UNLV University Libraries
University of Nevada, Las Vegas

UNLV Theses, Dissertations, Professional Papers, and Capstones

5-1-2019

Benchmarking Permutation Flow Shop Problem: Adaptive and Enumerative Approaches Implementations via Novel Threading Techniques

Hari Prasad Sapkota
harrysapkota@gmail.com

Follow this and additional works at: https://digitalscholarship.unlv.edu/thesesdissertations

 Part of the Computer Sciences Commons

Repository Citation

Sapkota, Hari Prasad, "Benchmarking Permutation Flow Shop Problem: Adaptive and Enumerative Approaches Implementations via Novel Threading Techniques" (2019). *UNLV Theses, Dissertations, Professional Papers, and Capstones*. 3668.
https://digitalscholarship.unlv.edu/thesesdissertations/3668

This Thesis is protected by copyright and/or related rights. It has been brought to you by Digital Scholarship@UNLV with permission from the rights-holder(s). You are free to use this Thesis in any way that is permitted by the copyright and related rights legislation that applies to your use. For other uses you need to obtain permission from the rights-holder(s) directly, unless additional rights are indicated by a Creative Commons license in the record and/or on the work itself.

This Thesis has been accepted for inclusion in UNLV Theses, Dissertations, Professional Papers, and Capstones by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.

BENCHMARKING PERMUTATION FLOW SHOP PROBLEM: ADAPTIVE AND ENUMERATIVE APPROACHES IMPLEMENTATIONS VIA NOVEL THREADING TECHNIQUES

By

Hari Prasad Sapkota

Bachelor of Computer Engineering
Tribhuvan University
2012

A thesis submitted in partial fulfillment
of the requirements for the

Master of Science in Computer Science

Department of Computer Science
Howard R. Hughes College of Engineering
The Graduate College

University of Nevada, Las Vegas
May 2019

© Hari Prasad Sapkota, 2019
All Rights Reserved

UNLV | GRADUATE COLLEGE

Thesis Approval

The Graduate College
The University of Nevada, Las Vegas

April 11, 2019

This thesis prepared by

Hari Prasad Sapkota

entitled

Benchmarking Permutation Flow Shop Problem: Adaptive and Enumerative Approaches Implementations via Novel Threading Techniques

is approved in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science
Department of Computer Science

Wolfgang Bein, Ph.D.
Examination Committee Chair

Kathryn Hausbeck Korgan, Ph.D.
Graduate College Dean

Ajoy Datta, Ph.D.
Examination Committee Member

Laxmi Gewali, Ph.D.
Examination Committee Member

Henry Selvaraj, Ph.D.
Graduate College Faculty Representative

Abstract

A large number of real-world planning problems are combinatorial optimization problems which are easy to state and have a finite but usually very large number of feasible solutions. The minimum spanning tree problem and the shortest path problem are some which are solvable through polynomial algorithms. Even though there are other problems such as crew scheduling, vehicle routing, production planning, and hotel room operations which have no properties such as to solve the problem with polynomial algorithms. All these problems are NP-hard. The permutation flow shop problem is also NP-hard problem and they require high computation. These problems are solvable as in the form of the optimal and near-optimal solution. Some approach to get optimal are exhaustive search and branch and bound whereas near optimal are achieved annealing, Genetic algorithm, and other various methods.

We here have used different approach exhaustive search, branch and bound and genetic algorithm. We optimize these algorithms to get performance in time as well as get the result closer to optimal. The exhaustive search and branch and bound gives all possible optimal solutions. We here have shown the comparative result of optimal calculation for 10 jobs with varying machine number up to 20. The genetic algorithm scales up and gives results to the instances with a larger number of jobs and machines.

Acknowledgements

"I would like to wholeheartedly thank my thesis advisor and chair, Dr. Wolfgang Bein, without whose invaluable constructive criticism, suggestive guidelines, support and encouragement, this thesis would not be in the form it is today.

I am grateful to Dr. Ajoy Datta, Dr. Laxmi Gewali and Dr. Henry Selvaraj for accepting to be on the committee and their tremendous support, invaluable feedback and encouragement.

I would like to express my sincere gratitude to Computer Science department at Howard R. Hughes College of Engineering, University of Nevada, Las Vegas for providing me with the learning opportunity as well as graduate assistantship.

I am thankful to my graduate assistantship instructor Dr. Edward Jorgensen for being helpful throughout my study and for valuable suggestions.

I am also grateful to all my professors for their suggestions and inspirational lectures which were beneficial for my thesis.

Finally, my thanks go to my family and all my friends who have offered their insightful suggestions and comments."

Hari Prasad Sapkota

University of Nevada, Las Vegas

May 2019

This thesis is dedicated to my parents and my wife for their love, endless support and encouragement.

Table of Contents

List of Tables

List of Figures

List of Algorithms

List of Acronyms

Cmax : Makespan

MPM : Multi Purpose Machine

NIQ : No Intermediate Queues

SDST : Sequence-Dependent Set-up Times

SMT : Surface Mount Technology

PCB : Printed Circuit Board

MILP : Mixed Integer Linear Program

Ex : Exhaustive Search

BB : Branch and Bound

GA : Genetic Algorithm

BFS : Breadth First Search

DFS : Depth First Search

BeFS : Best First Search

LB : Lower Bound

UB : Upper Bound

AR : Approximation Ratio

Chapter 1

Introduction

The world is developing in various fields with the aid of technologies. The sequencing is one of the common problem occurring frequently. The manufacturing sector, airlines sector, banking sector and various other day to day life activities involve sequencing to great extent [RWCM03]. In economic and industrial field flow shop can be implied to great extent.This has lured many researchers to work on problem with diverse classical assumptions and different objective functions and by implementing various optimization techniques.There are two main elements of flow shop problem: (i) M machines for job processing and (ii) N jobs to be processed on those machines. The jobs and machines have their criteria. All jobs sequence execute in same machine order. The job is not executed multiple times on same machine. A job cannot be executed on multiple machines neither a machine can execute multiple jobs. At first Conway et al. (1967) have devised the notation for flow shop with makespan criterion as $n/m/F/c_{max}$ [RWCM03] later Graham et al. (1979) introduce new notation as $F||c_{max}$ [R.L79]. The makespan criterion is defined as the total time of completing all the jobs on all machines.This problem has $(n!)^m$ alternative sequence of jobs over the machines. The classical flow shop problem is assumed to have a buffer or queue to hold the jobs between the machines (Allahverdi et al. 1999 [AGA99]). Later different variants were developed.Blocking flow shop problem was stated by Abadi, Hall and Sriskandarajah (1995) [AHS00]. Aldowaisan and Allahverdi (1998) gave a concept on no-wait flow shop problem [AA98]. The similar concept was earlier pitched by Piehler (1960), Reddi and Ramamoorthy (1972), Bonney and Gundry (1976), King and Spachis (1980), Gangadharan and Rajendran (1993) and Rock (1984) whereas no intermediate queues (NIQ) flow shop problem was mentioned by Stafford (1988), Stafford and Tseng (1990), and Wismer (1972) [S05] . The survey of Hall and Sriskandarajah (1996) aided to deduce the computational complexity for a varieties of approach to problems describing also the different

application [HS96]. Thus it make no-wait flow shop problem a topic of interest (e.g. Bertolissi 2000 [Ber00], Aldowaisan and Allahverdi 2004 [AA04], *passim*). In the beginning of 2000, hybrid flow shop problems came to eyes of researchers.

The flow shop environment in real life scenario is not as stated in classical flow shop problem where set-up times are assumed to be unaffected by the job's position in sequence. Hence set-up time can be added in processing time of job. In fact the real life flow shop problem has sequence-dependent set-up times (SDST). The surface mount technology (SMT) and printed circuit board (PCB) manufacturing environments have SDST flow shop problem. A mixed integer linear program (MILP) model is suggested first by Srikar and Ghosh (1986) for SDST flow shop problem [SG86]. Later, Stafford and Tseng (1990) [JT90] and Rios-Mercado and Bard (1999) [RMB99] did further work on the problem. Tseng and Stafford (2001 [TJ01],2002 [ST02]) proposed two MILP models. The flow shop problem with SDST and makespan criterion was taken into consideration by Ruiz *et al.* (2004) [RMA05]. The good performing metaheuristics of a regular flow shop problem as well as advanced genetic algorithms were proposed. The flow shop problem with setup times were divided into four categories by Allahverdi *et al.* (1999) [AGA99] as sequence independent non-batch set-ups, sequence-dependent non-batch set-ups, sequence independent batch set-ups, and sequence-dependent batch set-ups [AGA99]. The researchers put their effort based on these categories to solve the problem in real world. For PCB manufacturing environment Lee and Shaw (2000) have done a great deal of work through the minimizing the objective function concept [LS00]. The travelling salesman problem is used to model the problem F/no-idle/$C_{\max}$ by Saadani *et al.* (2004) [SGM05]. All the previous mentioned are serial flow shop however the concept of a concurrency in flow shop is shown by Lee *et al.* (1993) [LCL93] and Potts *et al.* (1995) [PSS+95].

Simultaneously, with the work on flow shop, the special case of the permutation flow shop was also studied by the researchers with great deal of interest since 1960. In the permutation flow shop problem the same job order is followed by all machines and is denoted as $F|prmu|c_{max}$ [R.L79]. Dudek and Teuton (1964) [DT64] had explained the basic assumptions for this problem in detail. The optimal calculation approaches had been approached earlier by Szwarc (1997) [Szw71], Lageweg *et al*(1978) [LLRK78], Potts (1980) [Pot80], and Carlier and Reba(1996) [CR96]. The permutation flow shop problem is known to be a NP-hard problem for three or more machines [GJS76]. Hence, heuristics approach had been a way of getting a near-optimal solutions. Yet, there is no perfect framework that had been the best. Many researchers had tried to classify problem and the heuristics for better results [Lou96]. Gupta(1979) [Gup79], King and Spachis(1980) [KS80] and Parl *et al.*(1984) [PPE84] had

given the review on development of heuristics. All these heuristics happen to have different nature such as computation time, complexity order or memory requirement. Further, Widmer and Hertz (1989) [WH89], Moccellin (1995) [Moc95] and Nawaz *et al.*(1983) [NEH83] have worked on new better heuristics. There were even earlier attempts to classify the flow shop heuristics as fixed functional heuristics, floating functional heuristics, and synthetic functional heuristics. Framimam *et al.*(2004) [FGL04] presented a general framework for the development of the heuristics. There are approaches to handle the uncertainties that may occur during the job execution. Studies had been initiated by Gholami, Zandieh, and Alem-Tabriz(2009) [GZAT09] and Ouelhadj and Petrovic(2009) [OP08] which later grab more researchers towards the area. While the researchers are investing their effort on various nature of permutation flow shop problem, we here took a simple dig on the exhaustive search and branch and bound method for the optimal. The both methods are consuming the multi-threading approach with some tweaks in the algorithm. In addition, for near-optimal solution, we took genetic algorithm and converted it into a concurrent approach with some tweaks in the algorithm. These methods requires extensive computation which we are trying to minimize and add the concurrent approach for faster execution. It is also known that the optimal solution though branch and bound techniques are most widely used to examine the performance for most studies involving heuristics [BH91]. Hence, we here have expected these approach will aid in further research in the field of static permutation flow shop problem.

In the first chapter we started with the previous related works followed by short introduction to scheduling and shop problems. Later, flow shop and permutation flow shop problem are explained in detail with example. We concluded first chapter with the algorithm for makespan calculation. In *chapter two*, we briefly describe branch and bound algorithm along with the related history and explained the implementation of BB algorithm in permutation flow shop problem. At the end, we showed the multi-threaded approach of the BB algorithm. Genetic algorithm and its implementation in detail is described in *chapter three*. The multi-threaded approach to genetic algorithm for permutation flow shop problem is illustrated at the end of the chapter. The next chapter is the results comparison and analysis followed by chapter with conclusion and future work.

1.1 Scheduling

A schedule is the allocation of intervals for each job to be processed on machines. The jobs can be of n different types and machines can be m. For the illustrative purpose of jobs and its relation to machines Gantt charts are good. There are two ways that are used in representing the Gantt

chart, first, machine-oriented and next is job-oriented.

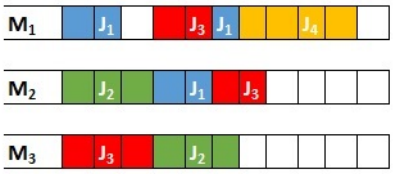


Figure 1.1: Machine oriented Gantt Chart

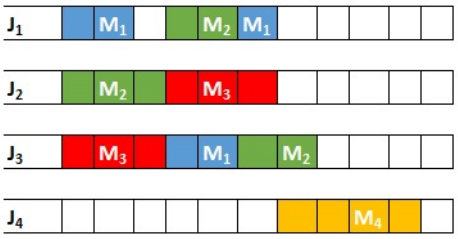


Figure 1.2: Job oriented Gantt Chart

As seen in the figure above, machine-oriented has machines fixed and jobs are places as per intervals. Similarly, job oriented has jobs fixed and machines are places as per intervals.

1.1.1 Job Data

We follow Peter Brucker[Bru07] notation stating for scheduling problems. J_i is a job consisting of operations O_{i1}, O_{i2}, ... , O_{in_i} where n_i is number of operations for the job. For each operation O_{ij}, there is certain processing time p_{ij}. Assume J_i has $\mathrm{n_i} = 1$, its operation is now O_{i1} only then processing time is denoted by p_i. r_i is a release date, on which the first operation of J_i becomes available for processing may be specified. Associated with each operation O_{ij} is a set of machines $\mu_{ij} \subseteq \{M_1, \ldots, M_m\}$. O_{ij} may be processed on any of the machines in μ_{ij}. Usually, all μ_{ij} are one element sets or all μ_{ij} are equal to the set of all machines. Machine-oriented has **dedicated machines** while job oriented has machines as **parallel**. **Multi-purpose machines** (MPM) are adjusted such that they can process operation. These MPM are used in real-world problems like flexible manufacturing has various steps of operations performed by machines with different tools. While processing O_{ij} may use all machines in the set μ_{ij}, scheduling problems as such are called **multiprocessor task scheduling problems.** A **cost function** $f_i(t)$ is defined by a **due date** d_i and a **weight** w_i measures the cost of completing J_i at time t. The symbols p_i, p_{ij}, r_i, d_i and w_i are all assumed to be integers. The scheduling problems are classified based on a three-fields $\alpha|\beta|\gamma$ classification where α specifies the **machine environment**, β specifies the **job characteristics** and γ denotes the **optimality criterion**. This classification scheme was introduced by Grahm *et al.* [R.L79].

