Travelling Salesman Problem using Prim Algorithm in High Performance Computing

by

Wan Nurhafizah bt Wan Harun

Dissertation submitted in partial fulfillment of the
requirements for the
Bachelor of Technology (Hons)
(Infromation Communication & Technology)

JANUARY 2007

Universiti Teknologi PETRONAS
Bandar Seri Iskandar
31750 Tronoh
Perak Darul Ridzuan

CERTIFICATION OF APPROVAL

Travelling Salesman Problem using Prim Algorithm in High Performance Computing

by

Wan Nurhafizah bt Wan Harun

A project dissertation submitted to the
Information Communication Technology Program
Universiti Teknologi PETRONAS
in partial fulfillment of the requirement for the
BACHELOR OF TECHNOLOGY (Hons)
(INFORMATION COMMUNICATION TECHNOLOGY)

Approved by,

[illegible]

(Izzatdin Abdul Aziz)

UNIVERSITI TEKNOLOGI PETRONAS
TRONOH, PERAK
January 2007

CERTIFICATION OF ORIGINALITY

This is to certify that I am responsible for the work submitted in this project, that the original work is my own except as specified in the references and acknowledgements, and that the original work contained herein have not been undertaken or done by unspecified sources or persons.

[illegible]

WAN NURHAFIZAH BT WAN HARUN

ACKNOWLEDGMENTS

In the name of Allah s.w.t The Most Gracious and The Most Merciful

I am indebted and grateful to everyone who has provided both direct and indirect assistance to the completion of this project.

First and foremost, I would like to dedicate my gratitude to my loving family for their continuous support and encouragement. Mom, Dad, you are superheroes, thanks for being there for me all the time.

I am greatly indebted to my project supervisor, Mr Izzatdin Abdul Aziz, who had continuously monitored my progress during the course of the project and never shows a sign of uninvited for my frequent show up in front of his office door. Without constructive comments, advices and guidance this project will not be a success.

Last but not least, I would also like to give credit to Suriana, Nazmi, Marlinda, Hamidah and Aida for their invaluable advice, comments and support.

ABSTRACT

Thanks to the advances in wide area network technology and the low cost of computing resources, High Performance Computing came into being and currently research area. One incentive of High Performance Computing is to summative the power of widely distributed resources, and provide non-trivial services to users. To achieve this goal an efficient job scheduling algorithm system is an essential part of the High Performance Computing. This preliminary report emphasizes on the basic terms of the efficient job scheduling algorithm for traveling salesman problem in high performance computing. Job scheduling algorithm will reduce the traffic between the processors and can help improve resource utilization and quality of service. Traveling salesman problem is finding is the shortest path connecting number of locations such as cities, visited by a traveling salesman on his sales route. TSP has been used in The Two-Period Travelling Salesman Problem Applied to Milk Collection in Ireland and Usefulness of Solution Algorithms of the Travelling Salesman Problem in the typing of Biological Sequences in a Clinical Laboratory Setting.

TABLE OF CONTENTS

ABBREVIATIONS

TSP	Travelling Salesman Problem
HPC	High Performance Computing
SIMD	Single Instruction Multiple Data Stream
MIMD	Multiple Instruction-multiple Data Stream
MPI	Message Passing Interface
UTP	Universiti Teknologi Petronas

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Background of study

The popularity of the Internet and the availability of powerful computers and high speed networks as low-cost commodity components are changing the way we use computers today. These opportunities have led to the possibility of using geographically distributed resources to solve large-scale problems in science, engineering, science and commerce. This paradigm is known as grid computing.

To achieve the potential distributed resources, effective and efficient job scheduling algorithms are fundamentally important. Job scheduling has the potential of improving the application's overall performance by redistributing the workload among the processing elements.

Scheduling algorithm can be broadly divided into two classes which are time sharing and space sharing. Time sharing algorithms divide time and processor into several discrete intervals or slot. These slots are then assigned to unique jobs. Hence, several jobs at any given time can share the same compute resource. Space sharing algorithms give the requested resources to a single job until the job completes execution.

"Mathematical problems related to the traveling salesman problem were treated in the 1800s by the Irish mathematician Sir William Rowan Hamilton and by the British mathematician Thomas Penyngton Kirkman. The picture below is a photograph of

Hamilton's Icosian Game that requires players to complete tours through the 20 points using only the specified connections." [1]

Traveling Salesman Problem has many algorithm can be applied on it but the author only do the research in Prim's Algorithm and Brute Force Algorithm to be applied in collapsing number. Prim's Algorithm is an algorithm in graph theory that finds a minimum spanning tree for a connected weighted graph. Brute Force algorithm is simply to calculate the total distance for every possible route and then select the shortest one.

How Prim's Algorithm works?

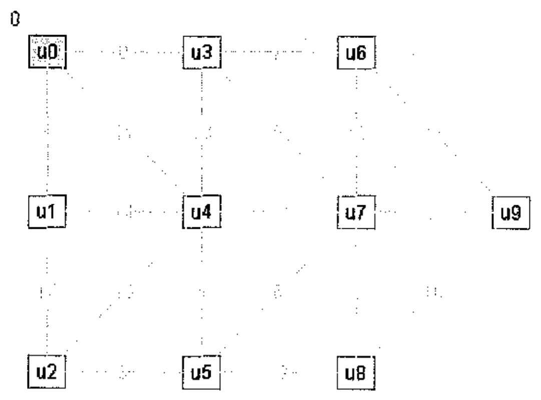


Figure 1.1 a Step 1 of 4

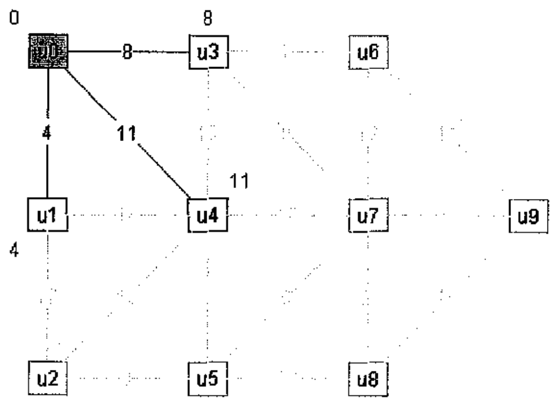


Figure 1.1 b Step 2 of 4

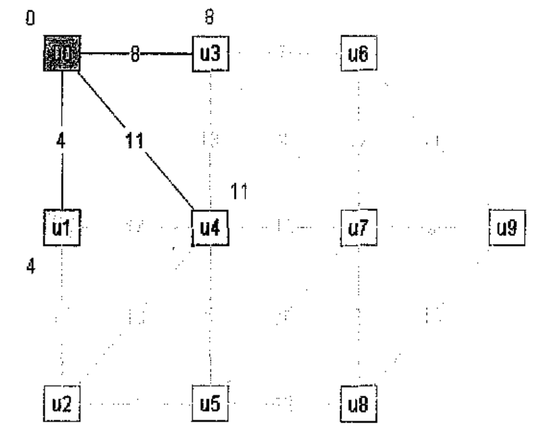


Figure 1.1 c Step 3 of 4

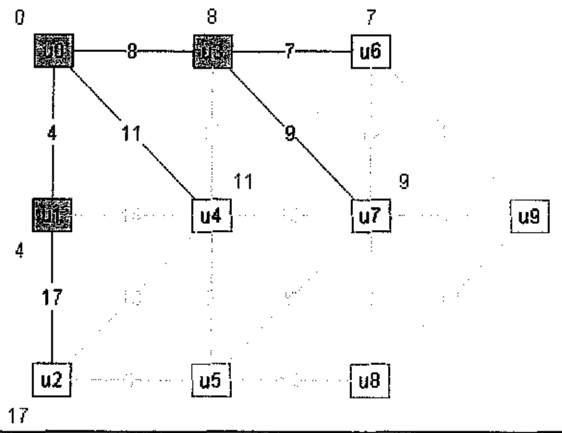


Figure 1.1 d Step 4 of 4

These figures explained how Prim's Algorithm work. There are 9 destinations (figure 1.1 a) and it start with first node which is u0. User will compare the shortest path between u1 and u3 and u4 (figure 1.1 b) and it obviously shows that u1's path is shorter than u3 so u1 will be selected (figure 1.1 c). After that user will compare between shortest path between u2, u3 and u4 and u3 is selected (figure 1.1 d). These processes will be evaluated until it reaches the last destination which is u9.

How Brute Force Algorithm work?

