Simulation and Optimization of Dynamic Resource Allocation Strategies in a Single Server Queuing Network

by

DOUGLAS DAVIDSON ZONE
M.A.L.D., Tufts University
(1986)
B.A. Economics, Emory University
(1982)

SUBMITTED TO THE M.I.T. SLOAN SCHOOL OF MANAGEMENT
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE
OF
MASTER OF SCIENCE IN MANAGEMENT
at the
Massachusetts Institute of Technology
June 1990

Signature of Author [illegible]
M.I.T. Sloan School of Management
May 31, 1990

Certified by [illegible]
Dimitris Bertsimas
Assistant Professor of Management Science
Thesis Supervisor

Accepted by [illegible]
Jeffrey A. Barks
Associate Dean
Master's and Bachelor's Program

Simulation and Optimization of Dynamic Resource Allocation Strategies in Single Server Queuing Networks

by
DOUGLAS DAVIDSON ZONE

SUBMITTED TO THE M.I.T. SLOAN SCHOOL OF MANAGEMENT
on May 31, 1990
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE
OF
MASTER OF SCIENCE IN MANAGEMENT

ABSTRACT

In this thesis the problem of a "probabilistic traveling repairman" in a network is discussed and studied empirically through the use of computer simulation techniques. The probabilistic traveling repairman problem can be described as a single server spatially distributed queuing system in which classes of jobs arrive at the nodes of an underlying network. The problem arises in the context of computer networks, manufacturing and vehicle routing in a dynamic environment.

This thesis proposes, examines and compares several policies for the problem. The major conclusion is that a nearest neighbor strategy is superior to all other strategies tested with respect to the minimization of the time spent in the system of the average customer. It is conjectured that this is indeed the optimal policy.

Thesis Supervisor:	Dimitris Bertsimas
Title:	Assistant Professor of Management Science

Table of Contents

1 Introduction

The goal of this thesis is to increase the understanding of a well known problem in operations research: a "traveling repairman problem" with stochastic service requirements and demand incidences (arrivals). The traveling repairman problem addressed in this thesis is straight forward: in an area customers demand on site service, i.e. IBM Mainframe owners with hardware problems. Each customer experiences problems with his equipment periodically, and requires the repairman to visit and spend time servicing the computer.

The objective is to maximize the utility of each customer by providing the best service possible. The quality of service is, in part, determined by promptness and predictability in service completion times. This thesis focuses on the minimization of the time between the moment a problem is reported by a customer arises and the time the service is completed. This thesis concentrates on the case of repair problems appearing by a probabilistic process. The repairman does not know where or when service requests will arise. His knowledge is limited to an understanding of the probabilities of a service call coming from a specific location at a given time and taking a certain amount of time to fix. Other well known problems include manufacturing, communications in computer networks and vehicle routing in a dynamic environment.

Although description of the problem is easy, it is far harder to describe it analytically. Unlike many single server and multiple class customer queueing problems, this traveling repairman problem cannot assume that service times are independent. In other words, the decision to serve a customer in location A will mean that the waiting time for the next customer in location B will include the travel time from A to B. This simple dependence is the major analytical hurdle in understanding and optimizing the problem. Despite this difficulty, the problem is common in the delivery of business and government services. Any increased knowledge of its parameters would be useful.

The description of the computer repairman is the most obvious incidence of a probabilistic traveling repairman problem (PTRP), but any case where the service time of one customer is dependant on the ordering of previous services will be analogous. One case that has been investigated is the transmission of data in packets on a communications network where each packet is accompanied by a request for a channel. This request absorbs a small amount of the transmitters time and consequently causes a delay to all other packets as the transmitter is occupied. Moreover, while waiting in line to sent each packet grows as more messages of its type are received by the transmitter. In this case the goal is to minimize the waiting time of each message by optimizing the order in which packets are sent. The selection of an order to send packets could be predetermined before the opening of the transmitter. Alternatively, the order be could determined "on line": i.e. selecting the packet to be sent by changing criteria such as always sending the longest packet first. Unlike a predetermined order a dynamic "on line" strategy requires the transmitter to keep track of the size of individual packets. Numerous static and dynamic routing strategies can be formulated. Bertsekas and Gallager have studied this problem in depth with a number of rigid strategies requiring the transmitter to "serve" packet types in an exact order. This thesis explicitly investigates these as well as other dynamic strategies using a simulation package to test how each performs.

Another example of a traveling repairman could a machine tool maintenance engineer at a manufacturing site. The engineer is required to adjust machine tools upon request from their operators. Requests arrive randomly and can be reasonably modeled by a Poisson process. Unlike breakdowns, adjustment requests can pile up without putting the machine tools being taken off the line. In other words each adjustment request is nor affected by the previous state of adjustment of the machine. In other words this scenario assumes independence of service (adjustment) times. This assumption is not necessary in much of the analysis, but the simulation package used in this thesis assumes independence. Although an interesting problem in itself, this thesis concentrates on a second type of service dependency: the choice of which machine to adjust will effect how long it takes to complete another machine by the amount of time it takes to get from one to the other. If the engineer choses to serve the machines in a line it will take less time than if he choses to serve machines on the opposite side of the plant alternately: thereby spending a good part of his time crisscrossing the plant and not adjusting machines. Despite the intuitive appeal of serving machines so as to minimize traveling time, will it be an optimal policy if some machines need to be adjusted more frequently than other. In other words would not it be more efficient to locate the engineer in the vicinity of the most troublesome machines and have him go to the machines that need less frequent adjustment only when all the "bad" machines are taken care of? This thesis studies both approaches and tries to find which policy would be superior and under what conditions.