1.1.2 Job Characteristics

There are at most six parameters to define the job characteristics usually β is used as β_1, β_2, β_3, β_4, β_5 ,and β_6. Preemption is indicated by β_1 as $\beta_1 = pmtn$ if allowed else β_1 is not present in β. Precedence relations is described through β_2. The values for β_2 varies as *prec*, *intree (outtree)*, *tree*, *chains* and *sp-graph* depending on different nature of jobs. If precedence relation is not there β_2 is not included in β. Release dates specification for a job is handled through $\beta_3 = r_i$. β_3 is not in β when $r_i = 0$ for all jobs. Processing time or operations behavior is stated as in β_4. Job deadline is specified through β_5. Batching nature is represented by β_6 as the values *p-batch* or *s-batch* if present else β_6 is not included in β.

1.1.3 Machine Environment

Similar to job characteristics, machine environment is addressed through α as α_1 and α_2. The values of α_1 are *o*, *P*, *Q*, *R*, *PMPM*, *QMPM*, *G*, *X*, *O*, *J* and *F* which has meaning as dedicated machines, identical parallel machines, uniform parallel machines, unrelated parallel machines, multi-purpose machines with identical speed, multi-purpose machines with uniform speed, general shop, mixed shop, open shop, job shop and flow shop respectively. α_2 represents the number of machines if specified else machines count is arbitrary.

1.1.4 Optimality Criteria

γ is used as optimality criteria definition. C_i is denoted as finishing time for job J_i and cost is $f_i(C_i)$. *Bottleneck objectives* and *sum objectives* are types of total cost functions denoted respectively below as equations.

$$f_{\max}(C) = max\{f_{\mathrm{i}}(C_{\mathrm{i}})|i = 1, ..., n\}$$

and

$$\sum f_{\mathrm{i}}(C) = \sum_{i=1}^{n} f_{\mathrm{i}}(C_{\mathrm{i}})$$

The schedule which minimizes these total cost functions is the aim of scheduling problem. Optimality Criteria γ is set as $\gamma = f_{\max}$ or $\gamma = \sum f_{\mathrm{i}}$ also there are special functions too. The makespan C_{max}, total flow time $\sum C_{\mathrm{i}}$ and weighted flow time $\sum w_{\mathrm{i}} C_{\mathrm{i}}$ are some common objective functions. In addition, objective functions with due date d_i associated jobs J_i have various definition as:

lateness $L_{\mathrm{i}} = C_{\mathrm{i}} - d_{\mathrm{i}}$

earliness $E_{\mathrm{i}} = max\{0, d_{\mathrm{i}} - C_{\mathrm{i}}\}$

tardiness $T_{\mathrm{i}} = max\{0, C_{\mathrm{i}} - d_{\mathrm{i}}\}$

absolute deviation $D_{\mathrm{i}} = |C_{\mathrm{i}} - d_{\mathrm{i}}|$

squared deviation $S_{\mathrm{i}} = (C_{\mathrm{i}} - d_{\mathrm{i}})^2$

unit penalty $U_{\mathrm{i}} = \begin{cases} 0 & \text{if } C_{\mathrm{i}} \leq d_{\mathrm{i}} \\ 1 & \text{otherwise} \end{cases}$

A regular objective function is nondecreasing with respect to all variables C_i. E_i, D_i and S_i are not regular and others are regular.

1.2 Flow Shop

It has been shown that the three or more machine permutation flow shop problems are NP-complete problems (Gonzalez and Sahni, 1978). Some of other shop scheduling problems are open shop problems, job shop problems, and mixed shop problems. These problems fall in general shop problems category. The general shop problem consists of n number of jobs J_1, J_2, ... , J_n. Each J_i having operations O_1, O_2, ... , O_{n_i} and each operations O_{ij} have processing time p_{ij}. There are m Machines M_1, M_2, ... , M_m. Each operations O_{ij} must be processed on a machine $\mu_{ij} \in \{M_1, M_2, \ldots, M_m\}$. The operations of jobs have a precedence relationship. A machine can process a job at a time and no two machines can process the same job at the same time. Job J_i has some objective function of finishing time C_i and our objective is to find a feasible schedule that minimizes the finishing times for all jobs J_1, J_2, ... , J_n. The assumption is that the objective function is regular.

1.2.1 Open Shop Problem

An open shop problem is defined such that there is no precedence relationship between operations and each job have exactly m operations where operation $\mathbf{O_{ij}}$ has to be processed in machine $\mathbf{M_j}$.

1.2.2 Job Shop Problem

A generalized version of the flow shop problem is job shop problem. There are n jobs and m machines. Operations vary based on the job number yet it preserves the order. Each operation with their processing time is needed to be processed in a particular machine. Considering the

finishing time of the last operation, the solution is to find the best possible schedule which has a minimum value for the objective function.

1.2.3 Mixed Shop Problem

The mixed shop problem comprises the open shop problem and the job shop problem thus has open shop and job shop jobs. There are bound of a job on number count as n_O for open shop and n_J for job shop.

1.2.4 Flow Shop Problem

There are Jobs such as J_1, J_2, ... , J_n and Machines such as M_1, M_2, M_3, ... , M_m. These Jobs has different operations such as O_1, O_2, O_3, ... , O_m to be ran on m different machines each. For particular Job say J_i running particular operation O_j on machine M_j it takes processing time as p_{ij} units.

There is a relationship between the job 's operations and machines. The job J_i's any operations O_j can only begin on machine M_j when operations O_1, O_2, O_3, ... , O_{j-1} for the job J_i are completed from machines M_1, M_2, M_3, ... , M_{j-1}. Alternatively, no two operations for the same jobs are processed at the same time and all preceding operations have to be completed before beginning a new one. A machine can take only a job at one time. The job order may differ considering the operations performed on the particular machine. As being flow shop, the operations order hence machine order are followed by each job as M_1 then M_2 then M_3 till M_m. Overall the aim of this problem is to reduce the makespan such that the completion times for all the jobs on all the machines is minimum. Hence the flow shop problem requires the effective job order for each machine.Computing for n jobs and m machines flow shop problem has $(n!)^m$ different solutions in order to get the optimal one. This is highly unlikely a feasible way to get optimal results. Therefore there are other approaches as permutation flow shop where problem size reduces to n!.

Example of Flow Shop Problem

The example shows three jobs and three machines problem with a solution.

The makespan for the example is found to be 20.

Machines / Jobs	M1	M2	M3
J1	2	1	2
J2	1	3	4
J3	5	1	2

Table 1.1: Flow Shop Problem Example Input

Figure 1.3: Gantt chart for the flow shop example

1.2.5 Permutation Flow Shop

The huge size of the solution of the flow shop problem is somewhat reduced using permutation flow shop. Here instead of considering different job orders for each machine, just only one job order is chosen for all the machines. The problem size hence turned to n!.

Example of Permutation Flow Shop Problem

The example shows three jobs and three machines problem with a solution for job order J_1, J_2 and J_3.

Machines / Jobs	M1	M2	M3
J1	3	1	2
J2	1	3	6
J3	5	3	2

Table 1.2: Permutation Flow Shop Problem Example Input

The makespan for the problem is found as 15.

Figure 1.4: Gantt chart for the permutation flow shop example

1.2.6 Makespan Calculation

The job and machine along with their values are represented as matrix **M**. I have assumed machines as rows and jobs are columns in matrix **M**.The total number of machines are as *totalMachine* and the total number of jobs are as *totalJob*. The order or sequence of a job to be computed through the machines is represented as an array **O**. The makespan calculated value is here represented as *Cmax* which is an integer value.

The *algorithm* 1 below shows steps for calculation of makespan.

Algorithm 1 Cmax Calculation Algorithm

Input: (i) Job Machine Matrix **M**, (ii) Job Order O
Output: Cmax Value

Initialize variables
Allocate space for cumulative job completion calculation matrix **J**
for each machine r in **M** **do**
    Allocate memory for the machine r and assign to matrix **J**
    if r = 0 **then**
        for each job c in **M** **do**
            if $c \neq 0$ **then**
                $\mathbf{J}[r][c] = \mathbf{J}[r][c-1] + \mathbf{M}[r][\mathbf{O}[c]]$
            else
                $\mathbf{J}[r][c] = \mathbf{M}[r][\mathbf{O}[c]]$
    else
        for each job c in **M** **do**
            if $c = 0$ **then**
                $\mathbf{J}[r][c] = \mathbf{J}[r-1][c] + \mathbf{M}[r][\mathbf{O}[c]]$
            else
                $\mathbf{J}[r][c] = (\mathbf{J}[r-1][c] > \mathbf{J}[r][c-1])?\mathbf{J}[r-1][c]\ \mathbf{J}[r][c-1] + \mathbf{M}[r][\mathbf{O}[c]]$
$Cmax \leftarrow \mathbf{J}[totalMachine - 1][totalJob - 1]$
Release Memory from matrix **J**

Chapter 2

Branch and Bound

2.1 History

Branch and Bound (B&B) is by far the most widely used tool for solving large scale NP-hard combinatorial optimization problems. B&B is, however, an algorithm paradigm, which has to be filled out for each specific problem type, and numerous choices for each of the components exist. Even then, principles for the design of efficient B&B algorithms have emerged over the years.

The branch and bound methods in flow shop scheduling have been widely used for finding optimal or near optimal solution methods. Ignall and Schrage (1965) [IS65], Lomnicki (1965) [Lom65], McMahon and Burton (1967) [MB67], Ashour (1970) [Ash70], Gupta (1971) [Gup71], Lageweg et al. (1978) [LLK78], and Bansal (1979) [Ban79] among others have developed different branch and bound methods for various measures of performance like makespan, mean flow time, mean tardiness and maximum tardiness. The difference and the efficiencies of a branch and bound algorithms are in the choice of the lower bound (LB) and elimination rules. The strong bounds and elimination rules eliminate relatively more nodes of the search tree which very often brings in more computation requirements as well. If such needs are excessively large, it may become advantageous to search through larger nodes using a weaker, but fast computable LB. However, the advantages of stronger bounds and elimination rules are more substantial in large scale problems (Baker, 1975) [Bak75]. In 1973 Salvador [Sal73] suggested the permutation flow shop problem's solution through Branch and bound. Kochhar and Morris (1987) [KM87] report work on the development of the heuristics. The heuristics developed try to minimize the effect of setup times and blocking. Further work has been reported by Brah and Hunsucker in the development of mathematical formulation, primarily useful for small size problems [BH91].

2.2 Overview

The concept of the branch and bound provides a strong basis for constructing the algorithms to solve NP-hard discrete optimization problems. Starting with the whole solutions the B&B algorithm searches for the best solution. However, the exponential number of solutions restricts explicit enumeration. In order to make solution space feasible for getting best solution bounds play a vital role. The bounds let search focus on only those space where there is a possibility of having the best solution.

The search begins with a pool containing all the possible solutions which are marked to be unexplored. The unexplored such subset is represented as nodes. For such node, B&B algorithm processes a node at a time. The operations of B&B algorithm consists of selection of a node, calculation of bound and branching. The order of these steps depends on the strategy of choosing the next node for processing. If the bound value of the subproblem is considered for selection as next subproblem, then branching is done after choosing the node. The branching means dividing the current node space into two or more subspaces base on certain criteria. The branching is done till the subspace has a single solution left which is compared to the current best solution and decision is made to keep or discard based on the result of the comparison. Otherwise, the bounding function for the subspace is calculated and compared to the current best solution. If the comparison results show the subspace cannot have an optimal solution it is pruned or discarded else it is kept for further iterations. Since the bounds are calculated first this is called eager strategy. Another way is calculating the bound of the selected node and then branch on the node if necessary. The nodes created are then stored together with the bound of the processed node. This strategy is called lazy strategy. This is good for the depth-first approach in the search tree. Ultimately, the search space is such reduced to have a solution and all live subspaces explored then the current best solution is the required solution.

2.3 Simple B&B algorithm

The simplest B&B algorithm concept as an eager strategy is enumerated below:

1. Consider the whole unexplored solution which is a live node **S** in live.
2. Repeat until there is no node in live
   (a) Get a node **P** from live

(b) Generate branches from **P**

(c) for each branch of **P**

i. Get bound value **X** for each branch of **P**,

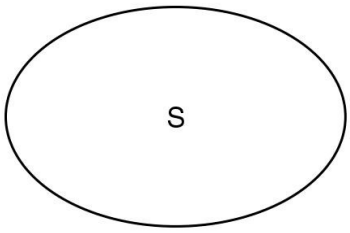
ii. If branch has only one solution then compare the current best value and update if better.

iii. If branch has solutions pool then compare the lower bound X obtained to global if better keep the branch in live else discard.

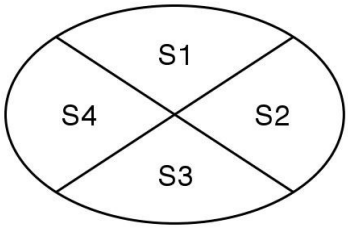
3. After completion the global results are required solution.

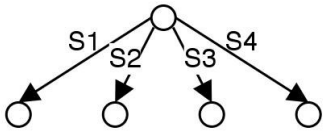
2.4 Figurative illustration of branch and bound

Let us understand the B&B concept through the figure. Figure 2.1 has images shown having problem representation in form of eclipse area and tree. The figure 2.1a shows the initial state of any problem, on the left as eclipse whereas to the right is the root node. Based on certain properties the area is divided into smaller regions which also can be illustrated as in the form of a tree as shown in figure 2.1b. Meanwhile, there occurs bounding or pruning or fathoming which restricts the need to explore a certain portion of the solutions. Thus all the solutions falling to the region are not needed to be computed. In the figure 2.1c below, it is assumed S1 and S3 are pruned as their LB is no better than the current best solution where S2 and S4 are further iterated to get more deeper into the tree and hence towards the optimal. Let us assume, that S21, S22, S41 and S42 each have a solution only. Each solution is compared to the global value. If the LB of of the solution is better then new global value are set else it is discarded. Finally, the global value is the required optimal for the problem.

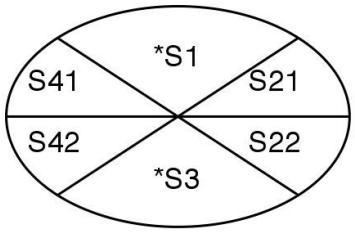


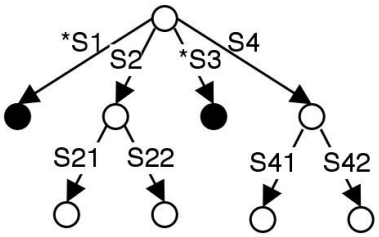
(a)





(b)





Assumed here S1 and S3 will yield no better result

(c)

Figure 2.1: Illustrative of Branch and Bound

2.5 B&B algorithm in Permutation Flow Shop Problem

The B&B requires the solutions to be searched for arranged in the form of a tree. The tree consists of nodes. In my case the nodes consist of node level, lower bound for the node, partial job order till that node, remaining job order, children count, children pointer, parent pointer, is fathomed and other navigation necessary properties. The strategy followed in my case is eager. The initial upper bound (UB) is obtained by iterating the random job order and computing makespan for the job order to a fixed number of times with a minimum makespan preserved from the iteration. We prune the node during traversal earlier by comparing UB to LB. This helps in the reduction of many branches and hence nodes search. The traversal in a node is the hybrid best first search (BeFS). The breadth-first search (BFS) is not suitable as it requires huge memory also depth-first search (DFS) may take a longer time to get an optimal solution.

