School of Physical Sciences and Engineering
MSc in Advanced Software Engineering
2006

Search-based Approaches to Understanding Project Management Choices

Project Report
by

Fahim Qureshi

Supervised
by

Professor Mark Harman

1st September 2006

ACKNOWLEDGEMENTS

I would like to express my gratitude to the following for their help and support throughout the project.

First of all I am thankful to Allah Almighty who gave me strength and ability to complete my MSc and this project. Without His will nothing is possible.

My dearest friend Ahmad Faruq Taraq who helped me and guided me at critical stages of the project with his analytical skills and exceptional expertise. I am indebted to him for his continuous support.

My lovely fiancé Sabin for her love, care and whole-hearted support.

My family – my father and mother and my all siblings for their confidence in me and encouragement they provided at every stage.

Kiarash Mahdavi for his valuable assistance during initial phase of the project that helped me define my project objectives.

Professor Mark Harman for his inspiring supervision, innovative ideas, valuable advice and continuous motivation that made this project possible.

ABSTRACT

Project managers always try to ensure that the project is completed in minimum possible time. Once breakdown and effort estimation of tasks is completed, next step is allocation of tasks to the project teams. This is critical stage of the project which determines the completion time of the project. An effective tasks allocation must ensure both minimum completion time and maximum resource utilization. Such decision making becomes difficult with the increase in project size. Also the complexity of such problem increase in the presence of inter-related tasks.

Based on notion of search-based software engineering, this problem can be categorized as search-based problem with no ideal solution. Heuristic techniques prove to be effective for such problems in obtaining near optimal solution. Work has been done in past to solve the problem of resource allocation using heuristic techniques and the results obtained are encouraging. This project aims at extending the work of solving project management problems using search-based approach.

While this project attempts to solve problem of efficient resource allocation, it also addresses the issue of communication and its importance in a project. Project managers tend to use estimation techniques that assume no communication takes place among team members and thus confuse effort with progress. This leads to false assumption that increase in human resource will result in reduction of completion time.

Brook's law is studied and various experiments are carried out, based on Brook's law and its variations, in this project to understand and address this problem of communication and enable project managers to better understand the relationship between time and human resource in presence of complex inter-related tasks. The aim is to provide project managers with techniques that can provide more accurate estimates and allow them to make realistic assumptions about the project progress and completion time to be achieved.

Results proved that this technique outperforms human decision making as far as resource allocation is concerned and allows better human resource utilization by allocating tasks more uniformly resulting in lesser completion time. Results from Brook's law experiments proved that communication requires significant amount of effort and it must be ensured that proper communication level must be maintained in order to complete tasks on time.

TABLE OF CONTENTS

LIST OF TABLES AND FIGURES

1. TABLES

2. FIGURES

1. INTRODUCTION

Most large scale software projects involve several teams and numerous individual work packages. These work packages are derived from Work Breakdown Structure (WBS). The optimal allocation of work packages to teams is an important problem and becomes more difficult as the size of the project increases. Such a resource allocation problem is an example of a bin packing problem, the solution of which is NP-hard and for which, heuristic algorithms are known to be effective[1,2,3].

Apart from efficient resource allocation another major problem faced by project managers while meeting the deadlines of the project is the use of estimation techniques that fallaciously confuse effort with progress, hiding the assumption that men (human resource) and months are interchangeable. This leads to a false assumption that increasing the human resources will directly reduce the completion time of the project. As a result when a project falls behind schedule project managers add more people to the project in an attempt to improve the progress. The result is more delay in the project. This happens due to the fact that project managers fail to understand the complex relationship between human resource and time in presence of inter-related tasks.

There is a need for technique that allows optimization of resource allocation in order to reduce the completion time of the project. Also there is need to understand the relationship between human resource and time in presence of complex inter-related tasks to prevent situations which might result in delay of the projects.

1.1 Objective

This project aims to achieve two main objectives:

- Optimization of resource allocation using search based techniques
- Study the effect of Brook's law in order to understand the relationship between men (human resource) and months (time) in presence of inter-related tasks.

1.2 Road Map

The document is divided into the following sections:

- Literature review gives a detailed review of related research papers and explanation of concepts which act as basis of this project.

- Research Questions and Experiment Design discusses the research questions to be answered by this project and how experiments will be designed to answer those questions
- Implementation chapter discusses in detail the implementation phase of the project which includes design and explanation of the algorithm. The chapter also discusses some main methods defined in the code to provide better understanding of the implementation.
- Testing and Results explains how testing was performed and what were the results. The analysis of results is also presented in this chapter.
- Future work suggests some future directions in which this work can be carried forward.
- Conclusion sums up what has been achieved and discovered by this project.
- There is glossary of terms and appendices of code and other details at the end of the report.

2. LITERATURE REVIEW

In this chapter we discuss the related research work done in past as well as explain various concepts which provide basis of this project.

2.1 Related Work

2.1.1 A Robust Search Based Approach to Project Management in Presence of Abandonment, Rework, Error and Uncertainty

This paper addresses the problem of uncertainty, abandonment and rework of work packages by combing genetic algorithms with queuing simulation model in a robust manner. A tandem genetic algorithm is used to search for the best sequence in which to process work packages and the best allocation of staff to project teams. The simulation model, that computes the project estimated completion date, guides the search. The possible impacts of rework, abandonment and erroneous or uncertain initial estimates are characterised by separate error distribution. The paper presents results from the application of these techniques to data obtained from large scale commercial software maintenance project.

The search-based approach suggested in the paper uses two GAs in tandem. The first GA aims to find optimal solution to the problem of determining the ideal order in which to present work packages to the process. The second GA aims to find a good allocation of staff to project teams to maximize the throughput for the WP ordering found by the first GA. The overall optimization process is iterative; the results of the second GA are fed back into the first GA, which attempts to fine tune the optimisation of WP ordering. The repetition continues until the search stabilises on a solution.

Once an optimal solution is found, a sensitivity analysis is performed to model the effect of i) uncertainty in the effort estimation, ii) abandonment, and iii) rework.

An empirical study is proposed in the paper based on the results obtained after applying the suggested technique on real life project. This study aims at defining a near optimal project staffing and scheduling for maintenance activities of WPs coming from the project.

This paper emphasises on effect of abandonment and rework of work packages on progress of the project which result in erroneous estimates and uncertainty. Thus the paper suggests a robust approach that is not affected by abandonment or rework of work packages.

2.1.2 Search-based Techniques for Optimizing Software Project Resource Allocation

This paper evaluates three optimization techniques namely hill climbing, simulated annealing and genetic algorithm in order to optimize resource allocation among project teams. Each technique is applied to two very different encoding strategies. Each encoding represents the way in which the work packages of the overall project are to be allocated to teams of programmers.

Results obtained from evaluation prove that the overall difference between the three approaches appear to be small whereas the scanning genome encoding was found to outperform vector-based genome encoding.

2.2 Brook's Law and the Mythical Man-Month

Brook's law states that **"adding more manpower to late software projects makes it later"**. The number of months of a project depends upon its sequential constraints. The maximum number of men depends upon the number of independent subtasks. From these two quantities one can derive schedules using different number of men and months.

2.2.1 The Man-Month [4]

Man-Month (now referred to as person-months) is the unit of effort used in estimating and scheduling. Cost does indeed vary as the product of number of men and the number of months. Progress does not. Hence the man-month as a unit for measuring the size of job is a dangerous and deceptive myth. It implies that man and months are interchangeable.

Men and month are interchangeable commodities only when a task can be partitioned among many workers with no communication among them. This is true of reaping wheat or picking cotton but not true of programming.

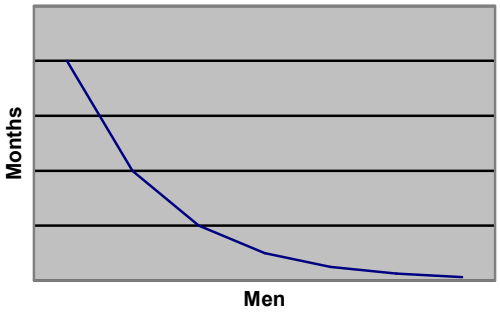


Figure 1: Time versus number of workers- perfectly partitioned tasks

When a task cannot be partitioned because of sequential constrain, the application of more effort has no effect on the schedule. The bearing of child takes nine months, no matter how many women are assigned. Many software tasks have this characteristic because of sequential nature of debugging.

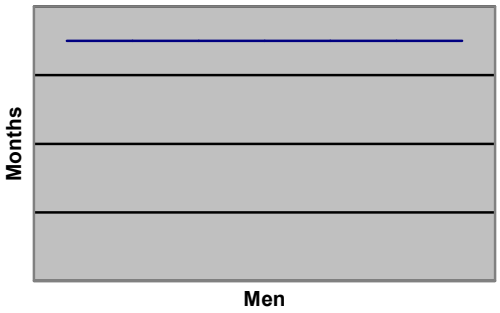


Figure 2: Time versus number of workers- unpartitioned tasks

In tasks that can be partitioned but require communication among subtasks, the effort of communication must be added to the amount of work to be done. Therefore the best that can be done is somewhat poorer than even trade of men for months.

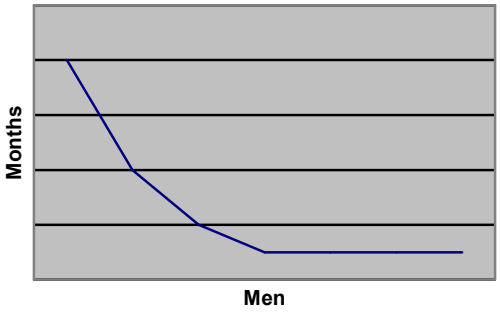


Figure. 3: Time versus number of workers- partitionable task requiring communication

Effort required for intercommunication is calculated in the following manner. If each part of the task must be separately coordinated with each other part, the effort increases as n(n-1)/2. Three workers require three times as much pair wise intercommunication as two; four requires six times as much as two. If moreover, there need to be conferences among three, four, etc. workers to resolve things jointly, matters get worse yet. The added effort of communication may fully counteract the division of original task and bring us to the situation of the following figure:

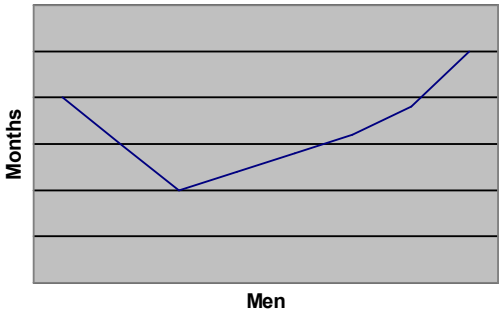


Figure 4: Time versus number of workers- task with complex interrelationships

2.2.2 Mathematical Representation [5]

The effort required for intercommunication among team members in order to complete partitioned task can be calculated as following:

Let MMc be the average effort expended by a pair of persons, working on the project, in communicating with each other - determined to be based on the project not *N (*where N = number of members). Then:

$\sum$ MMc =N (N-1)/2 x MMc = Pairs x MMc

Assuming no original level isolations but complete communication among team members.

MMn = Effort required without communication

MM = Total effort including communication:

MM=MMn + (N (N-1)/2) x MMc = N x T

T = MMn/N + ((N-1)/2) x MMc

2.3 Bin Packing Problem

In computational complexity theory, the bin packing problem is a combinatorial NP-hard problem. In it, objects of different volumes must be packed into a finite number of bins of capacity *V* in a way that minimizes the number of bins used [6].

There are many variations of this problem, such as 2D packing, linear packing, packing by weight, packing by cost, and so on. They have many applications, such as filling up containers, loading trucks with weight capacity, and creating file backup in removable media.

2.4 Heuristic Algorithms

Heuristic algorithms either give nearly the right answer or provide a solution for not all instances of the problem. Usually heuristic algorithms are used for problems that cannot be easily solved. Classes of time complexity are defined to distinguish problems according to their" hardness". Class P consists of all those problems that can be solved on a deterministic Turing machine in polynomial time from the size of the input. Turing machines are an abstraction that is used to formalize the notion of algorithm and computational complexity. Class NP consists of all those problems whose solution can be found in polynomial time on a non-deterministic Turing machine. Since such a machine does not exist, practically it means that an exponential algorithm can be written for an NP-problem, nothing is asserted whether a polynomial algorithm exists or not. A subclass of NP, class NP-complete includes problems such that a polynomial algorithm for one of them could be transformed to polynomial algorithms for solving all other NP problems. Finally, the class NP-hard can be understood as the class of problems that are NP-complete or harder. NP-hard problems have the same trait as NP-complete problems but they do not necessary belong to class NP, that is class NP-hard includes also problems for which no algorithms at all can be provided. In order to justify application of some heuristic algorithm we prove that the problem belongs to the classes NP-complete or NP-hard. Most likely there are no polynomial algorithms to solve such problems; therefore, for sufficiently great inputs heuristics are developed [7].

2.4.1 Genetic Algorithms

A genetic algorithm (or GA for short) is a programming technique that mimics biological evolution as a problem-solving strategy. Given a specific problem to solve, the input to the GA is a set of potential solutions to that problem, encoded in some fashion, and a metric called a fitness function that allows each candidate to be quantitatively evaluated. These candidates may be solutions already known to work, with the aim of the GA being to improve them, but more often they are generated at random [8].

The GA then evaluates each candidate according to the fitness function. In a pool of randomly generated candidates, of course, most will not work at all, and these will be deleted. However, purely by chance, a few may hold promise - they may show activity, even if only weak and imperfect activity, toward solving the problem.

These promising candidates are kept and allowed to reproduce. Multiple copies are made of them, but the copies are not perfect; random changes are introduced during the copying process. These digital offspring then go on to the next generation, forming a new pool of candidate solutions, and are subjected to a second round of fitness evaluation. Those candidate solutions which were worsened, or made no better, by the changes to their code are again deleted; but again, purely by chance, the random variations introduced into the population may have improved some individuals, making them into better, more complete or more efficient solutions to the problem at hand. Again these winning individuals are selected and copied over into the next generation with random changes, and the process repeats. The expectation is that the average fitness of the population will increase each round, and so by repeating this process for hundreds or thousands of rounds, very good solutions to the problem can be discovered [9].

Methods of Selection

- Elitist Selection
- Fitness Proportionate Selection
- Roulette-Wheel Selection
- Scaling Selection
- Tournament Selection
- Rank Selection

- Generation Selection
- Steady State Selection
- Hierarchical Selection

Methods of Change

- Mutation
- Crossover
    - Single Point Crossover
    - Uniform Crossover

2.4.2 Hill-climbing

Similar to genetic algorithms, though more systematic and less random, a hill-climbing algorithm begins with one initial solution to the problem at hand, usually chosen at random. The string is then mutated, and if the mutation results in higher fitness for the new solution than for the previous one, the new solution is kept; otherwise, the current solution is retained. The algorithm is then repeated until no mutation can be found that causes an increase in the current solution's fitness and this solution is returned as the result [11]. (To understand where the name of this technique comes from, imagine that the space of all possible solutions to a given problem is represented as a three-dimensional contour landscape. A given set of coordinates on that landscape represents one particular solution. Those solutions that are better are higher in altitude, forming hills and peaks; those that are worse are lower in altitude, forming valleys. A "hill-climber" is then an algorithm that starts out at a given point on the landscape and moves inexorably uphill.) Hill-climbing is what is known as a greedy algorithm, meaning it always makes the best choice available at each step in the hope that the overall best result can be achieved this way. By contrast, methods such as genetic algorithms and simulated annealing are not greedy; these methods sometimes make suboptimal choices in the hopes that they will lead to better solutions later on.