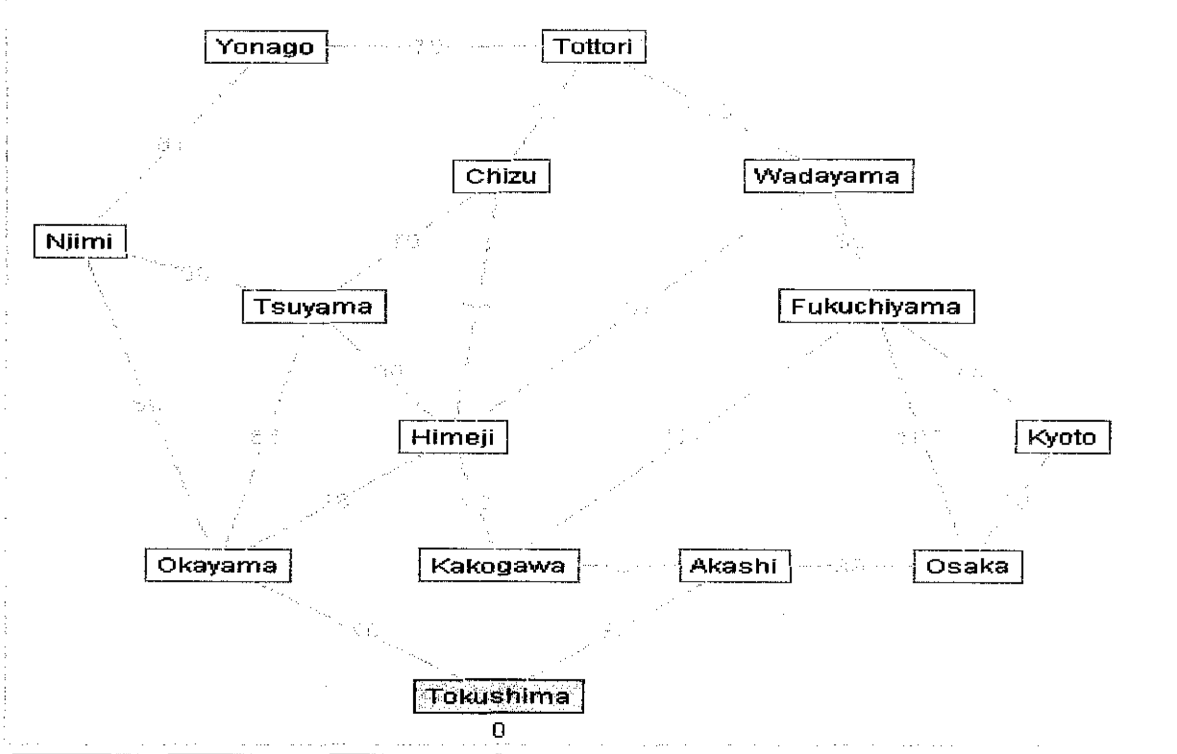


Figure 1.2 Brute Force algorithm

Figure 1.2 shows the 14 towns in Japan. The problem here is to find out which path is the shortest path from Tokushima to Tottori. Using this algorithm, the user will evaluate the entire path and calculate each route that he travels to reach Tottori. After go thru the entire path, he will compare which route is the shortest path.

1.2 Problem Statement

1.21 Problem Identification

- Current Travelling Salesman Problem is written in sequential, not in parallel.
- It has not been proven yet which algorithm would work efficiently with Travelling Salesman Problem running in High Performance Computing.

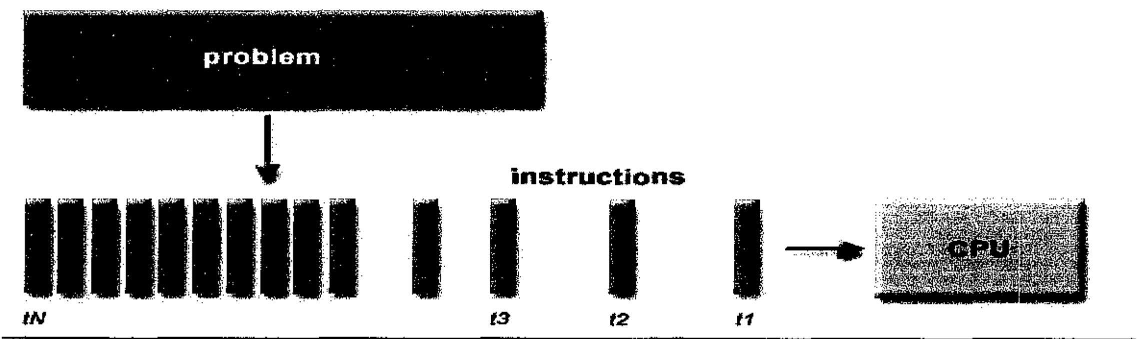


Figure 1.3 Sequential computing

Figure 1.3 shows the sequential computing where the problem will be assign to the one CPU sequentially. All the jobs must be queued while waiting to be assigning to the CPU. Only one problem can be assigned at a time.

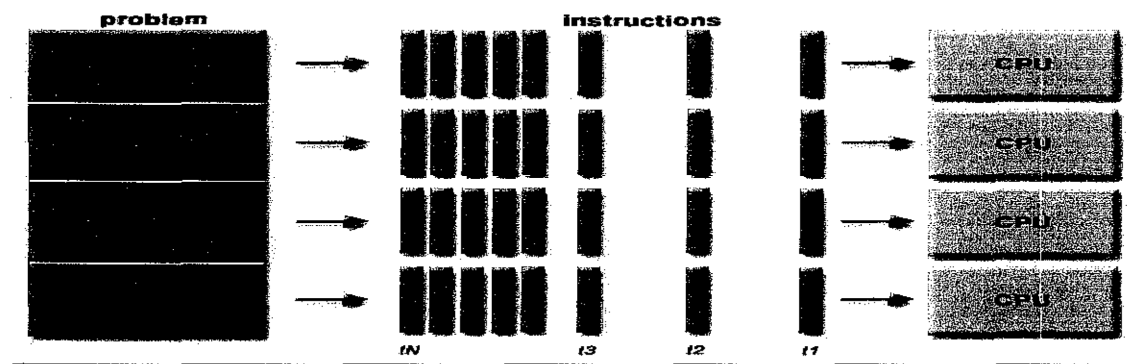


Figure 1.4 Parallel computing

Figure 1.4 explained how parallel computing works. Here, more than one problem can be located because there is more than one CPU to process the job. The process of handling the problem will be faster and more efficient compared to sequential.

1.22 Significant of the Project

This project will benefit to the people who involves in the industry, science and technology because job scheduling algorithm will help them in accelerating time to results, which allows for the provisioning of extra time and resources to solve problems that were previously unsolvable.

This is because job scheduling can reduce the traffic between the processors and can help improve resource utilization and quality of service.

1.3Objective of Study

- To conduct a study on Travelling Salesman Problem using Prim's and Brute Force Algorithm.
- To implement Travelling Salesman Problem using Message Passing Interface.
- To implement the Travelling Salesman Problem in parallel.

1.4 Scope of Study

- Rock Cluster
- Linux
- Message Passing Interface

1.5 Project Timeline

This project is going to be conducted in two semester time. This project is started with conducting studies, fact finding, and research in the first semester. The studies and research is done mostly through books from the library and also from the internet. The consultation hour with the supervisor also has helped the author a lot in understanding the core element of this project. Refer to figure 1.5a and figure 1.5b.

CHAPTER 2

LITERATURE REVIEW

2.1 Grid Computing

"Grid computing offers the power to address some of the world's most challenging problems; for example, struggles to prevent cancer and cure smallpox, to reliably predict earthquakes and global warming, and many others". [7]

Two key benefits of grid computing would enable these advances. First, grids tie together varied systems together into a mega computer, and therefore, can apply greater computational power to a task. Second, a grid virtualizes these varied resources, so that applications for the grid can be written as if for a single, local computer, vastly simplifying the development needed for such powerful applications.

"A computational grid is a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities." [8]

Grid computing is an emerging technology that enables large scale resource sharing and coordinated problem solving within distributed, coordinated group; this is the computational aspects of grids. Thus for the project, this fundamental element of grid computing is going to help a lot in breaking down the massive computational task into sub task to be performed by each slave node and at the same time controlled by the master node.

2.2 Job Scheduling Algorithm

"Today, several of these algorithms have been implemented in both commercial and open source job schedulers. Scheduling algorithms can be broadly divided into two classes: time-sharing and space-sharing. Time-sharing algorithms divide time on a processor g into several discrete intervals, or slots. These slots are then assigned to unique jobs." [2]

Several jobs at any given time can share the same compute resource. Conversely, *space-sharing* algorithms give the requested resources to a single job until the job completes execution. Most cluster schedulers operate in space-sharing mode.

2.3 Load balancing

"Dynamic load balancing aims to balance processor workloads at runtime while minimizing inter-processor communication." [3]

Dynamic load balancing can make the processor workloads balanced at runtime while the interaction inter-processor can be reduce or in other words it will reduce the traffic between the processors. Dynamic load balancing has become extremely important in several applications like scientific computing, task scheduling, sparse matrix computations, computations, parallel discrete event simulation, and data mining.

2.5 Message Passing Interface (MPI)

"*Almost everything in MPI can be summed up in the single idea of message sent – message received.*" [4]

The basic principle of MPI is that a multiple parallel processes work concurrently toward a common goal using messages as their means of communicating among each other. This is the mechanism of message-passing programming model. Message-passing is probably the most widely used parallel programming model today.

"*Message-passing model does not preclude the dynamic creation of tasks, the execution of multiple tasks per processor, or the execution of different programs by different tasks. However, in practice, most message-passing systems create a fixed number of identical tasks at program startup and do not allow tasks to be created or destroyed during program execution*". [5]

These kinds of systems are said to implement a single program multiple data (SPMD) programming model that is mentioned earlier. This is because each task executes the same program but operates on different data. Based on the reviewed that has be done on journals and book on MPI, it shows that in most MPI implementations, a fixed set of processes is created at program initialization and one process is created per processor.

However, as it is said that MPI does not preclude dynamic creation of tasks; MPI processes may also execute different programs. Thus the MPI programming model is sometimes referred to as multiple programs multiple data (MPMD) to distinguish it from the SPMD model in which every processor executes the same program.

For this project, MPI is chosen to be implemented together with job scheduling algorithm as there are seamless approach to parallel computing in C++ and MPI available through books and online resources via the internet. The author experiences in writing C++ program for some course project before is also the main reason for choosing the MPI.

