10	20	0.07	0.08	0.03	0.06	0.93	5.48 (6)
		30	0.14	0.16	0.05	0.27	1.05	3.92 (5)
S2	5	20	0.04	0.05	0.02	0.03	0.00	0.00 (10)
		30	0.09	0.09	0.02	0.03	0.62	3.14 (6)
		40	0.15	0.17	0.02	0.05	0.35	3.46 (9)
		50	0.25	0.28	0.02	0.03	0.60	1.89 (4)
	10	20	0.05	0.06	0.03	0.05	0.19	0.64 (7)
		30	0.14	0.16	0.03	0.03	0.04	0.43 (9)
S3	5	20	0.05	0.05	0.02	0.05	0.00	0.00 (10)
		30	0.09	0.09	0.02	0.03	0.03	0.25 (9)
		40	0.16	0.17	0.02	0.03	0.00	0.00 (10)
		50	0.26	0.27	0.03	0.03	0.04	0.61 (8)
	10	20	0.04	0.06	0.03	0.03	2.31	6.92 (3)
		30	0.14	0.16	0.03	0.05	0.13	0.77 (7)
C2								
S1	5	20	0.05	0.05	0.02	0.03	0.00	0.00 (10)
		30	0.09	0.09	0.02	0.03	0.49	1.26 (5)
		40	0.16	0.17	0.03	0.03	0.42	1.13 (5)
		50	0.25	0.27	0.03	0.05	0.22	1.18 (6)
	10	20	0.07	0.08	0.02	0.03	0.38	1.43 (7)
		30	0.14	0.16	0.03	0.03	1.05	3.92 (5)
S2	5	20	0.05	0.05	0.02	0.03	0.00	0.00 (10)
		30	0.09	0.09	0.02	0.03	0.62	3.14 (6)
		40	0.16	0.17	0.02	0.05	0.35	3.46 (9)
		50	0.25	0.27	0.03	0.05	0.60	1.89 (4)
	10	20	0.07	0.08	0.02	0.03	0.00	0.00 (10)
		30	0.14	0.16	0.03	0.05	0.13	0.43 (7)
S3	5	20	0.05	0.05	0.02	0.03	0.00	0.00 (10)
		30	0.09	0.09	0.02	0.03	0.00	0.00 (10)
		40	0.16	0.17	0.02	0.03	0.00	0.00 (10)
		50	0.26	0.27	0.03	0.05	0.05	0.61 (8)
	10	20	0.07	0.08	0.03	0.03	0.08	0.38 (8)
		30	0.14	0.16	0.03	0.05	0.03	0.27 (9)

***** The figures in parenthesis indicate the number of times the optimal solution is found.

Moreover it can be observed from Tables 6.16, A.1, A.2, A.3 and A.4 that increasing the number of periods causes a slight increase in the CPU times.

Finally, we can conclude that the tabu search algorithm returns solutions in consistent CPU times over all processing time and capacity sets.

As can be observed from the tables all average deviations are below 2.5% and the maximum deviation is 6.92% at worst. This indicates that our tabu search algorithm performs consistently well over all instances. We observe a slight increase in deviations when *m (n)* values increase for the fixed *n (m)* for most settings.

We could not observe any significant and predictable effect of *s* values on the deviation results. As can be observed from Tables A.1, A.2, A3 and A4 for *s* values of 2 and 3, the processing time sets affect the average deviations. The deviations are relatively higher for *S1* set and the lowest deviations usually occur for *S3* set when all the other factors are the fixed. For *s*=5 (Table 6.16) there are more exceptions to this observation.

An increase in the capacity factor does not have a negative effect on the deviation results, with a few exceptions. The deviation usually stays the same or decreases when the capacity factor is increased. This is due to the fact that increasing the agent capacities, increases the number of feasible improving shifts and swap moves that the algorithm makes.

The tabu search algorithm finds 862 optimal solutions out of 1200 problem instances, i.e., in about 72% of the instances the optimal solution is produced.

Based on these results we can say that our *TS* algorithm is quite successful in finding good quality and quick solutions.

We next study the performance of our Branch and Bound based heuristic, namely the α approximation scheme. We performed experiments for three values of α, 0.1, 0.05 and 0.02.

Table 6.17 shows the results for α=0.1, when *s*=5. Tables B.1-B.2 and B.3-B.4 in appendix report the results when *s*=2 and *s*=3, respectively.

We observe that the algorithm performs different for set *S1* and *m*=10. We call this combination as *S1M10*. We study the results in two parts: For *S1M10* and the other combinations. For the sets other than *S1M10*, all average and maximum CPU times are below 0.5 and 2.03 seconds, respectively. The average and maximum number of nodes are at most 243 and 1149, respectively. Moreover the average deviations are less than 1.8% for almost all instances and the maximum deviation is 5.77%. The performance of the algorithm decreases considerably for *S1M10* with respective average and maximum CPU times of 13 and 78.38 seconds. The results are similar for the number of nodes: the average and maximum number of nodes increase up to 6900 and 46000, respectively. The deviations are more consistent than the CPU times and are similar to the other instances with a maximum deviation of 6.67%. Hence we can conclude that the α approximation scheme performs consistently well over all instances in terms of % deviation but this good performance comes with an increase the CPU times. Since this is a Branch and Bound based heuristic the effects of the parameters on the CPU time is similar to those of the Branch and Bound algorithm. Also it can be observed from the tables that in more than 30% of the instances (388 out of 1200), the heuristic finds the optimal solution.

For α=0.05, Table 6.18 reports the results when *s*=5 and Tables C.1- C.2 and C.3-C.4 report the results when *s*=2 and *s*=3, respectively. It is clear that as α gets closer to zero, the behavior of the algorithm becomes closer to the original Branch and Bound algorithm, hence decreasing α increases the CPU time and number of nodes while providing solutions closer to optimal solutions.

Table 6.17: α Approximation Scheme Results for α=0.1, *s*=5

s=5								
C1								
			Solution Time (CPU seconds)		Number of nodes in *B&B* Tree		%dev	
	m	*n*	Avg.	Max.	Avg.	Max.	Avg.	Max.
S1	5	20	0.04	0.09	19	160	0.54	3.66 (7)
		30	0.05	0.06	1	1	1.68	3.45 (1)
		40	0.08	0.11	1	1	1.82	3.46 (2)
		50	0.11	0.17	1	1	0.96	1.90 (1)
	10	20	4.93	33.41	4525	29677	0.82	4.11 (6)
		30	12.28	77.38	6891	45520	1.31	3.64 (4)
S2	5	20	0.03	0.05	2	10	0.70	2.42 (4)
		30	0.06	0.08	1	1	1.06	3.14 (2)
		40	0.08	0.09	1	1	1.01	4.09 (1)
		50	0.15	0.22	1	1	1.04	1.89 (0)
	10	20	0.06	0.13	5	20	0.82	3.18 (6)
		30	0.08	0.14	4	10	1.18	2.92 (1)
S3	5	20	0.03	0.05	4	15	0.41	2.42 (5)
		30	0.05	0.06	1	5	0.20	0.63 (3)
		40	0.08	0.11	1	5	0.29	1.18 (2)
		50	0.14	0.23	1	5	0.41	0.99 (0)
	10	20	0.48	2.03	243	1149	1.31	5.02 (1)
		30	0.08	0.09	5	10	0.57	2.61 (5)
C2								
S1	5	20	0.04	0.09	19	160	0.54	3.66 (7)
		30	0.06	0.08	1	1	1.68	3.45 (1)
		40	0.09	0.13	1	1	1.82	3.46 (2)
		50	0.12	0.19	1	1	0.96	1.90 (1)
	10	20	4.72	31.63	4518	29750	0.82	4.11 (6)
		30	11.96	77.30	6599	45320	1.31	3.64 (4)
S2	5	20	0.03	0.03	1	1	0.70	2.42 (4)
		30	0.05	0.06	1	1	1.06	3.14 (2)
		40	0.08	0.09	1	1	1.01	4.09 (1)
		50	0.15	0.22	1	1	1.04	1.89 (0)
	10	20	0.04	0.06	1	1	1.13	3.18 (5)
		30	0.08	0.14	1	1	1.39	3.75 (1)
S3	5	20	0.03	0.03	1	1	0.31	1.35 (7)
		30	0.05	0.09	1	1	0.32	1.29 (3)
		40	0.09	0.11	1	1	0.31	1.18 (2)
		50	0.15	0.22	1	1	0.40	0.99 (0)
	10	20	0.05	0.05	1	1	0.89	2.36 (4)
		30	0.07	0.08	1	1	0.49	1.82 (3)