Hill climbing is used widely in artificial intelligence fields, for reaching a goal state from a starting node. Choice of next node/ starting node can be varied to give a list of related algorithms.

The algorithm [10] below describes steps involved in Hill Climbing.

```
Algo (Hill Climbing)
  bestEval = -INF;
  currentNode = startNode;
  bestNode = NULL;
  for MAX times
     if (EVAL(currentNode) > bestEval)
         bestEval = EVAL(currentNode);
         bestNode = currentNode;
     L = NEIGHBORS(currentNode);
     tempMaxEval = -INF;
     for all x in L
        if (EVAL(x) > tempMaxEval)
            currentNode = x;
            tempMaxEval = EVAL(x);
  return currentNode;
```

The problem with hill climbing is that it will find only local maxima. Unless the heuristic is good / smooth, it doesn't reach global maxima. There are many methods to overcome this problem, including iterated hill climbing, stochastic hill climbing, random walks, and simulated annealing.

2.4.3 Simulated Annealing

Another optimization technique similar to evolutionary algorithms is known as simulated annealing. The idea borrows its name from the industrial process of *annealing* in which a material is heated to above a critical point to soften it, then gradually cooled in order to erase defects in its crystalline structure, producing a more stable and regular lattice arrangement of atoms [12]. In simulated annealing, as in genetic algorithms, there is a fitness function that defines a fitness landscape; however, rather than a population of candidates as in GAs, there is only one candidate solution. Simulated annealing also adds the concept of "temperature", a global numerical quantity which gradually decreases over time. At each step of the algorithm, the solution mutates (which is equivalent to moving to an adjacent point of the fitness landscape). The fitness of the

new solution is then compared to the fitness of the previous solution; if it is higher, the new solution is kept. Otherwise, the algorithm makes a decision whether to keep or discard it based on temperature. If the temperature is high, as it is initially, even changes that cause significant decreases in fitness may be kept and used as the basis for the next round of the algorithm, but as temperature decreases, the algorithm becomes more and more inclined to only accept fitness-increasing changes. Finally, the temperature reaches zero and the system "freezes"; whatever configuration it is in at that point becomes the solution. Simulated annealing is often used for engineering design applications such as determining the physical layout of components on a computer chip [13].

3. RESEARCH QUESTIONS AND EXPERIMENT DESIGN

In this chapter we discuss the research questions that will be answered by this project. The implementation and testing strategy will be based on these research questions therefore it is important to fully understand these questions and how the implementation and testing strategy will be formulated.

3.1 Questions

The research questions to be addressed in this project are:

3.1.1 How to allocate work packages to teams in such a way that project completion time is minimized?

As we already discussed resource allocation is an important problem in project management and as size of the project increases, this problem becomes more complex. Such problem is an example of bin packing problem and search space for such problem is T^{Nw} where T is number of teams and Nw is number of work packages. The solution of such problem is NP-hard and for which heuristic techniques are known to be effective.

Once we have decided that heuristic techniques are effective for such problem, next step is to define the fitness function for this problem. Now during tasks allocation, since number of tasks is greater than number of teams, multiple tasks are allocated to teams simultaneously. The team that finishes in the end decides the completion time of the project because rest of the teams had finished their allocated tasks by then. Therefore in order to minimize the completion time of the project we have to minimize the maximum time taken by any team to complete the allocated tasks. Now we can define our fitness function as **"maximum time taken by any team to complete allocated tasks".** We will try to minimize the fitness value as low as possible; lower the value better will be the solution.

We will design an algorithm that will search for optimal configuration that will result in minimum completion time. Since the algorithm will be heuristic in nature, the solution will be near optimal. Following techniques will be implemented:

1. Random Search
2. Hill Climbing
3. Simulated Annealing

Random search will be implemented to compare its performance with hill climbing and simulated annealing. Hill climbing or simulated annealing will be selected for further testing based on their performance and results.

3.1.2 How to distribute work packages uniformly among teams in order to avoid starvation or exhaustion?

While minimizing project completion time we have to ensure that tasks distribution among teams is uniform and teams are allocated tasks according to their size. Project managers try to ensure that teams are allocated tasks uniformly to avoid situation that might lead to starvation and exhaustion therefore we must ensure that our technique must allocate tasks uniformly.

In order to solve this problem, we enhance the functionality of our technique in order to minimize standard deviation among teams. Once a configuration with minimum completion time is obtained, we will maximize the minimum completion time and ensure that the maximum value achieved previously is kept intact. In this way the standard deviation will be minimized and tasks distribution will become more uniform.

3.1.3 How to preserve dependencies among work packages during allocation in case of dependent work packages?

Till now we have been discussing how to allocate work packages in such a way that it minimizes the project completion time. We reshuffle the allocation in an attempt to find near optimal solution assuming that work packages are independent of each other which simplifies the problem as we do not have to consider the dependency while shuffling the configuration of work packages. But in real-life projects, work packages or tasks are dependent and dependency structure must be preserved in order to complete tasks. Most of the tasks depend on each other and one task cannot start until certain task is completed and the chain goes on.

In our first phase we will combine the dependent work packages to form a set of independent work packages and test our technique on the modified data. Once we achieve minimum completion time, we will modify our algorithm so that the dependency structure is preserved and work packages are allocated in order of their dependency. For this purpose the dependency structure will be provided as input to the program alongwith other data. The dependency is stored as array-list data structure and before allocation of each work package to team its dependency structure is checked and then it

is allocated accordingly. This allows dependent work packages to be allocated to different teams but in order of their completion. Thus with incorporation of dependency preservation, this technique becomes more practically applicable and the results obtained will be more reliable and realistic.

3.1.4 How to calculate the effort team members consume during communication in order to complete allocated tasks?

In literature review we discussed the Brook's law and concept of man-month. We also discussed how the effort for communication among team members in order to complete the allocated tasks can be calculated. Till this point in our technique, we are assuming no communication among team members which is highly optimistic assumption and might lead to unrealistic results. Communication is an integral part of team effort and team members need to communicate and synchronize their efforts for smooth operation. This communication requires some time and hence effort which adds to the overall effort for completing the allocated tasks.

In order to calculate the effort for communication, we incorporate the mathematical formula (discussed in literature review) in our algorithm that will calculate the total effort which includes effort to complete task as well as effort for communication. The value of coefficient of effort MMC is provided as input. Therefore we specify value of MMC in input file. Once the value of MMC is read from the input file it is stored in the variable of program and the effort is calculated according to value of MMC. Since MMC is user-defined we can vary its value and study how it affects the project completion time.

3.1.5 How communication affects overall progress of the project?

Now that we have modified our technique to calculate the effort required for communication, we will study how communication affects the overall progress of the project.

Since the value of co-efficient of communication MMC is user-defined we can vary its value and see how it affects the completion time of the project. We can plot a graph between MMC and project completion time and see how time increases with increase in MMC. This will allow us to understand what role communication plays in the completion time of the project.

We will also test variations of Brook's law, each variation representing how team members communication among themselves. According to Brook's law, team members

have N(N-1)/2 communication links (where N = Number of team members) among them and for each link an effort is required. We will test following variations:

1. N – Linear law
2. Log(N) – Log Law

With these variations, we will study the effect of communication on project completion time by plotting a graph between MMC and completion time. The result will help us in understand for each communication strategy how communication affects the progress of the project.

3.1.6 How can we define upper bound for the amount of effort to be dedicated to communication?

Once we plot a graph between MMC and project completion time we can analyze how completion time increases with increase in MMC. With this information we can define upper bound for the amount of effort to be consumed in communication. If we are required to complete a project in X days and we plot a graph between MMC and project completion time using our technique, we can look at maximum MMC value with which the project completion time is equal to or less than X. Thus we can define an upper limit for how much effort to be consumed in communication. This can prove to be very useful information for project managers as they can have specific information regarding what level of communication must be maintained in order to complete project on time.

3.1.7 How allocation process is affected by increase in communication?

After studying the effect of communication on completion time, we will study how allocation process is affected by communication. We will study the impact of increase in communication on work package configuration; how different the allocation becomes with increase in communication. This will enable us to understand if there is any relationship between allocation process and the communication among team members. As increase in communication causes increase in overall effort to complete the project, there is a possibility that the technique might change the configuration/allocation of work packages. The results from this study will enable us to understand relationship between communication and allocation process.

3.2 Motivation for Experimentation

The questions raised above will act as motivation for experimentation. The experiments will be based on these questions and will be carried out in a way as to answer these questions. The details of experiments will be discussed in Chapter 4: Testing and Results.

4. IMPLEMENTATION

In this chapter we discuss the algorithm design and its implementation details. The programming language used is C-Sharp (C#) and the development environment is Microsoft Visual Studio Dot Net 2005.

First we look into the algorithm and then how the algorithm is implemented in programming language. After discussing the algorithm we will look at details of various functions in the code for better understanding of the implementation.

4.1 Fitness Function

Fitness function acts as evaluation criteria for the solution to determine how good the solution is. As there is no absolute or ideal solution in heuristic technique, fitness function helps in determining how good the solution is and it helps in obtaining near optimal solution.

For this problem the fitness function is defined as the **"maximum time taken by any team to complete allocated tasks"**. As multiple teams will be involved in the project working concurrently on the allocated tasks, the team which takes the maximum time will determine the project completion time. Our aim will be to minimize the maximum time taken by any team to complete the allocated tasks. Lower the time better will be the solution.

4.2 Algorithm

We will discuss the improved version of algorithm which includes preservation of dependency structure and calculation of effort for internal communication. Following explains the working of algorithm:

1. Randomly assign work packages to teams while checking the dependency
2. Calculate the time taken by each team to complete their allocated work packages
3. Store the maximum time among teams which will be the fitness value
4. Mutate the current configuration/solution
5. Calculate the fitness value
6. If the fitness value calculated in step 5 is less than fitness value calculated in step 3, it means we have obtained better solution so overwrite the fitness value stored at step 3
7. Move to step 4

8. Repeat till minimum value is achieved
9. Once the maximum value is minimized, next step is to maximize the minimum value
10. The team with maximum time (X) is excluded from this search
11. For the rest of the teams, the minimum value is maximized by mutating the WPs assignment whilst ensuring that the maximized value do not exceed the fitness value calculated in previous search (X)
12. This process is repeated for same number of iterations

Figure 5 shows control flow graph of the algorithm.

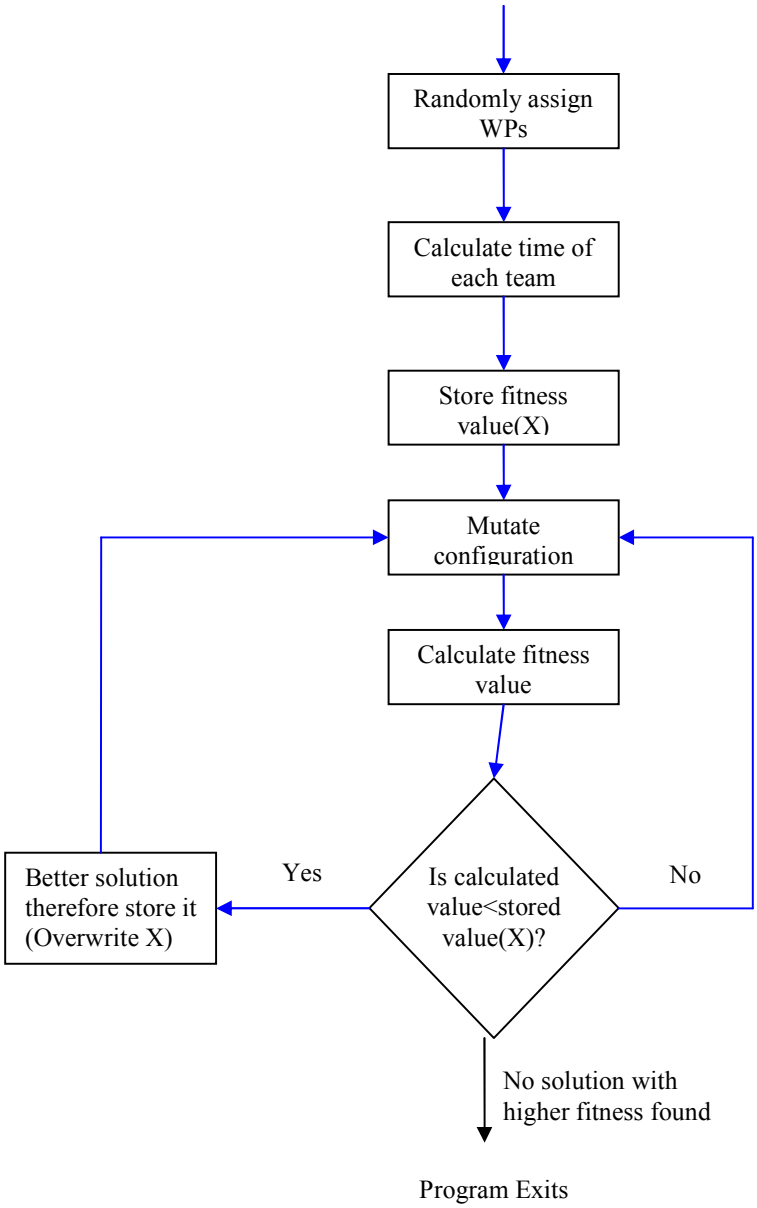


Figure 5: Control Flow Graph

4.3 Functions

Now we will look at some details of main functions in the code:

4.3.1 void AssignWorkPackage(int WorkID)

This function assigns work package to a team. It takes ID of the work package as input and has no return value. The assignment of work package to the team is random. While assigning the work packages, this function first checks the dependency structure in order to ensure that the work package must be assigned after completion of the work packages it is dependent on.

4.3.2 void CalcFitnessFunc()

This function calculates the fitness value of a solution. Once a solution is obtained after allocation of all the work packages to teams, this function is called to check the fitness of that solution. The completion time for each team is calculated. For minimizing the maximum time taken by team, the maximum time taken by any team is stored as fitness value and in case of maximizing the minimum time; the minimum value is stored as fitness value.

4.3.3 void CalculateWorkday(int WorkPaID, int TeamNo)

This function calculates the days required to complete the work package. This function also calculates the time and effort required for internal communication among the team members. For this purpose it takes ID of target work package and ID of team, to which it is allocated, as input. The time calculated by this function is sum of time required to complete the allocated work package and time consumed during communication within team members. There are 3 variations of work day calculation, each representing a strategy followed for internal communication.

- Case 1 is for the “N(N-1)/2” strategy
- Case 2 is for the “N” strategy and
- Case 3 is for the “Log (N)” strategy where N is number of team members.

This function is critical from Brook’s law point of view because it allows us to study the relationship between human resources and time by varying the values of input parameters.

If a team comprises of only 1 member then the effort required to complete the work package is returned as time to complete that work package.

4.3.4 void ClearAssignedWorkPackages()

This function simply clears the assigned work packages structure. This method is called in order to evaluate other solutions to find the most optimal solution. All attributes of work package list are set to -1 or false, depending upon the datatype, as well as list of assigned work packages of each team is cleared.