2.6 Parallel Computing

"Imagine a large hall like a theater, except that the circles and galleries go right round through the space usually occupied by the stage. The walls of this chamber are painted to form a map of the globe. A myriad of computers are at work upon the weather of the part of the map where each sits, but each computer attends only to one equation or part of an equation. The work each region is coordinated by an official of higher rank. From the floor of the pit tall pillar rises to half of the height of the hall. It carries a large pulpit on its top. In this sits the man in charge of the whole theater; he is surrounded by several assistants and messengers. One of his duties is to maintain a uniform speed of progress in all parts of the globe. He is like the conductor of an orchestra in which the instruments are the slide rules and calculating machines .But instead of waving a baton he turns a beam of blue light upon those who are behindhand." [6]

This prophetic quote describes quite accurately the many hardware and software ingredients that make up a parallel computer. It refers to the term of *multiple instruction-multiple data type* and involves decomposition as the mechanism of partitioning the work load. The concepts of master node that synchronize and coordinate the process as well as load balancing are also included in the quote.

"A parallel computer is a set of processors that are able to work cooperatively to solve a computational problem."[5]

The main interest of parallel computing is that it offers the potential to concentrate computational process; whether processors, memory, or I/O bandwidth on important computational process. It is important to note that the performance of a computer depends directly on the time required to perform a basic operation and the number of these basic operations that can be performed concurrently.

The time to perform a basic operation is limited by the *clock cycle* of the processor that is the time required to perform the most primitive operation. One way to overcome the decreasing clock cycle times is by incorporating multiple computers, each with its own processor, memory and associated interconnection mechanism, which is going to be implemented in this project.

"*Parallel computing is a divide-and-conquer strategy*". [3]

The idea of partitioning the work in parallel computing is for all processors to keep busy and none remain idle. Parallel computing is a natural extension of the concept of divide and conquer; we first begin with a problem that is need to be solved, then access the available resources that can be used to solve the problem; which is the number of processors that can be used, and attempt to partition the problem into manageable pieces that can be executed concurrently by each processor.

"*A popular taxonomy for parallel computers is the description introduced by Michael Flynn in the mid-1960s of the programming model as single instruction – multiple data stream (SIMD) or multiple instruction-multiple data stream (MIMD).*"[2]

On a SIMD computer, each processor performs the same arithmetic operation or stays idle during each computer clock, as controlled by a central control unit. SIMD applies the concept of master node synchronizes and coordinates the whole process. On the other hand, on a MIMD computer, each processor can execute a separate stream of instructions on its own local data.

MIMD computers can have either shared memory or distributed memory. Distributed memory means that memory is distributed among the processors, rather than placed in a central location. Shared memory means all processors share access to a common memory, typically via a bus or hierarchy of buses.

CHAPTER 3

METHODOLOGY

This project would be completed phase by phase and for the system development, the method applied is *evolutionary approach development*. The reason the author has chosen this method is because for development of this kind of system, a flexible choice of system development methodology is very important.

Traditional approach of system development methodology that needs to get the development model mostly right early in a project is impossible for this project as this project involves more than just one area of studies. There will be a lot of things need to be considered that cannot be foreseen at the beginning of the project. Thus different conditions and techniques would be evolved during project development phase from time to time.

Evolutionary development is an iterative and incremental approach to system development. The system will be delivered incrementally over time. Evolutionary development is new to many existing professional developer, and many traditional programmers as well. Figure 1 illustrates the phases involved in evolutionary development approach.

In the first phase which is specification, the author will get as much information about this project title to meet the project specification. To get the information, the author needs to do some research about this project. The author also required to do a lot of reading in order to dig up more information.

At the development phase, the author will install the appropriate software in order to develop the project. The system that will be developed is based in the specification that the author has done before.

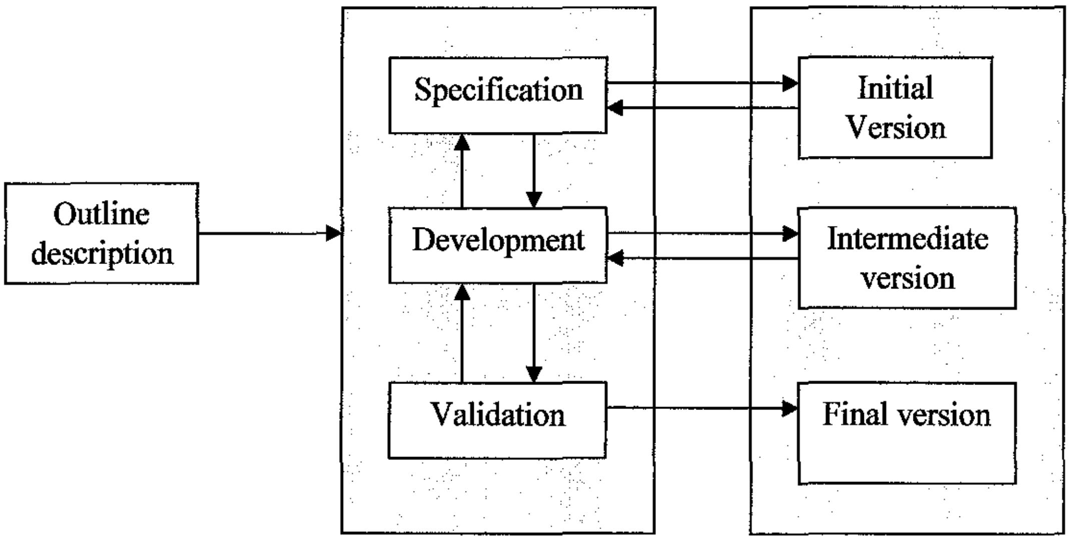


Figure 3.1: Phases involved in evolutionary development approach

At the third phase, the author will do validation for the system that has been developed to make sure whether the system meets the objective of the project or not. Testing also will be done in this phase to overcome errors that might occur during the development.

By developing the simple prototype in the beginning of project development, new constraints and requirements could be identified. Currently the author is working on the first prototype of the project that should be the end product for the first semester of developing the project.

CHAPTER 4

RESULT AND DISCUSSION

4.1 Result

	Prim's Algorithm	Brute Force Algorithm
Time	Less time	More time
Speed	Faster	Slower

Table 4.1 Comparison between Prim's and Brute Force performance

From the table above, it shows that Prim's algorithm is more efficient compared to Brute Force algorithm because it is faster and in term of speed and required less time.

4.2 Discussion

Recently, the author has done the study on Message Passing Interface. Message Passing Interface is a specification for message passing libraries, designed to be a standard for distributed memory, message passing, and parallel computing. The goal of the Message Passing Interface simply stated is to provide a widely used standard for writing message-passing programs. The interface attempts to establish a practical, portable, efficient, and flexible standard for message passing.

MPI resulted from the efforts of numerous individuals and groups over the course of a 2 year period between 1992 and 1994.

- 1980s - early 1990s: Distributed memory, parallel computing develops, as do a number of incompatible software tools for writing such programs - usually with tradeoffs between portability, performance, functionality and price. Recognition of the need for a standard arose.
- April, 1992: Workshop on Standards for Message Passing in a Distributed Memory Environment, sponsored by the Center for Research on Parallel Computing, Williamsburg, Virginia. The basic features essential to a standard message passing interface were discussed, and a working group established to continue the standardization process. Preliminary draft proposal developed subsequently.
- November 1992: - Working group meets in Minneapolis. MPI draft proposal (MPI1) from ORNL presented. Group adopts procedures and organization to form the MPI Forum. MPIF eventually comprised of about 175 individuals from 40 organizations including parallel computer vendors, software writers, academia and application scientists.
- November 1993: Supercomputing 93 conference - draft MPI standard presented. MPI-2 picked up where the first MPI specification left off, and addressed topics which go beyond the first MPI specification. The original MPI then became known as MPI-1. MPI-2 is briefly covered later. It was finalized in 1996.
- Today, MPI implementations are a combination of MPI-1 and MPI-2. A few implementations include the full functionality of both.

The reasons for using MPI:-

- Standardization - MPI is the only message passing library which can be considered a standard. It is supported on virtually all HPC platforms.
- Portability - there is no need to modify your source code when you port your application to a different platform which supports MPI.
- Performance - vendor implementations should be able to exploit native hardware features to optimize performance.
- Availability - a variety of implementations are available, both vendor and public domain.

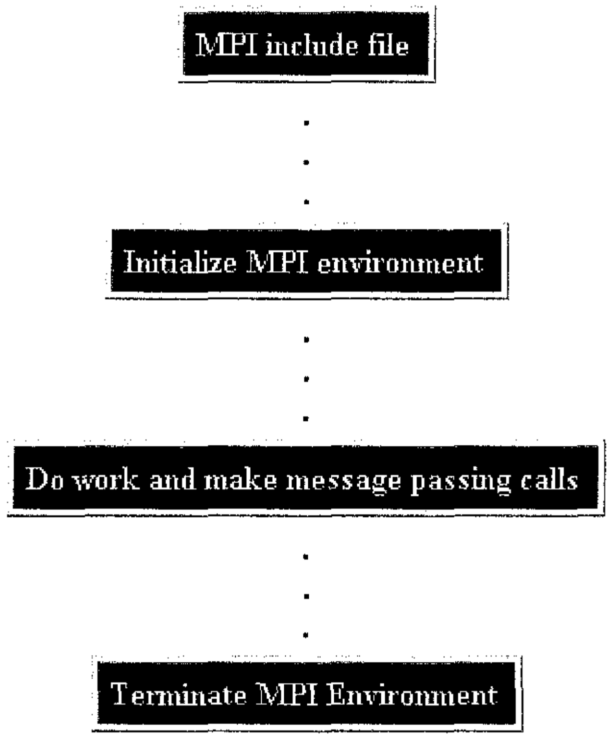

MPI include file
Initialize MPI environment
Do work and make message passing calls
Terminate MPI Environment

Figure 4.2 General MPI Program Structure

MPI uses objects called communicators and groups to define which collection of processes may communicate with each other. Most MPI routines require you to specify a communicator as an argument.