***** The figures in parenthesis indicate the number of times the optimal solution is found.

Table 6.18: α Approximation Scheme Results for α=0.05, *s*=5

s=5								
C1								
			Solution Time (CPU seconds)		Number of nodes in *B&B* Tree		%dev	
	m	*n*	Avg.	Max.	Avg.	Max.	Avg.	Max.
S1	5	20	0.09	0.26	97.6	360	0.00	0.00 (10)
		30	0.14	0.36	101.0	320	1.24	2.30 (1)
		40	0.10	0.17	7.3	50	1.35	3.29 (3)
		50	0.12	0.17	1.0	1	0.96	1.90 (1)
	10	20	20.21	122.97	24207.9	145089	0.55	4.11 (8)
		30	127.19	788.50	49217.0	253140	0.47	1.00 (5)
S2	5	20	0.06	0.09	3.1	10	0.55	2.15 (4)
		30	0.08	0.11	2.9	20	0.90	3.14 (2)
		40	0.10	0.23	1.4	5	0.65	1.37 (1)
		50	0.22	0.52	1.0	1	1.04	1.89 (0)
	10	20	0.08	0.22	19.2	110	0.63	3.14 (6)
		30	0.12	0.17	8.3	20	1.14	2.92 (2)
S3	5	20	0.04	0.08	3.6	15	0.21	0.58 (5)
		30	0.05	0.06	1.4	5	0.20	0.63 (3)
		40	0.10	0.14	1.4	5	0.29	1.18 (2)
		50	0.16	0.27	1.4	5	0.41	0.99 (0)
	10	20	0.76	4.45	395.6	2652	0.69	1.93 (2)
		30	0.10	0.17	7.3	10	0.21	0.79 (7)
C2								
S1	5	20	0.09	0.27	97.6	360	0.00	0.00 (10)
		30	0.14	0.38	101.0	320	1.24	2.30 (1)
		40	0.10	0.17	7.3	50	1.35	3.29 (3)
		50	0.12	0.17	1.0	1	0.96	1.90 (1)
	10	20	18.95	132.16	23247.0	162734	0.67	4.11 (7)
		30	82.16	433.86	41689.0	230070	0.47	1.00 (5)
S2	5	20	0.04	0.05	2.2	5	0.70	2.42 (4)
		30	0.06	0.13	2.9	20	0.90	3.14 (2)
		40	0.08	0.16	1.4	5	0.65	1.37 (1)
		50	0.15	0.23	1.0	1	1.04	1.89 (0)
	10	20	0.07	0.19	18.2	110	0.94	3.18 (6)
		30	0.11	0.20	6.4	10	1.22	3.75 (3)
S3	5	20	0.04	0.05	1.0	1	0.31	1.35 (7)
		30	0.05	0.08	1.0	1	0.32	1.29 (3)
		40	0.09	0.13	1.0	1	0.31	1.18 (2)
		50	0.15	0.23	1.0	1	0.40	0.99 (0)
	10	20	0.06	0.13	8.2	10	0.70	2.36 (5)
		30	0.10	0.17	4.6	10	0.44	1.82 (4)

***** The figures in parenthesis indicate the number of times the optimal solution is found.

The increase in the CPU times and number of nodes and the decrease in the deviations can also be observed from Tables 6.18, C.1, C.2, C.3 and C.4. For the instances other than *S1M10*, the increase in CPU time is not significant with a few exceptions. However for *S1M10* the increase in CPU time is much more remarkable (maximum average CPU time is 127.19 and the CPU time is 788.5 seconds for the worst case for *S1C1*, *m*=10 *n*=30, s=5, over all such instances). Since the heuristic runs in exponential time, 2 instances (*S1C1 m*=10, *n*=30, *s*=3 and *S1C2 m*=10, *n*=30, *s*=3) could not be solved in our time limit of 20 minutes. These instances are excluded from solution time computations. The same situation occurs in terms of the number of nodes. The average deviations are below 1.8% over all instances and the maximum deviation is 4.26%. As solution quality increases more instances are solved to optimality, in about 38% of the instances (453 out of 1198), the optimal solution is found.

The results of the algorithm with α =0.02 are reported in Tables 6.19 for *s*=5 and D.1-D.2, D.3-D.4 for *s*=2 and *s*=3, respectively. When we decrease α to 0.02, we observe similar effects on the CPU time, number of nodes and deviations. Since the algorithm becomes much closer to the original Branch and Bound algorithm, the CPU times and number of nodes increase while the number of instances that can be solved within the time limit decreases. As we guarantee to have better results by decreasing α, the deviations from the optimal solution decreases and the number of instances that finds the optimal solution increases. In about half of the reported instances (585 out of 1182) the algorithm returns the optimal solution.

Table 6.19: α Approximation Scheme Results for α=0.02 *s*=5

s=5								
C1								
			Solution Time (CPU seconds)		Number of nodes in *B&B* Tree		%dev	
	m	*n*	Avg.	Max.	Avg.	Max.	Avg.	Max.
S1	5	20	0.37	1.14	609	2125	0.00	0.00(10)
		30	3.06	9.72	3235	11270	0.38	1.26 (6)
		40	1.49	4.05	1343	4560	0.58	1.64 (5)
		50	1.01	6.44	908	7020	0.52	1.37 (2)
	10	20	203.41	1897.22	46360	325527	0.00	0.00(10)**
		30	348.31	1097.48	121436	260199	0.23	0.95 (6)**
S2	5	20	0.05	0.11	7	25	0.36	2.15 (6)
		30	0.13	0.48	45	375	0.32	0.80 (3)
		40	0.10	0.17	4	10	0.42	1.37 (3)
		50	0.30	0.92	60	540	0.65	1.46 (0)
	10	20	1.41	13.25	1274	12584	0.38	1.27 (6)
		30	0.24	0.69	54	217	0.72	1.31 (3)
S3	5	20	0.04	0.09	5	15	0.17	0.58 (6)
		30	0.06	0.09	4	20	0.15	0.63 (5)
		40	0.09	0.16	2	5	0.27	1.18 (2)
		50	0.14	0.23	2	5	0.41	0.99 (0)
	10	20	4.33	33.41	2508	20352	0.65	1.93 (3)
		30	0.12	0.19	10	10	0.21	0.79 (7)
C2								
S1	5	20	0.42	1.30	605	2090	0.00	0.00(10)
		30	3.06	10.17	3233	11255	0.38	1.26 (6)
		40	1.47	3.97	1343	4560	0.58	1.64 (5)
		50	0.95	5.89	908	7015	0.52	1.37 (2)
	10	20	320.78	3083.70	48001	327039	0.14	1.43 (9)**
		30	337.65	1132.75	119453	261640	0.23	0.95 (6)**
S2	5	20	0.05	0.19	17	125	0.46	2.15 (5)
		30	0.12	0.45	45	375	0.32	0.80 (3)
		40	0.10	0.17	4	10	0.42	1.37 (3)
		50	0.25	0.77	60	540	0.65	1.46 (0)
	10	20	1.47	14.05	1442	14313	0.50	2.45 (6)
		30	0.24	0.50	33	160	0.72	1.67 (4)
S3	5	20	0.04	0.06	4	5	0.06	0.58 (9)
		30	0.06	0.08	3	5	0.28	1.29 (4)
		40	0.10	0.16	1	5	0.27	1.18 (2)
		50	0.14	0.23	1	5	0.40	0.99 (0)
	10	20	0.28	1.25	161	779	0.31	1.15 (7)
		30	0.13	0.17	10	10	0.42	1.82 (5)