In particular, this thesis attempts to analyze routing strategies that use dynamic as opposed to static decision rules. In other words each step in the route is not determined a priori. Routing decisions are reevaluated after each service completion. The repairman updates his knowledge of the state of his customers at each step and decides what his next move will be if any. Herein lies a major difference between static and dynamic optimization techniques. Static policies require a priori knowledge of how long the server can expect to take to remedy the problem and how long it will take to travel from one location to another. In other words the static policy can use aggregate data about the network to formulate a policy before beginning service. The latter also requires knowledge of service and travel times, but this knowledge must be constantly updated. A static policy in real life might be a decision to take a particular route to a destination according to before hand calculations of distances and estimates of traffic loads. A dynamic policy would be a constant series of re-assessments on which lane in a super highway is quickest to travel on. In all more information is necessary for dynamic decisions. In the job shop example this dynamic information can include the number of jobs at each site and the amount of time to complete each job in turn. Static information would include the average frequency that a machine breaks down and the time it takes to travel between each job. In other words, dynamic strategies are harder to implement and control. Yet, of two dynamic policies, it is not clear whether a more information intensive policy will be superior. An "ignorant" policy might perform just as well as one that uses all the best information. In fact, this thesis finds that in the case of a policy that has the server follow a set route, the ability to tell which jobs do not need service, so that the server can take a short cut to a job that does need work, does not increase efficiency significantly.

This thesis attempts to increase the understanding of this particular probabilistic traveling salesman problem primarily through the use of simulation techniques. A simulation program was written to "put into practice" different dynamic strategies. The program creates a random customer network and a timeline of incidences of repair requests at each location. At this point a "repairman" is programmed to serve the timeline of service requests using approximately forty different strategies. The output of the program is the mean waiting time and variance of all jobs over a large time period. The objectives of the simulation are to:

- determine when a particular strategy will have the lowest average times in system, if at all,
- identify those parameters that determine the optimality or stability of particular strategies,
- test whether well known results for independent service times hold under PTRP,
- and find formulas that provide good empirical estimates for waiting times under each strategy.

The thesis is organized in the following chapters:

• Chapter Two: the probabilistic traveling repairman problem is analyzed using applicable results in queueing theory. A number of results that motivate some of the policies tested are discussed.

• Chapter Three: the simulation and the policies tested explained. Motivations for each test are given based on the discussion in Chapter Two.

• Chapter Four: the results of the simulation are discussed and explanations are conjectured for the results. The chapter ends with possible areas of further research.

• Chapter Five: the major findings of the thesis are presented.

• Appendix A: the exact mechanics of each strategy are explained. Instructions re give on use of the simulation program by interested users.

Appendix B: "C" language code is given to enable the reader to adapt the simulation to new strategies beyond the current forty-seven. The code is written to give the user a wide scope of experimentation with both strategies and their controlling parameters.

The thesis concludes that the traveling salesman and nearest neighbor policies are best in almost every circumstance in all but the limiting case where arc travel time fall to zero. The analysis of M/G/1 queues in section one attempts to provide the intuition behind this empirical result. The thesis also shows that certain policies that attempt to follow the analytical optimal results on "work conserving" M/G/1 queues behave as expected but are no longer optimal.

2 Analysis of the Probabilistic Traveling Repairman Problem

2.1 Terminology and Description of Problem

The traveling repairman problem can be broken down into various parts: the space the repairman travels on, the process in which the timing of customers request is generated and the process through which the amount of service required i⌐ ᵈetermined. These three combined with the routing stra‿egy of the server will completely the determine the behavior of the system.

The terminology for the map on which the server travels is derived from network analysis. The travel space of the server is termed the **graph** in this thesis. By graph it is implied that customers reside at **nodes** of the graph, and travel distances are represented by the **arc** lengths. The key point in this analysis is that nodes are discrete, non-random and immobile. This analysis is distinct from a situation in which customers service requests emanate from random locations in space: a situation that better describes the job requests for on-site car repairs or emergency medical housecalls.[1,] The terminology used for jobs is taken from the analysis of queuing systems.