#include "mpi.h" – header file, it is required for all programs which make MPI library call.

MPI_Init - Initializes the MPI execution environment. This function must be called in every MPI program, must be called before any other MPI functions and must be called only once in an MPI program. For C programs, MPI_Init may be used to pass the command line arguments to all processes, although this is not required by the standard and is implementation dependent.

```
MPI_Init(&argc,&argv)
MPI_INIT (ierr)
```

MPI_Comm_size - Determines the number of processes in the group associated with a communicator. Generally used within the communicator MPI_COMM_WORLD to determine the number of processes being used by your application.

```
MPI_Comm_size(comm,&size)
MPI_COMM_SIZE (comm,size,ierr)
```

MPI_Comm_rank - Determines the rank of the calling process within the communicator.

```
MPI_Comm_rank(comm,&rank)
MPI_COMM_RANK (comm,rank,ierr)
```

MPI_Abort - Terminates all MPI processes associated with the communicator. In most MPI implementations it terminates ALL processes regardless of the communicator specified.

```
MPI_Abort(comm,errorcode)
MPI_ABORT (comm,errorcode,ierr)
```

MPI_Get_processor _name - Returns the processor name. Also returns the length of the name. The buffer for "name" must be at least MPI_MAX_PROCESSOR_NAME characters in size. What is returned into "name" is implementation dependent - may not be the same as the output of the "hostname" or "host" shell commands.

```
MPI_Get_processor_name(&name,&resultlength)
MPI_GET_PROCESSOR_NAME (name,resultlength,ierr)
```

MPI_Initialize - Indicates whether MPI_Init has been called - returns flag as either logical true (1) or false(0). MPI requires that MPI_Init be called once and only once by each process. This may pose a problem for modules that want to use MPI and are prepared to call MPI_Init if necessary. MPI_Initialized solves this problem.

```
MPI_Initialized(&flag)
MPI_INITIALIZED (flag,ierr)
```

MPI_Wtime - Returns an elapsed wall clock time in seconds (double precision) on the calling processor.

```
MPI_Wtime()
MPI_WTIME ()
```

MPI_Wtick - Returns the resolution in seconds (double precision) of MPI_Wtime.

```
MPI_Wtick()
MPI_WTICK ()
```

MPI_Finalize - Terminates the MPI execution environment. This function should be the last MPI routine called in every MPI program - no other MPI routines may be called after it.

```
MPI_Finalize()
MPI_FINALIZE (ierr)
```

Example of the MPI program coding:

```
#include <stdio.h>
#include <mpi.h>


int
main(int argc, char *argv[])
{
  int rank, size;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  printf("Hello world! I am %d of %d\n", rank, size);

  MPI_Finalize();

  return 0;
}
```

The author also had met Mr.Dani Adhipta to ask him to create an account for the author to get an access to the UTP HPC Cluster. The author also had downloaded putty software and installed it in order to get access to the UTP HPC Cluster.

The author did some research on the prim's algorithm pseudocode and discussed the pseudocodes with the supervisor, Mr. Izzatdin which one is the best pseudocode to be used and implement in the project. Below is the pseudocode that has been chosen.

```
Given a graph, G, with edges E of the form (v1, v2) and vertices V

dist : array of distances from the source to each vertex
edges: array indicating, for a given vertex, which vertex in the tree
it
       is closest to
i    : loop index
F    : list of finished vertices
U    : list or heap unfinished vertices

/* Initialization: set every distance to INFINITY until we discover a
way to
link a vertex to the spanning tree */
for i = 0 to |V| - 1
```

```
    dist[i] = INFINITY
    edge[i] = NULL
end

pick a vertex, s, to be the seed for the minimum spanning tree

/* Since no edge is needed to add s to the minimum spanning tree, its
distance
from the tree is 0 */
dist[s] = 0


while(F is missing a vertex)
   pick the vertex, v, in U with the shortest edge to the group of
vertices in
       the spanning tree add v to F

   /* this loop looks through every neighbor of v and checks to see if
that
    * neighbour could reach the minimum spanning tree more cheaply
through v
    * than by linking through a previous vertex */
   for each edge of v, (v1, v2)
        if(length(v1, v2) < dist[v2])
            dist[v2] = length(v1, v2)
            edges[v2] = v1
            possibly update U, depending on implementation
        end if
    end for
end while
```

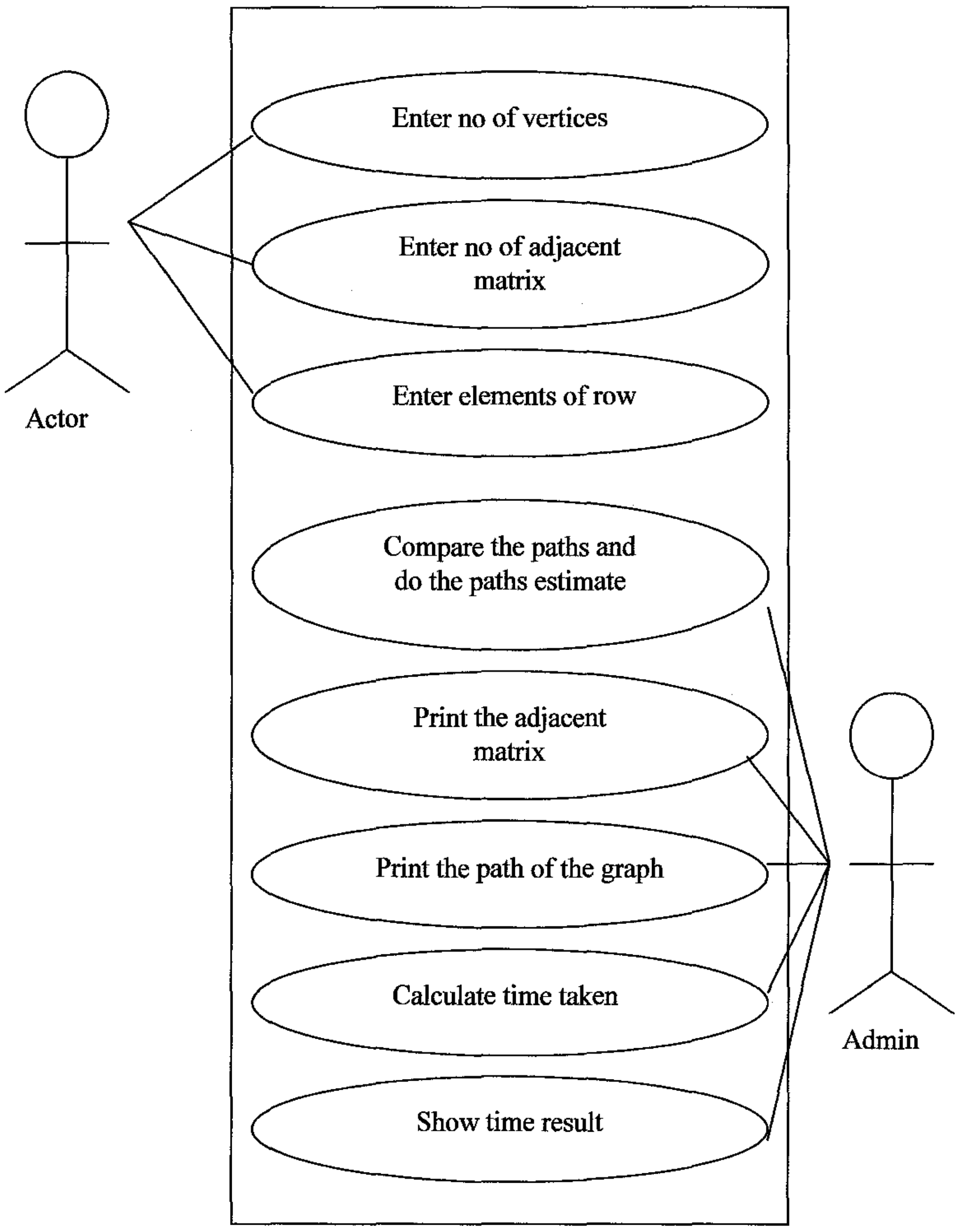

Enter no of vertices
Enter no of adjacent matrix
Enter elements of row
Actor
Compare the paths and do the paths estimate
Print the adjacent matrix
Print the path of the graph
Calculate time taken
Show time result
Admin