4.3.5 void DisplayConfig(string str)

This method displays the configuration of the work packages allocated to teams. It shows which work packages are allocated to which team and in what order. It also outputs the fitness value of each configuration. When we execute the code, first of all the initial configuration is dispayed. Next whenever a better solution is found with lower completion time, configuration of that solution is displayed. This also helps in analysing the behavior of the program.

4.3.6 private void DisplayLoadedData()

This method simply displays the data loaded from the input file. Information related to teams and work packages which has been read from the input file is displayed by this method.

4.3.7 public void HillClimb(int MaxTries, bool useMaxFit)

This function implements the core hill climbing algorithm. Starting from a random point this function evaluates neighbouring solutions at each point and moves toward the first better solution. The first parameter “MaxTries” specifies the number of evaluations to be performed and “useMaxFit” which is a boolean variable specifies either to maximize the minimum value or minimize the maximum value.

4.3.8 private void ReadFromFile(string filename)

This function reads all the data from input file. The input file contains following information:

- Number of teams
- Members in each team
- Number of work packages
- Effort required for each work package

- Work package dependency
- Value of MMC variable

The data is read and loaded into respective data structures for further processing.

5. TESTING AND RESULTS

Testing is carried out in two phases in order to achieve the objectives of the project and answer the questions raised in Research Questions chapter. In first phase, different heuristic techniques were tested on same dataset. The purpose was to find out which technique outperforms others in achieving the most optimal results and also compare the results obtained from technique with actual results i.e. results obtained without using the technique. The target in this case was to achieve minimum completion time for the project and the techniques tested were:

1. Random
2. Hill Climbing
3. Simulated Annealing

For further testing, hill climbing was selected to analyze other aspects of allocation process such as work packages and effort distribution.

In second phase the effect of Brook's law was analyzed by applying modified hill climbing algorithm on more detailed dataset. The modifications made in the algorithm were incorporation of communication effort calculation and work package dependency preservation. The dataset used in the testing contained additional information such as dependency structure of work packages and value of MMC (internal communication) variable.

5.1 First Phase Testing

Following are the results obtained from first phase testing:

Test Data

- Work packages = 48
- Total effort = 514 person days
- Teams = 5
- Total team member = 20
- Team1 = 3 members
- Team2 = 3 members
- Team3 = 7 members
- Team4 = 5 members
- Team5 = 2 members

Following results were obtained:

5.1.1 Completion Time

After running the program on test data, it was found that under the most optimal allocation configured by the program, the completion time is 25 days as compared to the duration of 31 days taken actually to complete the allocated tasks. Figure 6 shows behaviour of the program.

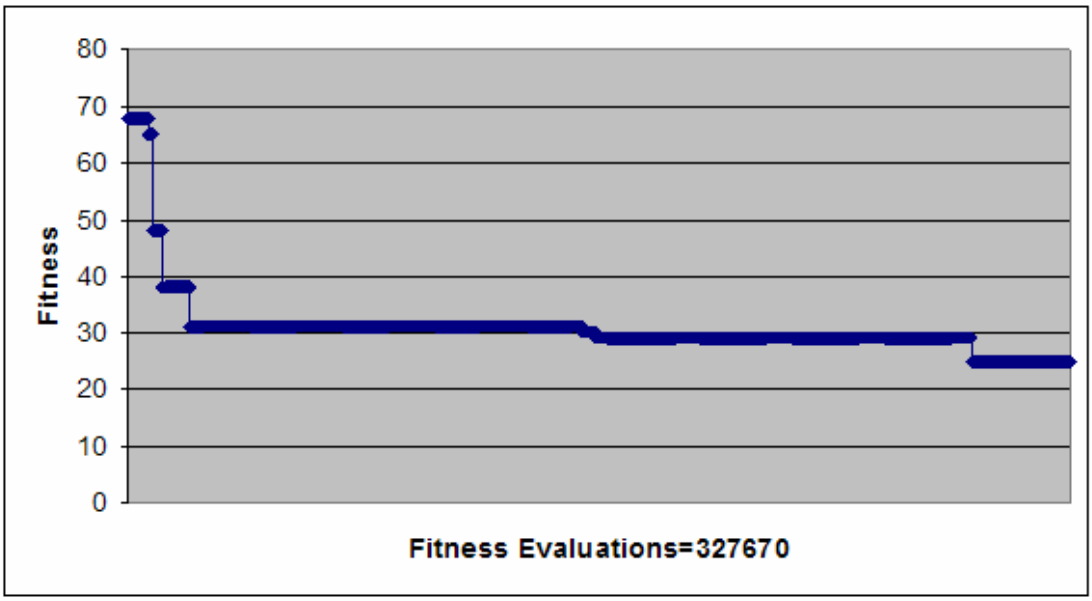


Figure 6: Completion time achieved with Hill Climbing
Hill climbing achieves minimum time of 25 days as compared to actual completion time of 31 days.

5.1.2 Work Packages Allocation

By analyzing the work packages allocation pattern of the program and comparing it with the actual allocation, we find that the automatic allocation (i.e. done by program) is more uniform and evenly distributed as compared to manual allocation (i.e. done by project manager). The program tends to minimize the standard deviation among allocated work packages in order to evenly distribute the tasks. Thus the program successfully avoids the situation of starvation or exhaustion of a team during resource allocation which is an important aspect from management point of view. Figure 7 shows work packages allocation done by program (algorithm) and by project manager (actual).

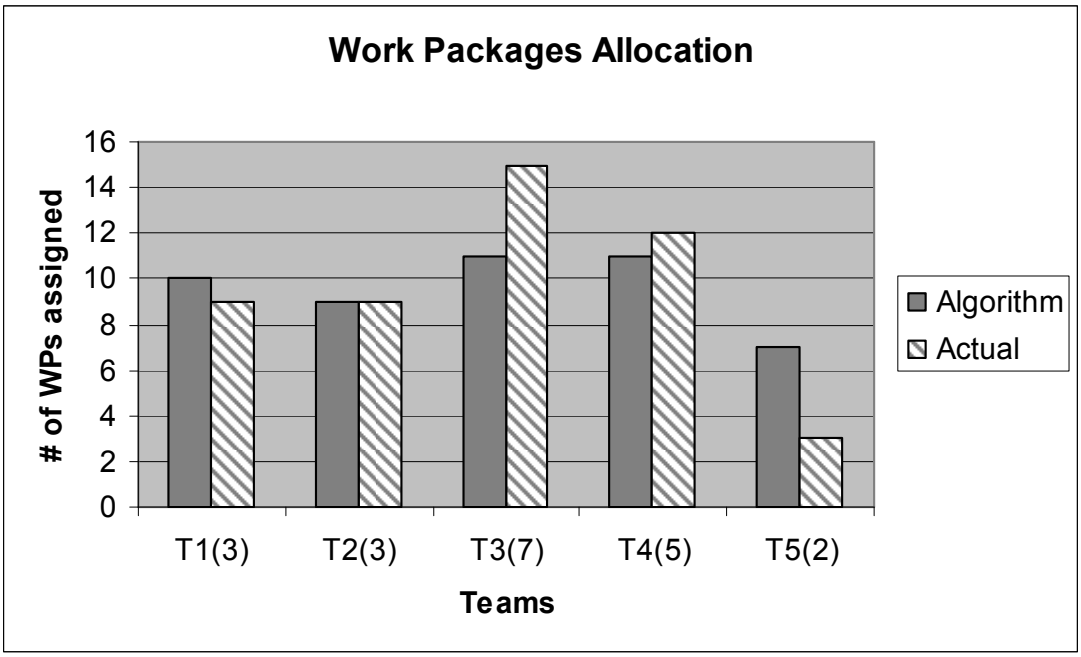


Figure 7: Actual versus Algorithm-Work Packages Allocation
The work packages are distributed more accordingly in case of algorithm while in actual case the distribution is not according to size of team

5.1.3 Effort Distribution

Since work packages vary in size and effort required, along with tasks distribution we also need to analyze how effort is distributed among teams. This will give us better idea of how much effort each team is required to do in order to complete the allocated tasks. In previous graph we observed that teams t3 and t4 are assigned equal number of work packages which might lead to the assumption that both teams are assigned equal effort to perform even though their size is different. But by looking at the graph below we observe that the effort allocated to them vary by a significant margin. This is due to the fact that team t3 having 7 members is assigned bigger work packages as compared to team t4 as a result team t3 is assigned 190 person days and t4 is assigned 120 person days.

Comparing it with actual effort distribution we can see that the automatic distribution is again more uniform and the standard deviation is less than that of actual distribution. We get the following when we compare the difference between maximum and minimum effort allocation done manually and automatically:

Algorithm: Effort of t3 (190) – Effort of t5 (45) = 145

Actual: Effort of t3 (224) – Effort of t5 (23) = 201

Clearly there is a significant difference in two values which further supports our claim that the suggested technique outperforms human effort from point of view of both minimizing completion time and uniform distribution of effort.

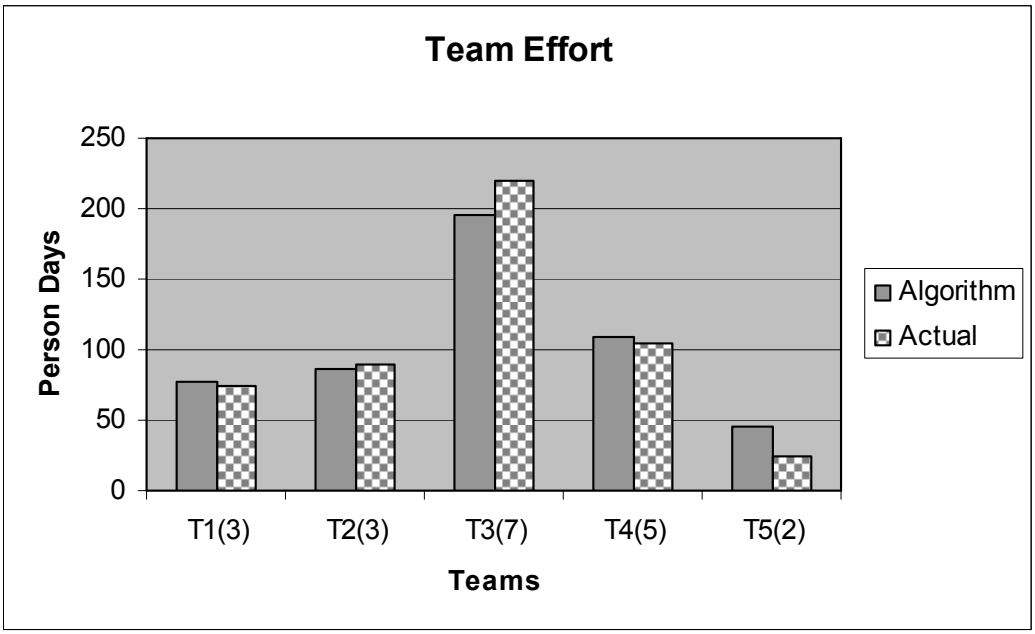


Figure 8: Actual versus Algorithm-Effort Distribution among Teams
The algorithm tends to distribute effort more uniformly as compared to human expert (actual), standard deviation in case of algorithm is less than actual.

Results obtained from other techniques are shown in figures below:

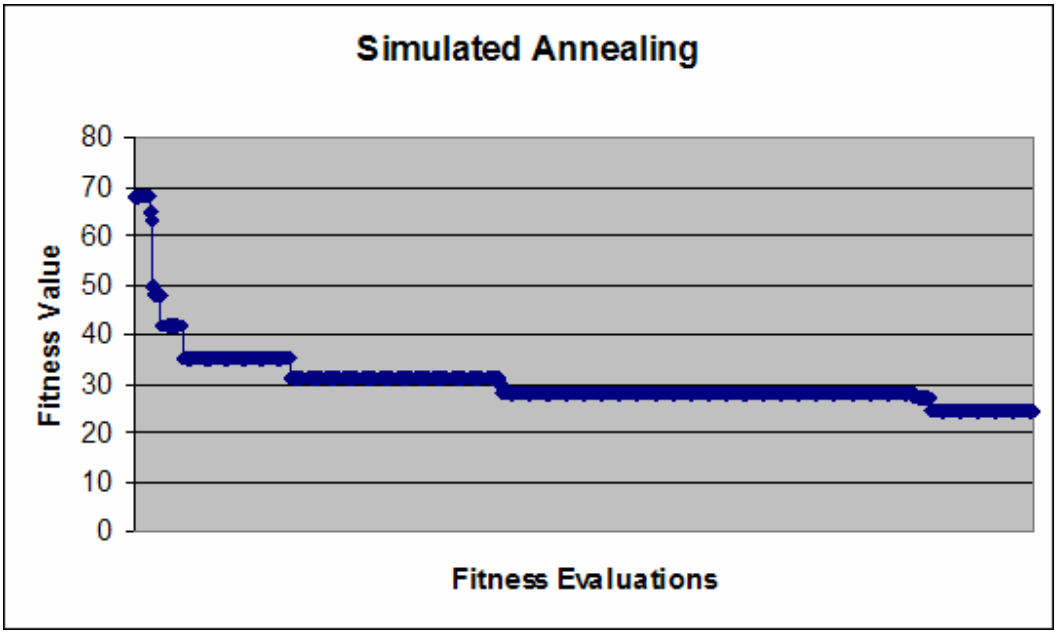


Figure 9: Completion time achieved by Simulated Annealing
Completion time achieved by simulated annealing is 24.5 days which is approximately equal to hill climbing (25 days) hence the two techniques produce almost similar results.

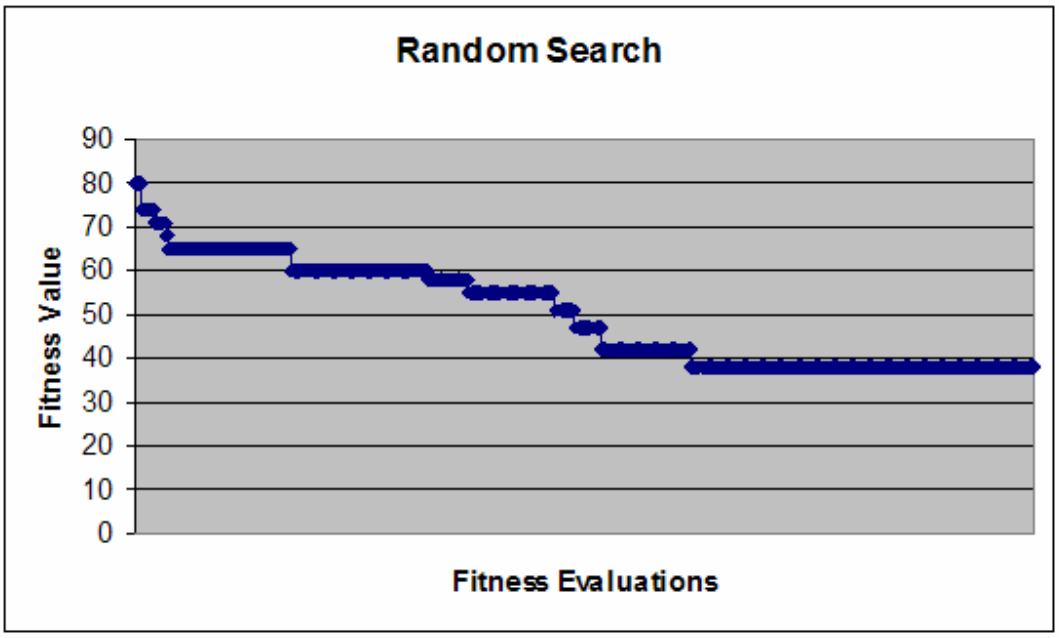


Figure 10: Completion time achieved by Random Search

Completion time achieved by random search is 38 days which is highest as compared to hill climbing and simulated annealing making it least efficient technique.