***** The figures in parenthesis indicate the number of times the optimal solution is found.

****** 2 out of 10 instances could not be solved in *S1C1 m*=10, *n*=30 and *S1C2 m*=10, *n*=30 sets.

When we compare the tabu search and α approximation scheme, we see that the *TS* algorithm produces solutions faster. When a satisfactory solution is required quickly one can use *TS*. However no guarantee for its performance can be stated. The α approximation scheme guarantees that the worst case performance is below a predefined limit, but at an expense of higher CPU times. The α approximation scheme runs in exponential time but has the advantage of flexibility. If time is a scarce resource, α=0.1 approximation scheme can be used. When better solutions are required in tolerable time α can be set to 0.05. Finally, if the decision maker is more after near optimal solutions than quick ones then α can be set to 0.02. Note that, the computational time of the α approximation scheme even with α = 0.02 is significantly smaller than that of Branch and Bound Algorithm.

CHAPTER 7

CONCLUSIONS AND FURTHER RESEARCH DIRECTIONS

In this study, we consider the Multi Period Agent Bottleneck Generalized Assignment Problem. Our aim is to minimize the maximum load over all agents. We are motivated from a practical problem arising in Heat Ventilation and Air Conditioning (HVAC) industry.

We develop a Branch and Bound algorithm for optimal solutions and heuristic algorithms for approximate solutions. To the best of our knowledge, our algorithms are first attempts to solve the problem.

In our Branch and Bound algorithm, we use the optimal solutions of the Linear Programming (LP) relaxations in finding lower and upper bounds and defining our branching scheme. Our motivation is the satisfactory behavior of the LP relaxation in producing solutions with few continuous variables and objective function values that are very close to the optimal objective function values. We also derive simpler lower bounds and use them as filtering mechanisms. Our hope is to reduce the number of LP relaxation problems solved.

Our Branch and Bound algorithm could find optimal solutions for the problems with up to 60 jobs when the number of agents is 5 and up to 30 jobs when the number of agents is 10 in our plausible limit of 20 minutes. We find that the number of jobs and the number of agents are dominant factors in defining the complexity of the problems. However, the number of periods does not have a significant effect on the performance. Moreover, we observe that the distribution

of the processing time affects the complexity and the hardest to solve instances are observed when the processing time distribution has low mean and high variance.

Our heuristic procedures are of two types. One is tabu search and the other is α approximation Branch and Bound algorithm. We find that tabu search produces very quick, high quality solutions, hence can be used to solve very large sized problem instances. On the other hand α approximation algorithm runs in exponential time, with guaranteed performance. Our experiments have revealed that the α approximation algorithm runs considerably faster than Branch and Bound algorithm, hence can be used when guaranteed performance with quick solution times is required.

To the best our knowledge our study is the first attempt to solve the bottleneck generalized assignment problem with multiple periods. We hope our results stimulate further research in generalized assignment problem literature. Some noteworthy extensions of our work can be listed as:

- Defining off periods for the job requirements.
- Defining skill levels for the agents such that some agents perform all opportunities at a higher pace.
- Finding the polynomially solvable special cases of the problem.
- Developing Lagrangean relaxation based lower bounds.
- Developing Branch and Bound based heuristic procedures, like beam search, filtered beam search, that benefit from our bounding mechanisms.
- Defining a neighborhood that allows infeasibility in tabu search. This will help to search the feasible region better and increase the solution quality, however at an expense of increased computational time brought by repair mechanisms.
- Incorporating the cost aspects: In addition to or as an alternative to minimizing maximum load, minimizing total cost can be studied.

- Incorporating stochastic aspects of the parameters. For example, the processing times of the opportunities may vary as time progresses.

REFERENCES

Alfandari L., Plateau A., Tolla P. (2002), “A two-phase path relinking algorithm for the generalized assignment problem”, Technical Report No. 378, CEDRIC, CNAM.

Amini M.M., Racer M. (1994), “A rigorous computational comparison of alternative solution methods for the generalized assignment problem”, *Management Science* 40 868–890.

Amini MM, Racer M. (1995) “A hybrid heuristic for the generalised assignment problem”, *European Journal of Operational Research* 88, 343-8.

Avella P., Boccia M., Vasliyev I. (2008), *Comput. Optim. Appl.*, DOI 10.1007/s 10589-008-9183-8.

Balachandran V. (1972), “An integer generalized transportation model for optimal job assignment in computer networks”, Working Paper 34-72-3, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh.

Balas E. and Martin, E. (1980), “Pivot and complement - A heuristic for 0-1 programming”, *Management Science* 26/1, 86-96.

Bean J.C. (1984), “A Lagrangian Algorithm for the Multiple Choice Integer Program”, *Operations Research* 32, 1185-1193.

Campbell G.M., Diaby M. (2002), “Development and evaluation of an assignment heuristic for allocating cross-trained workers”, *European Journal of Operational Research* 138(1) 9–20.

Cattrysse D.G. (1990), “Set partitioning approaches to combinatorial optimization problems”, Ph.D. Thesis, Katholieke Universiteit Leuven, Departement Wertuigkunde, Centrum Industrieel Beleid, Belgium.

Cattrysse D., Degraeve Z., Tistaert J. (1998), “Solving the generalised assignment problem using polyhedral results”, *European Journal of Operational Research* 108, 618-628.

Cattrysse D.G., Van Wassenhove L.N. (1992), “A survey of algorithms for the generalized assignment problem”, *European Journal of Operational Research* 60, 260-272.

Cattrysse DG, Salomon M, Van Wassenhove LN (1994), “A set partitioning heuristic for the generalized assignment problem”, *European Journal of Operational Research* 72: 167-174.

Chu P.C., Beasley J.E. (1997), “A genetic algorithm for the generalised assignment problem”, *Computers Operations Research* Vol. 24, No. 1, pp. 17-23.

Díaz J.A., Fernández E. (2001), “A tabu search heuristic for the generalized assignment problem”, *European Journal of Operational Research* 132, 22-38.

Farias Jr. I.E. de, Nemhauser G.L. (2001), “A family of inequalities for the generalized assignment polytope”, *Operations Research Letters* 29, 49-55.

Fisher M.L. and Jaikumar R. (1981) “A generalized assignment heuristic for vehicle routing”, *Networks* 11, 109-124.

Fisher M.L., Jaikumar R. and Van Wassenhove L. (1986), “A multiplier adjustment method for the generalized assignment problem”, *Management Science* 32/9, 1095-1103.

Francis R.L., White J.A. (1974), “Facility layout and location”, Prentice-Hall, Englewood Cliffs, New Jersey.

Gavish B., Pirkul H. (1982), “Allocation of databases and processors in a distributed computing system”, *Management of Distributed Data Processing*, J.Akoka (Ed.), North Holland Publishing Company, Amsterdam.

Gavish B., Pirkul H. (1986), “Computer and database location in distributed computer systems”, *IEEE Trans. Computers* 35-7, 583-590.

Gavish B., Pirkul H. (1991), “Algorithms for the multi-resource generalized assignment problem”, *Management Science* Vol. 37 No. 6, 695-713.