2.5.1 Lower bound for root

Initially, the LB computation based on each machine ($\mathrm{LB_m}$) is achieved by computing the complete job processing on the machine. Here, for the base machine, it is assumed all the jobs are aggregated and in addition the minimum of a job completion on all machines before the base machine is done and similarly the minimum of a job completion on all machine after the base machine is added. The lower bound of the root ($\mathrm{LB_{root}}$) is assigned by selecting the maximum value obtained from the lower bound computed on each machine base. The maximum value is selected because it is only the possible value. The mathematical representation is as:

$$LB_{\mathrm{m}} = \min_{i(0\ldots N-1)} \left(\sum_{j=0}^{j=m-1} p_{\mathrm{ij}} \right) + \sum_{i=0}^{i=N-1} p_{\mathrm{im}} + \min_{i(0\ldots N-1)} \left(\sum_{j=m+1}^{j=M-1} p_{\mathrm{ij}} \right) \tag{2.1}$$

$$LB_{\mathrm{root}} = \max_{m(0\ldots M-1)} \left(LB_{\mathrm{m}} \right) \tag{2.2}$$

In equation 2.1, m is the machine for which LB is to be calculated, i is any job, j is any machine, N is total jobs, M is total machines and p_{ij} is processing time for any job i for any machine j. It can be illustrated in the figure as below:

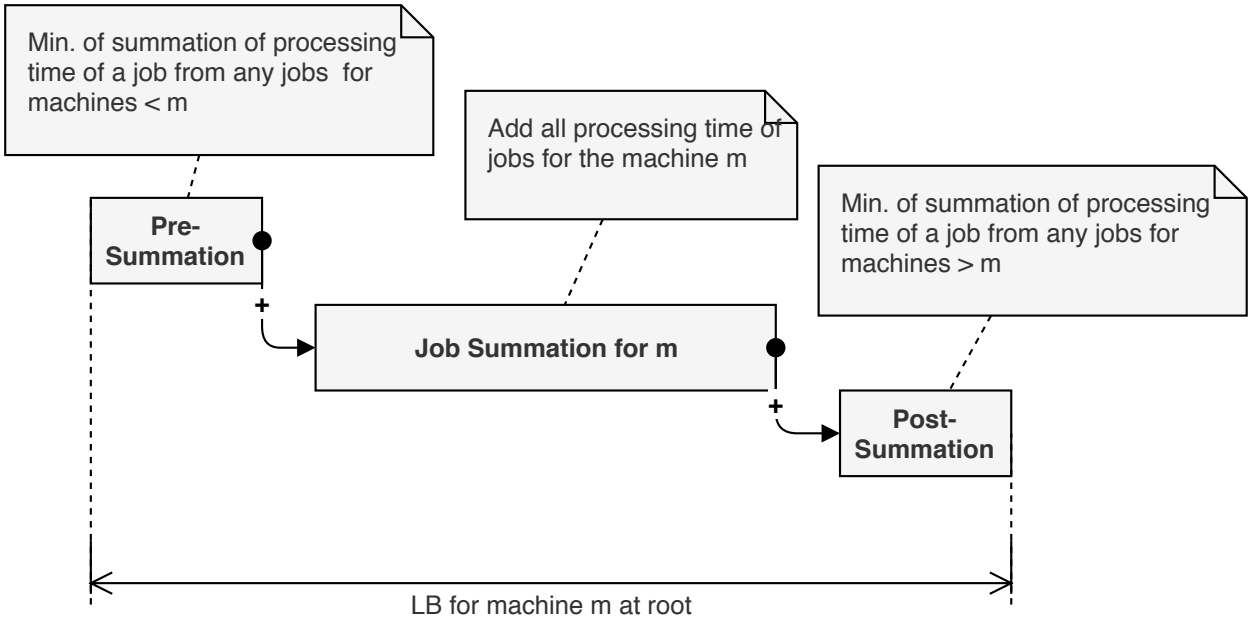


Figure 2.2: Lower bound for root calculation illustration figurative

2.5.2 Lower bound for node

Similar to LB for the root, lower bound for the node ($\mathrm{LB_m}$) also has three parts. The calculation is similar based on the machine. First part is partial Cmax based on partial order by considering only the machines up-to machine m. The second part is the summation of the processing time of jobs which are not in the partial order. The third part is the same as in case of LB for root. Taking only jobs that are not in partial order and machines that are after machine m in machine order, the minimum sum of any single job is the third portion. Lower bound for the node ($\mathrm{LB_{node}}$) is chosen from $\mathrm{LB_m}$ whichever is largest. The mathematical representation is:

$$LB_{\mathrm{m}} = Cmax_{\mathrm{rm}} + \sum_{i \notin R} p_{\mathrm{im}} + \min_{i \notin R} \left(\sum_{j=m+1}^{j=M-1} p_{\mathrm{ij}} \right) \tag{2.3}$$

$$LB_{\mathrm{node}} = \max_{m(0 \ldots M-1)} \left(LB_{\mathrm{m}} \right) \tag{2.4}$$

In equation 2.3, R is partial job order, r is last job of partial job order R and $\mathrm{Cmax_{rm}}$ is makespan value for order R and machines (0,...,m). Figurative explanation is shown below.

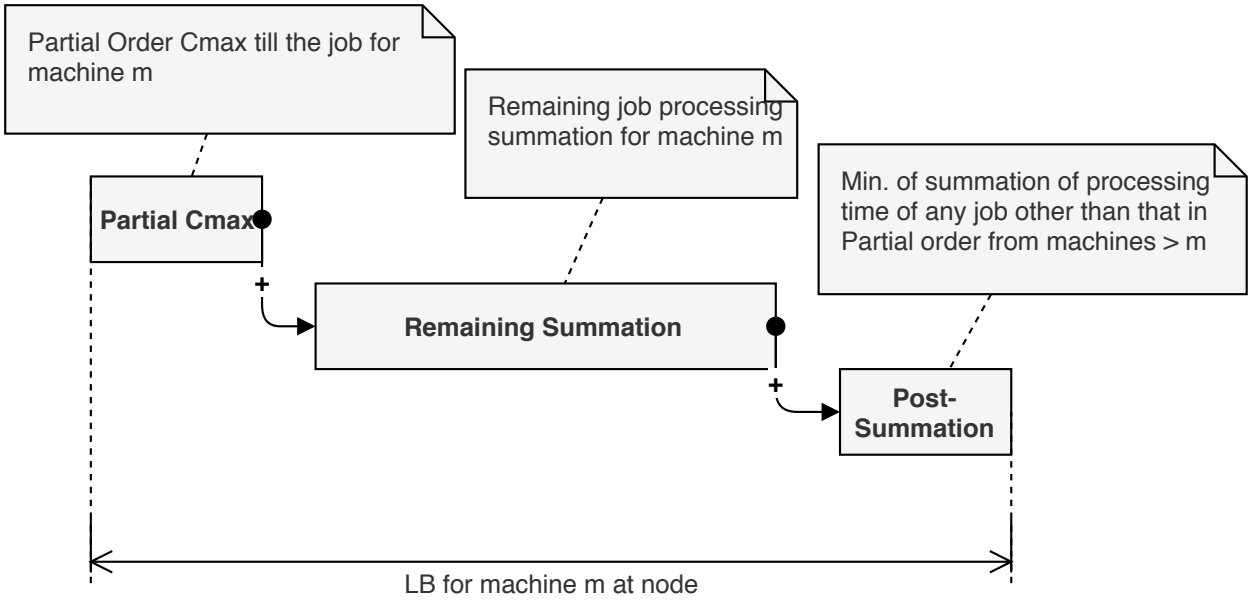


Figure 2.3: Lower bound for node calculation illustration figurative

2.5.3 Node selection and branching

As stated earlier, the selection is the hybrid BeFS. BeFS is beneficial as it consumes less memory than BFS and also reaches optimal faster than DFS. We have fused BeFS with DFS. The fusion with DFS enabled in less consumption of memory. The node with LB is selected and branched as the eager strategy. On each iteration, the search goes deeper into the tree until it reaches the leaf. Then, it tries to complete search on all sister nodes before searching back on the parent level. As search return back to parent it then release the memory occupied by the children. The branching is always done to nodes which have better values than the current UB. The branching is not done further than N-1 level because the child after the node is just one. Instead, the LB of its child is calculated and placed earlier. The selection algorithm is as:

In *algorithm* 2, the first while loop makes sure there are children node to be selected. On *line* 13, the child fathomed helps in reduction of the tree earlier and adds in finding optimal earlier. Similarly, memory release on line 16 adds more efficiency of the algorithm.

2.5.4 The B&B Algorithm

The B&B Algorithm is composed using *equation* 2.2, *equation* 2.4 and *algorithm* 2.

The root and the first children are created separately then *algorithm* 3 further is used. The method *GetPartialJoborders* (A) generates the partial job order and remaining job order for the

Algorithm 2 Hybrid BeFS algorithm for node selection

Input: Current Node C
Output: Next Node C'

```
1: Declare variable tempNext C'
2: if C is leaf node then
3:     C.isfathomed = true
4:     C ← C.parent
5: while C.allchildfathomed = true do
6:     C ← C.parent till root.
7: while true do
8:     for child c in C.children do
9:         if c.isfathomed = false then
10:            if c.lowerbound < GLOBAL.upperbound then
11:                C' ← c and break for loop
12:            else
13:                c.isfathomed = true
14:    if C.allchildfathomed = true then
15:        C.isfathomed = true
16:        Release memory occupied by all the children
17:        if C.parent = NULL then                    ▷ all nodes traversal completed.
18:            Return NULL
19:        else
20:            C ← C.parent
21: for child c in C'.parent.children do
22:    if c.lowerbound < C'.lowerbound and c.isfathomed = false then
23:        C' ← c
24: Return C'
```

child using the parent node. Based on the *equation* 2.4, *GetLBforTheNode* (A) method provide the LB for intermediate nodes. As stated earlier, on *line* 17 we have calculated the LB earlier in the parent node and reduced the further iteration shortening the tree. Also, the child is fathomed based on the calculated LB as they produce no better result further through the children. Eventually, reaching the last node of the tree, the result is saved if better else fathomed.

2.5.5 The B&B algorithm multi-threading

The single threaded approach takes more time. In order to improve the performance and reach the optimal solutions fast, we have implemented simple threading. The root's children are enqueued to a queue from where each thread picks a child and applies *algorithm* 3. The threading approach is shown below in the flow chart:

Algorithm 3 Branch and Bound algorithm

```
1: Declare variables myNode, child, children and childcount
2: while !nodeSearchComplete do
3:     myNode ← GetMimimumChildNode()
4:     if myNode = NULL then
5:         nodeSearchComplete ← true
6:         Break while loop
7:     childcount ← totalJobs − myNode.level
8:     if childcount > 1 then                                  ▷ Create children for myNode
9:         Allocate memory for myNode.children
10:        for i ← (0,...,childcount-1) do
11:            child.level ← myNode.level+1
12:            child.parent ← myNode
13:            child.fathomed ← false
14:            child.childAllFathomed ← false
15:            child.childCount ← 0
16:            GetPartialJoborders(child.partialOrder,child.remainingOrder,myNode,i)
17:            if childcount = 2 then
18:                child.partialOrder[totalJobs-1] ← child.remaingingOrder[0]
19:                child.lowerbound ← getCmax(child.partialOrder)          ▷ refer algorithm 1
20:            else                                                        ▷ refer equation 2.4
21:                child.lowerbound ← GetLBforTheNode(child.partialOrder,child.remainingOrder,child.level)
22:            if child.lowerbound > GLOBAL.upperbound then
23:                child.fathomed ← true
24:                child.childAllFathomed ← true
25:            myNode.children[i] ← child
26:    if childcount = 1 then
27:        if child.lowerbound <= GLOBAL.upperbound then
28:            saveResult(myNode.partialOrder,myNode.lowerbound)
29:        child.fathomed ← true
30:        child.childAllFathomed ← true
```

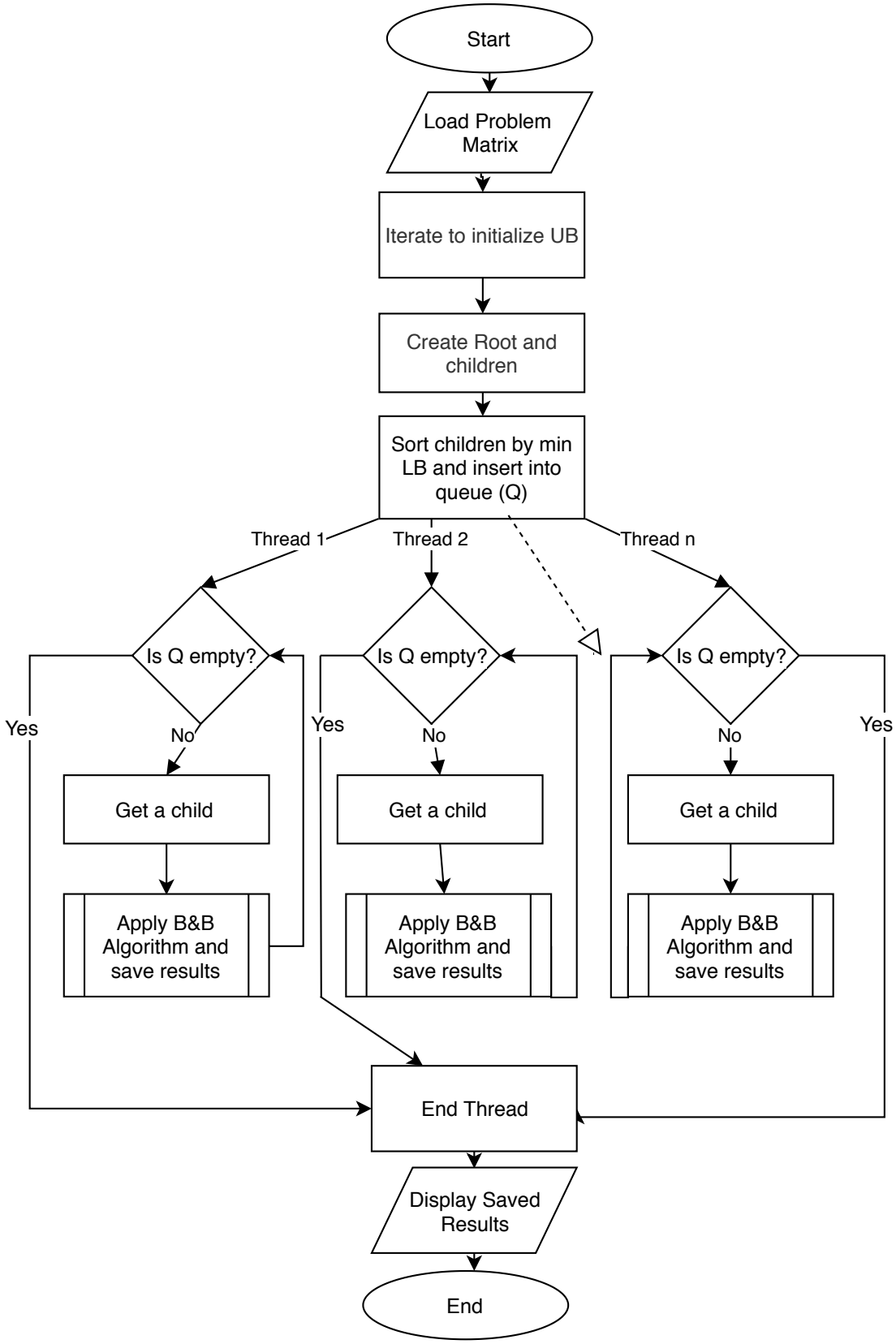


Figure 2.4: Multi-Thread B&B Approach illustration

Chapter 3

Genetic Algorithm

3.1 History

Earlier in the 1950s and the 1960s, independently several computer scientists studied evolutionary systems. They intend to use this idea for solving engineering problems. This started with the idea to evolve a population of candidate solutions to a given problem with the aid of operators mimicking natural selection and genetic variation. On that run, genetic algorithms (GA) were invented by John Holland and later Holland and his students and colleagues at the University of Michigan developed it. Holland, inspired by Darwinian theory, thought to implement similar in computer systems to design algorithms which follows adaptation phenomenon as in nature. A theoretical framework for adaptation under the GA emerge as Holland's book *Adaptation in Natural and Artificial Systems* was published in 1975. [Mit98]