Figure 4.3 UML diagram

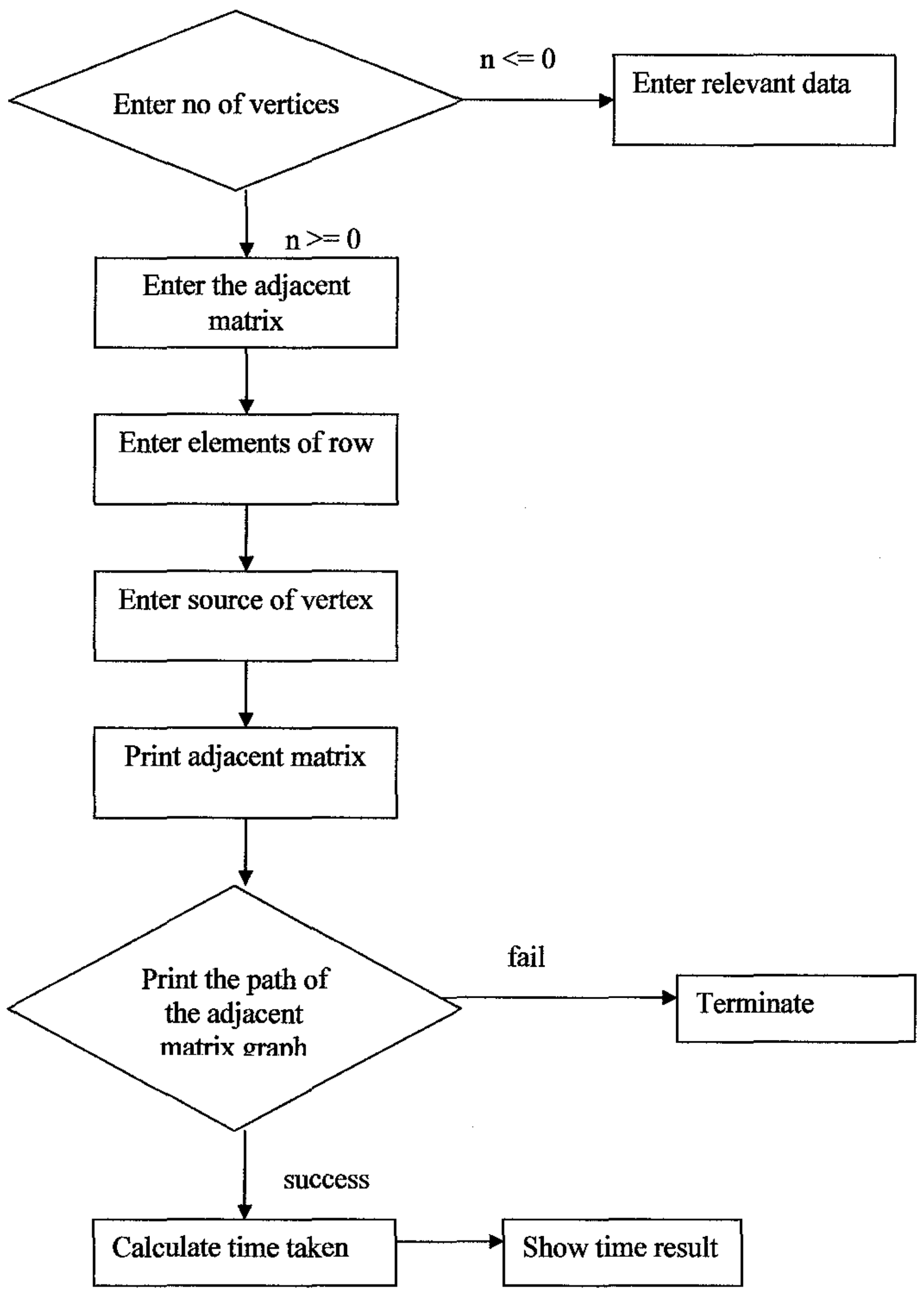


Figure 4.4 Data flow diagram

CHAPTER 5:

CONCLUSION AND RECOMMENDATION

5.1 Conclusion

The idea of this project is to parallelize the Travelling Salesman Problem using Prim algorithm instead the current TSP is in sequential. Apart from taking advantage of merging grid architecture of grid computing technology, the implementation of grid computing in this project is to catch up with the new technology in the IT field that is considered as a major paradigm shift in distributed computing principles. As for the author, parallel processing of Travelling Salesman Problem using Prim algorithm using grid computing is not an easy project. However, with all the resources available, the help of the supervisor and the right approach and methods that are going to be used, this project should achieve all the project objectives and feasible within the time frame given.

5.2 Recommendation

The future work recommended for this project will be to finish the parallelizing this system. It also recommended comparing all the algorithms that can be fit with TSP and come out with the best algorithm. From this result, the author will do the parallelizing part and most probably the result will be the best solution to find the TSP best algorithm for TSP.

REFERENCES

[1] Travelling Salesman Problem History by Mr.William Cook, an overview available at: http://www.tsp.gatech.edu/history/index.html, last accessed April 6, 2007

[2] Dell Power Solution Journal; Job Scheduling in HPC Cluster, by Saeed Iqbal, Rinku Gupta and Yung-Ching Fang, February 2005.

[3] Portable Parallel Programming for the Dynamic Load Balancing of Unstructured Grid Applications, by Rupak Biswas, Sajal K. Das, Daniel Harvey, and Leonid Oliker, November 2004.

[4] RSA and Public-key Cryptography by Richard A. Mollin, Chapman & Hall/CRC (2000)

[5] Parallel Computers and Computation by Ian Foster, an overview available at: www-unix.mcs.anl.gov/dbpp/text/node6.html

[6] Lewis F.Richardson, "Weather Prediction by Numerical Process (1922)

[7] IBM Systems Journal; Towards an Information Infrastructure for the Grid by S. Bourbonnais, V. M. Gogate, L. M. Haas, R. W. Horman, S. Malaika, I. Narang, and V. Raman, Volume 43, Number 4, 2004.

[8] IBM Systems Journal; Evolution of Grid Computing Architecture and Grid Adoption Models, by J. Joseph, M. Ernest, and C. Fellenstein Volume 43, Number 4, 2004.

No	Detail/week	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	Selection of project topic	▨	▨												
2	Preliminary research work		▨	▨	▨										
3	Submission of preliminary report				●										
4	Weekly report	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨
5	Project Research			▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	▨	
6	Submission of interim report													●	
7	Oral presentation														●

Table 1.5a Milestone for FYP a

No	Detail/week	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	Project work continue	▨	▨	▨											
2	Submission progress report 1				●										
3	Project work continue					▨	▨	▨							
4	Submission progress report 2								●						
5	Seminar									▨	▨				
6	Project work continue									▨	▨	▨	▨	▨	
7	Pre-EDX										●				
8	Submission of final report												●		
9	Oral presentation													●	
10	Submission of project dissertation														●

Table 1.5b Milestone for FYP b

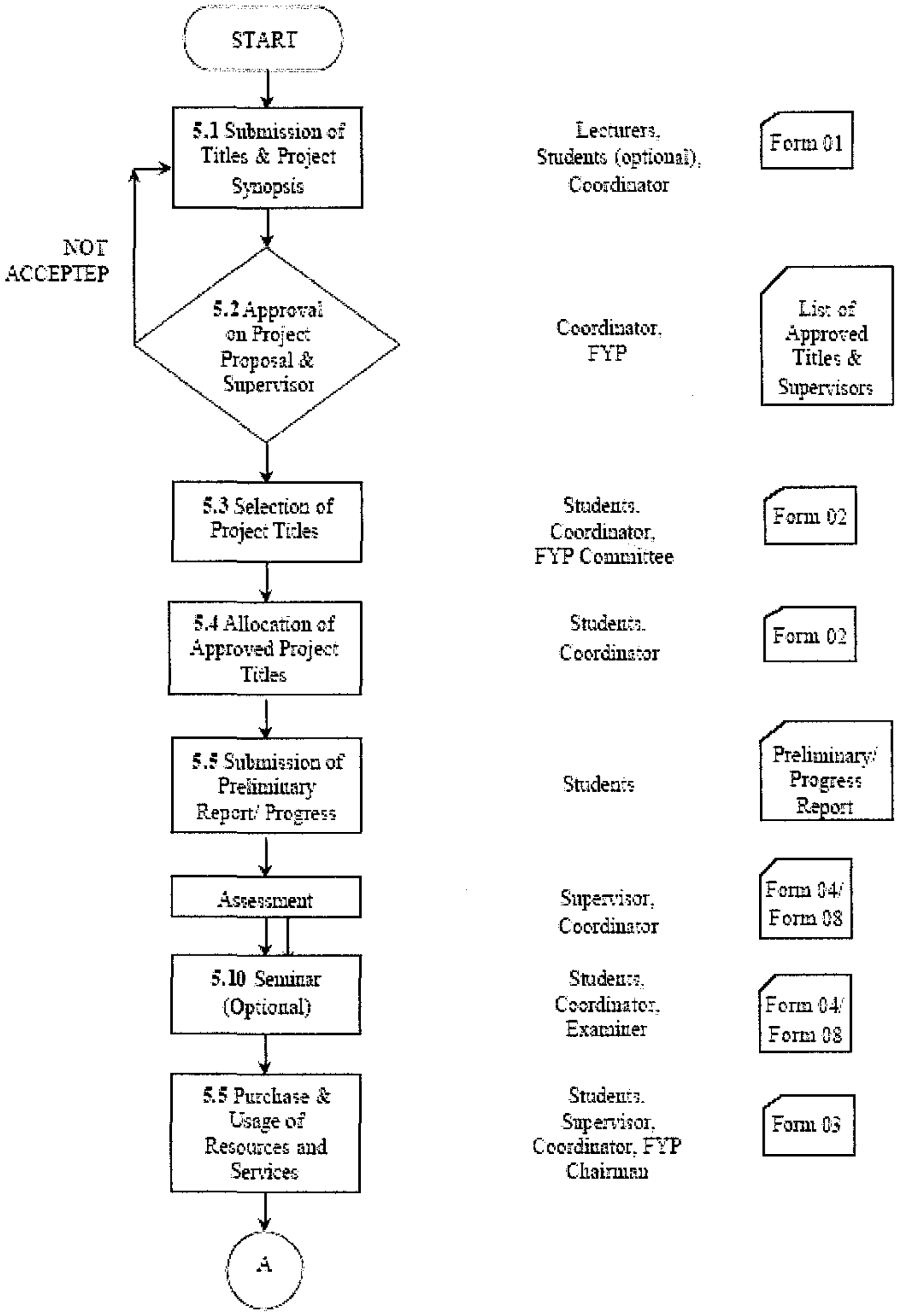

START
5.1 Submission of Titles & Project Synopsis
Lecturers, Students (optional), Coordinator
Form 01
NOT ACCEPTEP
5.2 Approval on Project Proposal & Supervisor
Coordinator, FYP
List of Approved Titles & Supervisors
5.3 Selection of Project Titles
Students, Coordinator, FYP Committee
Form 02
5.4 Allocation of Approved Project Titles
Students, Coordinator
Form 02
5.5 Submission of Preliminary Report/ Progress
Students
Preliminary/ Progress Report
Assessment
Supervisor, Coordinator
Form 04/ Form 08
5.10 Seminar (Optional)
Students, Coordinator, Examiner
Form 04/ Form 08
5.5 Purchase & Usage of Resources and Services
Students, Supervisor, Coordinator, FYP Chairman
Form 03
A