Glover, F. and M. Laguna (1997), “Tabu Search”, Kluwer Academic Publishers, Norwell, MA.

Grigoriadis M.D., Tang D.J. and Woo L.S. (1974), “Considerations in the optimal synthesis of some communications networks”, presented at the 45th Joint National Meeting of ORSA/TIMS, Boston, MA.

Gross D. and Pinkus C.E. (1972), “Optimal allocation of ships to yards for regular overhauls”, Technical Memorandum 63095, Institute of Management Science Engineering, George Washington University, Washington, DC.

Guignard M. and Rosenwein M. (1989a), “An improved dual-based algorithm for the generalized assignment problem”, *Operations Research* 37/4, 658-663.

Haddadi S. (1999), “Lagrangian decomposition based heuristic for the generalized assignment problem”, *Information Systems and Operational Research* 37, 392–402.

Haddadi S., Ouzia H. (2001), “Effective Lagrangian heuristic for the generalized assignment problem”, *Information Systems and Operational Research* 39 351–356.

Haddadi S., Ouzia H. (2004), “Effective algorithm and heuristic for the generalized assignment problem”, *European Journal of Operational Research* 153-1, 184-190.

Hallefjord A, Jörnsten KO, Värbrand P (1993) “Solving large scale generalised assignment problem”, *Oper*. *Res*. 64: 103-104.

Higgins A. J. (2001), “A dynamic tabu search for large-scale generalised assignment problems”, *Computers and Operations Research* 28, 1039-1048.

Jörnsten K.O. and Näsberg M. (1986), “A new lagrangean relaxation approach to the generalised assignment problem”, *European Journal of Operational Research* 27:313-323.

Jörnsten K.O. and Värbrand P. (1987), “Two algorithms for the generalized assignment problem”, Working paper, LiTH-Mat-R- 87-02, Linköping Institute of Technology, Sweden.

Jörnsten K.O., Värbrand P. (1991), “A hybrid algorithm for the generalized assignment problem”, *Optimization* 2:273-282.

Karabakal, N., J. C. Bean, and J. R. Lohmann (1992), “A Steepest Descent Multiplier Adjustment Method for the Generalized Assignment Problem”, Report 92-11, University of Michigan, Ann Arbor, MI.

Klastorin T.D. (1979), “An effective sub-gradient algorithm for the generalised assignment problem”, *Computers and Operations Research* 6: 155-164.

Laguna M., Kelly J.P., Gonzilez-Velarde J.L. and Clover F. (1995), “Tabu search for the multilevel generalized assignment problem.” *European Journal of Operational Research*, 82 176-189.

Lourenço H.R., Serra D. (2002), “Adaptive search heuristics for the generalized assignment problem”, *Mathware and Soft Computing* 9 209–234.

Martello, S. and Toth, P. (1981a), “An algorithm for the generalized assignment problem”, in: J.P. Brans (ed.), *Operational Research '81*, North-Holland, Amsterdam, 589-603.

Martello S, Toth P (1981b), “An algorithm for the generalised assignment problem”, In proceedings of the 9th IFORS conference, Hamburg, Germany.

Martello S., Toth P. (1990), “Knapsack Problems: Algorithms and Computer Implemetations”, John Wiley and Sons.

Martello S., Toth P. (1995), “The bottleneck generalized assignment problem”, *European Journal of Operational Research* 83 (3) 621–638.

Mazzola J.B. (1989), “Generalized assignment with nonlinear capacity interaction”, *Management Science* 35 923–941.

Mazzola J.B., Neebe A.W. (1988), “Bottleneck generalized assignment problems”, *Engineering Cost and Production Economics* 14, 61-65.

Mazzola . J.B., Neebe A.W. and Dunn C.V.R. (1989), “Production planning of a Flexible Manufacturing System in a material requirements planning environment”, *International Journal of Flexible Manufacturing Systems* 1/2, 115-142.

Mazzola J.B., Neebe A.W. (1993), “An algorithm for the bottleneck generalized assignment problem”, *Computers Ops. Res.* Vol. 20 No 4, 366-362.

Mazzola J.B. and Wilcox, S.P. (1988), “Algorithms for the generalized assignment problem with multiple resource constraints”, presented at the Joint National Meeting of ORSA/TIMS.

Mazzola J.B., Wilcox S.P. (2001), “Heuristics for the multi-resource generalized assignment problem”, *Naval Research Logistics* 48, 468-483.

Mitrović-Minić S., Punnen A. (2009), “Local search intesified: Very large scale variable neighborhood search for the multi-resource generalized assignment problem”, Discrete Optimization, doi: 10.1016/*j*.disopt.2009.04.004.

Murph R.A (1986), “A private fleet model with multi-stop backhaul”, Working Paper 103, Optimal Decision Systems, Green Bay, WI 54306.

Nauss R.M. (2003), “Solving the generalized assignment problem: An optimizing and heuristic approach”, *INFORMS J. Comput*. 15(3), 249–266.

Nonobe K. and Ibaraki T. (1998), “A tabu search approach to the CSP (constraint satisfaction problem) as a general problem solver”, *European Journal of Operational Research*, Special Issue on Tabu Search 106, 599-623.

Osman I.H. (1995), “Heuristics for the generalised assignment problem: simulated annealing and tabu search approaches”, *OR Spektrum* 17, 211-225.

Park J.S., Ha Lim B., Lee Y. (1998), “A Lagrangian Dual-Based Branch-and-Bound Algorithm for the Generalized Multi-Assignment Problem”, *Management Science* 44-12, S271-S281.

Pentico D.W. (2007), “Assignment problems: A golden anniversary survey”, *European Journal of Operational Research* 176, 774-793.

Pigatti A., Aragão M.P.de, Uchoa E. (2005), “Stabilized branch-and-cut-and-price for the generalized assignment problem”, *Electronic Notes in Discrete Mathematics* 19, 389–395.

Pirkul H. (1986), “An integer programming model for the allocation of databases in a distributed computer system”, *European Journal of Operational Research* 26-3, 401-411.

Racer M, Amini M (1994), “A robust heuristic for the generalized assignment problem”, *Ann*. *Oper*. *Res*. 50:487-503.

Ross G.T. and Soland R.M. (1975), “A branch-and-bound algorithm for the generalized assignment problem”, *Mathematical Programming* 8, 91-103.

Ross G.T. and Soland R.M. (1977), “Modeling facility location problems as generalized assignment problems”, *Management Science* 24/3, 345-357.

Savelsbergh M. (1997), “A branch and price algorithm for the generalized assignment problem”, *Operations Research* 45-6, 831-841.

Trick M.A. (1992), “A Linear Relaxation Heuristic for the Generalized Assignment Problem”, *Naval Research Logistics* 39, 137-152.

Wilcox S.P. (1989), “A new multiplier adjustment method for the generalized assignment problem”, Working paper, General Research Corporation, Mc Lean, VI.

Yagiura M., Ibaraki T., Glover F. (2006), “A path relinking approach with ejection chains for the generalized assignment problem”, *European Journal of Operational Research* 169, 548-569.

Yagiura M., Ibaraki T., Glover F. (2004), “An ejection chain approach for the generalized assignment problem”, *INFORMS Journal on Computing* 16 133–151.

Yagiura M., Ibaraki T., Glover R. (1999), “An Ejection Chain Approach for the Generalized Assignment Problem”, Technical Report #99013, Department of Applied Mathematics and Physics, Graduate School of Informatics, Kyoto University. Available at http://www.or.amp.*i*.kyotou.ac.jp/_yagiura/ papers/.

Yagiura M., Iwasaki S., Ibaraki T., Glover F. (2004), “A very large-scale neighborhood search algorithm for the multi-resource generalized assignment problem”, *Discrete Optimization* 1, 87-98.