5.2 Second Phase Testing

Second phase testing is aimed at analyzing the effect of Brook's law on project completion time and resource allocation. For testing purpose different variations of the formula are tested each variation representing how teams communicate among themselves and how much effort they consume in doing so. The variations are:

- N (N-1)/2 (Square Law)
- N (Linear Law)
- Log (N) (Logarithmic Law)

Where N is number of members in a project team

The effort required to complete the task and effort for communication is calculated as:

MMn = Effort required without communication

MM = Total effort including communication:

MM=MMn + (N (N-1)/2) x MMc = N x T

T = MMn/N + ((N-1)/2) x MMc

Test data for second phase testing can be found in Appendix D on page 60 of this report.

5.2.1 MMC

MMC is variable of effort estimate which is represented as percentage of effort required to complete a particular work package. This variable determines how much effort will be dedicated to communication among team members.

For each variation, the effect of increase in MMC on project completion time and resource allocation to team members is studied, details of which are as follows:

5.2.2 Project Completion Time

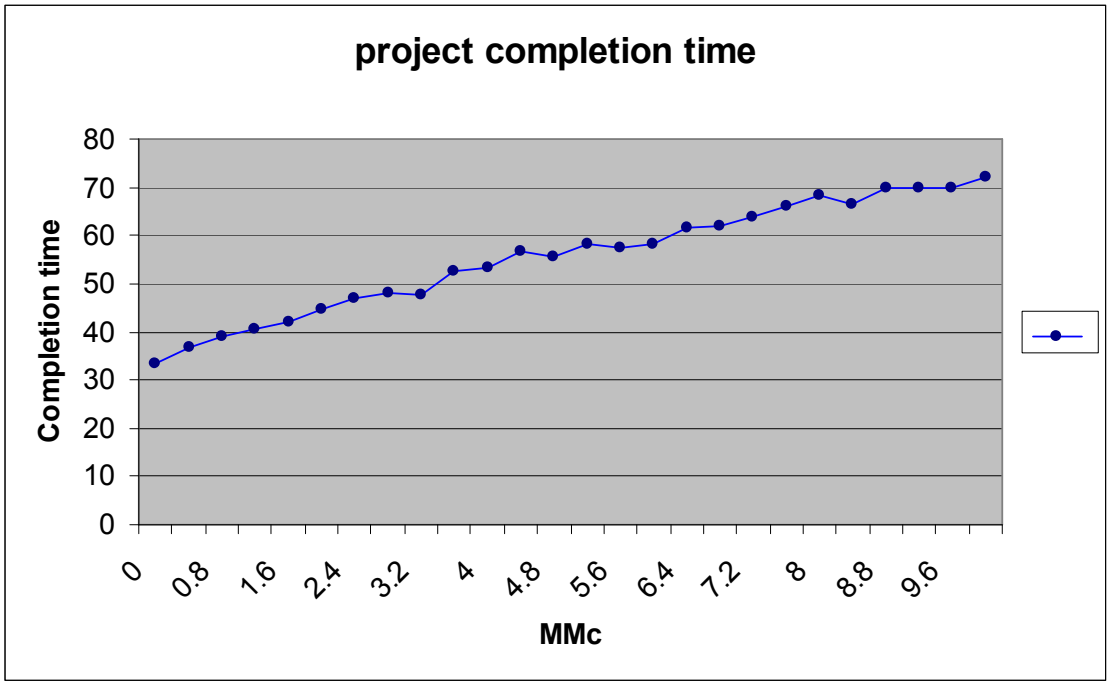


Figure 11: MMC versus Project Completion Time-N(N-1)/2 Strategy

This graph shows the effect of increasing communication on completion time of project in case of N(N-1)/2 communication links

This graph represents the most complex interaction among the team members. As the value of MMC increases, the completion time of the project increases significantly. With every 1% increase in MMC, the project is delayed by 4 to 5 days therefore the MMC should be kept minimum if the tasks require this form of communication among team members.

As more people are added in project which requires this form of communication, the project will get even more delayed due to the increase in effort required for interaction among members.

At some points in the graph, the completion time tends to reduce with increase in MMC; this is due the fact that the algorithm which is heuristic in nature finds a more optimal

solution as compared to solution with lower MMC and therefore calculates a lower value. The probability of finding such solution is very low due to which such behaviour is rarely observed in the graph.

From 0 to 10% increase in MMC the completion time increases from 33.5 days to 72.18 days causing a difference of approximately 40 days which shows how critical the MMC variable is in meeting the project deadlines. In the light of this analysis, the project managers must ensure that the communication effort must be minimized and it does not exceed a certain limit to reduce the risk of exceeding project deadlines.

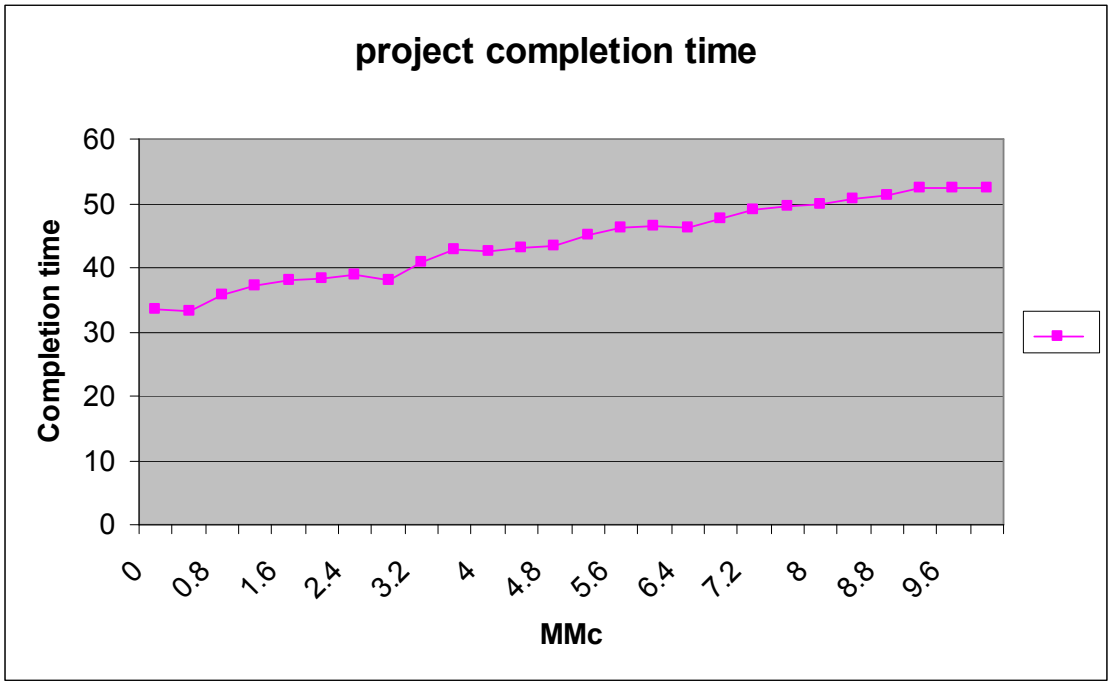


Figure 12: MMC versus Project Completion Time-N Strategy
This graph shows the effect of increasing communication on completion time of project in case of N communication links

This is slightly less strict form of communication. The delay in completion time with increase in MMC is less than the previous case which is due to the fact that in this form of communication there will be N communication links among team members as compared to N (N-1)/2 links. Due to this linear behavior the increase in MMC is causing slightly less significant difference in the project completion time.

With every 1% increase in the MMC, the completion time of the project is delayed by 2 to 3 days and the overall difference from 0 to 10% MMC is 19 days (52.4 – 33.4) which is still crucial.

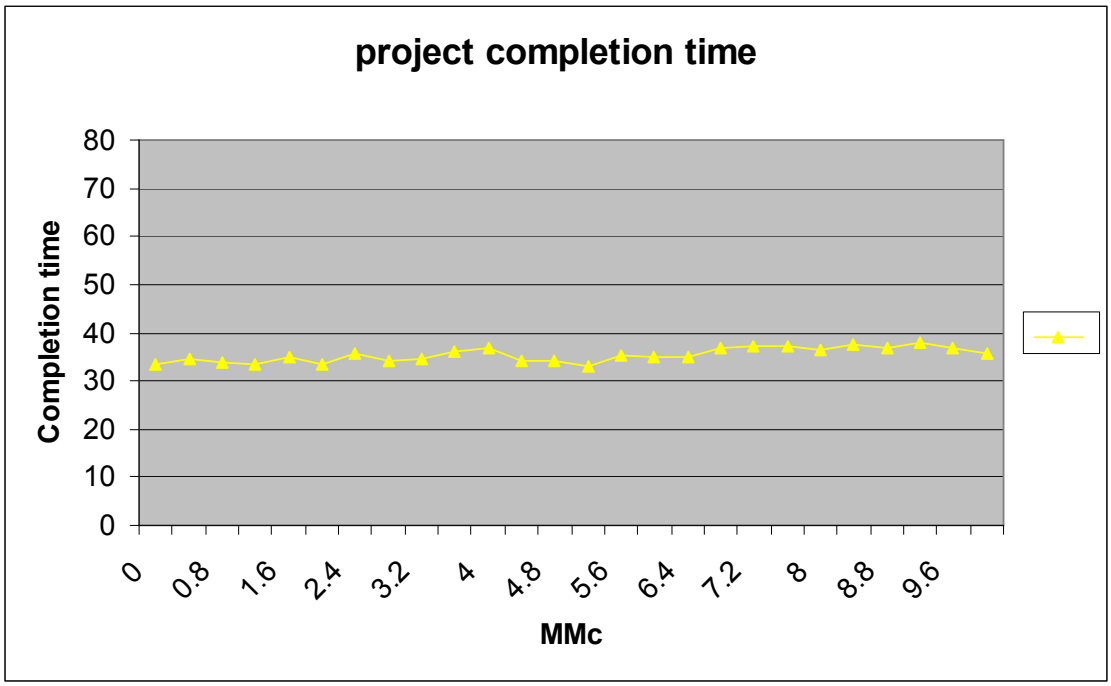


Figure 13: MMC versus Project Completion Time-Log(N) Strategy
This graph shows the effect of increasing communication on completion time of project in case of Log(N) communication links

This is the least strict form of communication in which the increase in MMC causes negligible effect on the project completion time. This is due to minimum number of communication links among team members. As the links are significantly reduced, the effort required for such communication will not have any significant impact on the project completion time.

As seen in the graph the completion time stays between 30 to 40 days for 10% increase in MMC from 0 to 10 which shows that MMC has practically no impact in this case.

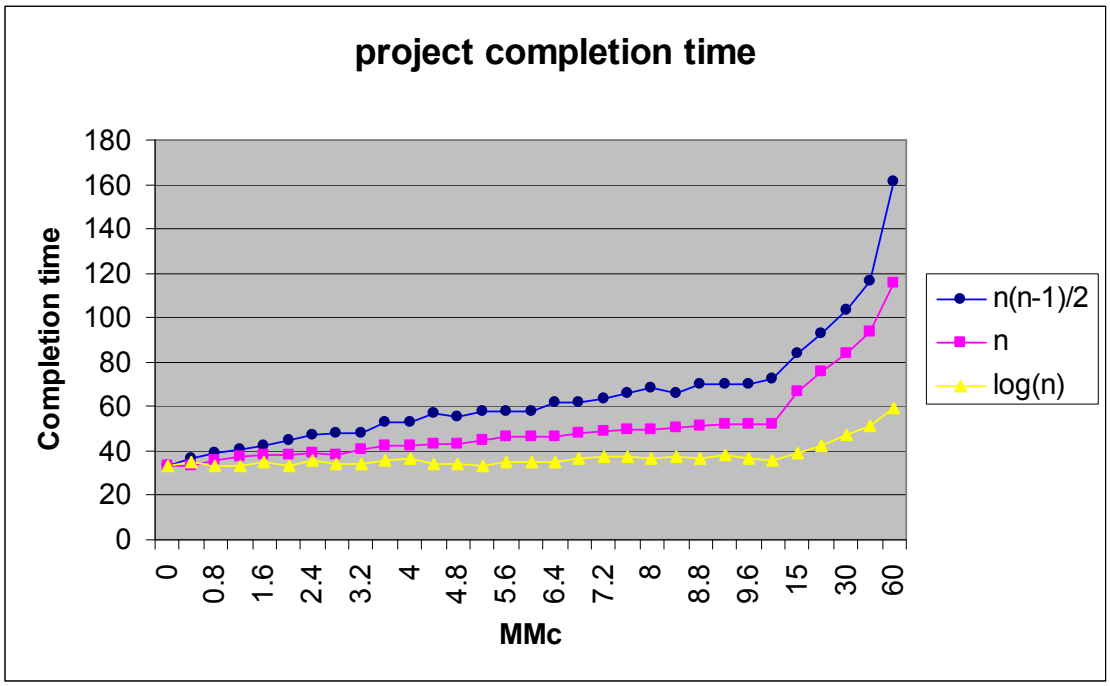


Figure 14: MMC versus Project Completion Time-Comparison

Value of MMC is increased to 60% and results of the 3 techniques are compared. We can notice an increase in yellow line as we move from 10% to 60% while blue and pink lines increase drastically.

By comparing the three strategies and extending the value of coefficient, it can be clearly observed that MMC has maximum impact on "N (N-1)/2"strategy and the completion time tends to increase significantly with increase in MMC. This increase is maximum as compared to the remaining strategies. The "N" strategy is also affected by MMC and the completion time increases by an average of 2 days for every 2% increase in MMC. Finally the "Log (N)"strategy do not seem to be affected by increase in MMC till 10% but as we increase the value of MMC with greater margin we see a slight increase in completion time of log(n) strategy.

5.2.3 Significance of the Results

These results provide an upper bound for the effort to be consumed in communication among the team members to complete the allocated task. Once effort has been estimated for all the tasks and they are allocated to the project teams, the project manager can then project the completion date of the project using the suggested technique. Then project manager can study the impact of increasing communication effort on completion time and can set an upper bound for the communication effort in order to complete project on time. With upper bound defined at the initial phase of the project, the progress can be monitored more effectively.

5.2.4 Resource Allocation/Effort Distribution

Now we will analyse how allocation of effort (person days) to teams is affected by change in communication variable MMC. Below are the results obtained from the study.