Jobs requested "**arrive**" at each node and wait to be **serviced**. The jobs are referred to as **arrivals**, for instance a request by customer one for service is termed as an "**arrival at node 1**". The backlog of jobs at location i is called the "**queue at node i**". The number of jobs backed up is the **queue length**, while the amount of total service time in a backlog is called the "**time in queue.**" **Service**

1, See Bertsimas and Van Ryzin in "A Stochastic and Dynamic Vehicle Routing Problem in the Euclidean Plane" pages 3-4.

times (termed loosely **service requirements**) are defined as the difference between the moment the repairman initiates a job and the instant that he completes it. Arrival epochs mean the moment a job request is initiated. **Delay** for job i is defined as the difference between the time a job i arrives at a node and the time it is finally serviced plus its service time. **Time in System** is used interchangeably with delay. Completion of service is called a **departure** from the queue and system. The determination of service requirements and arrival times are termed the **service and arrival processes** respectively.

Below are some common symbols used in the analysis in queues that are used throughout this thesis:

λ	"lambda". This term represents the arrival rate of jobs at a node. It is measured in arrivals per unit time. In the system under consideration here (Poisson arrivals) λ is the expected arrival rate as well.
$E(s)$	"the expected service time or requirement". This term is the mean amount of time it takes per service. In this simulation it is defined without reference to any particular probabilistic process. Nevertheless a normal distribution is used for purposes of the simulation.

σ_s^2	"the variance of the service times". This term is measures the variability in the time it takes to serve a job. Along with expected service times, this parameter determines the behavior of the random variable generated for each job's service requirement.
ρ	"rho". This term is defined as the product of lambda and the expected service time. Intuitively it represents the proportion of time the server is occupied.
M/G/1	This is the abbreviation for a Exponential (Poisson) arrival, General service, 1 server queuing system. This system has a wide area of application. It is especially suited to spatially distributed queues studied in this thesis.
d_{ij}	"distance", this term represents the distance from node i to node j in the underlying network. In most of the simulations this distance is equal the distance between two points on a plain.

I'	"velocity" or speed. Throughout the thesis we are concerned with the travel spent by a server in traveling. Varying this quantity for experimental purposes can be achieved by changing it directly, by changing distances or by changing speeds. This thesis changes speeds in that it makes a mode intuitive control variable in real world applications. Distances are kept constant throughout each simulation run.

Using these terms the object of this thesis is to analyze the effect of different strategies for moving from one node to another on the average delays of all jobs in the system. The thesis also investigates the effect of different arrival and service processes, different graph configurations and travel speeds under each strategy. A **strategy** is defined as a collection of decision rules that will completely determine the routing decision of a server at any time. (This is not strictly accurate in the context of the simulation. The decision rules used in this thesis are all based on the instant of time that service of the last job is completed. In other words, a server is committed to the decision he makes after completing his last job: he cannot change his mind while traveling or serving. This thesis does **not** investigate **preemptive** queuing regimes.)

A number of assumptions need to be highlighted that restrict the applicability of the this thesis conclusions and findings:

- There is a single mobile server who travels at constant speed. A multiple server system might be more realistic.
- The costs of travel and service to the server are ignored, the objective function consists only of the expected time of delay for all jobs in the system and/or its variance.
- Costs are linear in time and costs are equal across all classes of arrivals (nodes).
- The network is fixed and connected, although it is not required to be planar nor totally connected.
- Arrival epochs and service requirements (omitting the travel component) are independent.
- Servers can pass through a node without rendering service. Unless the strategy being tested requires the server to serve a node, there is no obligation to give service even if a job's and server's location coincide. In the planar the server can travel to his destination the way the crow files.

2.2 Description of System as a Queuing System

The analysis of PTRP as a system of queues is natural. Jobs arrive in the servers "area of responsibility" in a probabilistic manner. It is therefore impossible for the server to anticipate where he will be needed next. Since it is probable that the repairman will either not be at a site when a job arrives or will be occupied with a previous job, one would expect jobs to experience a delay as the server travels to the job's location and/or finishes its current service activities. Unless jobs arise only rarely one would expect that every once in a while jobs will pile up at customers location before the server has a chance to "get to them". Consequently, the PTRP has two key elements of a queuing system: delays in initiating service and pile-ups of jobs waiting in line to be served.

Analytically, the probabilistic traveling repairman problem can be understood as an extension of a single server priority queuing system. Priority queues are systems that allocate service to classes of customers by a priority system: the jobs in the class with the highest priority are served before those jobs in lower priorities. The optimization of priority queues is based on either the optimal allocation of jobs into different classes with different priorities or, conversely, assigning priorities to classes of jobs that already exist. The object of this section is to draw an analogy between priority queues and the PTRP and examine whether the known priority queue results are applicable to the case at hand.

In the PTRP the job classes are straightforward: they are the arrivals associated with each node. In that each node has its own arrival and service processes, the association of nodes to priorities is natural. Since the server must decide travel to a unique node to serve a member of each class, he must make explicit decisions on which class to serve. By default the act of making a decision to travel is the same as assigning each job to a class. Therefore, the results for priority queues would seem to apply readily to PTRP.