Yagiura M., Yamaguchi T., Ibaraki T. (1998), “A variable depth search algorithm with branching search for the generalized assignment problem”, *Optimization Methods* & *Software,* Vol. 10, 419-441.

Yagiura M., Yamaguchi T. and Ibaraki T. (1997), “A variable depth search algorithm for the generalized assignment problem”, *Proc. 2nd Metaheuristics Internationul Conference* (MIC) 97, 129- 130.

APPENDIX A

TABU SEARCH RESULTS FOR *s*=2 AND *s*=3

Table A.1: Tabu Search Results for *s*=2, *C1*

C1								
			Tabu time		Construction time		% dev	
	m	*n*	Avg.	Max.	Avg.	Max.	Avg.	Max.
S1	5	20	0.03	0.03	0.02	0.03	0.00	0.00(10)
		30	0.06	0.08	0.02	0.03	0.52	2.06 (6)
		40	0.11	0.13	0.02	0.03	0.39	0.81(5)
		50	0.19	0.20	0.02	0.03	0.24	0.63 (6)
		60	0.28	0.30	0.03	0.05	0.41	1.00 (4)
	10	20	0.05	0.06	0.02	0.03	1.60	6.45 (6)
		30	0.10	0.11	0.02	0.03	1.60	4.35 (4)
S2	5	20	0.03	0.03	0.02	0.03	0.08	0.76 (9)
		30	0.06	0.08	0.02	0.03	0.00	0.00 (10)
		40	0.09	0.13	0.02	0.03	0.15	1.15 (8)
		50	0.13	0.19	0.02	0.03	0.40	1.55 (4)
		60	0.25	0.31	0.04	0.11	0.47	1.32 (4)
	10	20	0.05	0.06	0.05	0.20	0.16	1.59 (9)
		30	0.10	0.11	0.02	0.03	0.22	1.08 (8)
S3	5	20	0.03	0.03	0.02	0.05	0.00	0.00 (10)
		30	0.06	0.06	0.02	0.03	0.06	0.32 (8)
		40	0.10	0.13	0.02	0.03	0.10	0.73 (8)
		50	0.17	0.22	0.03	0.05	0.06	0.20 (7)
		60	0.25	0.33	0.03	0.05	0.15	0.81 (5)
	10	20	0.05	0.05	0.03	0.05	0.10	0.96 (9)
		30	0.10	0.11	0.02	0.03	0.00	0.00 (10)

*The figures in parenthesis indicate the number of times the optimal solution is found.

Table A.2: Tabu Search Results for *s*=2, *C2*

C2								
			Tabu time		Construction time		% dev	
	m	*n*	Avg.	Max.	Avg.	Max.	Avg.	Max.
S1	5	20	0.03	0.03	0.02	0.03	0.00	0.00 (10)
		30	0.06	0.06	0.02	0.03	0.52	2.06 (6)
		40	0.12	0.13	0.02	0.03	0.39	0.81 (5)
		50	0.19	0.20	0.03	0.05	0.24	0.63 (6)
		60	0.28	0.30	0.02	0.03	0.41	1.00 (4)
	10	20	0.05	0.06	0.02	0.03	1.90	6.45 (6)
		30	0.11	0.11	0.02	0.05	1.60	4.35 (4)
S2	5	20	0.03	0.03	0.02	0.03	0.00	0.00 (10)
		30	0.07	0.08	0.02	0.03	0.00	0.00 (10)
		40	0.09	0.13	0.02	0.03	0.15	1.15 (8)
		50	0.13	0.19	0.04	0.17	0.40	1.55 (4)
		60	0.25	0.31	0.03	0.05	0.47	1.32 (4)
	10	20	0.05	0.06	0.02	0.03	0.16	1.59 (9)
		30	0.10	0.11	0.02	0.03	0.22	1.08 (8)
S3	5	20	0.02	0.03	0.02	0.03	0.00	0.00 (10)
		30	0.06	0.06	0.02	0.03	0.06	0.32 (8)
		40	0.11	0.13	0.02	0.03	0.10	0.73 (8)
		50	0.17	0.20	0.03	0.05	0.06	0.20 (7)
		60	0.24	0.28	0.03	0.05	0.15	0.81 (5)
	10	20	0.05	0.05	0.02	0.03	0.19	0.97 (9)
		30	0.11	0.11	0.02	0.03	0.00	0.00 (10)

*The figures in parenthesis indicate the number of times the optimal solution is found.

Table A.3: Tabu Search Results for *s*=3, *C1*

C1								
			Tabu time		Construction time		% dev	
	m	*n*	Avg.	Max.	Avg.	Max.	Avg.	Max.
S1	5	20	0.03	0.05	0.02	0.08	0.20	1.96 (9)
		30	0.08	0.08	0.02	0.03	0.26	1.95 (8)
		40	0.14	0.14	0.02	0.03	0.62	1.49 (3)
		50	0.22	0.23	0.03	0.05	0.49	1.25 (4)
		60	0.32	0.34	0.03	0.03	0.26	0.75 (5)
	10	20	0.06	0.06	0.03	0.06	1.48	6.38 (6)
		30	0.12	0.13	0.02	0.05	2.22	6.67 (2)
S2	5	20	0.04	0.05	0.02	0.05	0.00	0.00 (10)
		30	0.08	0.09	0.02	0.03	0.03	0.34 (9)
		40	0.13	0.16	0.02	0.03	0.13	0.77 (8)
		50	0.19	0.24	0.03	0.03	0.55	2.06 (5)
		60	0.27	0.36	0.03	0.03	0.42	1.92 (8)
	10	20	0.05	0.06	0.02	0.05	0.00	0.00 (10)
		30	0.12	0.13	0.03	0.06	0.14	0.71 (8)
S3	5	20	0.03	0.05	0.02	0.05	0.00	0.00 (10)
		30	0.07	0.08	0.02	0.03	0.00	0.00 (10)
		40	0.14	0.19	0.03	0.05	0.03	0.16 (8)
		50	0.20	0.23	0.03	0.03	0.23	1.93 (6)
		60	0.30	0.34	0.03	0.05	0.22	1.42 (6)
	10	20	0.04	0.05	0.03	0.06	0.06	0.65 (9)
		30	0.12	0.13	0.03	0.05	0.09	0.46 (8)

***** The figures in parenthesis indicate the number of times the optimal solution is found.

Table A.4: Tabu Search Results for *s*=3, *C2*

C2								
			Tabu time		Construction time		% dev	
	m	*n*	Avg.	Max.	Avg.	Max.	Avg.	Max.
S1	5	20	0.03	0.03	0.03	0.16	0.20	1.96 (9)
		30	0.08	0.08	0.02	0.03	0.26	1.95 (8)
		40	0.14	0.14	0.03	0.05	0.62	1.49 (7)
		50	0.22	0.23	0.02	0.03	0.49	1.25 (4)
		60	0.32	0.34	0.03	0.03	0.26	0.75 (5)
	10	20	0.06	0.06	0.02	0.03	1.28	6.38 (7)
		30	0.12	0.13	0.03	0.03	2.22	6.67 (2)
S2	5	20	0.03	0.05	0.02	0.03	0.00	0.00 (10)
		30	0.08	0.09	0.02	0.03	0.03	0.34 (9)
		40	0.12	0.16	0.03	0.03	0.15	1.03 (8)
		50	0.20	0.24	0.02	0.05	0.55	2.06 (5)
		60	0.28	0.38	0.05	0.16	0.42	1.92 (8)
	10	20	0.06	0.08	0.02	0.05	0.00	0.00 (10)
		30	0.12	0.14	0.02	0.03	0.07	0.70 (9)
S3	5	20	0.03	0.05	0.02	0.03	0.00	0.00 (10)
		30	0.08	0.08	0.02	0.03	0.00	0.00 (10)
		40	0.14	0.16	0.02	0.03	0.03	0.16 (8)
		50	0.20	0.23	0.03	0.03	0.23	1.93 (6)
		60	0.30	0.36	0.03	0.03	0.22	1.42 (6)
	10	20	0.06	0.06	0.04	0.22	0.00	0.00 (10)
		30	0.12	0.13	0.03	0.05	0.09	0.46 (8)