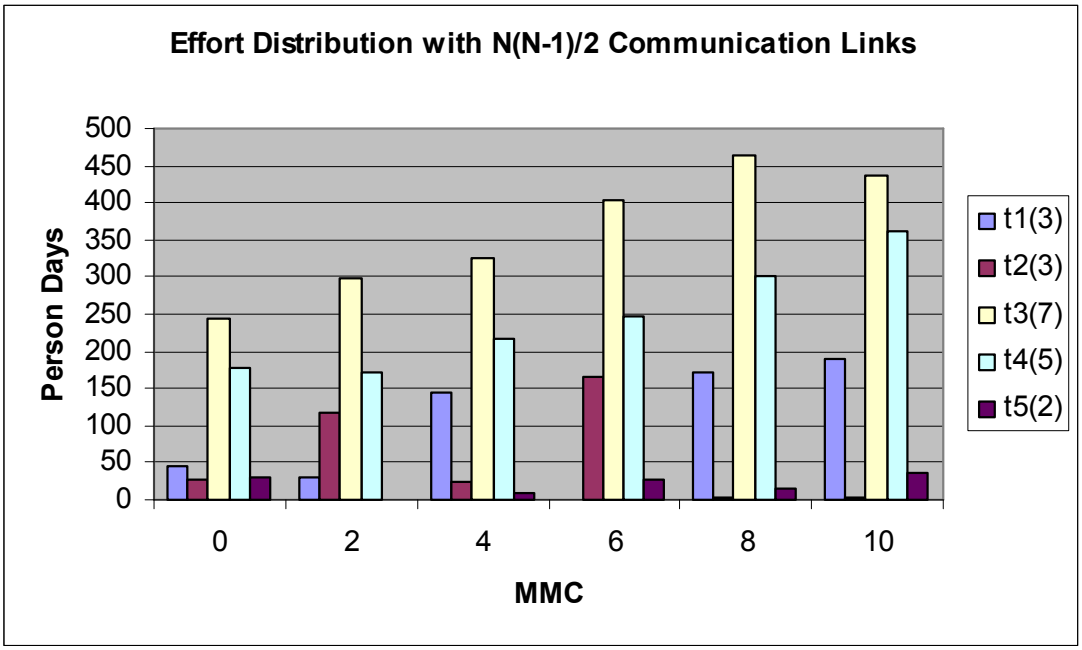


Figure 15: MMC versus Effort Distribution-N(N-1)/2 Strategy

As seen in figure 15, the algorithm tends to allocate more work packages to teams with higher number of members (t3=7 and t4=5). As a result at MMC=0, approximately 82% of effort is allocated to teams t3 and t4 (t3=46.5% and t4=34.1%). As we move along the x-axis, this ratio is maintained with the increase in MMC and at MMC=10 t3 and t4 are assigned approximately 78% of total effort (t3=42.4% and t3=35.1%). There is no significant change in effort allocation to team t5. This is due to the fact that team t5 has only 2 members therefore the algorithm tends to allocation minimum effort to that team. Allocation to t1 (3 members) and t2 (3 members) appear to be random as no regular behaviour is observed with the increase in MMC.

Due to strict dependencies among work packages, the effort distribution is not uniform among project teams. As the number of independent tasks increase, the effort distribution becomes uniform.

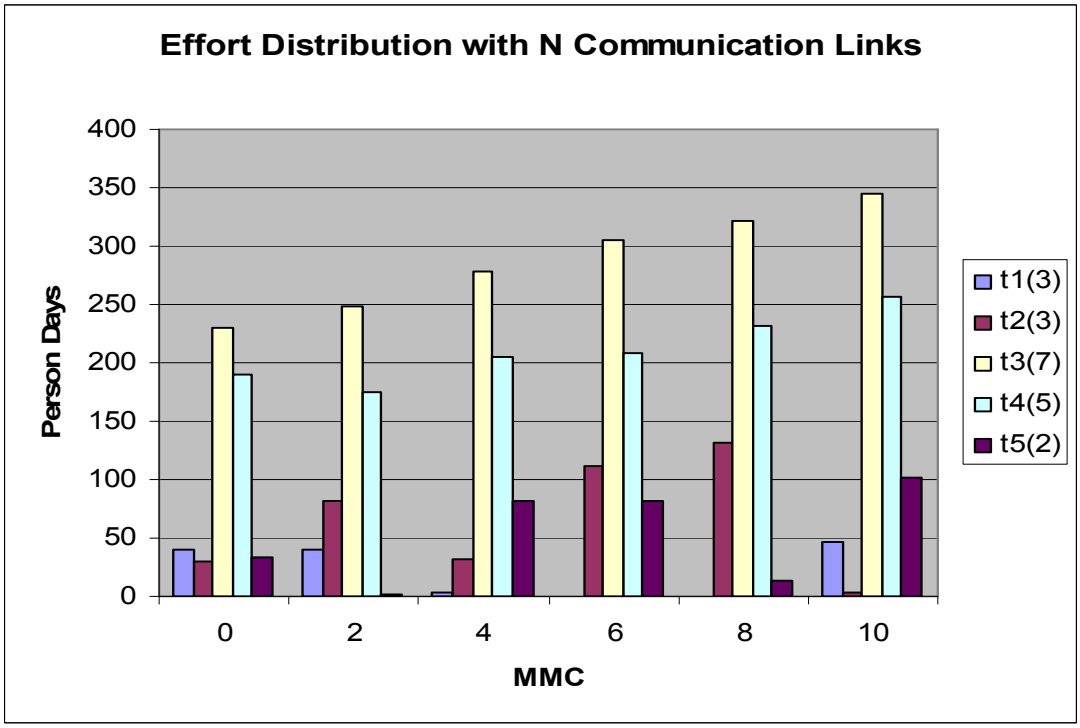


Figure 16: MMC versus Effort Distribution-N Strategy

With N communication links (figure 16), major portion of the total effort is assigned to teams t3 and t4. In this case t3 and t4 are assigned 80% of the total effort (t3=43.8% and t4=36.2%) which is approximately same as in case of N (N-1)/2 links and at MMC=10 this value is again 80% (t3=45.7% and t4=34.1%). As we move along x-axis, t3 and t4 are regularly assigned 80% or more of the effort while rest of the distribution is random. In case of t1 the effort allocation is 40, 40, 3.3, 0, 0 and 47 respectively while for t2 it is 30, 81, 31, 111, 132 and 3 respectively. Team t5 is assigned more effort as compared to previous case and at MMC=10, t5 is assigned an effort of 102 person days.

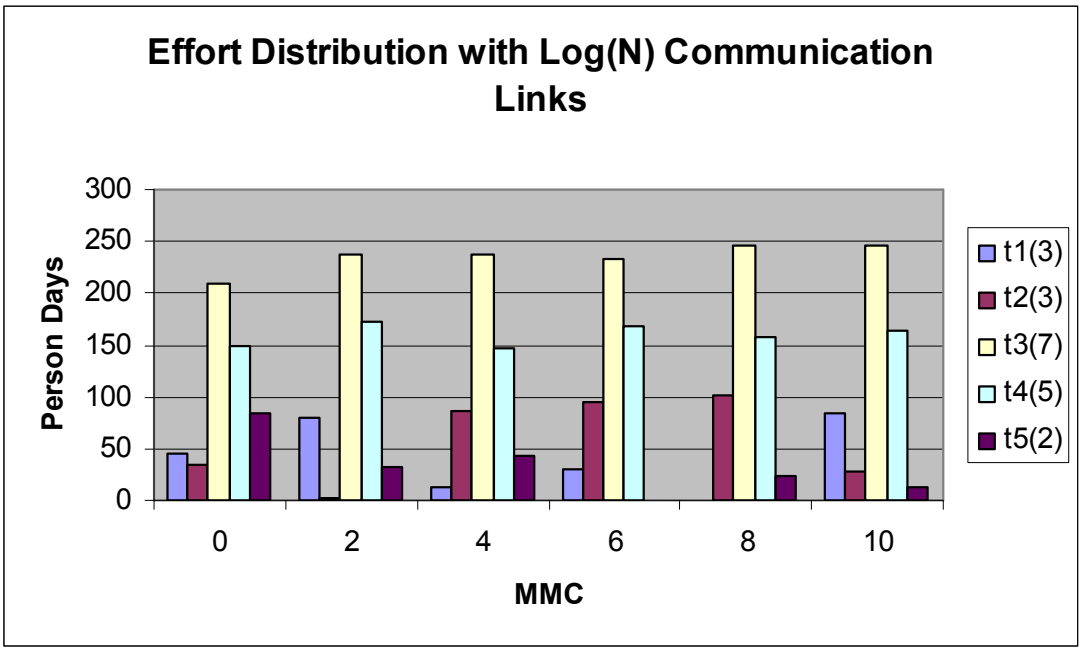


Figure 17: MMC versus Effort Distribution-Log(N) Strategy

With Log(N) communication links (figure 17), we observe the same behaviour during allocation of effort to teams. While teams t3 and t4 are assigned most of the effort, allocation to the remaining teams tend to stabilize a bit and at MMC=0 and MMC=4 we have a slightly even effort distribution.

5.2.5 Summarizing the Results

By analysing the three cases above it can be observed that size of the team plays major role in effort allocation and teams with higher number of members are assigned more effort. MMC variable do not seem to affect the allocation process as seen in all three cases.

5.2.6 Summary

The table below summarises the analysis and results discussed in this chapter. It also compares completion time and resource allocation results of the three strategies in form of graph.

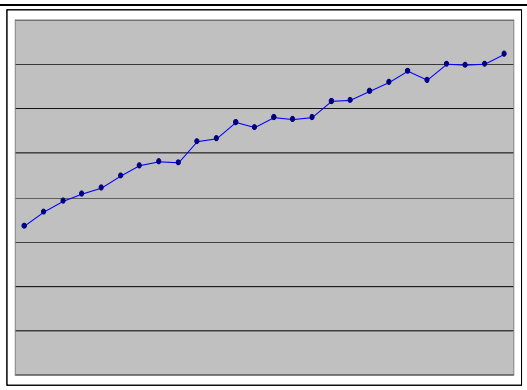
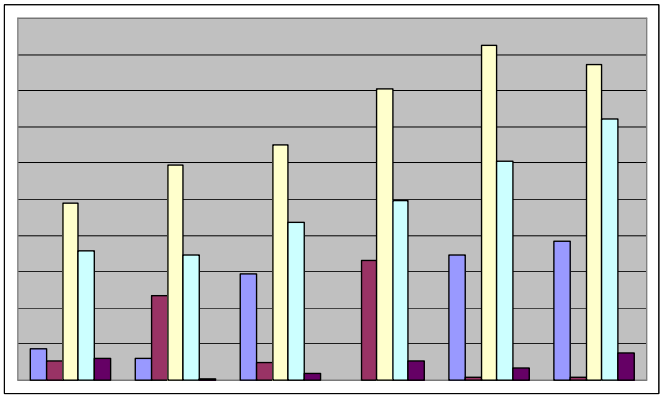
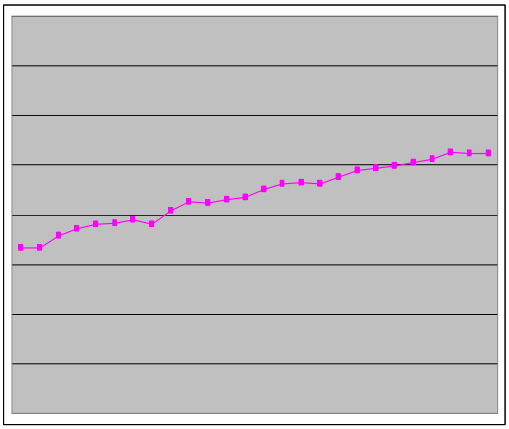
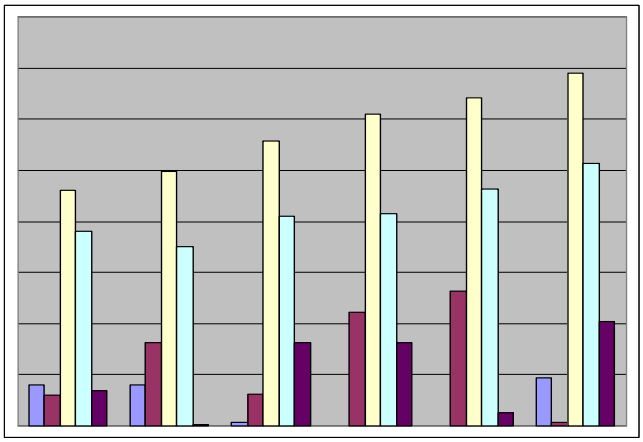
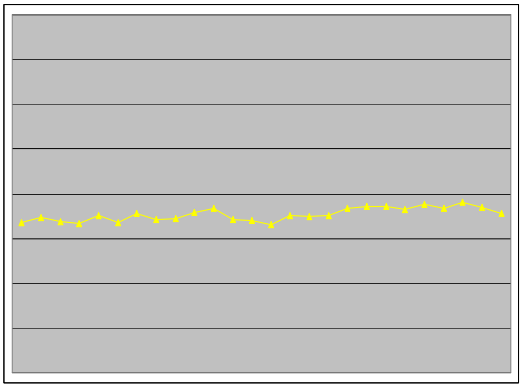
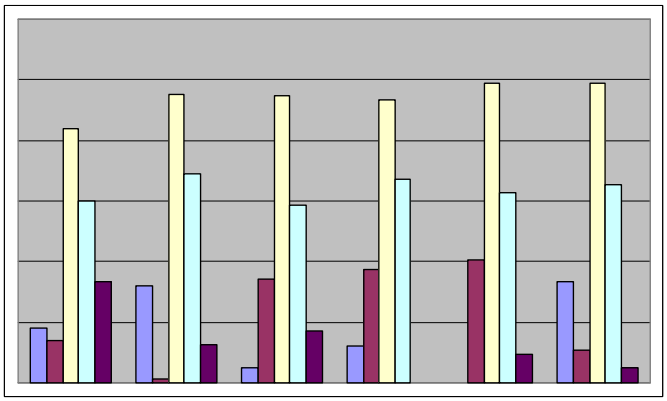
Strategy	Completion Time	Resource Allocation
N(N-1)/2		
N		
Log (N)		

Table 1: Summary of second phase testing results

6. FUTURE WORK

Some suggestions for future work in this direction can be:

6.1 Specialization

The allocation process can be improved by incorporation of factors such as:

- Type of Work Package
- Expertise of Team

With these factors incorporated in the technique, the allocation will be more effective and practically applicable as work packages will be allocated to relevant teams with required expertise level.

6.2 Data Sensitivity Test

Data sensitivity test can be performed on the data to find out which work packages are more crucial from point of view of allocation and completion. By varying the estimated effort of work packages and analyzing its effect on project completion time, a sensitivity analysis can be performed which will help project managers in determining the priority of work packages.

6.3 Determination of Ideal Communication Level

The study and analysis carried out in this project can be extended to determine the ideal communication level to be maintained among the team members throughout the project life cycle in order to complete project on time. A more detailed analysis can be performed to determine the ideal effort to be consumed during communication in order to complete the project on time. With such information, project managers will have precise value of how much effort to be consumed for communication purposes in order to complete the allocated tasks.

7. CONCLUSION

This project addressed some of the major problems of project management and attempted to solve those using heuristic techniques. The first problem addressed is of tasks allocation to project teams in such a way that the project completion time is reduced. This problem is classified as NP-Hard due to excessively large search space and for which heuristic techniques prove to be effective.

Once the problem is defined and interpreted in technical terms, next step was to design the algorithm which is heuristic in nature. For this purpose three techniques namely Random, Hill Climbing and Simulated Annealing were implemented. Study showed that Random search was least efficient while Hill Climbing and Simulated Annealing produced almost similar results. Hill Climbing was selected for further study and analysis.