Yet, as will be discussed below, the separation of nodes by a travel delay invalidates some of the key assumptions used to derive the known optimal policies, thereby making their application to our case questionable. Nevertheless, it is essential to note that the probabilistic traveling repairman problem in the limit does not violate these assumptions. As traveling distances fall to zero or the traveling speed goes to infinity, delays in switching service from one queue to another fall to zero. Without these delays the PTRP reduces to a single queue with multiple classes of customers. The known results apply to this type of system. To understand PTRP it is essential to understand how the assumptions are used in deriving optimal policies in the zero distance case. In this manner we can gain an idea of how optimal policies for PTRP can be found.

2.3 Priority Queues and the M/G/1 Queuing System

This section will begin with brief description of the common results in queues with priorities. Key assumptions that PTRP violates as a priority queue will be highlighted. The section will end with "loose" derivation of the Pollaczek-Khintchine formula for average delay of a job arriving in the graph at a random time. Though not analytically correct, it gives some intuitive understanding to two important service strategies studied in the Section 3: a strategy based on the shortest circuit and another based on the proximity of the nearest node.

An M/G/1 queue is defined as a system in which jobs arrive according to a Poisson process, are served by a general service process, and are served by a single "server". The M/G/1 system is particularly well suited to spatial queues. Spatial separation of jobs, makes the identity of the last job served a determinant of the time it takes to serve the current job. If the last job was served in close proximity to the current node, the service time of the next job will be reduced by the savings in travel times. In fact, PTRP is an M/G/1 queue but with dependant service times. The key result for M/G/1 queues are the Pollaczec-Khintchine (P-K) formulas used to determine average job delays. Ideally, this equation could form a basis of an optimization scheme for PTRP. Yet the derivation of these formulas assumes that the choice of job to serve is independent of the service requirements for the job. Unfortunately, most strategies studied in this thesis have highly dependent service times. This dependence is manifested in node travel times between adjacent nodes or in policies that investigate the service requirement at a node as a criterion for serving that node or not.

2.3.1 Priority Results

In the following priority theorm there exists a key assumption: **work is conserved.** This is the assumption that is invalid for the probabilistic traveling repairman outside the zero travel time case. Wolff defines work conservation as follows:

> Call S the service requirement, α the amount of service obtained so far and t the current time. If

the priority rule is work conserving then at any time t the remaining service time is equal to S - α.[2,3]

The PTRP is not work-conserving in that the remaining service time may fluctuate with the location of the server. To see this imagine that the first job served at a node is allocated the service time used in travel. (This allocation is to ensure that travel times are taken in consideration in calculating delays). If the server moves between several nodes (priorities) before coming to a node of reference it is easy to see that the first job in this final node may have a number of potential remaining service times = $S-\alpha+d_{ir}$, or = $S - \alpha + d_{jr}$, etc..

Nevertheless, it is possible that priority rules will be valid if travel times are either negligible or contribute only marginally to reducing average time in system. This limit case of PTRP is a good point from which to begin our analysis. The following subsections describe rules that are applicable to minimizing system times in M/G/1 queues with independent service requirements.

2, The definition in the text is paraphrased from the definition on page 437 of R. Wolff (1989) Stochastic Modeling and the Theory of Queues, Prentice- Hall, page 437.

3, D. Heyman and M. Sobel in Stochastic Models in Operations Research (1981), page 418,provide the following definition of work conservation: "A queue discipline is called work-conserving if (a) no servers are free when a customer is in queue and (b) the discipline does not effect the amount of service time given to a customer or the arrival time of any customer."

2.3.2 The Shortest Remaining Processing Time Discipline:

The SRPT Optimality theorem states that if preemption does not change the time required by a job, then a discipline that places into service the customer with the smallest amount of remaining service will minimize expected waiting time in the system. This result is clearly not applicable to the PTRP system studied in this thesis since there are no pre-emptions allowed. Nevertheless, once a server is within a queue the act of staying put at the node may be an implicit SRPT policy. Consider that the remaining service time of a job at other nodes as their service requirement plus the travel time that the server must undertake to get there. If travel times are large enough then the probability that the remaining service time of a job in another node is shorter than any of the remaining service times at the current node will be negligible. Therefore, by staying put until all the members of the current node are served may be implicitly following SRPT.(In other words: "exhaustive" policies may exhibit SRPT behavior). Now consider the moment that the server **must** depart the just emptied current node if he does not want to remain idle. The remaining service times of his next potential job will now include the traveling time. So in our network a server following a SRPT discipline would go to the node j that minimizes sum of 1) the travel time to node j and 2) the shortest remaining service times among jobs at node j. **Again if travel times are sufficiently large, the SRPT's would be determined primarily by the arc lengths.** But choice of which node to serve, now, will effect the remaining service times in the future as the server is closer or farther from the rest of the nodes. Therefore there is no work conservation and the SRPT theorm is not applicable to PTRP.