***** The figures in parenthesis indicate the number of times the optimal solution is found.

APPENDIX B

α APPROXIMATION SCHEME RESULTS FOR α = 0.1

Table B.1: α Approximation Scheme Results for α=0.1, *s*=2, *C1*

C1								
			Solution Time (CPU seconds)		Number of nodes in *B&B* Tree		%dev	
	m	*n*	Avg.	Max.	Avg.	Max.	Avg.	Max.
S1	5	20	0.05	0.11	9	85	0.97	2.78 (5)
		30	0.05	0.09	1	1	1.33	3.19 (2)
		40	0.07	0.09	1	1	0.99	3.88 (2)
		50	0.10	0.27	1	1	1.40	2.52 (0)
		60	0.15	0.19	1	1	0.77	2.14 (3)
	10	20	1.94	5.72	1956	5720	0.30	3.03 (9)
		30	6.72	49.15	3651	25230	2.04	4.55 (3)
S2	5	20	0.02	0.03	1	1	0.75	2.96 (5)
		30	0.05	0.08	1	5	0.25	1.02 (6)
		40	0.07	0.09	1	1	1.07	3.44 (2)
		50	0.10	0.13	1	1	1.42	3.41 (1)
		60	0.12	0.16	1	1	0.96	1.82 (2)
	10	20	0.04	0.06	2	10	0.92	4.62 (6)
		30	0.07	0.11	2	10	0.74	2.15 (5)
S3	5	20	0.02	0.05	1	1	0.57	1.44 (4)
		30	0.04	0.05	1	5	0.16	0.65 (7)
		40	0.06	0.08	1	1	0.63	1.23 (2)
		50	0.10	0.13	1	1	0.17	0.58 (4)
		60	0.13	0.19	1	1	0.53	2.60 (1)
	10	20	0.03	0.06	1	1	0.87	5.77 (7)
		30	0.06	0.08	1	1	0.79	2.65 (5)

* The figures in parenthesis indicate the number of times the optimal solution is found.

Table B.2: α Approximation Scheme Results for α=0.1, *s*=2, *C2*

C2								
			Solution Time (CPU seconds)		Number of nodes in *B&B* Tree		%dev	
	m	*n*	Avg.	Max.	Avg.	Max.	Avg.	Max.
S1	5	20	0.03	0.13	9	85	0.97	2.78 (5)
		30	0.04	0.06	1	1	1.33	3.19 (2)
		40	0.06	0.08	1	1	0.99	3.88 (2)
		50	0.08	0.14	1	1	1.40	2.52 (0)
		60	0.13	0.19	1	1	0.77	2.14 (3)
	10	20	1.97	6.31	2175	7850	0.30	3.03 (9)
		30	8.37	42.79	4821	23000	1.83	4.55 (3)
S2	5	20	0.02	0.03	1	1	0.75	2.96 (5)
		30	0.05	0.08	1	1	0.28	1.02 (5)
		40	0.06	0.08	1	1	1.07	3.44 (2)
		50	0.10	0.14	1	1	1.42	3.41 (1)
		60	0.12	0.14	1	1	0.96	1.82 (2)
	10	20	0.04	0.06	2	10	0.92	4.62 (6)
		30	0.07	0.11	2	10	0.74	2.15 (5)
S3	5	20	0.02	0.03	1	1	0.57	1.44 (4)
		30	0.03	0.05	1	1	0.16	0.65 (7)
		40	0.06	0.09	1	1	0.63	1.23 (2)
		50	0.09	0.13	1	1	0.17	0.58 (4)
		60	0.13	0.19	1	1	0.53	2.60 (1)
	10	20	0.03	0.05	1	1	0.87	5.77 (7)
		30	0.06	0.08	1	1	0.79	2.65 (5)

***** The figures in parenthesis indicate the number of times the optimal solution is found.

Table B.3: α Approximation Scheme Results for α=0.1, *s*=3, *C1*

C1								
			Solution Time (CPU seconds)		Number of nodes in *B&B* Tree		%dev	
	m	*n*	Avg.	Max.	Avg.	Max.	Avg.	Max.
S1	5	20	0.03	0.05	2	5	0.93	4.39 (7)
		30	0.04	0.06	1	1	1.80	4.55 (3)
		40	0.08	0.13	1	1	1.78	3.09 (0)
		50	0.11	0.16	1	1	1.03	3.35 (2)
		60	0.16	0.22	1	1	0.94	2.25 (1)
	10	20	4.11	14.23	5015	18865	1.05	6.38 (8)
		30	5.34	13.56	3232	9360	2.05	6.67 (2)
S2	5	20	0.03	0.03	1	1	0.40	1.49 (5)
		30	0.05	0.06	1	5	0.95	3.68 (3)
		40	0.07	0.13	1	1	0.54	1.50 (5)
		50	0.08	0.16	1	1	0.94	2.47 (1)
		60	0.15	0.28	1	1	0.78	1.92 (0)
	10	20	0.04	0.06	3	10	0.93	2.08 (4)
		30	0.05	0.11	1	1	0.91	3.50 (5)
S3	5	20	0.03	0.05	3	15	0.38	1.57 (6)
		30	0.05	0.06	2	5	0.67	2.17 (2)
		40	0.07	0.09	1	1	0.57	2.37 (2)
		50	0.10	0.13	2	10	0.58	1.93 (1)
		60	0.17	0.22	1	1	0.54	1.97 (1)
	10	20	0.05	0.11	6	30	1.28	3.23 (2)
		30	0.07	0.09	2	10	0.66	2.59 (3)

* The figures in parenthesis indicate the number of times the optimal solution is found.

Table B.4: α Approximation Scheme Results for α=0.1, *s*=3, *C2*

C2								
			Solution Time (CPU seconds)		Number of nodes in *B&B* Tree		%dev	
	m	*n*	Avg.	Max.	Avg.	Max.	Avg.	Max.
S1	5	20	0.03	0.06	2	5	0.93	4.39 (7)
		30	0.05	0.06	1	1	1.80	4.55 (3)
		40	0.07	0.11	1	1	1.78	3.09 (0)
		50	0.11	0.16	1	1	1.03	3.35 (2)
		60	0.16	0.22	1	1	0.94	2.25 (1)
	10	20	3.56	14.84	4459	20068	0.83	4.26(8)
		30	5.35	13.48	3260	9360	2.05	6.67 (2)
S2	5	20	0.03	0.03	1	1	0.40	1.49 (5)
		30	0.05	0.06	1	1	0.58	1.70 (4)
		40	0.07	0.13	1	1	0.56	1.50 (5)
		50	0.09	0.16	1	1	0.94	2.47 (1)
		60	0.15	0.28	1	1	0.78	1.92 (0)
	10	20	0.04	0.06	2	10	0.93	2.08 (4)
		30	0.06	0.11	1	1	0.91	3.50 (5)
S3	5	20	0.03	0.03	1	1	0.38	1.57 (5)
		30	0.05	0.08	1	1	0.67	1.27 (1)
		40	0.07	0.09	1	1	0.57	2.37 (2)
		50	0.10	0.14	1	1	0.58	1.93 (1)
		60	0.17	0.23	1	1	0.54	1.97 (1)
	10	20	0.04	0.05	1	1	1.03	2.55 (1)
		30	0.06	0.09	1	1	1.00	2.59 (2)

* The figures in parenthesis indicate the number of times the optimal solution is found.