Our study showed that the completion time achieved through the algorithm is significantly less than the actual completion time achieved which acted as first step towards our argument that the suggested technique outperforms human decision making. But this cannot be claimed as our suggested technique was independent of several factors that influence such decision making. Further study proved that the suggested technique tend to allocated work packages and effort more uniformly as compared to the actual allocation resulting in better utilization of human resources.

In next phase we studied the effect of Brook's law. The algorithm was modified to cater for work packages dependency and to calculate the effort required for internal communication among the team members in order to complete the allocated tasks. Three different communication schemes were studied. Once the modified algorithm was implemented and results obtained, the coefficient for internal communication (MMC) was varied to study the effect of communication on project completion time.

Results showed that "N(N-1)/2" strategy was most affected by increase in communication variable (MMC) while "Log(N)" remained least effected. This study was important from point of view of setting maximum limit for effort to be consumed in communication among project teams in order to complete allocated tasks on time.

Then we studied the effect of communication variable (MMC) on allocation process and studied how change in MMC affects the allocation of work packages to team members. Results showed that there was no significant change in allocation with increase in MMC and teams with higher number of members were regularly allocated more work packages irrespective of change in MMC.

In light of analysis and results obtained we conclude that the suggested technique can assist project managers in finding a better solution to the problems addressed in this project. While the role of human in decision making cannot be obliterated we can safely say that this technique can help managers make better decisions at initial stage of the project and also keep track of progress throughout the project lifecycle.

8. REFERENCES

[1] L. Davis. Job-shop scheduling with genetic algorithms. In International *Conference on GAs*, pages 136-140. Lawrence Erlbaum, 1985.

[2] E. Falkenauer. Genetic Algorithms and Grouping Problems. Wiley-Inter Science, Wiley – NY, 1998.

[3] E. Hart, D. Corne, and P. Ross. The state of the art in evolutionary scheduling. *Genetic Programming and Evolvable Machines*, 2004.

[4] Chapter 2: Mythical Man-Month, *The Mythical Man-Month* by Frederick Brooks.

[5] The Mythical Man-Month: Essays on Software Engineering by Frederick Brooks.

[6] Bin Packing Problem, - http://en.wikipedia.org/wiki/Bin_packing_problem.

[7] Natallia Kokash, *An Introduction to Heuristic Algorithms*, http://dit.unitn.it/~kokash/documents/Heuristic_algorithms.pdf

[8] Genetic Algorithms and Evolutionary Computation by Adam Marczyk, The Talk Origins Archive, http://www.talkorigins.org/faqs/genalg/genalg.html

[9] Chapter 1: Introduction, *An Introduction to Genetic Algorithms for Scientists and Engineers* by David A. Coley.

[10] Hill Climbing–Definition and Explanation, http://en.wikipedia.org/wiki/Hill_climbing

[11] Koza, John, Martin Keane, Matthew Streeter, William Mydlowec, Jessen Yu and Guido Lanza. Genetic Programming IV: Routine Human-Competitive Machine Intelligence. Kluwer Academic Publishers, 2003.

[12] Haupt, Randy and Sue Ellen Haupt. *Practical Genetic Algorithms*. John Wiley and Sons, 1998.

[13] Kirkpatrick, S., C.D. Gelatt and M.P. Vecchi. *Optimization by Simulated Annealing. Science*, vol. 220, p. 671-678 (1983).

9. GLOSSARY

WBS Stands for Work Breakdown Structure. It is a project management document which consists of all the tasks in the project.

WP Stands for Work Package. It is a task or finite set of tasks derived from Work Breakdown Structure.

Person-Months A unit of effort refers to a person working 8 hours a day 5 days a week. The effort required to complete a task is calculated in person days or person months.

NP-Hard NP-hard (Non-deterministic Polynomial-time hard) refers to the class containing all problems *H*, such that for every decision problem *L* in NP there exists a polynomial-time many-one reduction to *H*, written $L \leq_p H$. If *H* itself is in NP, then *H* is called NP-complete.

White Box A testing technique where data is selected based on tester's knowledge of the system. The data is chosen to exercise different paths in the code satisfying a certain criteria. The results are compared to behaviour expected from the program.

Black Box A testing technique where the user has no knowledge of the internal working of the program. The tester only knows the input and what the expected output should be.

Resource Allocation Refers to allocation of tasks to teams. In order to complete a project, several tasks are distributed among project teams.

Fitness Function A fitness function is a particular type of objective function that quantifies the optimality of a solution.

10. Appendix A

10.1 Source Code

The programming language used is C-Sharp (C#) and below is the source code of both versions of the program.

10.1.1 Version 1.0 Source Code

It is a single file named *Program.cs*.

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Text;
using System.IO;


namespace SPM
{
   public class Team
   {
      public string name;
      public int members;
      public ArrayList assignedWork;
      public int fitness;
      public Team()
      {
         members = -1;
         assignedWork = new ArrayList();
         fitness = -1;
      }
   }
   public class WorkPackage
   {
      public string name;
      public int personDays;
   }
   class Program
   {
      ArrayList TList = new ArrayList();
      ArrayList WPList = new ArrayList();


      static void Main(string[] args)
      {
         Program P = new Program();
         P.ReadFromFile("c:\\input1.txt");
         P.HillClimb(Int16.MaxValue, true);
         Console.WriteLine("\n\n Now Using Min Fitness Method.");
         P.HillClimb(Int16.MaxValue, false);
         Console.ReadLine();

      }
```

```
        // This Function will asssign all the work packages to Different teams;
        public void AssignWorkPackages_Randomly()
        {
            ClearAssignedWorkPackages();
            Random Rand = new Random(System.DateTime.Now.Millisecond);
            for (int i=0;i<WPList.Count ;i++)
                ((Team)TList[Rand.Next(0, int.MaxValue)%TList.Count]).assignedWork.Add(WPList[i]);
        }
        void ClearAssignedWorkPackages()
        {
            for (int i = 0; i < TList.Count; i++)
                ((Team)TList[i]).assignedWork.Clear();
        }
        void CalcFitnessFunc()
        {
            try
            {
                ArrayList WorkPs;
                Team CurrTeam;
                for (int i = 0; i < TList.Count; i++){
                    WorkPs = ((Team)TList[i]).assignedWork;
                    CurrTeam = ((Team)TList[i]);
                    CurrTeam.fitness = 0;
                    for (int j = 0; j < WorkPs.Count; j++)
                        CurrTeam.fitness+=((WorkPackage)WorkPs[j]).personDays /
((Team)TList[i]).members;
                }
            }
            catch(Exception e)
            {
                Console.WriteLine("Exception Check Input File: " +e.ToString());
            }
        }
        int MaxFit()
        {
            CalcFitnessFunc();
            int fitnessValue = -1;
            for (int i = 0; i < TList.Count; i++)
                if (fitnessValue < ((Team)TList[i]).fitness)
                    fitnessValue = ((Team)TList[i]).fitness;
            return fitnessValue;
        }
        int MinFit()
        {
            CalcFitnessFunc();
            int fitnessValue = int.MaxValue;
            for (int i = 0; i < TList.Count; i++)
                if (fitnessValue > ((Team)TList[i]).fitness)
                    fitnessValue = ((Team)TList[i]).fitness;
            return fitnessValue;
        }
        void DisplayConfig(string str)
        {
            Console.WriteLine(str);

            ArrayList WorkPs;
```

```
            Team CurrTeam;
            for (int i = 0; i < TList.Count; i++)
            {
                Console.WriteLine();
                WorkPs = ((Team)TList[i]).assignedWork;
                CurrTeam = ((Team)TList[i]);
                Console.Write(CurrTeam.name + " (" + CurrTeam.members+") :-");
                for (int j = 0; j < WorkPs.Count; j++)
                    Console.Write( "   "+((WorkPackage)WorkPs[j]).name +"-
"+((WorkPackage)WorkPs[j]).personDays);
            }

        }

        /// <summary>
        /// Two approaches are Supported.
        /// 1. Max Fitness Function Which Minimize the Maximum.
        /// 2. Min Fitness Function which Maximize the Minimum.
        /// </summary>
        /// <param name="MaxTries"></param>
        /// <param name="useMaxFit"></param>
        public void HillClimb (int MaxTries, bool useMaxFit)
        {
            AssignWorkPackages_Randomly();
            int CurrValue;
            if (useMaxFit)
            {
                CurrValue = MaxFit();
                // Save Curr Configuration Here , it Should be usefull
                DisplayConfig("First Time Configuration.  FitnessValue = " +CurrValue );

                for (int i = 0; i < MaxTries; i++)
                {
                    AssignWorkPackages_Randomly();
                    if (CurrValue > MaxFit())
                    {
                        CurrValue = MaxFit();
                        DisplayConfig("\nFound A Better Configuration.  FitnessValue = "+CurrValue);
                        // Save this Configuration or Display IT
                        i = -1; continue;
                    }
                }
            }
            else
            {
                CurrValue = MinFit();
                // Save Curr Configuration Here , it Should be usefull
                DisplayConfig("First Time Configuration.  FitnessValue = " + CurrValue);
                for (int i = 0; i < MaxTries; i++)
                {
                    AssignWorkPackages_Randomly();
                    if (CurrValue < MinFit())
                    {
                        DisplayConfig("\nFound A Better Configuration.  FitnessValue = " +CurrValue);
                        // Save this Configuration or Display IT
                        CurrValue = MinFit();
```

```
                i = -1; continue;
            }
        }
    }
}
private void ReadFromFile(string filename)
{

    Team t1 = new Team();
    WorkPackage w=new WorkPackage();
    StreamReader SR;
    string S;
    SR = File.OpenText(filename);
    S = SR.ReadToEnd();
    int len = S.Length;
    char[] fl=S.ToCharArray();

    string eff = "";
    string mem = "";

    int i = 0;

        while (fl[i] != 'W' && fl[i + 1] != 'P')
        {
            t1 = new Team();
            if (fl[i] == 'T' && fl[i + 1] != 'e')
            {
                t1.name = "";
                while (fl[i] != '=')
                {
                    t1.name = t1.name + fl[i].ToString();
                    i++;
                }


                if (fl[i] == '=')
                {
                    mem = "";
                    while (!(fl[i + 1] == ',' || fl[i + 1] == '\r'))
                    {

                        mem = mem + fl[i + 1].ToString();
                        i++;

                    }
                    t1.members = Convert.ToInt32(mem);
                    Console.WriteLine(t1.name + "=" + mem);

                }
                i++;
                TList.Add(t1);
            }
            i++;
        }
        //***********************************************
```

```
            while (fl[i] != 'T')
            {
                if (fl[i] == 'W' && fl[i + 1] != 'P')
                {
                    w = new WorkPackage();
                    w.name = "";
                    while (fl[i] != '=')
                    {
                        w.name = w.name + fl[i].ToString();
                        i++;
                    }
                    if (fl[i] == '=')
                    {
                        eff = "";
                        while (!(fl[i + 1] == ',' || fl[i + 1] == '\r'))
                        {

                            eff = eff + fl[i + 1].ToString();
                            i++;

                        }
                        Console.WriteLine(w.name + "=" + eff);
                        w.personDays = Convert.ToInt32(eff);
                    }
                    WPList.Add(w);
                }
                i++;
            }