Figure 4.3 FYP flow process

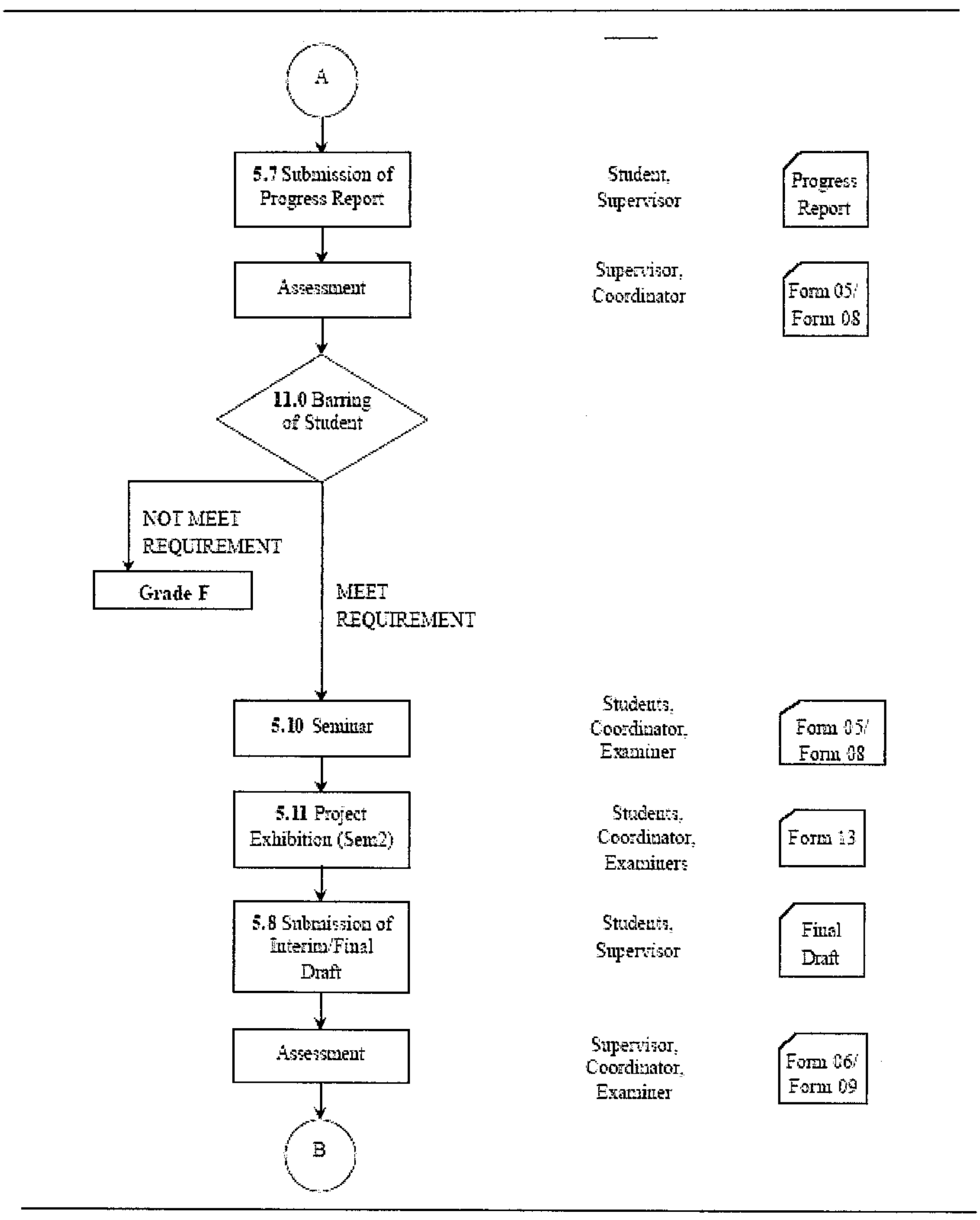


Figure 4.4 FYP flow process

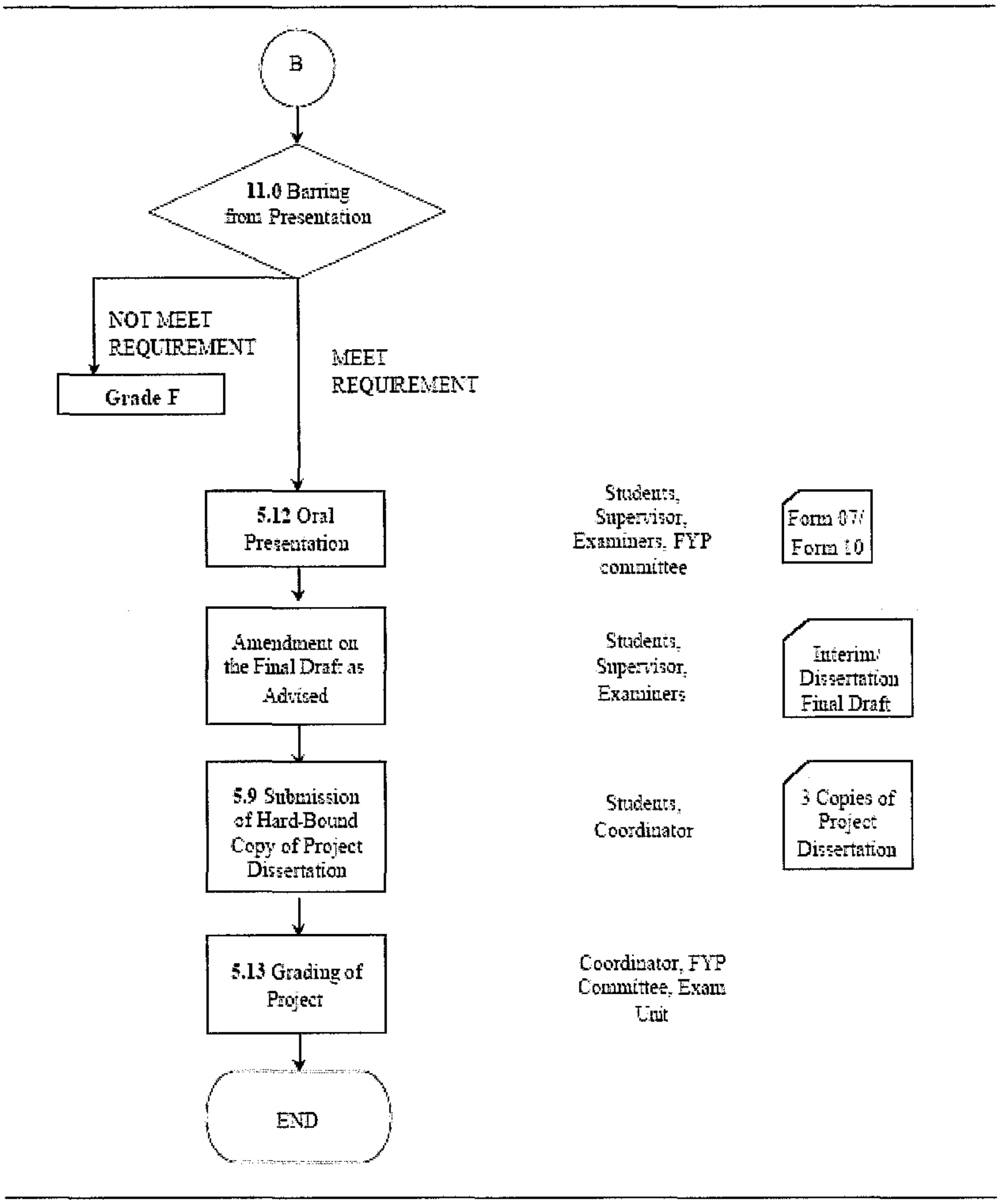
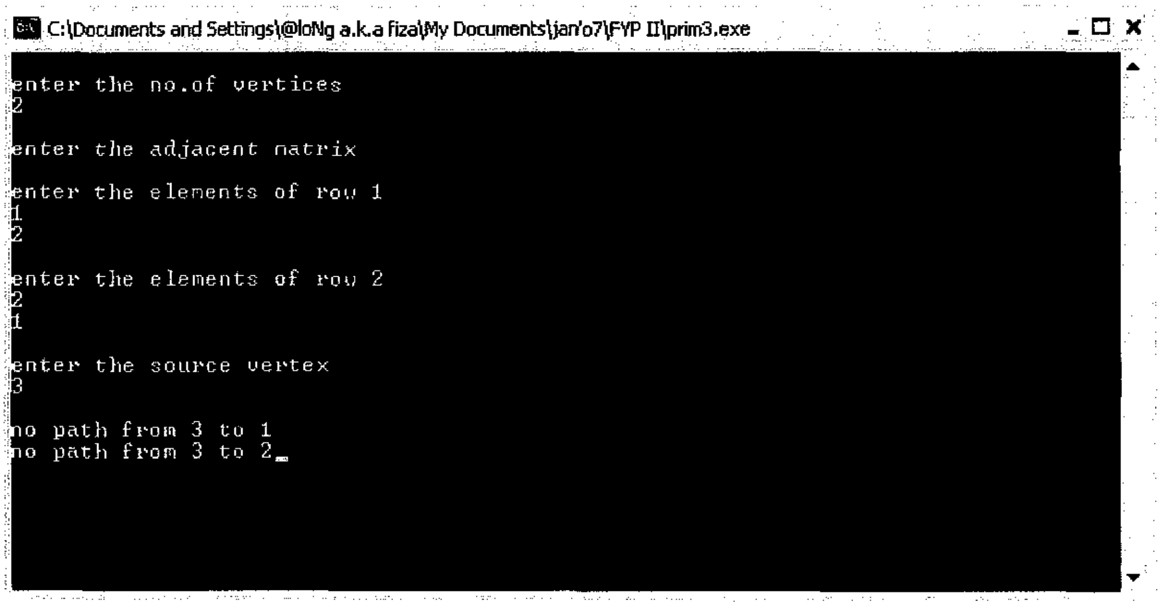