APPENDIX C

α APPROXIMATION SCHEME RESULTS FOR α = 0.05

Table C.1: α Approximation Scheme Results for α=0.05, *s*=2, *C1*

C1								
			Solution Time (CPU seconds)		Number of nodes in *B&B* Tree		%dev	
	m	*n*	Avg.	Max.	Avg.	Max.	Avg.	Max.
S1	5	20	0.15	0.50	231	780	0.13	1.32 (9)
		30	0.08	0.41	68	555	1.13	3.19 (3)
		40	0.07	0.16	3	30	0.60	1.43 (3)
		50	0.08	0.13	1	1	1.40	2.52 (0)
		60	0.13	0.17	1	1	0.77	2.14 (3)
	10	20	9.57	30.02	12069	40638	0.30	3.03 (9)
		30	70.31	374.55	37927	194510	0.92	2.38 (6)
S2	5	20	0.03	0.03	1	5	0.75	2.96 (5)
		30	0.04	0.05	1	5	0.25	1.02 (6)
		40	0.07	0.16	4	40	0.73	2.24 (3)
		50	0.09	0.14	1	1	1.42	3.41 (1)
		60	0.12	0.16	1	1	0.96	1.82 (2)
	10	20	0.06	0.11	11	30	0.47	1.59 (7)
		30	0.09	0.09	5	10	0.74	2.15 (5)
S3	5	20	0.02	0.03	1	1	0.57	1.44 (4)
		30	0.03	0.05	1	5	0.16	0.65 (7)
		40	0.06	0.08	1	1	0.63	1.23 (2)
		50	0.09	0.13	1	1	0.17	0.58 (4)
		60	0.13	0.20	1	1	0.53	2.60 (1)
	10	20	0.04	0.06	4	10	0.49	1.98 (7)
		30	0.06	0.11	2	10	0.79	2.65 (5)

* The figures in parenthesis indicate the number of times the optimal solution is found.

Table C.2: α Approximation Scheme Results for α=0.05, *s*=2, *C2*

C2								
			Solution Time (CPU seconds)		Number of nodes in *B&B* Tree		%dev	
	m	*n*	Avg.	Max.	Avg.	Max.	Avg.	Max.
S1	5	20	0.14	0.47	231	780	0.13	1.32 (9)
		30	0.09	0.41	68	555	1.13	3.19 (3)
		40	0.07	0.16	3	30	0.60	1.43 (3)
		50	0.08	0.14	1	1	1.40	2.52 (0)
		60	0.14	0.19	1	1	0.77	2.14 (3)
	10	20	8.69	26.67	11060	29889	0.30	3.03 (9)
		30	72.05	363.84	39634	193840	0.92	2.38 (6)
S2	5	20	0.03	0.03	1	5	0.75	2.96 (5)
		30	0.04	0.06	1	1	0.28	1.02 (5)
		40	0.07	0.16	4	40	0.73	2.24 (3)
		50	0.09	0.14	1	1	1.42	3.41 (1)
		60	0.11	0.14	1	1	0.96	1.82 (2)
	10	20	0.05	0.08	9	10	0.47	1.59 (7)
		30	0.08	0.09	5	10	0.74	2.15 (5)
S3	5	20	0.02	0.03	1	1	0.57	1.44 (4)
		30	0.04	0.05	1	1	0.16	0.65 (7)
		40	0.06	0.08	1	1	0.63	1.23 (2)
		50	0.09	0.11	1	1	0.17	0.58 (4)
		60	0.12	0.19	1	1	0.53	2.60 (1)
	10	20	0.04	0.06	4	10	0.49	1.98 (7)
		30	0.07	0.11	2	10	0.79	2.65 (5)

* The figures in parenthesis indicate the number of times the optimal solution is found.

Table C.3: α Approximation Scheme Results for α=0.05, *s*=3, *C1*

C1								
			Solution Time (CPU seconds)		Number of nodes in *B&B* Tree		%dev	
	m	*n*	Avg.	Max.	Avg.	Max.	Avg.	Max.
S1	5	20	0.09	0.25	83	295	0.20	1.96 (9)
		30	0.12	0.41	96	435	0.68	1.90 (5)
		40	0.08	0.11	6	55	1.78	3.09 (0)
		50	0.11	0.19	1	1	1.03	3.35 (2)
		60	0.16	0.22	1	1	0.94	2.25 (1)
	10	20	15.26	59.34	19328	71793	0.83	4.26 (8)
		30	130.35	459.88	70288	250250	0.70	1.67 (5)
S2	5	20	0.03	0.05	1	1	0.40	1.49 (5)
		30	0.05	0.09	1	10	0.58	1.70 (4)
		40	0.09	0.14	1	1	0.54	1.50 (5)
		50	0.10	0.17	1	1	0.94	2.47 (1)
		60	0.17	0.30	1	1	0.78	1.92 (0)
	10	20	1.06	10.11	837	8304	0.62	2.08 (5)
		30	4.82	47.34	1829	18244	0.70	3.50 (5)
S3	5	20	0.04	0.06	2	15	0.38	1.57 (6)
		30	0.05	0.06	1	5	0.67	2.17 (2)
		40	0.07	0.09	1	1	0.57	2.37 (2)
		50	0.10	0.14	1	10	0.58	1.93 (1)
		60	0.17	0.22	1	1	0.54	1.97 (1)
	10	20	0.14	0.66	42	336	1.28	3.23 (2)
		30	0.08	0.13	3	10	0.40	1.37 (4)

***** The figures in parenthesis indicate the number of times the optimal solution is found.

Table C.4: α Approximation Scheme Results for α=0.05, *s*=3, *C2*

C2								
			Solution Time (CPU seconds)		Number of nodes in *B&B* Tree		%dev	
	m	*n*	Avg.	Max.	Avg.	Max.	Avg.	Max.
S1	5	20	0.08	0.22	83	295	0.20	1.96 (9)
		30	0.12	0.38	96	435	0.68	1.90 (5)
		40	0.08	0.13	6	55	1.78	3.09 (0)
		50	0.11	0.16	1	1	1.03	3.35 (2)
		60	0.16	0.22	1	1	0.94	2.25 (1)
	10	20	14.98	60.70	18888	71488	0.83	4.26 (8)
		30	152.60	600.05	70273	249920	0.70	1.67 (5)
S2	5	20	0.04	0.05	1	1	0.40	1.49 (5)
		30	0.05	0.08	1	1	0.58	1.70 (4)
		40	0.07	0.13	1	1	0.56	1.50 (5)
		50	0.08	0.14	1	1	0.94	2.47 (1)
		60	0.16	0.30	1	1	0.78	1.92 (0)
	10	20	1.12	10.75	1026	10190	0.62	2.08 (5)
		30	4.90	48.17	2228	22230	0.70	3.50 (5)
S3	5	20	0.03	0.03	1	1	0.38	1.57 (5)
		30	0.05	0.06	1	1	0.67	1.27 (1)
		40	0.07	0.09	1	1	0.57	2.37 (2)
		50	0.10	0.14	1	1	0.58	1.93 (1)
		60	0.17	0.22	1	1	0.54	1.97 (1)
	10	20	0.10	0.51	36	279	0.96	2.55 (2)
		30	0.08	0.23	4	10	0.57	2.18 (4)