3.2 Basic Concept

A new population is created going through a method described by GA. The method follows natural selection in creating the new population as well the essence of genetics are imprinted through operators crossover, mutation and inversion. The population consists of chromosomes basically strings of 0's and 1's. Each gene is composed of an allele (0 or 1) and such genes join together to form a chromosome. The new population is created as the **selection** operator chooses the chromosomes that better fits. Sub-parts of two chromosomes are exchanged forming a **crossover**,**mutation** occurs randomly altering the allele values of some genes in the chromosome and a contiguous section's order is reversed through **inversion** all these operations makes a new array of genes in the chromosome. These operators idea is that the newly formed chromosome may provide a better

solution to the problem.

Simple GA Algorithm

Now, let us visualize a simple genetic algorithm concept in the digital world. It is explained as:

1. Initially create a population of size n having chromosomes of length l-bits randomly.
2. Calculate the fitness function *f(x)* over each chromosome x in the population.
3. Repeat the steps until n offspring have been created
   (a) From the current population, select a pair of parent chromosomes. The chromosomes are selected based on the fitness function's value higher or lower as preferred for type of problem. The selection is with replacement meaning same parent can be selected more than once.
   (b) Now do crossover based on crossover probability p_c. If no crossover occurred then offspring are parent themselves. The point for crossover in chromosome is determined randomly. The crossover can be single point or multi-point. Ultimately, 2 offspring are formed.
   (c) With a mutation probability p_m, mutate the offspring and put in new population.
4. Make the newly formed population as current population.
5. Based on required generations, go to step 2 and repeat.

Likewise, the biological generation, here a **generation** is marked as those process of an iteration. Depending upon the need number of generations varies. The entire set of generations forms a **run**. In overall operation, probabilities and randomness play a vital role hence at the end of the run we get different results.

3.2.1 Simple GA Example

Let's take an illustrative example. Assume chromosome has a string of length l 7 bit, fitness function *f(x)* be the count of 1's in the string, population size n be 4, crossover probability p_c be 0.8 and mutation probability p_m be 0.005.

Consider randomly generated initial population as

Chromosome Name	Chromosome String	Fitness Value
P	0011110	4
Q	1101101	5
R	1010101	4
S	0001000	1

Table 3.1: A Simple GA Example - Initial Data

Simply using "roulette-wheel sampling" as an implementation of fitness-proportionate selection in selection method we get parents. A circular roulette wheel is divided into arc areas based on individual fitness from the current population and each chromosome are assigned accordingly. The wheel is spun as to population size times to get that many parents here 4 times. Let from first 2 spins we get P and Q then on later we get Q and R. With $p_c = 0.8$, the parents P and Q crossover at the second bit to form T = 0001101 and U = 1111110. While Q and R didn't crossover so they became offspring. Next step, the mutation occurred on offspring T on the second locus to form T'= 0101101 and R on the fourth locus to form R'= 1011101. The final result then is:

Chromosome Name	Chromosome String	Fitness Value
T'	0101101	4
U	1111110	6
Q	1101101	5
R'	1011101	5

Table 3.2: A Simple GA Example - Result Data

Here we got a new better fit U (6), the average fitness also rose from 14/4 to 20/4. Eventually, with further iterations we will get a string with all ones.

3.3 GA in Permutation Flow Shop Problem

There are few adjustments in simple GA are made in order to advocate the issue of permutation flow shop problem. The chromosomes strings are job numbers exactly one time instead of 0's and 1's. We here represent chromosome string as job order. The crossover is two point and mutation swaps any two jobs in the job order. The crossover probability p_c is assumed to be 1. Based on the fitness function or objective function value from the current population, top certain (say $\delta\%$) job orders are taken directly to the new population. The remaining needed population are obtained through the same steps as in simple GA algorithm assuming $p_c = 1$.

3.3.1 Initial Population

As the first step, the initial population is generated. A initial job order is set say J_1, J_2, ... , J_n and with an advanced random number generator, we get any two position values in the job order to swap and create new job order. This newly created job order is used as the base job order to swap and generate another new job order. This process is repeated until the population size is reached.

Below are related algorithms : The next job order generator plays a vital role in providing randomly distributed candidate solutions.

Algorithm 4 GA Next Job Order Generator Algorithm

Input: Job Order **O**
Output: Job Order **O**

Initialize the required parameters a, b
flag $\leftarrow true$
while flag **do**
    $a = mynrand(0, totalJobs - 1)$ ▷ refer appendix A
    $b = mynrand(0, totalJobs - 1)$
    if $a \neq b$ **then** $flag \leftarrow false$
$swap(\mathbf{O}, a, b)$ ▷ refer appendix A
return **O**

In the algorithm 4, I have insured the random numbers generated are in the range and are unique. This makes sure that resultant job order **O** after swapping is different from that of input.

The random population generation algorithm is as follows:

Algorithm 5 Initial GA Population Generation

Input: (i) Job Machine Matrix **M**, (ii) Job Order **O**
Output: (i) Population **P** (ii) Top δ% job orders **T**

Initialize the required parameters, i = 0, Cmax, **P**, **T**
while i < population size **do**
    O $\leftarrow$ getNextJobOrder(**O**) ▷ refer algorithm 4
    $Cmax \leftarrow jobSumulate(\mathbf{M}, \mathbf{O})$ ▷ refer algorithm 1
    Bind $Cmax$ with **O**
    Insert **O** into $\mathbf{P_i}$
    for each element t in **T** **do** ▷ To get ranked top δ% job orders.
        if $t.Cmax \leq Cmax$ **then**
            Replace in **T** then terminate loop.
Return **P**, **T**

The *algorithm* 5 uses *algorithm* 1 and *algorithm* 4 to yield the initial random population **P**.The process of ranking the whole population is expensive in terms of performance to get the top δ% job orders. We here avoid a load of ranking by comparing and inserting the better job orders in

a fixed sized array **T**. This array **T** is transferred to new population directly. The remaining job orders for the new population are obtained through selection, crossover, and mutation.

3.3.2 Selection

The selection is done from the current population **P** with replacement. The job orders along with their $Cmax$ values are used to create a range in which higher $Cmax$ occupies more range. The range begins from 0 to sum of $Cmax$ from the current population **P**. I have designed an algorithm which helps in getting a better job order out of the current population for crossover. The algorithm is as:

Algorithm 6 Higher Probable Job Order Index Finder Algorithm

Input: (i) Cumulative Cmax Array $\mathbf{P^c}$ (ii) Population Size **S**
Output: Higher Probable Job Order Index for Current Population Array **P**

1: Initialize the required parameters, Cumulative Cmax $\mathbf{C^T} \leftarrow \mathbf{P^c}$[S]
2: $index \leftarrow \mathbf{S}$
3: $random \leftarrow mynrand(1, \mathbf{C^T})$ ▷ refer appendix A
4: **for** $i = \mathbf{S} - 1$ till $i = 0$ **do**
5: **if** $random \leq \mathbf{P^c}[i]$ **then**
6: $index \leftarrow index - 1$
7: Return $index$

The Cumulative Cmax Array $\mathbf{P^c}$ is created at the time of population generation so as to accommodate the selection process later. The index obtained from this *algorithm* 6 is then used as index in the current population **P** to get a parent for crossover. In addition, to make sure the two job order obtained are different, we have added following snippet before each index is fed for crossover.

```
...
a = higherProbJob(P^c,S)                ▷ refer algorithm 6
b = higherProbJob(P^c,S)
while a = b do
    b = higherProbJob(P^c,S)
...
```

3.3.3 Crossover

The two unique job orders ($\mathbf{O_1}$ and $\mathbf{O_2}$) are obtained through selection operation. The crossover we are implementing here is two point. Randomly the points are obtained such that first point

is always less than half of the job order and second is either equal to half or greater. This point selection can be of other ways too. The offspring generation has mainly three steps, copying the job order from the beginning until the start point from a parent (say $\mathbf{O_1}$), copying the job order from the start point to the end point inclusive from another parent (say $\mathbf{O_2}$) and finally copying the remaining job order from parent again(say $\mathbf{O_1}$). While copying from the start point onward it is made sure that the job number has not occurred previously in the sequence. If, for a position in the new job order, the job number matches then the next job number is taken and if it reaches the end of the job order in parent then it again scans from the beginning to find unmatched job number and then copy it to the new job order. This occurs twice to create two offspring. Below is the algorithm illustrating it.

Algorithm 7 Crossover Algorithm

Input: (i) Job Order $\mathbf{O_1}$ (ii) Job Order $\mathbf{O_2}$
Output: (i) New Job Order $\mathbf{O_1'}$ (ii) New Job Order $\mathbf{O_2'}$

1: Initialize the required parameters, start point *sp*, end point *ep*
2: $sp \leftarrow mynrand(1, (totalJob/2) - 1)$ ▷ refer appendix A
3: $ep \leftarrow mynrand(totalJob/2, totalJob - 1)$
4: **while** $i < sp$ **do**
5: $\quad \mathbf{O_1'}[i] \leftarrow \mathbf{O_1}[i]$
6: $\quad \mathbf{O_2'}[i] \leftarrow \mathbf{O_2}[i]$
7: **for** $i = sp$ till $i = ep$ **do** ▷ for $\mathbf{O_1'}$
8: $\quad$ flag $\leftarrow true$
9: $\quad$ **while** flag **do**
10: $\quad\quad$ **if** ($\mathbf{O_1}[i] \neq \mathbf{O_2}[i]$) and ($\mathbf{O_2}[i]$ does not exists in any $\mathbf{O_1'}[i]$) **then**
11: $\quad\quad\quad \mathbf{O_1'}[i] \leftarrow \mathbf{O_2}[i]$
12: $\quad\quad\quad$ flag $\leftarrow false$
13: $\quad\quad$ **else**
14: $\quad\quad\quad$ Select next job order from $\mathbf{O_2}$
15: Repeat similarly from step 7 to 14 for $\mathbf{O_2'}$
16: Return $\mathbf{O_1'}$,$\mathbf{O_2'}$

3.3.4 Mutation

The obtained new job orders $\mathbf{O'}$ after crossover are subjected to mutation. The mutation occurs based on the mutation probability p_m. Usually, p_m is very low.

Algorithm 8 Mutation Algorithm

Input: Job Order **O'**
Output: Mutated Job Order **O'**

Initialize the required parameters a, b
flag $\leftarrow true$
while flag **do**
    $a = mynrand(0, totalJobs - 1)$ ▷ refer appendix A
    $b = mynrand(0, totalJobs - 1)$
    if $a \neq b$ **then** $flag \leftarrow false$
$swap(\mathbf{O'}, a, b)$ ▷ refer appendix A
return **O'**

3.4 Population Generation

The new population for each generation are generated using the operators selection, crossover and mutation. The steps for new population generation uses *algorithm* 6, *algorithm* 7 and *algorithm* 8. There is chances of memory issues so proper cleanup are required after each generation. The algorithm for new population generation is illustrated as in *algorithm* 9 below:

3.5 GA multi-threading

Multi-threads are used in mainly two places in order to gain in performance. The initial population generation is done by multiple threads. Later the new population pool generation is also multi-threaded. The threading approach is illustrated in the diagram below:

Algorithm 9 New GA Population Generation

Input: (i) Job Machine Matrix **M** (ii) Population **P** (iii) Cumulative Cmax Array $\mathbf{P^c}$ (iv) Population Size **S** (v) Top δ% job orders **T**

Output: (i) New Population $\mathbf{P'}$ (ii) Top δ% job orders **T** (iii) New Cumulative Cmax Array $\mathbf{P'^c}$

Initialize the required parameters, i = 0, $\mathbf{P'}$
i $\leftarrow$ ($\mathbf{S} * \delta\%$) $-$ 1
$\mathbf{P'} \xleftarrow{insertall} \mathbf{T}$ ▷ insert top job orders from current population to new.
Update $\mathbf{P'^c}$ using **T**
while i $<$ **S do**
    a = higherProbJob($\mathbf{P^c}$,**S**) ▷ refer algorithm 6
    b = higherProbJob($\mathbf{P^c}$,**S**)
    while $a = b$ **do**
        b = higherProbJob($\mathbf{P^c}$,**S**)
    $\mathbf{O_1} \leftarrow$ **P[a]**
    $\mathbf{O_2} \leftarrow$ **P[b]**
    $\mathbf{O'_1}$,$\mathbf{O'_2} \leftarrow$ crossover($\mathbf{O_1}$,$\mathbf{O_2}$) ▷ refer algorithm 7
    $\mathbf{O^m_1} \leftarrow$ mutate($\mathbf{O'_1}$) ▷ refer algorithm 8
    $\mathbf{O^m_2} \leftarrow$ mutate($\mathbf{O'_2}$)
    Insert $\mathbf{O^m_1}$ and $\mathbf{O^m_2}$ into $\mathbf{P'}$ and update $\mathbf{P'^c}$
    for each element t in **T do** ▷ To get ranked top δ% job orders.
        if $t.Cmax \leq \mathbf{O^m_1}.Cmax$ **then**
            Replace in **T** with $\mathbf{O^m_1}$ then terminate loop.
    for each element t in **T do**
        if $t.Cmax \leq \mathbf{O^m_2}.Cmax$ **then**
            Replace in **T** with $\mathbf{O^m_2}$ then terminate loop.
Release memory of **P**,$\mathbf{P'^c}$
Return $\mathbf{P'}$, $\mathbf{P'^c}$,**T**