        }
    }
}
```

10.1.2 Version 2.0 Source Code

The file name is *Program.cs*.

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Text;
using System.IO;
using System.Threading;

namespace fyp
{
   public class Team
   {
      public string name;
      public int members;
      public ArrayList assignedWork;
      public float fitness;
      public float MythicalManMonth;
      public Team()
      {
         members = -1;
         assignedWork = new ArrayList();
         fitness = -1;
         MythicalManMonth = 0;
      }
   }

   public class WorkPackage
   {
      public string name;
      public int personDays;
      public ArrayList Dependency;

      public short teamNo;
      public short assignedPosition;
      public double workdays;
      public double totalWorkdays;
      public bool asignedToTeam;


      public WorkPackage()
      {
         Dependency = new ArrayList();
         teamNo = -1;
         assignedPosition = -1;
         asignedToTeam = false;
         workdays = -1;
         totalWorkdays = -1;
      }
   }
   class Program
   {
      ArrayList TList = new ArrayList();
      ArrayList WPList = new ArrayList();
      ArrayList SampleSpace = new ArrayList();
```

```
string output;
double MMC;
// Set 1 for n(n-1)/2
// Set 2 for n
// Set 3 for log(n)

int MythicalManMonthtype=3;

static void Main(string[] args)
{
    Program P = new Program();


    P.ReadFromFile("c:\\input2.txt");
    P.DisplayLoadedData();
    P.HillClimb(Int16.MaxValue * 20, true);
    //Console.WriteLine("\n\nNow Using Min Fitness Method.");
    //P.HillClimb(Int16.MaxValue * 10, false);
    Console.WriteLine("\nfinished searching");
    Console.ReadLine();
    // File.WriteAllText("c:\\output.txt",P.output);
    //Console.ReadLine();
    //P.FinalMax();


}
public void FinalMax ()
{
    float CurrValue = MaxFit();
    DisplayConfig("Final Max.  FitnessValue = " + CurrValue);

    CurrValue = MinFit();
    DisplayConfig("Final Min.  FitnessValue = " + CurrValue);
}
private int[] RandomizeArray (int Size)
{
    int[] RandArray = new int[Size];
    int Swap1, Swap2,temp;

    Random Rand = new Random(System.DateTime.Now.Millisecond);

    for(int i =0;i<Size; i++)
        RandArray[i]=i;

    for (int i = 0; i < Size; i++)
    {
        Swap1 = Rand.Next(0, int.MaxValue) % Size;
        Swap2 = Rand.Next(0, int.MaxValue) % Size;

        temp = RandArray[Swap1];
        RandArray[Swap1] = RandArray[Swap2];
        RandArray[Swap2] = temp;
    }


    return RandArray;
```

```
        }
        private void DisplayLoadedData()
        {
            for (int i = 0; i < WPList.Count; i++)
            {
                Console.Write(((WorkPackage)WPList[i]).name + ":" +
((WorkPackage)WPList[i]).personDays + "Dep -> ");
                for (int j = 0; j < ((WorkPackage)WPList[i]).Dependency.Count; j++)
                {
                    Console.Write(" "+
((WorkPackage)WPList[(int)(((WorkPackage)WPList[i]).Dependency[j])]).name  );
                }
                Console.WriteLine();
            }
        }

        // This Function will assign all the work packages to Different teams;
        public void AssignWorkPackages_Randomly(Boolean ConsiderDependencies)
        {
            ClearAssignedWorkPackages();
            if(ConsiderDependencies)
            {
                // add random package selection
                int[] Packages = RandomizeArray(WPList.Count);

                for (int i = 0; i < Packages.Length; i++)
                {
                    AssignWorkPackage(Packages[i]);
                }
            }
            else
            {
                Random Rand = new Random(System.DateTime.Now.Millisecond);
                for (int i = 0; i < WPList.Count; i++)
                    ((Team)TList[Rand.Next(0, int.MaxValue) %
TList.Count]).assignedWork.Add(WPList[i]);
            }
        }
        void CalculateWorkday(int WorkPaID, int TeamNo)
        {
            // also add functionality to add mythical man month
            if (((Team)TList[TeamNo]).members > 1)
            {
                // Set 1 for n(n-1)/2
                if(MythicalManMonthtype==1)
                    ((WorkPackage)WPList[WorkPaID]).workdays =
(((double)((WorkPackage)WPList[WorkPaID]).personDays * MMC) *
(((((Team)TList[TeamNo]).members - 1) * ((Team)TList[TeamNo]).members) / 2) +
(double)(((WorkPackage)WPList[WorkPaID]).personDays));// ((Team)TList[TeamNo]).members;
                // Set 2 for n
                    if (MythicalManMonthtype == 2)
                        ((WorkPackage)WPList[WorkPaID]).workdays =
(((double)((WorkPackage)WPList[WorkPaID]).personDays * MMC) *
((Team)TList[TeamNo]).members  +
(double)(((WorkPackage)WPList[WorkPaID]).personDays));// ((Team)TList[TeamNo]).members;
                // Set 3 for log(n)
```

```
                    if (MythicalManMonthtype == 3)
                        ((WorkPackage)WPList[WorkPaID]).workdays =
(((double)((WorkPackage)WPList[WorkPaID]).personDays * MMC) *
Math.Log10(((Team)TList[TeamNo]).members)  +
(double)(((WorkPackage)WPList[WorkPaID]).personDays));// ((Team)TList[TeamNo]).members;

            }
            else
                ((WorkPackage)WPList[WorkPaID]).workdays =
(double)(((WorkPackage)WPList[WorkPaID]).personDays);
        }
        void AssignWorkPackage(int WorkID)
        {
            byte[] b;
            int RandTeamNo;
            if (((WorkPackage)WPList[WorkID]).asignedToTeam)
                return;
            if (((WorkPackage)WPList[WorkID]).Dependency.Count == 0)
            {
                b = Guid.NewGuid().ToByteArray();
                RandTeamNo = System.Convert.ToInt32(b[0]) % TList.Count;

                ((Team)TList[RandTeamNo]).assignedWork.Add(WPList[WorkID]);

                //  Calculate Workday using Mythical man month
                CalculateWorkday(WorkID,RandTeamNo);

                // also add TotalWorkdays'
                if (((Team)TList[RandTeamNo]).assignedWork.Count > 1)
                    ((WorkPackage)WPList[WorkID]).totalWorkdays =
(short)(((WorkPackage)WPList[WorkID]).workdays +
((WorkPackage)((Team)TList[RandTeamNo]).assignedWork[((Team)TList[RandTeamNo]).assign
edWork.Count - 2]).totalWorkdays);
                else
                    ((WorkPackage)WPList[WorkID]).totalWorkdays =
((WorkPackage)WPList[WorkID]).workdays;

                ((WorkPackage)WPList[WorkID]).assignedPosition =
(short)(((Team)TList[RandTeamNo]).assignedWork.Count - 1);
                ((WorkPackage)WPList[WorkID]).teamNo = (short)RandTeamNo;
                ((WorkPackage)WPList[WorkID]).asignedToTeam=true;
                return;
            }

            for (int i = 0; i < ((WorkPackage)WPList[WorkID]).Dependency.Count; i++)
                AssignWorkPackage((int)((WorkPackage)WPList[WorkID]).Dependency[i]);

            while(true)
            {
                b = Guid.NewGuid().ToByteArray();
                RandTeamNo = System.Convert.ToInt32(b[0]) % TList.Count;

                for (int j = 0; j < ((WorkPackage)WPList[WorkID]).Dependency.Count; j++ )
                {
                    if (((Team)TList[RandTeamNo]).assignedWork.Count == 0)
                        break;
```

```
                if ( ((WorkPackage)
((Team)TList[RandTeamNo]).assignedWork[((Team)TList[RandTeamNo]).assignedWork.Count-
1]).totalWorkdays <
((WorkPackage)((Team)TList[((WorkPackage)WPList[((int)((WorkPackage)WPList[WorkID]).Dep
endency[j])]).teamNo]).assignedWork[((WorkPackage)WPList[((int)((WorkPackage)WPList[WorkI
D]).Dependency[j])]).assignedPosition]).totalWorkdays)
                    break;
                if (j == ((WorkPackage)WPList[WorkID]).Dependency.Count - 1)
                {// Found add

                    ((Team)TList[RandTeamNo]).assignedWork.Add(WPList[WorkID]);

                    //  Calculate WorkDAy uing Mythical man month
                    CalculateWorkday(WorkID,RandTeamNo);

                    // also add TotalWorkdays'
                    if (((Team)TList[RandTeamNo]).assignedWork.Count > 1)
                        ((WorkPackage)WPList[WorkID]).totalWorkdays =
(short)(((WorkPackage)WPList[WorkID]).workdays +
((WorkPackage)((Team)TList[RandTeamNo]).assignedWork[((Team)TList[RandTeamNo]).assign
edWork.Count - 2]).totalWorkdays);
                    else
                        ((WorkPackage)WPList[WorkID]).totalWorkdays =
((WorkPackage)WPList[WorkID]).workdays;

                    ((WorkPackage)WPList[WorkID]).assignedPosition =
(short)(((Team)TList[RandTeamNo]).assignedWork.Count - 1);
                    ((WorkPackage)WPList[WorkID]).teamNo = (short)RandTeamNo;
                    ((WorkPackage)WPList[WorkID]).asignedToTeam = true;

                    return;
                }
            }
        }
    }

    void ClearAssignedWorkPackages()
    {
        for (int i = 0; i < TList.Count; i++)
            ((Team)TList[i]).assignedWork.Clear();
        for (int i = 0; i < WPList.Count; i++)
        {
            ((WorkPackage)WPList[i]).asignedToTeam = false;
            ((WorkPackage)WPList[i]).assignedPosition = -1;
            ((WorkPackage)WPList[i]).teamNo = -1;
            ((WorkPackage)WPList[i]).totalWorkdays = -1;
            ((WorkPackage)WPList[i]).workdays = -1;
        }
    }
    void CalcFitnessFunc()
    {
        try
        {
            ArrayList WorkPs;
            Team CurrTeam;
            for (int i = 0; i < TList.Count; i++)
```

```
            {
                WorkPs = ((Team)TList[i]).assignedWork;
                CurrTeam = ((Team)TList[i]);
                CurrTeam.fitness = 0;
                for (int j = 0; j < WorkPs.Count; j++)
                    CurrTeam.fitness += (float)(((WorkPackage)WorkPs[j]).workdays /
((Team)TList[i]).members);
            }
        }
        catch (Exception e)
        {
            Console.WriteLine("Exception Check Input File: " + e.ToString());
        }
    }
    float MaxFit()
    {
        CalcFitnessFunc();
        float fitnessValue = -1;
        for (int i = 0; i < TList.Count; i++)
            if (fitnessValue < ((Team)TList[i]).fitness)
                fitnessValue = ((Team)TList[i]).fitness;
        return fitnessValue;
    }
    float MinFit()
    {
        CalcFitnessFunc();
        float fitnessValue = int.MaxValue;
        for (int i = 0; i < TList.Count; i++)
            if (fitnessValue > ((Team)TList[i]).fitness)
                fitnessValue = ((Team)TList[i]).fitness;
        return fitnessValue;
    }
    void DisplayConfig(string str)
    {
        Console.WriteLine(str);

        ArrayList WorkPs;
        Team CurrTeam;
        for (int i = 0; i < TList.Count; i++)
        {
            Console.WriteLine();
            WorkPs = ((Team)TList[i]).assignedWork;
            CurrTeam = ((Team)TList[i]);
            output += "\n" + CurrTeam.name + " (" + CurrTeam.members + ") :-";
            Console.Write(CurrTeam.name + " (" + CurrTeam.members.ToString() + ") :-");
            for (int j = 0; j < WorkPs.Count; j++)
            {
                Console.Write("   " + ((WorkPackage)WorkPs[j]).name + "-" +
((WorkPackage)WorkPs[j]).personDays);
                output += "\n" + ((WorkPackage)WorkPs[j]).name + "-" +
((WorkPackage)WorkPs[j]).personDays.ToString();
            }
        }
          //*************************************
        /*ArrayList WorkPs;
        Team CurrTeam;
```

```
            for (int i = 0; i < TList.Count; i++)
            {
                Console.WriteLine();
                WorkPs = ((Team)TList[i]).assignedWork;
                CurrTeam = ((Team)TList[i]);
                output += "\n" + CurrTeam.name + " (" + CurrTeam.members + ") :-";
                Console.Write(CurrTeam.name + " (" + CurrTeam.members.ToString() + ") :-");
                if (WorkPs.Count != 0)
                {
                    Console.Write(((WorkPackage)WorkPs[WorkPs.Count - 1]).totalWorkdays);
                    output += ((WorkPackage)WorkPs[WorkPs.Count - 1]).totalWorkdays;
                }
            }
            */


        }

        /// <summary>
        /// Climbing The Hill.
        /// Two approaches are Supported.
        /// 1. Max Fitness Function Which Minimize the Maximum.
        /// 2. Min Fitness Function which Maximize the Minimum.
        /// </summary>
        /// <param name="MaxTries"></param>
        /// <param name="useMaxFit"></param>
        public void HillClimb(int MaxTries, bool useMaxFit)
        {
            //StreamWriter sw = new StreamWriter("c:\\testing.txt");
            if (useMaxFit) AssignWorkPackages_Randomly(true);
            float CurrValue;

            if (useMaxFit)
            {
                CurrValue = MaxFit();
                // Save Curr Configuration Here , it Should be useful
                DisplayConfig("First Time Configuration.  FitnessValue = " + CurrValue);

                for (int i = 0; i < MaxTries; i++)
                {
                    //sw.WriteLine(CurrValue); //storing fitness value in file
                    //sw.WriteLine("-" + (i + 1));
                    AssignWorkPackages_Randomly(true);
                    if (CurrValue > MaxFit())
                    {
                        CurrValue = MaxFit();
                        //sw.Write(CurrValue); //storing fitness value in file
                        //sw.WriteLine("-" + (i + 2));
                        DisplayConfig("\nFound A Better Configuration at iteration number " + i + ".
FitnessValue = " + CurrValue);
                        // Save this Configuration or Display IT
                        i = -1; continue;
                    }
                }
            }
```

```
            else
            {
                CurrValue = MinFit();
                // Save Curr Configuration Here , it Should be usefull
                DisplayConfig("First Time Configuration.  FitnessValue = " + CurrValue);
                for (int i = 0; i < MaxTries; i++)
                {
                    AssignWorkPackages_Randomly(true);
                    if (CurrValue < MinFit())
                    {
                        DisplayConfig("\nFound A Better Configuration at iteration number " + i + ".
FitnessValue = " + CurrValue);
                        // Save this Configuration or Display IT
                        CurrValue = MinFit();
                        i = -1; continue;
                    }
                }
            }
            //sw.Close();
        }

        private void ReadFromFile(string filename)
        {

            Team t1 ;
            WorkPackage w ;

            StreamReader SR;
            SR = File.OpenText(filename);
            #region MyRegion

            /*int len = S.Length;
            char[] fl = S.ToCharArray();

            string eff = "";
            string mem = "";

            int i = 0;

            while (fl[i] != 'W' && fl[i + 1] != 'P')
            {
                t1 = new Team();
                if (fl[i] == 'T' && fl[i + 1] != 'e')
                {
                    t1.name = "";
                    while (fl[i] != '=')
                    {
                        t1.name = t1.name + fl[i].ToString();
                        i++;
                    }


                    if (fl[i] == '=')
                    {
                        mem = "";
                        while (!(fl[i + 1] == ',' || fl[i + 1] == '\r'))
```

```
            {

                mem = mem + fl[i + 1].ToString();
                i++;

            }
            t1.members = Convert.ToInt32(mem);
            Console.WriteLine(t1.name + "=" + mem);

        }
        i++;
        TList.Add(t1);
    }
    i++;
}
//************************************************

while (fl[i] != 'T')
{
    if (fl[i] == 'W' && fl[i + 1] != 'P')
    {
        w = new WorkPackage();
        w.name = "";
        while (fl[i] != '=')
        {
            w.name = w.name + fl[i].ToString();
            i++;
        }
        if (fl[i] == '=')
        {
            eff = "";
            while (!(fl[i + 1] == ',' || fl[i + 1] == '\r'))
            {

                eff = eff + fl[i + 1].ToString();
                i++;

            }
            Console.WriteLine(w.name + "=" + eff);
            w.personDays = Convert.ToInt32(eff);
        }
        WPList.Add(w);
    }
    i++;
}*/
#endregion

try
{
    while (!SR.EndOfStream)
    {
        string str = SR.ReadLine();
        str = RemoveSpaces(str);
        if (str.StartsWith("Team:"))
        {
            str = str.Replace("Team:", "");
```

```
                string[] TeamsList = str.Split(",=".ToCharArray());

                for (int i = 0; i < TeamsList.Length; i++,i++)
                {
                    t1 = new Team();
                    t1.name = TeamsList[i];
                    t1.members = System.Convert.ToInt32(TeamsList[i + 1]);
                    TList.Add(t1);
                    //Console.WriteLine(t1.name + "=" + t1.members);
                }
            }
            else if (str.StartsWith("WP"))
            {
                str = str.Replace("WP:", "");
                string[] WorkList = str.Split(",=".ToCharArray());

                for (int i = 0; i < WorkList.Length; i++, i++)
                {
                    w = new WorkPackage();
                    w.name = WorkList[i];
                    w.personDays= System.Convert.ToInt32(WorkList[i + 1]);
                    WPList.Add(w);
                    //Console.WriteLine(w.name + "=" + w.personDays);
                }
            }
            else if (str.StartsWith("Dependencies"))
            {
                str = str.Replace("Dependencies:", "");
                string[] DepList = str.Split(",>".ToCharArray());

                for (int i = 0; i < DepList.Length; i++, i++)

((WorkPackage)WPList[FindWPIndex(DepList[i])]).Dependency.Add(FindWPIndex(DepList[i+1]));
            }
            else if (str.StartsWith("MMc:"))
            {
                str = str.Replace("MMc:", "");
                str = str.Replace("%", "");

                MMC = System.Convert.ToDouble(str)/100;
            }
            else
            { Console.WriteLine("Unrecognized Line, Format not correct"); }
        }
    }
    catch
    {
        Console.WriteLine("Unrecognized Line, Format not correct");
    }
}

private String RemoveSpaces(String str)
{
    for (int i = 0; i < str.Length; i++)
        if (str[i] == ' ')
        { str = str.Remove(i, 1); i--; }
```

```
            return str;
        }

        private int FindWPIndex(string name)
        {
            for (int i = 0; i < WPList.Count; i++)
                if (((WorkPackage)WPList[i]).name.Equals(name))
                    return i;
            return -1;
        }
    }
}
```

11. Appendix B

11.1 Program Output

Output of the program is the configuration of work packages allocated to teams. The program displays the configuration at each step, whenever a better solution is achieved i.e. the algorithm moves to the next step, the configuration is displayed alongwith fitness value of that configuration. This helps in analyzing the behaviour of the program.

11.1.1 Version 1.0 Output

Figure B1 shows the output produced by program version 1.0. At each step the allocation of work packages to each team is displayed alongwith fitness value of that particular configuration.