***** The figures in parenthesis indicate the number of times the optimal solution is found.

APPENDIX D

α APPROXIMATION SCHEME RESULTS FOR α = 0.02

Table D.1: α Approximation Scheme Results for α=0.02, *s*=2, *C1*

C1								
			Solution Time (CPU seconds)		Number of nodes in *B&B* Tree		%dev	
	m	*n*	Avg.	Max.	Avg.	Max.	Avg.	Max.
S1	5	20	0.70	2.20	1225	4210	0.13	1.32 (9)
		30	2.01	10.28	2266	11000	0.40	1.06 (6)
		40	0.74	3.08	695	3425	0.46	0.81 (4)
		50	0.29	1.19	200	1310	0.91	1.28 (1)
		60	0.31	1.39	142	1125	0.56	1.50 (3)
	10	20	14.76	41.45	21072	64122	0.00	0.00(10)
		30	65.80	139.53	33280	70320	0.00	0.00(8)**
S2	5	20	0.05	0.17	27	202	0.08	0.78 (9)
		30	0.05	0.09	3	5	0.10	0.51 (8)
		40	0.61	5.33	285	2825	0.47	1.56 (3)
		50	0.17	0.36	22	85	0.58	1.23 (1)
		60	0.23	0.64	47	285	0.52	1.32 (3)
	10	20	0.23	1.06	188	1136	0.16	1.59 (9)
		30	1.90	17.28	1060	10211	0.32	1.08 (7)
S3	5	20	0.03	0.03	4	5	0.33	0.97 (6)
		30	0.04	0.05	4	15	0.16	0.65 (7)
		40	0.13	0.56	24	220	0.34	0.98 (4)
		50	0.09	0.11	1	5	0.17	0.58 (4)
		60	0.16	0.50	15	145	0.40	1.30 (1)
	10	20	0.10	0.52	1	1	0.19	0.97 (8)
		30	0.11	0.16	1	1	0.53	1.34 (5)

***** The figures in parenthesis indicate the number of times the optimal solution is found.

****** 2 out of 10 instances could not be solved in *S1 m*=10, *n*=30 set.

Table D.2: α Approximation Scheme Results for α=0.02, *s*=2, *C2*

C2								
			Solution Time (CPU seconds)		Number of nodes in *B&B* Tree		%dev	
	m	*n*	Avg.	Max.	Avg.	Max.	Avg.	Max.
S1	5	20	0.77	2.47	1219	4220	0.13	1.32 (9)
		30	1.99	9.98	2269	11030	0.40	1.06 (6)
		40	0.76	3.14	695	3425	0.46	0.81 (4)
		50	0.28	1.11	200	1310	0.91	1.28 (1)
		60	0.30	1.34	142	1125	0.56	1.50 (3)
	10	20	20.39	90.38	24519	94889	0.00	0.00(10)
		30	66.59	126.08	32690	70950	0.00	0.00(8)**
S2	5	20	0.06	0.19	27	205	0.08	0.78 (9)
		30	0.06	0.08	3	5	0.11	0.51(7)**
		40	0.62	5.50	285	2825	0.47	1.56 (3)
		50	0.18	0.42	22	85	0.58	1.23 (1)
		60	0.22	0.63	47	285	0.52	1.32 (39
	10	20	0.22	1.00	212	1334	0.16	1.59 (9)
		30	4.76	46.19	3396	33660	0.32	1.08 (7)
S3	5	20	0.02	0.03	4	5	0.33	0.97 (6)
		30	0.04	0.05	3	5	0.16	0.65 (7)
		40	0.12	0.56	24	220	0.34	0.98 (4)
		50	0.10	0.14	1	5	0.17	0.58 (4)
		60	0.16	0.53	15	145	0.40	1.30 (1)
	10	20	0.10	0.45	1	1	0.19	0.97 (8)
		30	0.10	0.16	1	1	0.53	1.34 (5)

***** The figures in parenthesis indicate the number of times the optimal solution is found.

****** 2 out of 10 instances could not be solved in *S1* *m*=10, *n*=30 set.

1 out of 10 instances could not be solved in *S2* *m*=5, *n*=30 set.

Table D.3: α Approximation Scheme Results for α=0.02, *s*=3, *C1*

C1								
			Solution Time (CPU seconds)		Number of nodes in *B&B* Tree		%dev	
	m	*n*	Avg.	Max.	Avg.	Max.	Avg.	Max.
S1	5	20	0.22	0.44	328	785	0.00	0.00(10)
		30	1.20	2.66	1567	3995	0.40	1.40 (6)
		40	1.52	3.25	1575	3175	0.51	1.49 (4)
		50	0.67	3.95	497	3220	0.45	0.87 (3)
		60	0.63	2.16	293	1220	0.62	1.50 (1)
	10	20	22.11	66.30	29301	88688	0.00	0.00(10)
		30	229.36	925.19	75975	254187	0.25	1.52(5)**
S2	5	20	0.04	0.06	8	28	0.15	0.52 (7)
		30	0.53	4.69	351	3445	0.52	1.70 (5)
		40	0.12	0.25	15	95	0.39	1.29 (6)
		50	0.26	0.69	63	365	0.39	1.03 (3)
		60	0.29	1.31	50	480	0.56	1.19 (0)
	10	20	25.51	234.99	25118	235058	0.52	2.08 (6)
		30	0.56	2.59	63	477	0.47	1.43(4)**
S3	5	20	0.05	0.13	7	40	0.22	1.24 (7)
		30	0.08	0.13	4	10	0.39	0.88 (3)
		40	0.10	0.17	3	15	0.37	0.94 (2)
		50	0.14	0.25	2	10	0.39	1.04 (2)
		60	0.23	0.39	5	35	0.38	1.52 (1)
	10	20	5.18	50.34	3697	36328	0.38	0.65 (4)
		30	0.38	2.92	125	1161	0.40	1.37 (4)

***** The figures in parenthesis indicate the number of times the optimal solution is found.

****** 4 out of 10 instances could not be solved in *S1* *m*=10, *n*=30 set.

1 out of 10 instances could not be solved in *S2* *m*=10, *n*=30 set.

Table D.4: α Approximation Scheme Results for α=0.02 *s*=3, *C2*

C2								
			Solution Time (CPU seconds)		Number of nodes in *B&B* Tree		%dev	
	m	*n*	Avg.	Max.	Avg.	Max.	Avg.	Max.
S1	5	20	0.22	0.47	328	785	0.00	0.00(10)
		30	1.27	2.78	1567	3995	0.40	1.40 (6)
		40	1.52	2.88	1575	3175	0.51	1.49 (4)
		50	0.67	3.98	500	3220	0.45	0.87 (3)
		60	0.66	2.28	294	1220	0.62	1.50 (1)
	10	20	24.57	87.20	29557	87619	0.00	0.00(10)
		30	163.16	452.05	64458	179690	0.00	0.00(6)**
S2	5	20	0.05	0.08	9	30	0.15	0.52 (7)
		30	0.52	4.77	352	3490	0.52	1.70 (5)
		40	0.11	0.23	15	95	0.41	1.29 (5)
		50	0.25	0.66	63	365	0.39	1.03 (3)
		60	0.29	1.30	50	480	0.56	1.19 (0)
	10	20	95.85	924.80	29127	278208	0.52	2.08 (6)
		30	0.58	2.41	63	480	0.47	1.43(4)**
S3	5	20	0.05	0.09	3	5	0.10	0.64 (8)
		30	0.07	0.16	3	5	0.47	0.88 (2)
		40	0.10	0.20	3	15	0.37	0.94 (2)
		50	0.14	0.25	2	10	0.43	1.04 (1)
		60	0.26	0.56	5	35	0.38	1.52 (1)
	10	20	0.75	5.53	554	4207	0.58	1.94 (4)
		30	0.37	2.66	128	1182	0.40	1.37 (4)

***** The figures in parenthesis indicate the number of times the optimal solution is found.

****** 4 out of 10 instances could not be solved in *S1* *m*=10, *n*=30 set.

1 out of 10 instances could not be solved in *S2* *m*=10, *n*=30 set.