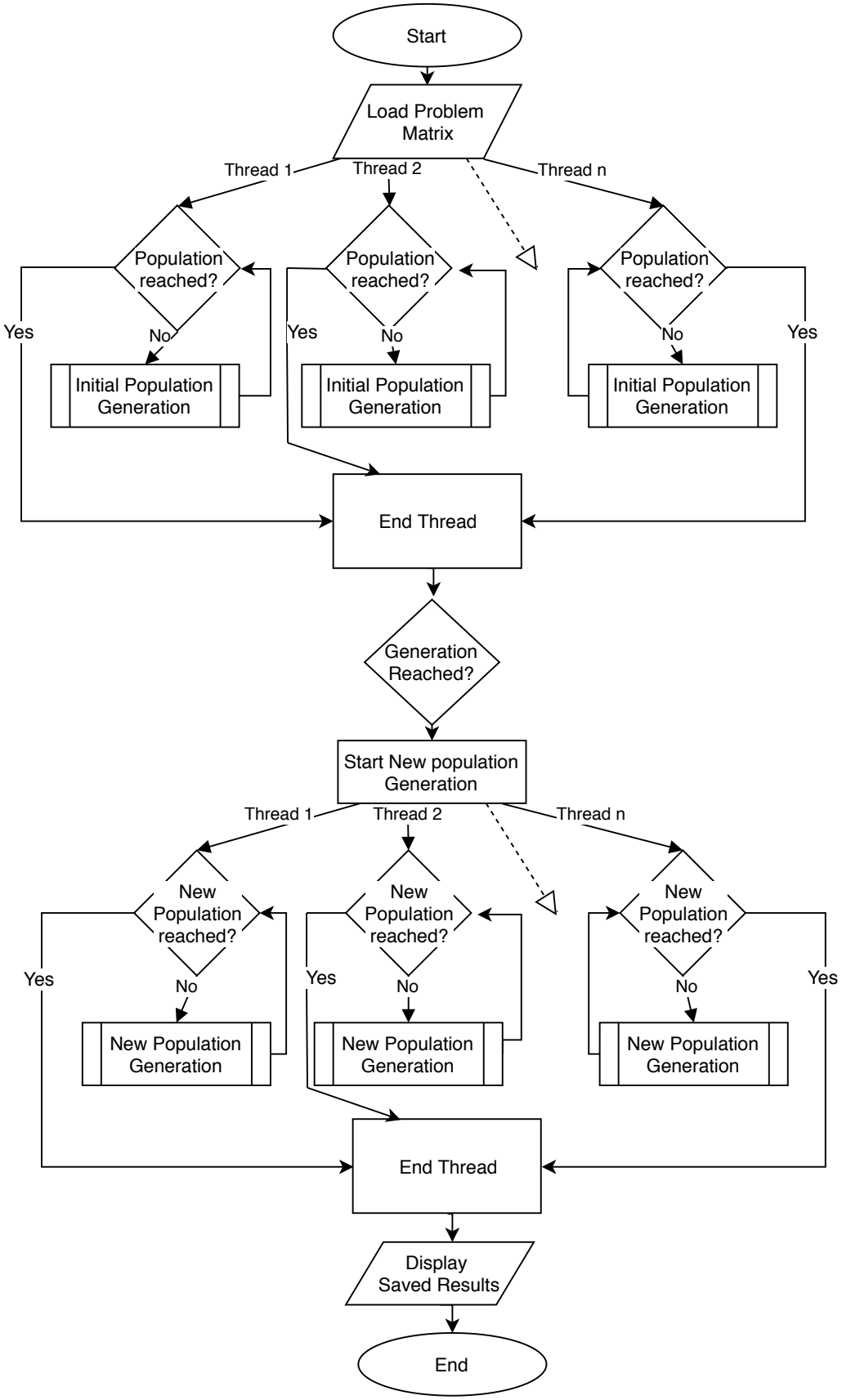


Figure 3.1: Multi-Thread GA Approach illustration

Chapter 4

Results

The algorithms were converted to program and results were obtained. The results were computed as a single thread as well as multi-thread. For the case of multiple threads, we have used eight threads. We have used C programming of version C99. The machine we have used is *java.cs.unlv.edu* and visual studio 2017 IDE is used. The problem files were obtained from *New hard benchmark for flow shop scheduling problems minimising makespan* [Eva15]. The machine configuration we have used is:

```
OS
GNU/Linux
Kernel Version
#1 SMP Fri Feb 1 14:54:57 UTC 2019
Kernel Release
3.10.0-957.5.1.el7.x86_64

CPU
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
CPU(s):                8
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 58
Model name:            Intel(R) Xeon(R) CPU E3-1240 V2 @ 3.40GHz
CPU MHz:               1643.969
CPU max MHz:           3800.0000
CPU min MHz:           1600.0000
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              8192K

MEMORY
Memory block size:       128M
Total online memory:      32G
Total offline memory:      0B
```

Figure 4.1: Machine Configuration

4.1 Exhaustive Search Results

The results were generated only from problems of jobs 10 and machines 5, 10, 15 and 20 which are in small set problems from VFR benchmark problems. Each problem has 10 different sets. The number of solutions to calculate in order to get all the optimal is given by the factorial of the number of jobs (for 10 jobs, 10! = 3628800 *solutions*). In addition, we have done the calculation for all the optimal for the problem. The speedup is calculated by dividing the execution time of a single thread by the execution time of multi-thread. The comparative results for single thread and multi-thread along with speedup is shown below:

S.N.	**Problem**	**Count**	**Time (Th 1)**	**Time (Th 8)**	**Speedup**
1	10_5_1	2228	1341.94202	932.01001	1.43984
2	10_5_2	30	1310.16406	922.09302	1.42086
3	10_5_3	36	1333.31397	930.43103	1.43301
4	10_5_4	26	1327.45300	921.53998	1.44047
5	10_5_5	12	1333.36597	939.31598	1.41951
6	10_5_6	323	1312.76599	923.65302	1.42128
7	10_5_7	66	1328.79004	930.16998	1.42855
8	10_5_8	12	1329.18799	922.89801	1.44023
9	10_5_9	18	1331.31006	923.76599	1.44118
10	10_5_10	48	1325.68201	929.86597	1.42567
11	10_10_1	2	2445.39307	1576.40100	1.55125
12	10_10_2	548	2458.28809	1558.51807	1.57732
13	10_10_3	24	2427.65991	1564.85107	1.55137
14	10_10_4	4	2455.43604	1560.08594	1.57391
15	10_10_5	5	2421.36206	1575.70496	1.53668
16	10_10_6	317	2426.29590	1582.04297	1.53365
17	10_10_7	1	2432.15088	1566.06494	1.55303
18	10_10_8	48	2437.12695	1556.12598	1.56615
19	10_10_9	6	2451.75708	1572.51502	1.55913
20	10_10_10	15	2447.73193	1574.06604	1.55504

Table 4.1 continued from previous page

S.N.	Problem	Count	Time (Th 1)	Time (Th 8)	Speedup
21	10_15_1	1	3527.47998	2177.64111	1.61986
22	10_15_2	1	3527.82007	2190.71191	1.61035
23	10_15_3	1	3509.75708	2196.44092	1.59793
24	10_15_4	5	3520.12891	2171.95313	1.62072
25	10_15_5	2	3503.35400	2196.25610	1.59515
26	10_15_6	1	3498.88794	2195.74512	1.59349
27	10_15_7	56	3539.15405	2186.43799	1.61868
28	10_15_8	9	3514.88208	2176.32690	1.61505
29	10_15_9	15	3531.58789	2202.57910	1.60339
30	10_15_10	1	3529.57495	2199.30005	1.60486
31	10_20_1	2	4639.98682	2852.52002	1.62663
32	10_20_2	1	4600.63721	2872.04102	1.60187
33	10_20_3	1	4594.14600	2889.40210	1.59000
34	10_20_4	4	4628.41992	2859.86401	1.61841
35	10_20_5	2	4594.84180	2841.83594	1.61686
36	10_20_6	218	4621.86523	2874.50195	1.60788
37	10_20_7	1	4625.17676	2847.19702	1.62447
38	10_20_8	2	4645.15820	2872.38989	1.61718
39	10_20_9	3	4601.27197	2869.10010	1.60373
40	10_20_10	2	4621.06104	2865.68604	1.61255

Table 4.1: Exhaustive search result single and multi threads with optimal results count.

The *table* 4.1 is summarized in terms of the number of machines and we get the following result.

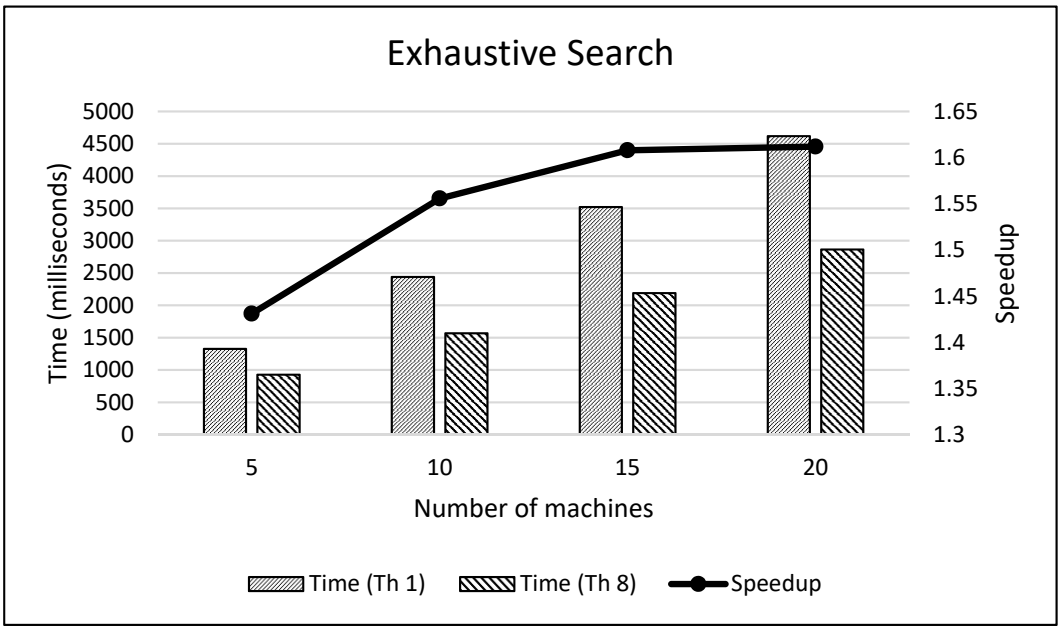


Figure 4.2: Machine based Threaded and Non-Threaded comparison with speedup

The threading seems to give an average speedup of 1.5. It is expected to increase the speed of execution by 8 times as we use eight threads but only 1.5 is obtained. This happened because of Amdahl's law [Gen67], which states, there is only a certain percentage (i.e. less than 100%) of execution time which can be subjected to the speedup. But as we go on increasing more jobs this approach wouldn't be enough to calculate optimal as the number of solution size increases exponentially.

4.2 Branch and Bound Algorithm Results

The next implementation Branch and Bound algorithm provided better results. The results for the same problem as of exhaustive search was taken. The comparative analysis of the algorithm based on execution time and nodes visited with and without threading along with speedup is tabulated below:

S.N.	Problem Size	BB		BB Threaded		BB /BB Th	Ex Th /BB Th
		Time	Node Visited	Time	Node Visited		
1	10_5_1	385.05801	1289946	250.62300	1124397	1.53640	3.71877
2	10_5_2	44.01800	172967	25.69200	144731	1.71330	35.89028
3	10_5_3	62.76900	250422	37.32700	217280	1.68160	24.92649
4	10_5_4	58.16800	234954	40.32100	214046	1.44262	22.85509
5	10_5_5	4.04000	12520	3.64300	16025	1.10898	257.84133
6	10_5_6	12.77600	47784	13.14300	51025	0.97208	70.27718
7	10_5_7	35.44700	140861	24.56200	132755	1.44316	37.87029
8	10_5_8	60.72302	244797	32.65100	174194	1.85976	28.26553
9	10_5_9	1.89900	4279	1.57200	4507	1.20802	587.63740
10	10_5_10	89.51600	361014	46.94700	280575	1.90675	19.80672
11	10_10_1	57.54300	151556	29.01200	143617	1.98342	54.33617
12	10_10_2	271.76502	790482	142.21503	731582	1.91094	10.95888
13	10_10_3	151.82001	245277	43.97800	229746	3.45218	35.58259
14	10_10_4	95.69500	224473	40.99800	200530	2.33414	38.05273
15	10_10_5	24.67600	57958	11.19000	55244	2.20518	140.81367
16	10_10_6	158.48300	453318	80.11900	422942	1.97810	19.74616
17	10_10_7	276.18103	803231	118.64200	632401	2.32785	13.19992
18	10_10_8	95.78300	130833	27.13900	124448	3.52935	57.33911
19	10_10_9	25.94100	41289	9.45500	38058	2.74363	166.31571
20	10_10_10	157.42000	384958	80.25500	407163	1.96150	19.61331
21	10_15_1	81.05500	133817	35.69400	130673	2.27083	61.00860
22	10_15_2	104.81200	189062	40.79900	159475	2.56898	53.69524
23	10_15_3	234.21800	435419	65.86400	285716	3.55609	33.34813
24	10_15_4	364.97803	725653	158.66400	671070	2.30032	13.68901
25	10_15_5	52.03400	70898	22.34500	78818	2.32866	98.28849
26	10_15_6	60.20400	97532	23.55500	89527	2.55589	93.21779
27	10_15_7	242.08200	475161	122.84802	458486	1.97058	17.79791

Table 4.2 continued from previous page

S.N.	Problem Size	BB		BB Threaded		BB /BB Th	Ex Th /BB Th
		Time	Node Visited	Time	Node Visited		
28	10_15_8	164.58000	310466	57.57000	230850	2.85878	37.80314
29	10_15_9	162.23900	293375	63.21900	276696	2.56630	34.84046
30	10_15_10	71.40997	112719	31.05200	113137	2.29969	70.82636
31	10_20_1	249.46500	295685	80.60500	267074	3.09491	35.38887
32	10_20_2	166.57800	197686	61.16300	181222	2.72351	46.95717
33	10_20_3	418.05899	594717	132.83200	475108	3.14728	21.75230
34	10_20_4	79.84800	82861	35.50800	92656	2.24873	80.54140
35	10_20_5	241.64301	299718	97.46400	271912	2.47931	29.15780
36	10_20_6	144.31500	171803	58.46000	173378	2.46861	49.17041
37	10_20_7	100.94400	109576	34.58300	93880	2.91889	82.32938
38	10_20_8	82.06900	84133	35.27500	84345	2.32655	81.42848
39	10_20_9	98.76000	109283	49.70100	144366	1.98708	57.72721
40	10_20_10	161.92900	186145	76.32200	219059	2.12166	37.54731

Table 4.2: Execution time and node visited results for single and multi-threaded comparison with speedup

In *table* 4.2, it is clearly observed from the results that we gain performance when multi-threading is used. The execution time of single and multi-threading are summarized in terms of average time, maximum time, minimum time and median time in order to understand the algorithm's performance from different prospect.