Now consider a case in which the arc lengths change randomly after each service. In other words the travel time between each node has a general distribution defined over time. (Imagine a caterer who only takes drink orders at a wedding as the server, and the cliques of relatives as the nodes that accumulate jobs as drink orders. The cliques constantly wander around the reception floors en masse.) In this case, the SRPT regime would maintain independence of service times. The sample path of the server would be unaffected by the decision to serve the closest node: the next closest node will be determined randomly by the relocated nodes. Choosing a farther node now, believing that she will be closer to other nodes in the future and therefore have shorter remaining service times, would be an even odds gamble. If the nodes are far enough apart a policy of exhausting all the service requirements at a particular node before going on to the nearest neighbor would be essentially a SRPT discipline where work conservation would effectively hold.

Now take the case of the stationary nodes again. Again assume inter-node distances make travel times exceed any job's service requirement, (making an exhaustive policy SRPT by default). If a server is incapable of predicting an optimal sequence of nodes to visit in order to minimize remaining service times in the future as well as the present, he is in the same situation as a server in a "chaotic" network. Thus, a policy of exhausting the queue at the current node and then traveling to the nearest neighbor may exhibit "quasi"-independent service times and resemble an SRPT discipline. In order for the server to be incapable of selecting a dependant optimal path the following may need to be true:

- The server is ignorant of all but the travel distances and speeds between nodes and the probability that a queue is empty at any given moment is high. In this case, path selection would be futile.
- The server has knowledge of queues at each node but there is sufficient probability that such nodes will become non-empty as to make the selected optimal path invalid.

The following conjecture can be made. If an optimal path is no better than a dynamic nearest neighbor path (due to light traffic) and if nodes are sufficiently separated as to make the job with the SRPT always the job at the nearest neighbor, then the nearest neighbor policy may closely approximate a SRPT discipline and therefore be optimal.

2.3.3 Other M/G/1 Disciplines

There exist strong M/G/1 optimality results beyond SRPT, but they use different objective functions and are less applicable to this thesis simulation. Two policies are of interest though: the FIFO regime and its effect of minimizing the variance of times in system, and the "$c-\mu$" rule which minimizes the objective function: $\Sigma_{i=1}^{N} c_i\rho_i + \Sigma_{i=1}^{N} c_i\lambda_i \overline{W}_{qi}$. A corollary to this last policy states that minimization of average waiting times can be achieved by assigning priorities according to expected service times for each class of user.[4] Therefore, one would expect that as PTRP traveling times approach 0 that a priority system favoring nodes with the smallest expected service times would perform the best. In particular, one would expect the policy to work the best when the server is allowed to switch nodes without emptying it first as long as the server moves to serve a higher priority job immediately. An interesting empirical question would be to find out at what travel times do the "$\mathrm{Min}\,E(S)$" policy and the SRPT policy and the optimal policy on a non-zero distance, (which I conjecture to be a Nearest Neighbor policy), perform equally well.

2.3.4 Applicable Results Derived From M/G/1 Queues

A number of results that expand on the P-K formulas and apply them to spatially dependant service times exist. The key fact about each of these formulations is that they are specific to certain strategies. The most notable result is presented by Bertsekas and Gallager for a routing strategy on a network with uniform service and arrival parameters across all nodes. The strategy mandates that each node be served in a fixed order. The server is required to serve each node until it is empty and is obligated to visit each node whether it is empty

4, See R. Larson and A. Odoni (1981), Urban Operation Research, Prentice-Hall pages 237-239 for a discussion of this "corollary."

or not. The strategy "44" in the simulation uses these rules exactly. The result presented by Bertsekas and Gallager resembles the form conjectured in the previous section. It shows that a queuing strategy on a network with non-zero travel times will have average system times highly dependant on rho and the variance. The travel time term interacts exclusively with rho. In a test of the effects of expected service times, variance of service time and travel times for other strategies it was found that, as opposed to the strategy used by Bertsekas and Gallager, a strong interaction exists between the variance and speed in determining average time in system. Below is the formula derived for expected time in system when the server "exhausts" the queue in his current node. The strategy is based on the server following a fixed circuit around the nodes on a continual basis.[5] A is defined as the travel time it takes the server to complete a circuit.

$$\overline{W} = \frac{\lambda E(s^2)}{2(1-\rho)} + \frac{A}{2}\frac{\left(1-\left(\frac{\rho}{m}\right)\right)}{(1-\rho)}$$

where:

$\overline{W}$ =Average time in system,

m = the number of nodes in the network,

A = the time to travel around the specified circuit if the server were uninterrupted.

$E(s^2)$ = the second moment of the service time distribution. It can be derived as $\sigma_s^2 + (E(s))^2$

5, see D. Bertsekas and R. Gallager (1987), Data Networks, Prentice Hall, page 157.

The above formula suggests that different strategies may have similar forms with a distinct coefficient for A. Nevertheless, this policy is unique in that its server route is independent of the arrival process's mean or variance. One would expect that a strategy that allows even minimal server discretion based on the state of the system would have a different functional form with a coefficient on the firsts term also.

2.3.5 Conclusion

At this point we have two sets of results that apply to the PTRP problem: the accepted priority theorems that clearly apply to PTRP (though under a restricted set of strategies) at the limit, and the less than rigorous indications of the previous section. Without a theory to combine both conceptualizations we are left with a number of interesting empirical questions:

- at what point do travel times grow big enough to make the priority rules invalid for PTRP?
- do the priority optimization results apply at high travel time scenarios, but are essentially masked by the large distance saving effects of routing optimization policies? In other words, can priority rules improve the performance of routing optimization policies?