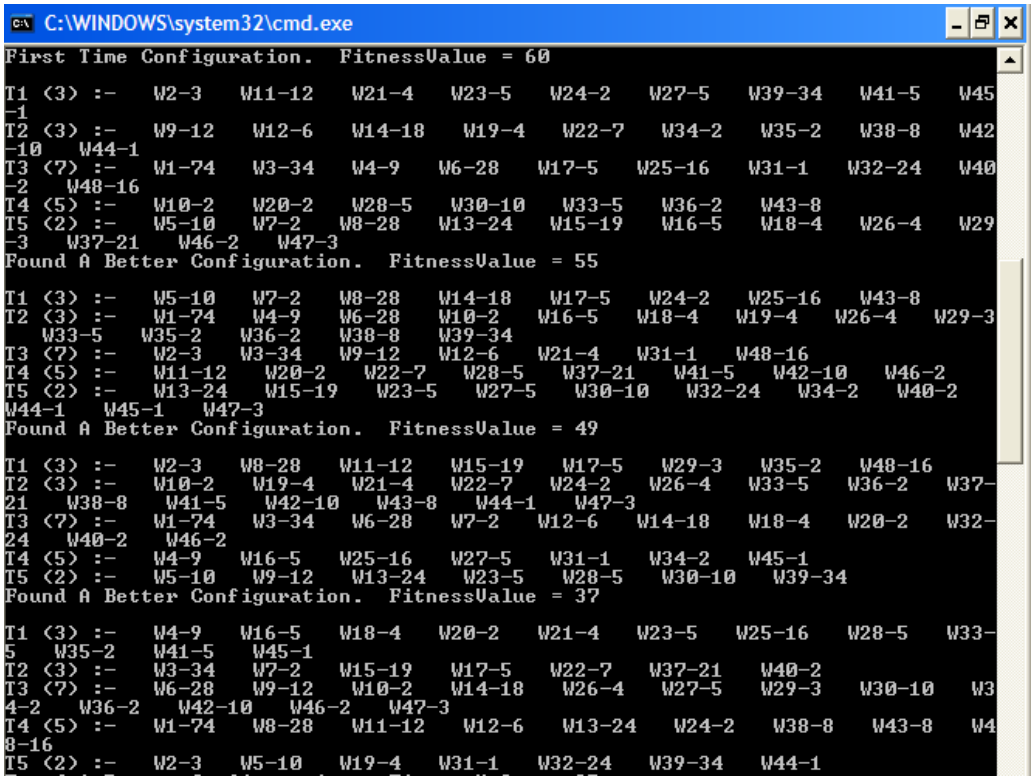


Figure B1: Program Version 1.0 Output

11.1.2 Version 2.0 Output

Figure B2 shows the output produced from program version 2.0. There is not much difference in the output as compared to version 1.0 except that the number of work packages has increased. The pattern is same with fitness value appearing at top of each configuration.

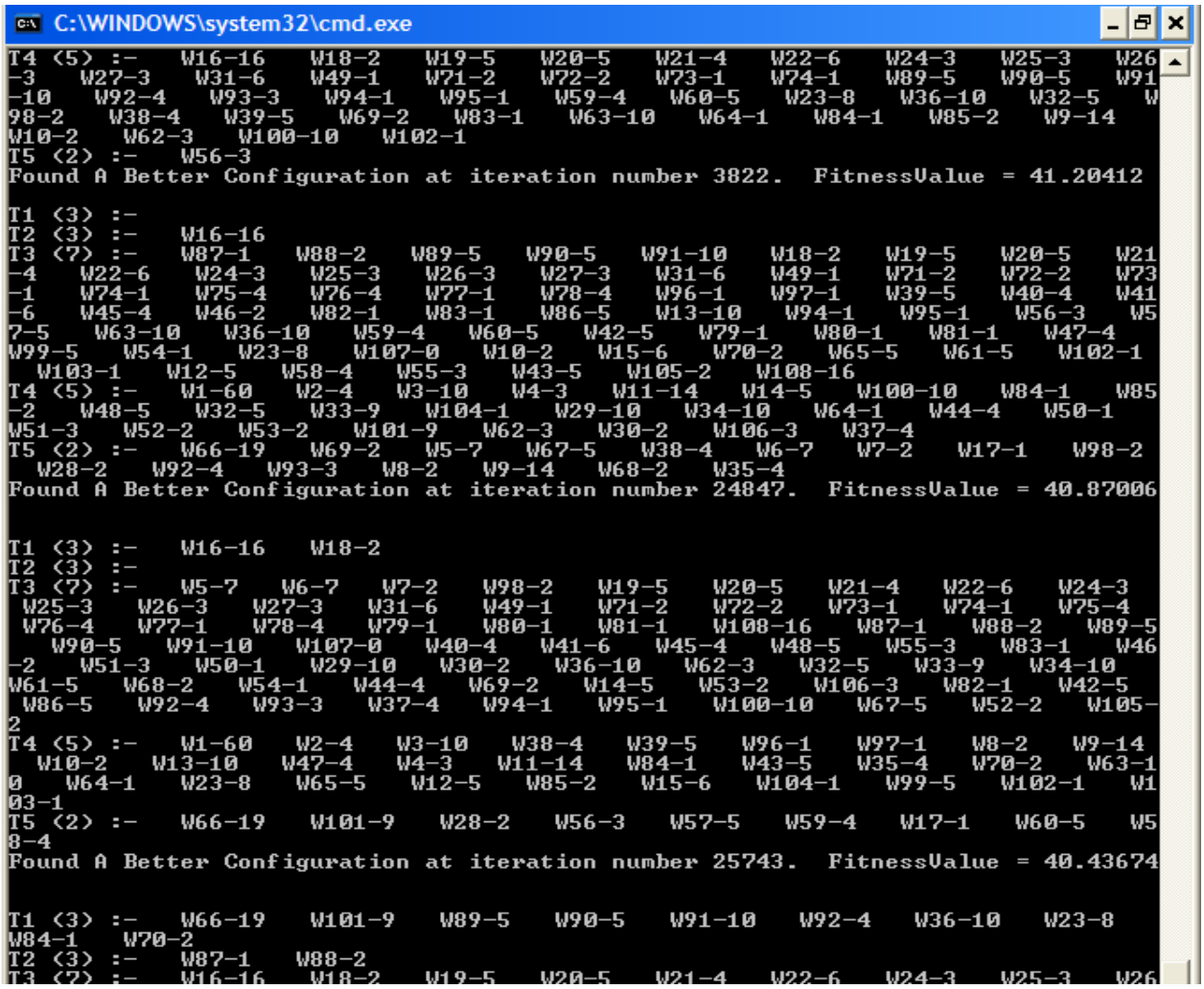


Figure B2: Program Version 2.0 Output

12. APPENDIX C

12.1 Input File Version 1.0

This input file is required to run version 1.0 of the program. This is simple form of input provided to the program. The input file consists of following in information:

- Number of Teams
- Members in Each Team
- Number of Work Packages
- Effort Required to Complete Each Work Package

Figure C1 shows the snapshot of input file version 1.0

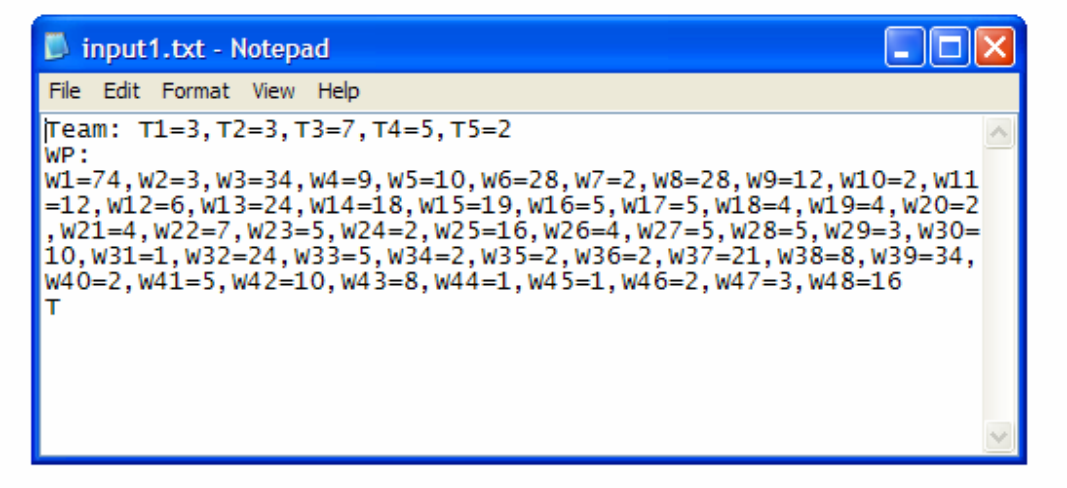


Figure C1: Input File Version 1.0 Snapshot

For simplicity purpose, the dependent work packages are combined together to form a single work package, thus resulting work packages are independent of each other. As a result we have fewer number of work packages.

12.2 Input File Version 2.0

This input file is required to execute version 2.0 of the program. This file contains additional information alongwith the information present in version 1.0. The work packages are not combined and dependency structure is also included in the file. The value of communication variable MMC is also included in the file.

Figure C2 shows the snapshot of input file version 2.0

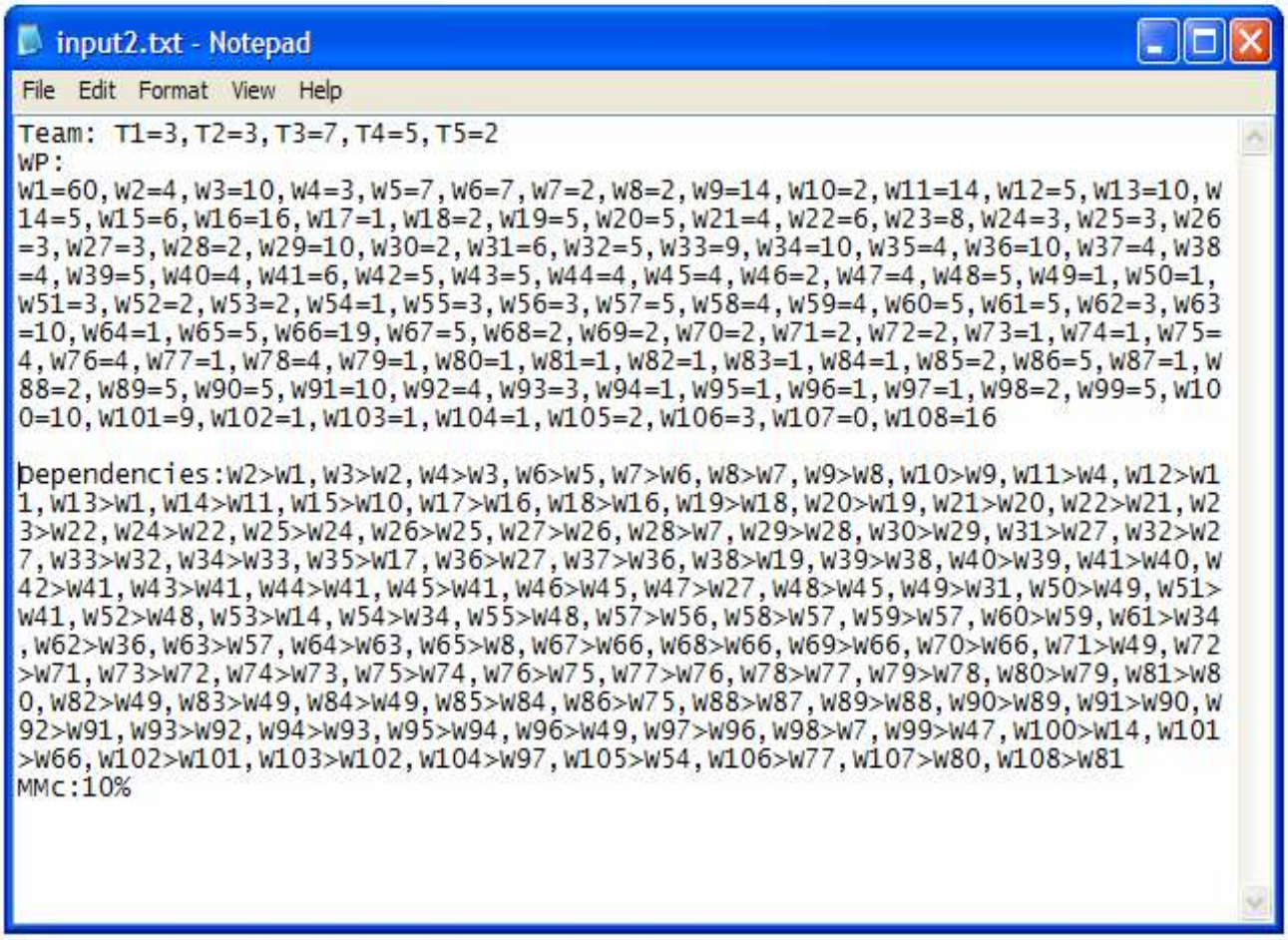


Figure C2: Input File Version 2.0 Snapshot

13. APPENDIX D

13.1 Test Data

For confidentiality purpose, the names of work packages have been changed.

- Number Teams = 5
- Number of Members in Team 1 = 3
- Number of Members in Team 2 = 3
- Number of Members in Team 3 = 7
- Number of Members in Team 4 = 5
- Number of Members in Team 5 = 2

Detail of work packages is given in table below:

Work Package	Effort Required (in person days)	Dependent On
W1	60	-
W2	4	W1
W3	10	W2
W4	3	W3
W5	7	-
W6	7	W5
W7	2	W6
W8	2	W7
W9	14	W8
W10	2	W9
W11	14	W4
W12	5	W11
W13	10	W1
W14	5	W11
W15	6	W10
W16	16	
W17	1	W16
W18	2	W16

W19	5	W18
W20	5	W19
W21	4	W20
W22	6	W21
W23	8	W22
W24	3	W22
W25	3	W24
W26	3	W25
W27	3	W26
W28	2	W7
W29	10	W28
W30	2	W29
W31	6	W27
W32	5	W27
W33	9	W32
W34	10	W33
W35	4	W17
W36	10	W27
W37	4	W36
W38	4	W19
W39	5	W38
W40	4	W39
W41	6	W40
W42	5	W41
W43	5	W41
W44	4	W41
W45	4	W41
W46	2	W45
W47	4	W27
W48	5	W45
W49	1	W31
W50	1	W49
W51	3	W41

W52	2	W48
W53	2	W14
W54	1	W34
W55	3	W48
W56	3	-
W57	5	W56
W58	4	W57
W59	4	W57
W60	5	W59
W61	5	W34
W62	3	W36
W63	10	W57
W64	1	W63
W65	5	W8
W66	19	-
W67	5	W66
W68	2	W66
W69	2	W66
W70	2	W66
W71	2	W49
W72	2	W71
W73	1	W72
W74	1	W73
W75	4	W74
W76	4	W75
W77	1	W76
W78	4	W77
W79	1	W78
W80	1	W79
W81	1	W80
W82	1	W49
W83	1	W49
W84	1	W49

W85	2	W84
W86	5	W75
W87	1	-
W88	2	W87
W89	5	W88
W90	5	W89
W91	10	W90
W92	4	W91
W93	3	W92
W94	1	W93
W95	1	W94
W96	1	W49
W97	1	W96
W98	2	W7
W99	5	W47
W100	10	W14
W101	9	W66
W102	1	W101
W103	1	W102
W104	1	W97
W105	2	W54
W106	3	W77
W107	0	W80
W108	16	W81

Figure D1: Work Packages-Effort and Dependency