We found out the execution time was decreased for multi-threading which can be easily seen in figure 4.3. The data obtained is summarized and we get speedup summary as the maximum 3.556, minimum 0.972, average 2.252 and median 2.285. Since the median is higher than average we can say the speedup is more skewed towards the left which is low. Yet, we can see there is good speedup factor. Similarly, we have calculated the execution time of multi-threaded exhaustive search to multi-threaded branch and bound on *table* 4.2. The summarized speedup results are way

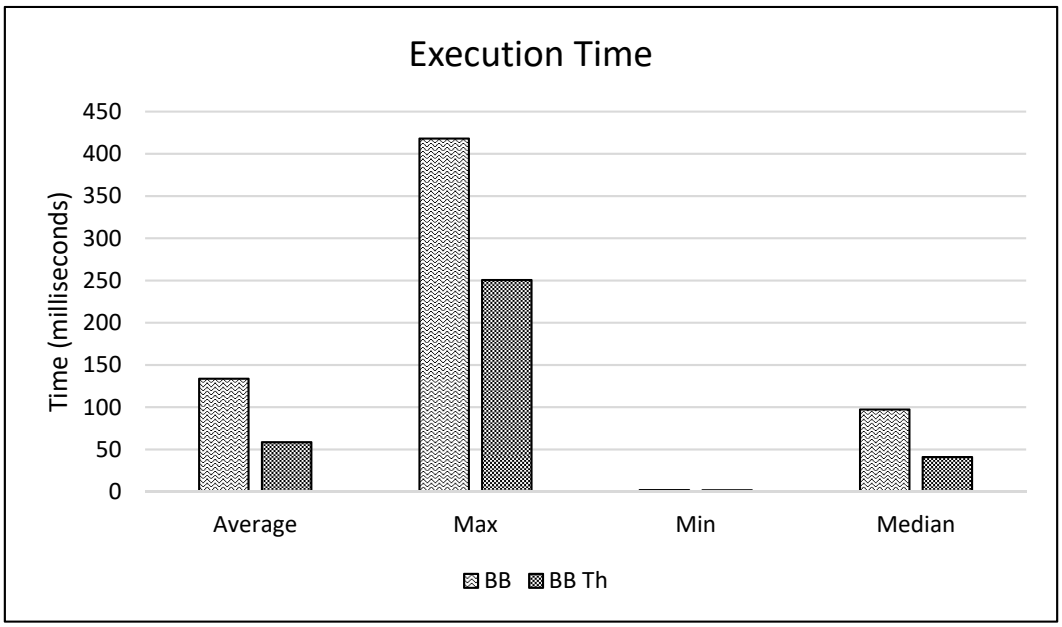


Figure 4.3: Execution Time comparison for Single and Multi-Threaded

better with average 67.039, maximum 587.637, minimum 3.718 and median 37.836. The median is lower than average which means there are higher speedup values in comparison to lower ones.

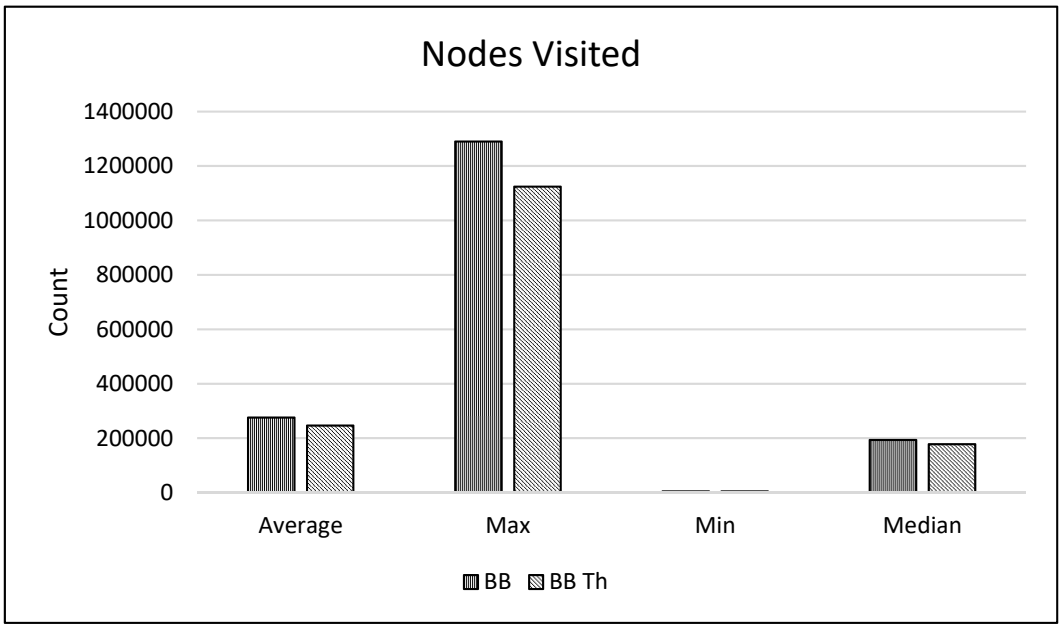


Figure 4.4: Nodes Visited comparison for Single and Multi-Threaded

The next is, we put the nodes visited by the branch and bound method based on single and multi-

thread are summarized in terms of their average node visited, maximum nodes visited, minimum nodes visited and median of nodes visited. It is clearly visible from *figure* 4.4, in every aspect the multi-threaded approach outperforms the single thread. In order to get optimal, for the case of 10 jobs, the exhaustive search has to calculate 3628800 solutions which we can see has been drastically reduced in case of branch and bound. The number of nodes visited is summarized as maximum 1124397, minimum 4507, average 246068 and median 177708. This shows that we were able to reduce a lot of computation (only 30% get optimal even for maximum nodes visited).

The result is also compared to the result from *A branch-and-bound method to minimize the makespan in a permutation flow shop with blocking and setup times* [TN17]. Only the relevant problems were taken and compared:

Problem Size	**BB(TN)**		**BB(here)**		**Speedup**
	Time	**Nodes**	**Time**	**Nodes**	
10_5	54930	199743.05	47.64	235953.50	1153.02
10_10	129950	270314.60	58.30	298573.10	2228.98

Table 4.3: BB results comparison with Takano and Nagano [TN17]

The *table* 4.3 shows our method visits more node in comparison to that of Takano and Nagano's results. However, there is a significant gain in terms of the execution time. In addition, the speedup is even better for a larger problem.

4.3 Genetic Algorithm Results

The genetic algorithm was used to calculate near-optimal solutions for all the small size problems from VFR benchmark problems with varied population size and generations. The results are generated varying the generation and the population size. Further, the set of 10 problems of each problem size are averaged in terms of the approximation ratio (AR) and the execution time. The AR here denotes how close to optimal we were able to get the makespan. In addition, the exact optimal obtained for that problem size is also noted.

We have results summarized for twenty-four different problem size into three different tables. Each table has records for two different values of generation and population. In the first table, we took 50 population size and generation of 50 and 100. Similarly on the second table population is increased to 100 and generations to 100 and 1000. The third table has records for generation 2000 and population 100 as well for generations 1000 and population 1000.

S.N.	Problem Size	Generation 50 Population 50			Generation 100 Population 50		
		Ave AR	Ave Time	Count	Ave AR	Ave Time	Count
1	10_05	1.00539	41.87093	4	1.00162	79.26700	7
2	10_10	1.00483	33.35743	4	1.00643	78.45760	3
3	10_15	1.00950	33.33077	0	1.00248	77.08480	2
4	10_20	1.00920	33.92823	1	1.00410	78.52070	2
5	20_05	1.04371	39.08390	0	1.03076	79.60833	0
6	20_10	1.07757	38.15840	0	1.06539	73.98053	0
7	20_15	1.07059	38.61563	0	1.06261	70.66593	0
8	20_20	1.06446	37.74750	0	1.05550	68.19177	0
9	30_05	1.02147	39.68787	0	1.01935	74.48783	0
10	30_10	1.10117	38.28437	0	1.09125	71.27473	0
11	30_15	1.11074	36.78903	0	1.09393	68.17823	0
12	30_20	1.09615	35.49190	0	1.08396	65.73140	0
13	40_05	1.02954	40.26257	0	1.02016	72.73687	1
14	40_10	1.09974	37.50417	0	1.08995	73.94120	0
15	40_15	1.11996	35.35737	0	1.10796	70.73783	0
16	40_20	1.12134	36.75620	0	1.10704	69.20780	0
17	50_05	1.02196	35.74777	0	1.01411	69.48460	0
18	50_10	1.09544	33.94417	0	1.08130	66.22093	0
19	50_15	1.13573	33.30110	0	1.11449	64.89777	0
20	50_20	1.13615	33.40837	0	1.12478	64.60513	0
21	60_05	1.03200	29.98170	0	1.01900	57.68743	0
22	60_10	1.10462	29.02387	0	1.08971	56.20353	0
23	60_15	1.13586	28.81720	0	1.12293	56.09087	0
24	60_20	1.14475	28.95177	0	1.13094	56.20207	0

Table 4.4: Genetic algorithm summary results 1

S.N.	Problem Size	Generation 100 Population 100			Generation 1000 Population 100		
		Ave AR	Ave Time	Count	Ave AR	Ave Time	Count
1	10_05	1.00184	70.42243	8	1.00014	637.15124	9
2	10_10	1.00304	67.30763	5	1.00055	563.00970	8
3	10_15	1.00451	66.25617	3	1.00145	554.71808	7
4	10_20	1.00441	67.27426	3	1.00000	549.31816	10
5	20_05	1.03209	68.53643	0	1.01390	558.81753	0
6	20_10	1.05253	63.50790	0	1.03567	548.49307	0
7	20_15	1.05213	55.56420	0	1.02947	540.93880	0
8	20_20	1.04463	55.47463	0	1.02646	537.31767	0
9	30_05	1.01604	56.46510	1	1.00770	551.06350	3
10	30_10	1.07758	55.59820	0	1.05031	541.44977	0
11	30_15	1.09166	55.69220	0	1.05935	1235.39293	0
12	30_20	1.07884	56.01157	0	1.04972	2297.76910	0
13	40_05	1.01835	56.01090	1	1.00590	998.44786	3
14	40_10	1.08696	55.90403	0	1.05410	630.12128	0
15	40_15	1.10128	55.95207	0	1.06491	544.53797	0
16	40_20	1.10191	56.79460	0	1.07217	552.93111	0
17	50_05	1.01267	56.25187	1	1.00408	630.04404	3
18	50_10	1.07916	56.33490	0	1.04923	547.22150	0
19	50_15	1.11664	56.96643	0	1.07751	556.66084	0
20	50_20	1.11920	58.15757	0	1.08277	565.84767	0
21	60_05	1.02101	56.49667	0	1.00436	1189.01226	2
22	60_10	1.08601	57.13873	0	1.05288	2486.81500	0
23	60_15	1.11879	58.59247	0	1.07877	635.25043	0
24	60_20	1.12831	59.89063	0	1.08899	580.39360	0

Table 4.5: Genetic algorithm summary results 2

S.N.	Problem Size	Generation 2000 Population 100			Generation 1000 Population 1000		
		Ave AR	Ave Time	Count	Ave AR	Ave Time	Count
1	10_05	1	1231.2886	10	1	1912.20807	10
2	10_10	1.00085	1127.04913	8	1	1937.68647	10
3	10_15	1.00044	1109.87139	8	1	1944.28724	10
4	10_20	1	1099.07697	10	1	1933.09803	10
5	20_05	1.00956	1188.34078	0	1.00356	1902.75586	5
6	20_10	1.03018	1097.1213	0	1.01747	1913.47974	0
7	20_15	1.0255	1082.8509	0	1.01744	1913.43444	0
8	20_20	1.01966	1075.11493	0	1.01378	2023.56199	0
9	30_05	1.00662	1166.30976	3	1.00544	1945.76337	5
10	30_10	1.04901	1083.87214	0	1.0433	1968.38396	0
11	30_15	1.04903	1083.37943	0	1.04395	2069.26316	0
12	30_20	1.04918	2744.781	0	1.03957	2277.33846	0
13	40_05	1.00336	1148.62089	5	1.00301	1998.68504	4
14	40_10	1.04807	1091.90557	0	1.04423	2140.90675	0
15	40_15	1.06168	1238.30443	0	1.05737	2420.30684	0
16	40_20	1.06874	4779.18795	0	1.06045	2785.26738	0
17	50_05	1.00187	1171.33963	4	1.00287	2271.09966	3
18	50_10	1.04451	1094.29702	0	1.04193	2452.44397	0
19	50_15	1.07411	2802.18254	0	1.06927	4018.52802	0
20	50_20	1.07756	4038.72514	0	1.07054	3988.47355	0
21	60_05	1.00315	1172.21559	3	1.00462	2716.75994	2
22	60_10	1.05169	1106.39646	0	1.04525	6880.58966	0
23	60_15	1.06941	5149.50628	0	1.06913	3383.70755	0
24	60_20	1.08586	1252.06204	0	1.0759	6904.88014	0

Table 4.6: Genetic algorithm summary results 3

We accumulated the results of *table* 4.4, *table* 4.5 and *table* 4.6 based on number of jobs and number of machines. The results are compared in terms of the execution time, the AR achieved

and the number of exact optimal reached. The accuracy calculation formula is:

$$AR = \frac{GA\ Value}{Optimal}$$

The AR = 1 means the algorithm has generated the optimal perfectly. Any value higher than 1 means the result from GA is deviated from the optimal.

Analysis per number of jobs

The *figure* 4.5 shows the time comparison. It is seen with an increase in the number of jobs the execution time increase. The execution time increase for change in configuration values which is also increasing.