B
11.0 Barring from Presentation
NOT MEET REQUIREMENT
Grade F
MEET REQUIREMENT
5.12 Oral Presentation
Students, Supervisor, Examiners, FYP committee
Form 07/ Form 10
Amendment on the Final Draft as Advised
Students, Supervisor, Examiners
Interim/ Dissertation Final Draft
5.9 Submission of Hard-Bound Copy of Project Dissertation
Students, Coordinator
3 Copies of Project Dissertation
5.13 Grading of Project
Coordinator, FYP Committee, Exam Unit
END

Figure 4.5 FYP flow process

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
 int graph[15][15],s[15],pathestimate[15],mark[15];
 int num_of_vertices,source,i,j,u,predecessor[15];
 int count=0;
 int minimum(int a[],int m[],int k);
 void printpath(int,int,int[]);
 printf("\nenter the no.of vertices\n");
 scanf("%d",&num_of_vertices);
 if(num_of_vertices<=0)
 {
  printf("\nthis is meaningless\n");
   exit(1);
 }
 printf("\nenter the adjacent matrix\n");
   for(i=1;i<=num_of_vertices;i++)
 {
  printf("\nenter the elements of row %d\n",i);
  for(j=1;j<=num_of_vertices;j++)
  {
   scanf("%d",&graph[i][j]);
  }
 }
 printf("\nenter the source vertex\n");
 scanf("%d",&source);
 for(j=1;j<=num_of_vertices;j++)
 {
  mark[j]=0;
  pathestimate[j]=999;
  predecessor[j]=0;
 }
 pathestimate[source]=0;

 while(count<num_of_vertices)
 {
  u=minimum(pathestimate,mark,num_of_vertices);
  s[++count]=u;
  mark[u]=1;
  for(i=1;i<=num_of_vertices;i++)
  {
   if(graph[u][i]>0)
   {
    if(mark[i]!=1)
    {
    if(pathestimate[i]>pathestimate[u]+graph[u][i])
    {
     pathestimate[i]=pathestimate[u]+graph[u][i];
     predecessor[i]=u;
    }
    }
   }
  }
 }
 for(i=1;i<=num_of_vertices;i++)
 {
 printpath(source,i,predecessor);
 if(pathestimate[i]!=999)
 printf("->(%d)\n",pathestimate[i]);
 }
}
int minimum(int a[],int m[],int k)
{
 int mi=999;
 int i,t;
 for(i=1;i<=k;i++)
 {
  if(m[i]!=1)
  {
```

```
 if(mi>=a[i])
 {
  mi=a[i];
  t=i;
 }
 }}
 return t;
}
void printpath(int x,int i,int p[])
{
 printf("\n");
if(i==x)
{
printf("%d",x);
}
else if(p[i]==0)
printf("no path from %d to %d",x,i);
else
{
printpath(x,p[i],p);
printf("..%d",i);
}
}
```



C:\Documents and Settings\@loNg a.k.a fiza\My Documents\jan'o7\FYP II\prim3.exe

```
enter the no.of vertices
2

enter the adjacent matrix

enter the elements of row 1
1
2

enter the elements of row 2
2
1

enter the source vertex
3

no path from 3 to 1
no path from 3 to 2_
```

```
#include <iostream>
#include <string>
#include <fstream>
#include <set>

using namespace std;

struct gData{
   int v1;
   int v2;
   int w;
};

int main()
{
  ifstream ifp;
  string fileRead;
  int i = 0;
  gData arrayS[1000];
  gData arrayF[1000];
  gData tmp;
  char toss;

  cout << "Name of file to be read." << endl;
  cin >> fileRead;

  ifp.open(fileRead.c_str());

  if(!ifp.fail())
   {
    while(ifp >> arrayS[i].v1 >> toss >> arrayS[i].v2 >> toss >> arrayS[i].w)
     {
      i++;
     }
    ifp.close();
   }
  else
    cout << "Error opening file." << endl;

  //sort so you don't have to check weight later
  cout << "Before sort: " << endl;
  for(int b = 0; b < i; b++)
     cout << "V1: " << arrayS[b].v1 << " V2: " << arrayS[b].v2 <<" W: " << arrayS[b].w << endl;

  for(int cnt = 0; cnt < i - 1; cnt++)
   {
    int min = cnt;
    for(int sur = cnt + 1; sur < i; sur++)
     {
      if(arrayS[sur].w < arrayS[min].w)
       {
        tmp = arrayS[min];
        arrayS[min] = arrayS[sur];
        arrayS[sur] = tmp;
       }
     }
   }
 cout << "After Sort: " << endl;
 for(int j = 0; j < i; j++)
   cout << "V1: " << arrayS[j].v1 << " V2: " << arrayS[j].v2 << " W: " << arrayS[j].w << endl;

  cout << endl;

  //make and set and put first edge w/ smallest weight inside
  int a = 0;
  set<int> mySet;
  //chose 1st vertex
  mySet.insert(arrayS[a].v1);
  mySet.insert(arrayS[a].v2);
  int wTotal = arrayS[a].w;
```

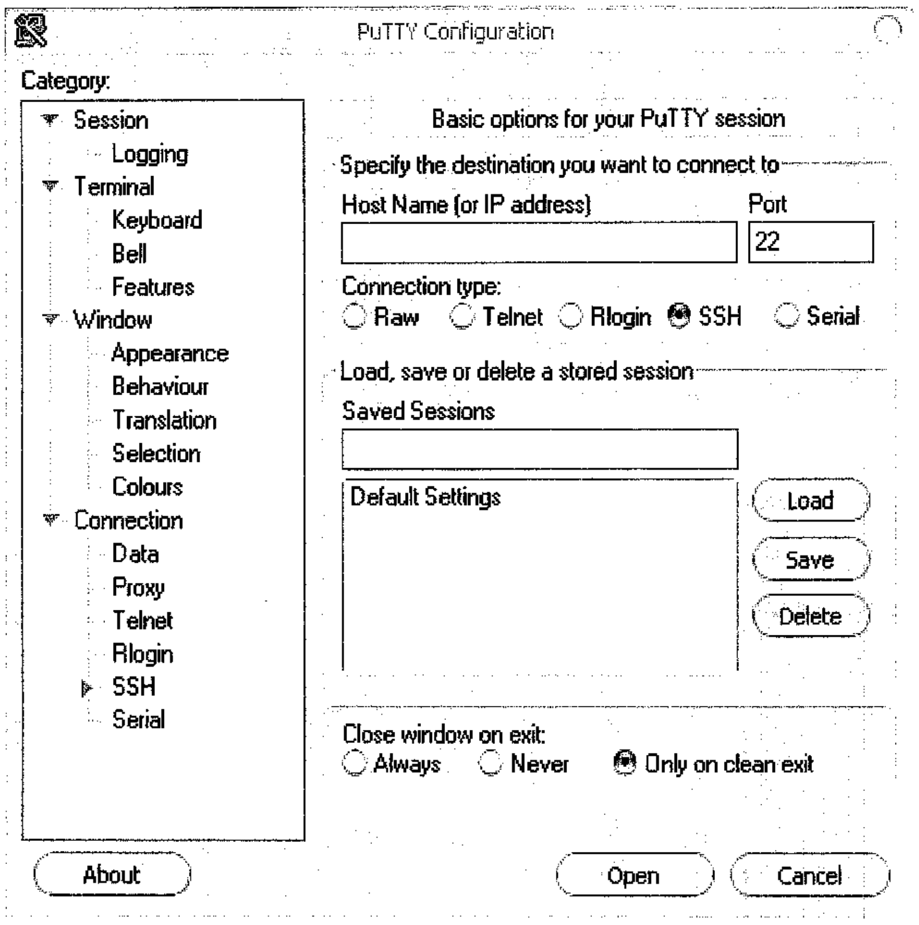
```
arrayF[a] = arrayS[a];