In the chapter 4 section we discuss these issues. In most cases the data confirm the intuition the models discussed above provide.

3 Simulation: Description and Motivations of Strategy Formulations

3.1 Overview of Strategies

In this chapter, the thesis concentrates on formulating potential strategies that will minimize the average system times for all jobs in the system. A majority of policies draw directly from the discussion in the last chapter. Some are tested on intuitive grounds with little theory behind them. In brief, three broad classes of strategies were formulated. First, strategies based on pre-assigning priorities to each node were drawn up. These are the least dynamic of all the strategies: they essentially assign the priorities before initialization and require the server to serve the highest priority node that has a non-empty node at all times. The second set of strategies use current statistics to base routing decisions. The server, for instance, is required to serve the node with the highest accumulated amount of service at any one time. These strategies need the most information to run and can be considered the most "dynamic". The last set of policies to be studied are the "routing" disciplines. The server is required to serve either a preset route or make all his decisions on routing criteria. For instance the nearest neighbor policy requires the server to proceed to the closest non-empty node after completing service at the current queue. A combination of the last two policies is also tested. These policies, (called "vicinity" strategies), have the server use nearest neighbor policies within a subset of the nodes in the network. The subset of nodes, on the other hand, are chosen by the second set of criteria: either longest vicinity-wide queue or largest vicinity-wide amount of accumulated service.

3.2 Gating Strategies

In almost every category a number of gating schemes were formulated. Gating involves setting a maximum or minimum number of jobs the repairman can take on before being required to serve another queue. Initially, the setting of a maximum service number per queue was felt to be way to prevent the server from serving new customers just arriving in his present location at the expense of older customers elsewhere. In other words, if the server did not serve one population for the benefit of another, it would appear the ignored groups time in system would push up the whole populations mean. As it turns out, this logic is misleading. If one considers that each individual in the system, by not being served, contributes the same **marginal** amount to the mean time in system, it is clear that a gating strategy that minimizes the number of jobs in the system regardless of how long they have been in the system will be superior. If one gating strategy reduces the system queue by 4 recent arrivals and another by 3 old arrivals, it is the former strategy that is superior in keeping down average times in system. This is a direct consequence of the linear time cost function. It would not be true if longer times in system cost more proportionally. An implication of this reasoning is that any policy that minimizes the amount of time the server is idle or on the move will improve performance. In other words, a policy such as exhaustive service at each node may be better than a policy that forces the server to move earlier. A counter argument may be that by forcing the server to move early he is being prevented from ignoring a queue in which a large number of jobs could be "knocked off" quickly. Going back to the discussion of SPRT in the last chapter, if inter-node travel times are great enough any gains from giving up the present node to serve a node where remaining processing times are shorter would be lost in the move. An interesting point that is not investigated in this thesis is at what travel times do exhaustive policies begin dominate?

3.3 Priority Strategies

The minimization of times in system in M/G/1 queues without service times dependency can achieved through the use of the $c\mu$ rule. The application of this rule is unclear in PTRP since $\mu = \frac{1}{E(s)}$ is now made up of travel time as well as the job,s service requirement. Therefore, the expected service times cannot be ordered as in the simple non-spatial case. Nevertheless, since PTRP in the limit will be optimized by the $c\mu$ rule, it is worth investigating whether it is valid outside the domain of very small travel times. A priority system was set up with the node having the lowest E(s) being given the highest priority. As in the non-spatial case this strategy was set up to ensure that the server would take care of the highest priority queue as soon as it became non-empty. Nonetheless, given the intuition behind exhaustive gating, this policy was modified to ensure that the server exhaust or partially exhaust the node he was in. The mixture of these two concepts is grounded more in experimentation than any formal reasoning. In order to ascertain whether this policy had any effect at high travel times, its opposite priority scheme was also tested giving the lowest priority to the minimum E(s). If these policies are not significantly different at high travel times we can say that it has no effect at all.

In order to bring a spatial element into the above priority scheme, a system with the highest priority being given to the node with the lowest average travel time plus expected service time was tested. It was felt that the $c\mu$ would hold into higher travel time environments if it reflected the effect of travel. The use of average distance is crude, but it left in tact the benefit priority schemes have in being able to set up strategies before initialization.

A number of other priority schemes were tested without the analytical background of the previous priority schemes. These were priorities by maximum and minimum λ and ρ. The maximum arrival policy may have some use at high travel times in combination with an exhaustive node management policy. If directing the server to a high arrival node prevents him from moving (since exhausting the queue is difficult), then, by the SPRT analogy in the previous chapter, this priority scheme might outperform the others.