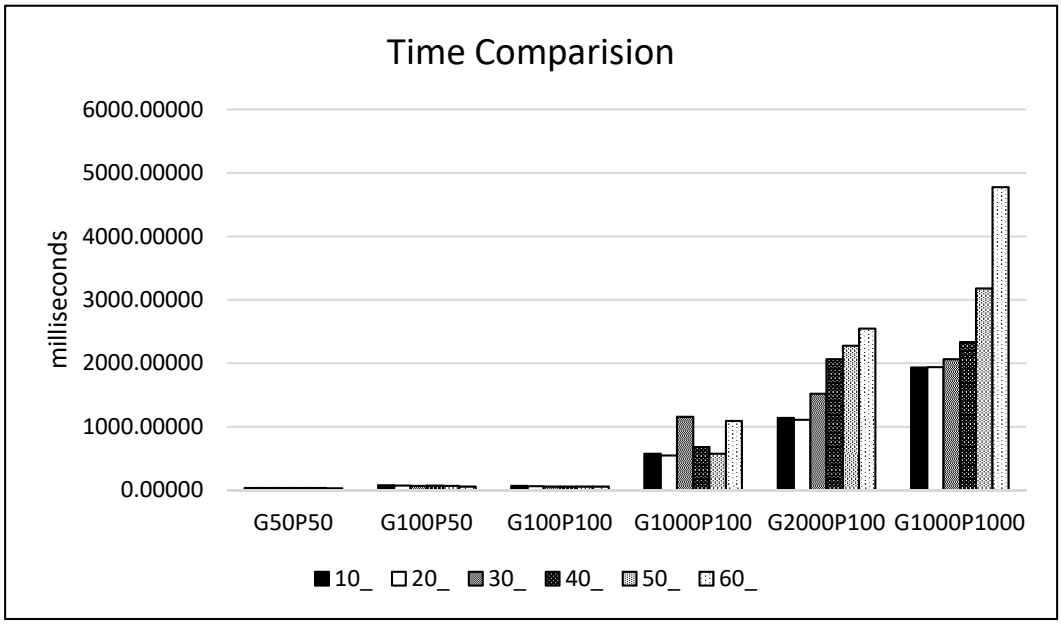


Figure 4.5: Averaged execution time comparison summarized over number of jobs

For an increase in the number of jobs the AR indicate more deviation from the optimal as seen in the *figure* 4.6. But with a change in configuration GA provides better results and converge towards the optimal.

The third comparison in the *figure* 4.7 shows that more optimal is achieved as we increase the value of generation and population size. For a lower number of job we are able to get more optimal.

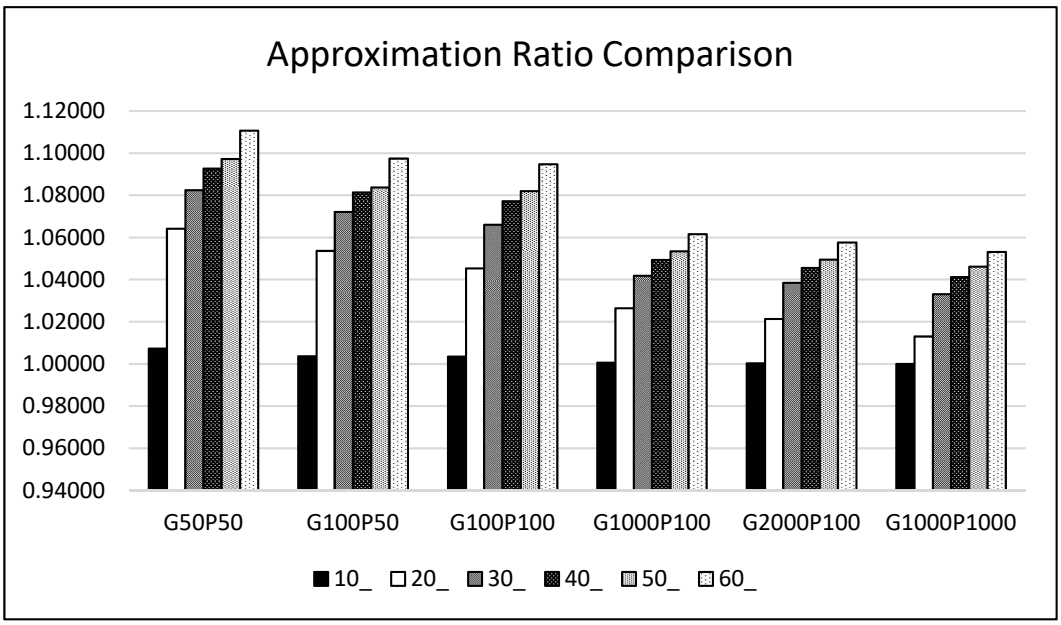


Figure 4.6: Averaged AR comparison summarized over number of jobs

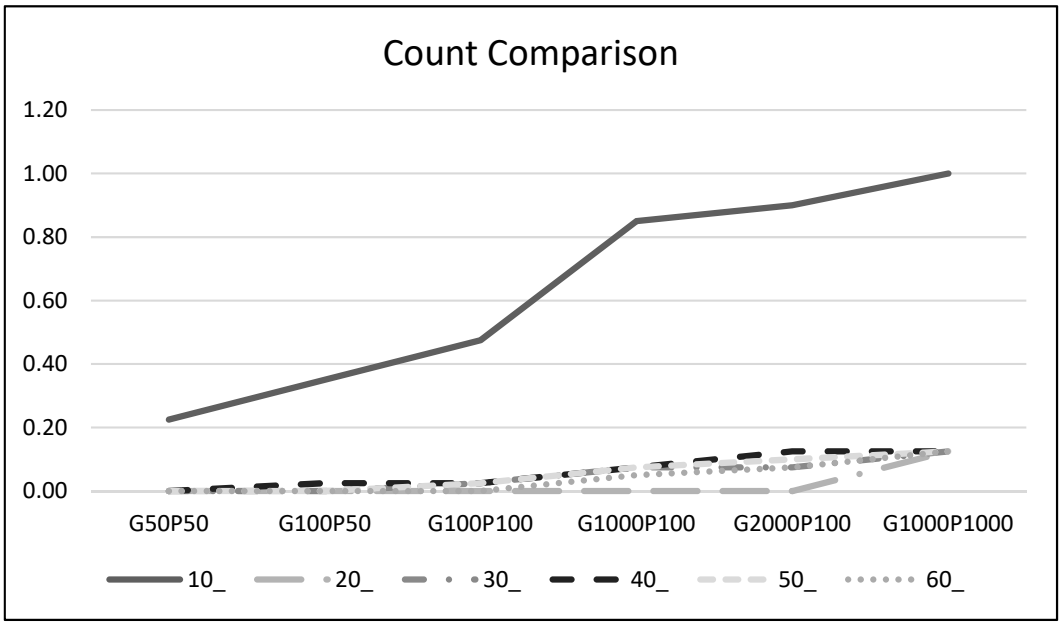


Figure 4.7: Exact optimal count comparison summarized over number of jobs

Analysis per number of machines

Based on the machine and the configuration values the GA tend to have an increase in the execution time. The respective execution time plot is shown in figure 4.8:

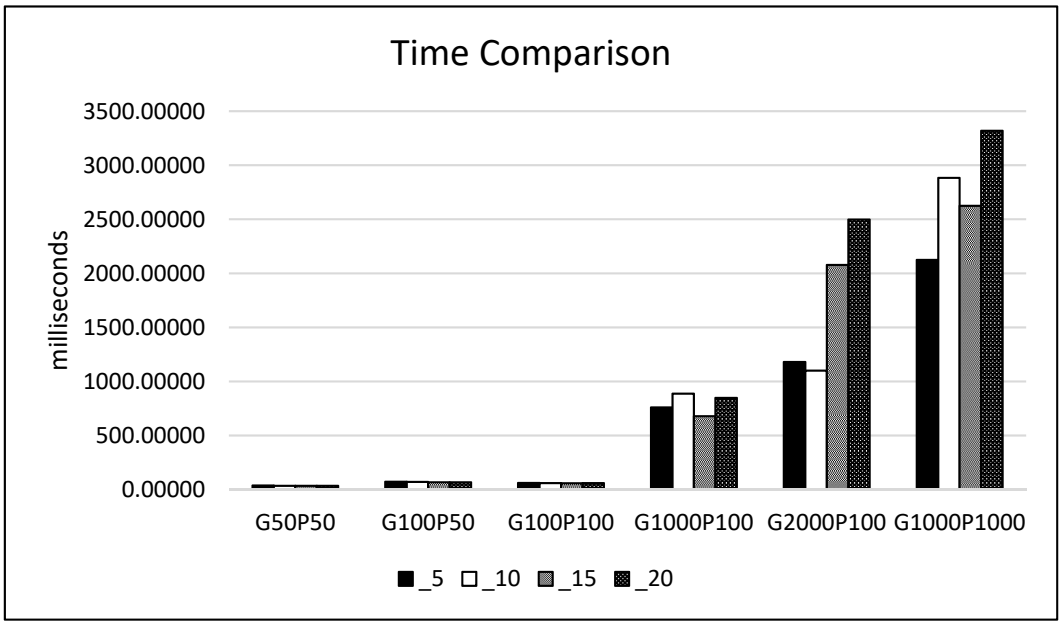


Figure 4.8: Averaged execution time comparison summarized over number of machines

The AR comparison is similar to the AR comparison based on the number of jobs as shown earlier. The result tends to be nearer to the optimal as the configuration is changed to higher values. Also, AR is better for problems with less number of machines. It is shown in the *figure* 4.9:

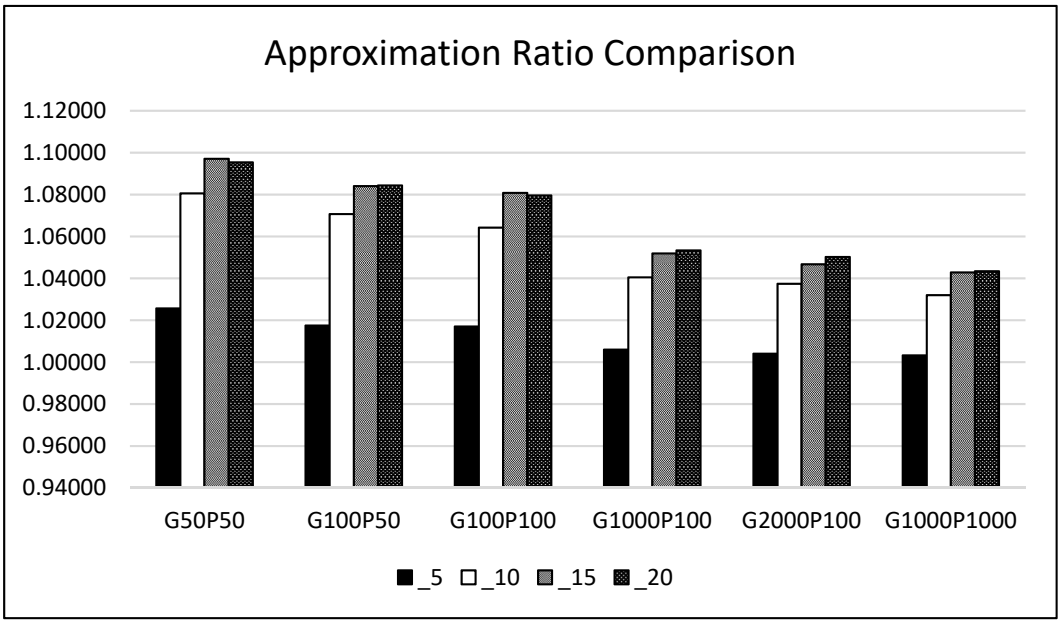


Figure 4.9: Averaged AR comparison summarized over number of machines

The trend is similar to that of job-based comparison for the number of optimal counts. More optimal is achieved as we change the configuration to higher values and count are higher for less number of machines. It is shown in the *figure* 4.10:

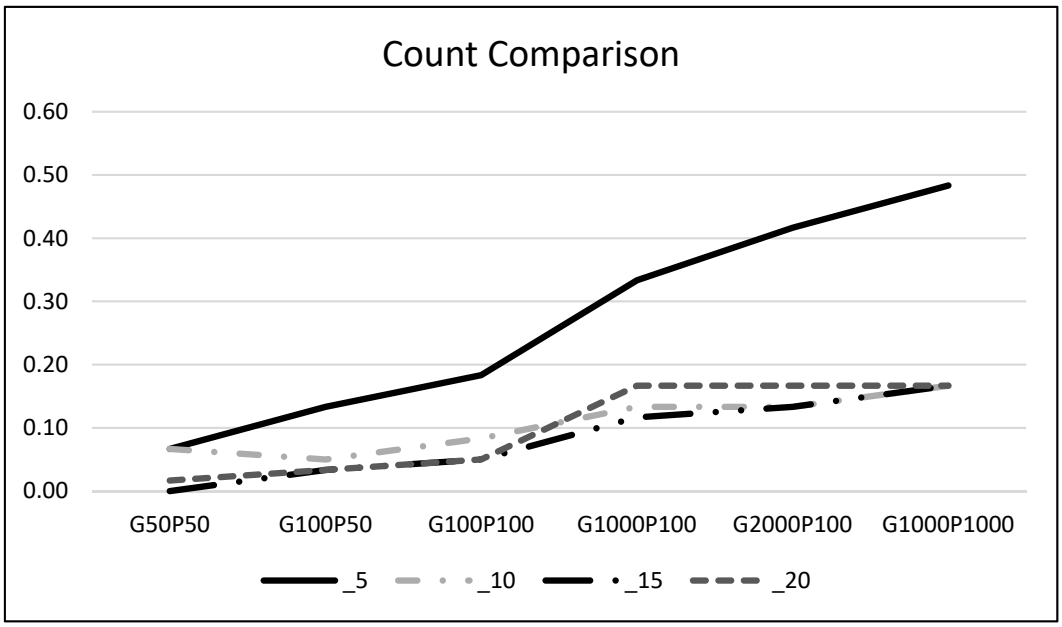


Figure 4.10: Exact optimal count comparison summarized over number of machines

4.4 Problems Faced

There are various technical problems that occurred during the thesis. The problem given by VFR was in a different format to read which was adjusted while reading from the file and creating the matrix. The multi-threading implementation initially raised the issue of race conditions as the original algorithm's approach didn't fit for the concurrent approach. With the use of multi-thread, the issue of memory leak arose which was handled by proper allocation and release of memory. The default library in C wasn't good enough for generating higher value random number. A different random number generator was used for the case. There was an issue of lack of memory for some case so many other problems weren't computed. For results comparison, there were no other results found in case of the VFR problems which are different from that of Taillard's.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

We here have shown the improvement in algorithms for performance as well compared the obtained results. We achieved a maximum speedup of 1.6 in case of exhaustive search which can be further increased with an increase in the number of threads. We also have computed for all the optimal for the problem which can be reflected as a basis for near-optimal methods.

The next part, Branch and Bound performs way better in comparison to that of exhaustive search. A minimum speedup of 3 to 587 is obtained for the same problem set in compared to exhaustive search. Moreover, in terms of execution time, our branch and bound algorithm had speedup over 1153.02 compared to that of the branch and bound algorithm of Takano and Nagano. This suggests our approach of the branch and bound served well.