//loop through entire array
 for(a = 1; a < i; a++)
 {

   //if neither V connects to existing V
   if((mySet.count(arrayS[a].v1) == 0) && (mySet.count(arrayS[a].v2) == 0))
    {
     //don't use. keep looking
    }
   //if both are in it
   else if((mySet.count(arrayS[a].v1) == 1) && (mySet.count(arrayS[a].v2) == 1))
    {
     //don't create a cycle, but...
     cout << "throw out larger weight:" << endl;
     for(int l = 0; l < a; l++)
      {
       cout << "ArrayS weight: " << arrayS[a].w << " ArrayF Weight: " << arrayF[l].w << endl;
       //check for corresponding point
       int cyclePoint;
       if(mySet.count(arrayS[a].v1) == 1)
        {
         cyclePoint = arrayS[a].v1;
        }
       else if(mySet.count(arrayS[a].v2) == 1)
        {
         cyclePoint = arrayS[a].v2;
        }
       //if either V matches the cycle point
       if((arrayF[l].v1 == cyclePoint) || (arrayF[l].v2 == cyclePoint))
        {
         //and if the new weight is less than the old weight
         if(arrayS[a].w < arrayF[l].w)
          {
            cout << "Found one" << endl;
            cout << "Weight: " << arrayF[l].w << " " << arrayS[a].w << endl;
            //the old spot is replaced with the new spot & weight
            arrayF[l] = arrayS[a];

            cout <<"Array F: " << arrayF[l].v1 << " " << arrayF[l].v2 << " Array S: " << arrayS[a].v1 << " " << arrayS[a].v2 <<
endl;

            wTotal = (wTotal - arrayS[l].w) + arrayS[a].w;
          }
         //if the new weight is greater than the old weight, nothing changes
        }
      }
    }

   //if one is in it
   else if((mySet.count(arrayS[a].v1) == 1) || (mySet.count(arrayS[a].v2) == 1))
    {
     if(mySet.count(arrayS[a].v1) == 1)
      {
       //add other V
       mySet.insert(arrayS[a].v2);
       arrayF[a] = arrayS[a];
       wTotal = wTotal + arrayS[a].w;
      }
     else if(mySet.count(arrayS[a].v2) == 1)
      {
       //add other V
       mySet.insert(arrayS[a].v1);
       arrayF[a] = arrayS[a];
       wTotal = wTotal + arrayS[a].w;
      }
    }
 }
 for(set<int>::const_iterator it = mySet.begin(); it != mySet.end(); it++)
   {
    cout << "My set: " << *it << endl;
   }
 cout << "Final weight: " << wTotal << endl;

 cout << "Final array: " << endl;
 for(int t = 0; t < i; t++)
   {
    if((arrayF[t].v1 != 0) || (arrayF[t].v2 != 0))
      {
       cout << "V1: " << arrayF[t].v1 << " V2: " << arrayF[t].v2 << " W: " << arrayF[t].w << endl;
      }
   }
 return 0;
}
```

```
#include<iostream.h>
class dijkstra
{
private:
 int graph[15][15];
 int set[15],predecessor[15],mark[15],pathestimate[15];
 int source;
 int num_of_vertices;
public:
 int minimum();
 void read();
 void initialize();
 void printpath(int);
 void algorithm();
 void output();
};
void dijkstra::read()
{
 cout<<"enter the number of vertices\n";
 cin>>num_of_vertices;
 while(num_of_vertices<=0)

 {
  cout<<"\nthis is meaningless,enter the number carefully\n";

cin>>num_of_vertices;

 }
 cout<<"enter the adjacent matrix:\n";
 for(int i=1;i<=num_of_vertices;i++)
  {
  cout<<"\nenter the weights for the row\n"<<i;
  for(int j=1;j<=num_of_vertices;j++)
   {
   cin>>graph[i][j];
   while(graph[i][j]<0)
    {
    cout<<"\nu should enter the positive valued weights only\nenter the value again\n";
    cin>>graph[i][j];
    }
   }
  }
cout<<"\nenter the source vertex\n";
cin>>source;
}

void dijkstra::initialize()
{
 for(int i=1;i<=num_of_vertices;i++)
  {
  mark[i]=0;
  pathestimate[i]=999;
  predecessor[i]=0;
  }
 pathestimate[source]=0;
}
void dijkstra::algorithm()
{
 initialize();
 int count=0;
 int i;
 int u;
while(count<num_of_vertices)
 {
  u=minimum();
  set[++count]=u;
  mark[u]=1;
  for(i=1;i<=num_of_vertices;i++)
 {
 if(graph[u][i]>0)
  {
  if(mark[i]!=1)
   {
   if(pathestimate[i]>pathestimate[u]+graph[u][i])
   {
    pathestimate[i]=pathestimate[u]+graph[u][i];
    predecessor[i]=u;
    }
   }
  }
 }
}

}

void dijkstra::printpath(int i)
{
cout<<endl;
if(i==source)
{
cout<<source;
}
else if(predecessor[i]==0)
cout<<"no path from "<<source<<" to "<<i;
else
{
printpath(predecessor[i]);
cout<<".."<<i;
}
}
void dijkstra::output()
{
 for(int i=1;i<=num_of_vertices;i++)
 {
 printpath(i);
 if(pathestimate[i]!=999)
  cout<<"->("<<pathestimate[i]<<")\n";
 }
 cout<<endl;
}
int dijkstra::minimum()
{
int min=999;
 int i,t;
 for(i=1;i<=num_of_vertices;i++)
 {
  if(mark[i]!=1)
  {
  if(min>=pathestimate[i])
   {
  min=pathestimate[i];
  t=i;
   }
  }
 }
 return t;
}
void main()
{
 dijkstra s;
 s.read();
 s.algorithm();
 s.output();
}
```

```
#ifndef BOOST_GRAPH_MST_PRIM_HPP
#define BOOST_GRAPH_MST_PRIM_HPP

#include <functional>
#include <boost/graph/graph_traits.hpp>
#include <boost/graph/dijkstra_shortest_paths.hpp>

namespace boost {

  namespace detail {
    // this should be somewhere else in boost...
    template <class U, class V> struct _project2nd {
      V operator()(U, V v) const { return v; }
    };
  }

  namespace detail {

    // This is Prim's algorithm to calculate the Minimum Spanning Tree
    // for an undirected graph with weighted edges.

    template <class Graph, class P, class T, class R, class Weight>
    inline void
    prim_mst_impl(const Graph& G,
                  typename graph_traits<Graph>::vertex_descriptor s,
                  const bgl_named_params<P,T,R>& params,
                  Weight)
    {
      typedef typename property_traits<Weight>::value_type W;
      std::less<W> compare;
      detail::_project2nd<W,W> combine;
      dijkstra_shortest_paths(G, s, params.distance_compare(compare).
                              distance_combine(combine));
    }
  } // namespace detail

  template <class VertexListGraph, class DijkstraVisitor,
            class PredecessorMap, class DistanceMap,
            class WeightMap, class IndexMap>
  inline void
  prim_minimum_spanning_tree
    (const VertexListGraph& g,
     typename graph_traits<VertexListGraph>::vertex_descriptor s,
     PredecessorMap predecessor, DistanceMap distance, WeightMap weight,
     IndexMap index_map,
     DijkstraVisitor vis)
  {
    typedef typename property_traits<WeightMap>::value_type W;
    std::less<W> compare;
    detail::_project2nd<W,W> combine;
    dijkstra_shortest_paths(g, s, predecessor, distance, weight, index_map,
                            compare, combine, std::numeric_limits<W>::max(), 0,
                            vis);
  }

  template <class VertexListGraph, class PredecessorMap,
            class P, class T, class R>
  inline void prim_minimum_spanning_tree
    (const VertexListGraph& g,
     PredecessorMap p_map,
     const bgl_named_params<P,T,R>& params)
  {
    detail::prim_mst_impl
      (g,
       choose_param(get_param(params, root_vertex_t()), *vertices(g).first),
       params.predecessor_map(p_map),
       choose_const_pmap(get_param(params, edge_weight), g, edge_weight));
  }
```

```
 template <class VertexListGraph, class PredecessorMap>
 inline void prim_minimum_spanning_tree
  (const VertexListGraph& g, PredecessorMap p_map)
 {
  detail::prim_mst_impl
   (g, *vertices(g).first, predecessor_map(p_map).
    weight_map(get(edge_weight, g)),
    get(edge_weight, g));
 }

} // namespace boost

#endif // BOOST_GRAPH_MST_PRIM_HPP
```

```
Pseudocode for Prim Algorithm

Given a graph, G, with edges E of the form (v1, v2) and vertices V

dist : array of distances from the source to each vertex
edges: array indicating, for a given vertex, which vertex in the tree
it
       is closest to
i    : loop index
F    : list of finished vertices
U    : list or heap unfinished vertices

/* Initialization: set every distance to INFINITY until we discover a
way to
link a vertex to the spanning tree */
for i = 0 to |V| - 1
    dist[i] = INFINITY
    edge[i] = NULL
end

pick a vertex, s, to be the seed for the minimum spanning tree

/* Since no edge is needed to add s to the minimum spanning tree, its
distance
from the tree is 0 */
dist[s] = 0


while(F is missing a vertex)
   pick the vertex, v, in U with the shortest edge to the group of
vertices in
       the spanning tree add v to F

   /* this loop looks through every neighbor of v and checks to see if
that
    * neighbor could reach the minimum spanning tree more cheaply
through v
    * than by linking through a previous vertex */
   for each edge of v, (v1, v2)
        if(length(v1, v2) < dist[v2])
            dist[v2] = length(v1, v2)
            edges[v2] = v1
            possibly update U, depending on implementation
        end if
    end for
end while
```


PuTTY Configuration

Category:

- Session
  - Logging
- Terminal
  - Keyboard
  - Bell
  - Features
- Window
  - Appearance
  - Behaviour
  - Translation
  - Selection
  - Colours
- Connection
  - Data
  - Proxy
  - Telnet
  - Rlogin
  - SSH
  - Serial

Basic options for your PuTTY session

Specify the destination you want to connect to

Host Name (or IP address):

Port: 22

Connection type:
Raw Telnet Rlogin [✓ SSH] Serial

Load, save or delete a stored session

Saved Sessions

Default Settings

Close window on exit:
Always Never [✓ Only on clean exit]