3.4 Routing Strategies

The routing strategies in terms of queuing theory have little intuitive appeal. Each strategy is based on minimizing the distance traveled by the server. No attention is paid to service, arrival, queue length, or accumulated service time statistics. Yet as discussed in Chapter One, the act of minimizing time spent traveling may be a de facto Shortest Remaining Processing Time discipline if travel times are large enough. Three policies that depend on routing are tested: a Nearest Neighbor policy, a Traveling Salesman strategy which allows shortcutting past empty nodes and a Traveling Salesman policy without shortcuts. Shortcuts with TSP are allowed to make it more comparable with NN's (Nearest Neighbor) "intelligence". Nearest Neighbor can see if a neighboring node is empty and will not chose to visit it. TSP had to be given the same ability or else a comparison between the two would be biased by the information asymmetries. Other routing policies based on a combination of NN and TSP or based on a type of node coverage were not tested but would be of great interest. Finally, a policy of measuring closest by travel time plus the average service time of a job at the node was tested. It was felt that this could be an improvement on NN, making it imitate SPRT more closely.

3.5 Strategies Based on Queue Statistics

A number of strategies were tested on the conjecture that more current information will benefit any serving policy. They are based on an idea that the server should go where he is most needed: to the node with the longest queue or the node with the most service time added up. As mentioned in Section 3.2, the urge to serve the oldest in the queue first does not make sense on the grounds of minimizing average time in system, unless time costs increase at an increasing rate. (Though these policies may be beneficial in reducing the variance of time in system though). Yet, if these policies cause the server to stay at a node longer than he would have at an average node, then their might be some advantages similar to those derived by exhaustive regimes.

3.6 Vicinity Strategies: Combining the Best of Routing and Current Information Strategies?

Vicinity strategies attempt to improve on the NN policy with the information used in the previous section. These policies do this by restricting the travel of the server to a subset of the nodes in the network in which the arc lengths are minimized. Within the subset the nodes are selected by a nearest neighbor policy. Each subset (vicinity) is selected by the longest sum of queues or the most accumulated sum of service criterion. If the subset of nodes is equal to all the nodes, these policies are equivalent to the NN policy. If the subset is equal to one these policies are equivalent to the ones in the last section. A vicinity is defined as the set of beta nodes surrounding each node with the minimum summed arc distances. (Therefore, for each node there is a corresponding vicinity). The object of a vicinity policy is to keep the server from traveling between nodes as much as possible by sending him to the nodes with the most service needed while making sure that if he does travel he goes the least distance possible. In light of the SPRT intuition, another vicinity policy requiring the server to move to the node subset with the least accumulated service was also tested.

3.7 Conclusion

One policy tested that does not fit into the categories above is a system-wide FIFO discipline. Theoretical results tell us that under FIFO the variance in times in system would be minimized. It would be interesting to see if this will hold under significant travel time regimes. Intuition would suggest that system-wide FIFO (SFIFO) would behave poorly in holding down average times in system. In effect SFIFO, would have the server stay at a node only if it experienced two or more consecutive arrivals, otherwise he would be required to travel to each node in proportion to the probability that an arrival occurs at that node. Though not the maximum length path, this routing will have significantly more travel times than NN or TSP, even when they are gated at one service.

Finally, one policy of interest is not tested. This strategy is a refinement of NN to imitate SPRT more closely. In this strategy the choice to move to the next node would be made after each service. The server would compare the minimum remaining service time of the jobs in the current node with the minimum of the sum of travel times to other nodes and their minimum remaining service times. Only if the latter is smaller or the current node is exhausted would the server change nodes. In this modification, one would expect the NN policy to remain an SPRT discipline even as travel times fall to zero. Nonetheless, of the strategies suggested so far, this one would be the most information intensive. In fact it is the only policy suggested that would need to keep track of individual service requirements.

4 Results of Simulations

4.1 Overview of Results

In this chapter, the thesis concentrates on the empirical results of the dynamic strategies tested in the simulation package. The strategies were selected for three primary purposes: one, to find which policies perform the best, two, to investigate the potential of policies found to be optimal in similar systems to PTRP and, finally, to give an indication where theoretical work on finding an optimal PTRP should go. Overall, it was found that shortest circuit (TSP) and nearest neighbor strategies had the lowest mean time in system as well as lower than the average variances among the group tested. Optimal M/G/1 priority disciplines such as $c\mu$ policies deteriorated in performance as travel times in the network increased. Yet those policies that performed well under high travel times did not perform worse than average under the zero travel time case where PTRP essentially becomes a priority queue. On a theoretical level tests performed showed that the exhaustive policies were superior to policies that gate service as travel times became significant. This gives an indication that an analytical approach based on SRPT may be fruitful. Finally, the simulation showed that TSP based strategies would dominate NN as travel times exceed the mean service requirement. This result is not explained in this thesis and leaves a interesting theoretical question open for research.

4.2 Description of Results

In the following sections simulation results are described and explained. In order to understand the data some explanations of units of measurement are necessary. In the first place, time and distance are presented in generic units. A speed of 2000 can be translated as two thousand distance units per time unit.