In the case of the genetic algorithm, we scale up our problem solving capacity than that of the other two approaches. Comparatively, we have observed the variation of results based on the configuration of generations and population size. It can be clearly seen that the higher the number of generation and population size the better are the results. But, there is the cost of higher execution time. Also, the population size seems to have a larger impact on the result than the generations.

Overall, we have computed a benchmark for the VFR permutation flow shop problems. These problems seem to have higher complexity with an increase in the number of jobs as well as an increase in the number of machines.

5.2 Future Work

The exhaustive search approach can be made as distributed with multi-threading approach which could give results for a larger problem. The branch and bound algorithm can be extended as distributed with multi-threading approach. In addition, the proper memory management may increase the algorithm's capacity of solving a larger problem. In genetic algorithm, a varying configuration can help in determining the best values of generations and population size based on the problem size. A proper approach to determine mutation percentage and best carry over chromosome percentage may yield better results. Also, properly random initial random population generation method may provide better result.

Appendix A

Selected Source Code

Random Generator

```
/***Provides higher range random number ***/
unsigned long long llrand() {
        unsigned long long r = 0;
        for (int i = 0; i < 5; ++i) {
                r = (r << 15) | (rand() & 0x7FFF);
        }
        return r & 0xFFFFFFFFFFFFFFFFULL;
}
/***Provides a random number between specified range***/
unsigned int mynrand(unsigned int lower, unsigned int upper)
{
        return (lower + llrand() % (upper - lower + 1));
}
```

Swap

```
/***Swaps values at specified places***/
void swap(unsigned int *a, unsigned int i, unsigned int j)
{
        unsigned int temp = a[i];
        a[i] = a[j];
        a[j] = temp;
}
```

Lower Bound for Root

```
unsigned int GetInitialLBforRoot() {
        unsigned int a,  c, LB = 0, tempLB = 0;
```

```
        unsigned int* sum1 = (unsigned int *)malloc(totalJobs * sizeof(unsigned int));
        unsigned int* sum3 = (unsigned int *)malloc(totalJobs * sizeof(unsigned int));
        unsigned int sum2 = 0;
        unsigned int minFirstJob = 0;
        unsigned int minLastJob = 0;
        for (unsigned int m = 0;m < totalMachines;m++) {
                tempLB = 0;
                sum2 = 0;
                minFirstJob = 0;
                minLastJob = 0;
                //find min sum for first jobs before the machine m.
                for (c = 0;c < totalJobs;c++) {
                        sum1[c] = 0;
                        for (a = 0;a < m;a++) {
                                sum1[c] += originalMatrix[a][c];
                        }
                        //sum all jobs in the machine m
                        sum2 += originalMatrix[m][c];
                        //find sum of last jobs on remaining machine after machine m
                        sum3[c] = 0;
                        for (a = m + 1;a < totalMachines;a++) {
                                sum3[c] += originalMatrix[a][c];
                        }
                        if (c != 0) {
                                if (sum1[minFirstJob] > sum1[c]) {
                                        minFirstJob = c;
                                }
                                if (sum3[minLastJob] > sum3[c]) {
                                        minLastJob = c;
                                }
                        }
                }
                tempLB = sum1[minFirstJob] + sum2 + sum3[minLastJob];
                if (tempLB > LB) {
                        LB = tempLB;
                }
        }
        free(sum1);
        free(sum3);
        return LB;
}
```

Lower Bound for Node

```
unsigned int GetLBforTheNode(unsigned int*jOrder, unsigned int*rOrder,
```

```
                        unsigned int level) {
        register unsigned int m, c, a;
        unsigned int tempLB, tempClr, sum2, minLastJobs, LB = 0;
        unsigned int* sum3 = malloc(totalJobs * sizeof(unsigned int));
        for (m = 0;m < totalMachines;m++) {
                tempLB = sum2 = minLastJobs = 0;
                tempClr = GetPartialCmax(jOrder, level, m);
                for (c = 0;c < totalJobs - level;c++) {
                        sum2 += originalMatrix[m][rOrder[c]];
                        sum3[c] = 0;
                        for (a = m + 1;a < totalMachines;a++) {
                                sum3[c] += originalMatrix[a][rOrder[c]];
                        }
                        if (c != 0) {
                                if (sum3[minLastJobs] > sum3[c]) {
                                        minLastJobs = c;
                                }
                        }
                }
                tempLB = tempClr + sum2 + sum3[minLastJobs];
                if (tempLB > LB) {
                        LB = tempLB;
                }
        }
        free(sum3);
        return LB;
}
```

Generate Partial Order

```
void GetPartialJoborders(unsigned int *partialJobOrder, unsigned int *remJobOrder,
                        unsigned int currLevel, unsigned int remJobNo) {
        register unsigned int r = 0, c = 0;
        //transferred specified job from remaining order to fixed order
        partialJobOrder[currLevel] = remJobOrder[remJobNo];
        //remove the job no from remJobOrder
        for (c;c < totalJobs - currLevel;c++) {
                if (c != remJobNo) {
                        remJobOrder[r] = remJobOrder[c];
                        r++;
                }
        }
```

Bibliography

[AA98] A. Allahverdi and T. Aldowaisan. Job lateness in flowshops with setup and removal times separated. *Journal of the Operational Research Society*, 49(9):1001–1006, 1998.

[AA04] Tariq Aldowaisan and Ali Allahverdi. New heuristics for m-machine no-wait flowshop to minimize total completion time. *Omega*, 32(5):345 – 352, 2004.

[AGA99] Ali Allahverdi, Jatinder N.D Gupta, and Tariq Aldowaisan. A review of scheduling research involving setup considerations. *Omega*, 27(2):219 – 239, 1999.

[AHS00] I. N. Kamal Abadi, Nicholas G. Hall, and Chelliah Sriskandarajah. Minimizing cycle time in a blocking flowshop. *Operations Research*, 48(1):177–180, 2000.

[Ash70] Said Ashour. A branch-and-bound algorithm for flow shop scheduling problems. *A I I E Transactions*, 2(2):172–176, 1970.

[Bak75] Kenneth R. Baker. A comparative study of flow-shop algorithms. *Operations Research*, 23(1):62–73, 1975.

[Ban79] S. P. Bansal. On lower bounds in permutation flow shop problems. *International Journal of Production Research*, 17(4):411–418, 1979.

[Ber00] Edy Bertolissi. Heuristic algorithm for scheduling in the no-wait flow-shop. *Journal of Materials Processing Technology*, 107(1):459 – 465, 2000.

[BH91] Shaukat A. Brah and John L. Hunsucker. Branch and bound algorithm for the flow shop with multiple processors. *European Journal of Operational Research*, 51(1):88 – 99, 1991.

[Bru07] Peter Brucker. *Scheduling Algorithms*. Springer Publications, 5th edition, 2007.

[CR96] Jacques Carlier and Ismal Reba. Two branch and bound algorithms for the permutation flow shop problem. *European Journal of Operational Research*, 90(2):238 – 251, 1996.

[DT64] Richard A. Dudek and Ottis Foy Teuton. Development of m-stage decision rule for scheduling n jobs through m machines. *Oper. Res.*, 12(3):471–497, 1964.

[Eva15] Eva Vallada, Rubn Ruiz, Jose M. Framinan. New hard benchmark for flowshop scheduling problems minimising makespan. *European Journal of Operational Research*, 240:666–677, 2015.

[FGL04] J. M. Framinan, J. N. D. Gupta, and R. Leisten. A review and classification of heuristics for permutation flow-shop scheduling with makespan objective. *The Journal of the Operational Research Society*, 55(12):1243–1255, 2004.

[Gen67] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS spring joint computer conference*, 1967.

[GJS76] M. R. Garey, D. S. Johnson, and Ravi Sethi. The complexity of flowshop and jobshop scheduling. *Math. Oper. Res.*, 1(2):117–129, 1976.

[Gup71] Jatinder N. D. Gupta. A functional heuristic algorithm for the flowshop scheduling problem. *Journal of the Operational Research Society*, 22(1):39–47, 1971.

[Gup79] Jatinder N. D. Gupta. *A review of flowshop scheduling research*, pages 363–388. Springer Netherlands, Dordrecht, 1979.

[GZAT09] M. Gholami, M. Zandieh, and A. Alem-Tabriz. Scheduling hybrid flow shop with sequence-dependent setup times and machines with random breakdowns. *The International Journal of Advanced Manufacturing Technology*, 42(1):189–201, May 2009.

[HS96] Nicholas G. Hall and Chelliah Sriskandarajah. A survey of machine scheduling problems with blocking and no-wait in process. *Operations Research*, 44(3):510–525, 1996.

[IS65] Edward Ignall and Linus Schrage. Application of the branch and bound technique to some flow-shop scheduling problems. *Operations Research*, 13(3):400–412, 1965.

[JT90] Edward F. Stafford Jr and Fan T. Tseng. On the srikar-ghosh milp model for the ivx m sdst flowshop problem. *International Journal of Production Research*, 28(10):1817–1830, 1990.

[KM87] Sandeep Kochhar and Robert J.T. Morris. Heuristic methods for flexible flow line scheduling. *Journal of Manufacturing Systems*, 6(4):299 – 314, 1987.

[KS80] J. R. King and A. S. Spachis. Heuristics for flow-shop scheduling. *International Journal of Production Research*, 18(3):345–357, 1980.

[LCL93] Chung-Yee Lee, T. C. E. Cheng, and B. M. T. Lin. Minimizing the makespan in the 3-machine assembly-type flowshop scheduling problem. *Management Science*, 39(5):616–625, 1993.

[LLK78] B. J. Lageweg, J. K. Lenstra, and A. H. G. Rinnooy Kan. A general bounding scheme for the permutation flow-shop problem. *Operations Research*, 26(1):53–67, 1978.

[LLRK78] B. J. Lageweg, J. K. Lenstra, and A. H. G. Rinnooy Kan. A general bounding scheme for the permutation flow-shop problem. *Oper. Res.*, 26(1):53–67, 1978.

[Lom65] Z. A. Lomnicki. A "branch-and-bound" algorithm for the exact solution of the three-machine scheduling problem. *OR*, 16(1):89–100, 1965.

[Lou96] Helena Ramalhinho Loureno. Sevast'yanov's algorithm for the flow-shop scheduling problem. *European Journal of Operational Research*, 91(1):176 – 189, 1996.

[LS00] In Lee and Michael J Shaw. A neural-net approach to real time flow-shop sequencing. *Computers & Industrial Engineering*, 38(1):125 – 147, 2000.

[MB67] G. B. McMahon and P. G. Burton. Flow-shop scheduling with the branch-and-bound method. *Operations Research*, 15(3):473–481, 1967.

[Mit98] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1st edition, 1998.

[Moc95] Joo Vitor Moccellin. A new heuristic method for the permutation flow shop scheduling problem. *Journal of the Operational Research Society*, 46(7):883–886, 1995.

[NEH83] Muhammad Nawaz, E Emory Enscore, and Inyong Ham. A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. *Omega*, 11(1):91 – 95, 1983.

[OP08] Djamila Ouelhadj and Sanja Petrovic. A survey of dynamic scheduling in manufacturing systems. *Journal of Scheduling*, 12(4):417, Oct 2008.

[Pot80] C.N. Potts. An adaptive branching rule for the permutation flow-shop problem. *European Journal of Operational Research*, 5(1):19 – 25, 1980.

[PPE84] Yang Byung Park, C. Dennis Pegden, and E. Emory Enscore. A survey and evaluation of static flowshop scheduling heuristics. *International Journal of Production Research*, 22(1):127–141, 1984.

[PSS+95] C. N. Potts, S. V. Sevast'janov, V. A. Strusevich, L. N. van Wassenhove, and C. M. Zwaneveld. The two-stage assembly scheduling problem: Complexity and approximation. *Operations Research*, 43(2):346–355, 1995.

[R.L79] R.L Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[RMA05] Rubn Ruiz, Concepcin Maroto, and Javier Alcaraz. Solving the flowshop scheduling problem with sequence dependent setup times using advanced metaheuristics. *European Journal of Operational Research*, 165(1):34 – 54, 2005.

[RMB99] Roger Z. Ríos-Mercado and Jonathan F. Bard. A branch-and-bound algorithm for permutation flow shops with sequence-dependent setup times. *IIE Transactions*, 31(8):721–731, 1999.

[RWCM03] William L. Maxwell Richard W. Conway and Louis W. Miller. *Theory of Scheduling*. Dover Publications, Inc., 1st edition, 2003.

[S05] S. Reza Hejazi * and S. Saghafian. Flowshop-scheduling problems with makespan criterion: a review. *International Journal of Production Research*, 43(14):2895–2929, 2005.

[Sal73] Michael S. Salvador. A solution to a special class of flow shop scheduling problems. In Salah E. Elmaghraby, editor, *Symposium on the Theory of Scheduling and Its Applications*, pages 83–91, Berlin, Heidelberg, 1973. Springer Berlin Heidelberg.

[SG86] Bellur N. Srikar and Soumen Ghosh. A milp model for the n-job, m-stage flowshop with sequence dependent set-up times. *International Journal of Production Research*, 24(6):1459–1474, 1986.

[SGM05] Nour El Houda Saadani, Alain Guinet, and Mohamed Moalla. A travelling salesman approach to solve the f/no-idle/cmax problem. *European Journal of Operational Research*, 161(1):11 – 20, 2005. IEPM: Focus on Scheduling.

[ST02] Edward F. Stafford and Fan T. Tseng. Two models for a family of flowshop sequencing problems. *European Journal of Operational Research*, 142(2):282 – 293, 2002.

[Szw71] Wlodzimierz Szwarc. Elimination methods in the m n sequencing problem. *Naval Research Logistics Quarterly*, 18(3):295–305, 1971.

[TJ01] Fan T. Tseng and Edward F. Stafford Jr. Two milp models for the n m sdst flowshop sequencing problem. *International Journal of Production Research*, 39(8):1777–1809, 2001.

[TN17] Mauricio Iwama Takano and Marcelo Seido Nagano. A branch-and-bound method to minimize the makespan in a permutation flow shop with blocking and setup times. *Cogent Engineering*, 4(1):1389638, 2017.

[WH89] Marino Widmer and Alain Hertz. A new heuristic method for the flow shop sequencing problem. *European Journal of Operational Research*, 41(2):186 – 193, 1989.

Curriculum Vitae

Graduate College
University of Nevada, Las Vegas

Hari Prasad Sapkota
harrysapkota@gmail.com

Degrees:

Master of Science in Computer Science 2019
University of Nevada Las Vegas

Thesis Title: Benchmarking Permutation Flow Shop Problem: Adaptive and Enumerative Approaches Implementations via Novel Threading Techniques

Thesis Examination Committee:

Chairperson, Dr. Wolfgang Bein, Ph.D.
Committee Member, Dr. Ajoy Datta, Ph.D.
Committee Member, Dr. Laxmi Gewali, Ph.D.
Graduate Faculty Representative, Dr. Henry Selvaraj, Ph.D.