Throughout the results the networks are defined on a square plane with sides of 1000 distance units. Average distance between nodes are theoretically one third the lengths of the square in the x direction and in the y direction. With 24 and 48 nodes in the square the actual average should be fairly close. Service times and arrival times are also measured in units of time per event. In other words a E(s) of .347 means that it is expected to take .347 time units to complete a job. Lambda "i" in the tables is the average arrival rate at each node. Lambda is the arrival rate for the system as a whole, (typically lambda "i" times the number of nodes in the network.)

For presentation purposes each strategy discussed in the last chapter is given an identification number from one to forty seven. A list of policies are given in the following table. Notice that each category includes various alternative "gating" disciplines, which usually includes one exhaustive and three gated disciplines. The gating on "until empty" means the server stays at a node until it is completely empty. Likewise the gating on "until all originals served" forces the server to move after the jobs that were in queue when he began at the present node are served. "Until at least beta served" means the server must move after serving beta customers or until the queue is empty; whichever comes first. Similarly, "until beta*rho worth of service at most" is a gating that forces the server to move if he has served for beta x rho worth of time at his current position. If the queue empties before this time has passed the server also moves on.

Many of the forty-seven policies are included as control studies. For instance, the priority policy that gives the node with the minimum expected service time the highest priority (reflecting a $c\mu$ result) is accompanied by an opposite policy that gives the same node the lowest priority.

Complete explanations of each policy are given in the appendix.

4.2.1 List of Strategies Tested

ID	Category-Type	Special Features
	System-wide First In First Out	
1	Serve by System FIFO (SFIFO)	Move to next oldest arrival.
	Information Based Strategies	
2	Serve the longest queue	Until empty.
3	Serve the longest queue	Until all originals served.
	Route Based Strategies (NN and TSP)	
4	Serve the closest queue (NN)	Until empty.
5	Serve the closest queue	Until all originals served.
6	Serve the closest queue	Until at most Beta served.
7	Serve the closest queue	Until Beta*Rho worth of service at most.
42	Serve a vicinity of nodes with the lightest work load	Until empty.
8	Serve TSP route (TSP)	Serve at most one customer.
9	Serve TSP route	Until empty.
10	Serve TSP route	Until all originals served.
11	Serve TSP route	Until at most Beta served.
12	Serve TSP route	Until Beta*Rho worth of service at most.

43	Serve Route: w/o "shortcuts"	Move to Preferred if non-empty.
44	Serve Route: w/o "shortcuts"	Until empty.
45	Serve Route: w/o "shortcuts"	Until all originals served.
46	Serve Route: w/o "shortcuts"	Until at least Beta served.
47	Serve Route: w/o "shortcuts"	Until Beta*Rho worth of service at most.
	<u>Mixture of Route and Information based strategies</u>	
13	Serve vicinity w/ Max. Service	Until empty.
14	Serve vicinity w/ Max. Service	Until all originals served.
15	Serve vicinity w/ Max. Queue	Until at most Beta served.
16	Serve vicinity w/ Max. Queue	Until Beta*Rho worth of service at most.
41	Serve the closest where close= E(S)+Travel Time	Until empty.
	<u>Priority Queuing Systems</u>	
17	Preferred: Min. E(service)	Move to Preferred if non-empty.
18	Preferred: Min. E(service)	Until empty.
19	Preferred: Min. E(service)	Until all originals served.
20	Preferred: Max. E(service)	Move to Preferred if non-empty.
21	Preferred: Max. E(service)	Until empty.
22	Preferred: Max. E(service)	Until all Originals Served

23	Preferred: Min. Rho	Move to Preferred if non-empty.
24	Preferred: Min. Rho	Until empty.
25	Preferred: Min. Rho	Until all originals served.
26	Preferred: Max. Rho	Move to Preferred if non-empty.
27	Preferred: Max. Rho	Until empty.
28	Preferred: Max. Rho	Until all originals served.
29	Preferred: Min. Arrival	Move to Preferred if non-empty.
30	Preferred: Min. Arrival	Until empty.
31	Preferred: Min. Arrival	Until all originals served.
32	Preferred: Max. Arrival	Move to Preferred if non-empty.
33	Preferred: Max. Arrival	Until empty.
34	Preferred: Max. Arrival	Until all originals served.
35	Prefer: Min.E(S)+Travel	Move to Preferred if non-empty.
36	Prefer: Min.E(S)+Travel	Until empty.
37	Prefer: Min.E(S)+Travel	Until all originals served.
38	Prefer: Max.E(S)+Travel	Move to Preferred if non-empty.
39	Prefer: Max.E(S)+Travel	Until empty.
40	Prefer: Max.E(S)+Travel	Until all originals served.

4.3 Important Simulation Details

A number of special features of the simulation need to be emphasized. First, the server treats each node queue with a FIFO discipline. Though the SRPT would perform better, it is felt that replacing FIFO with shortest remaining processing time rule would not change the relative ranking of the inter-node routing strategies. Nonetheless, in the case of closely ranked NN (nearest neighbor) and TSP policies a change in the internal node discipline from FIFO may cause a reversal in their rankings. The simulation is also limited in the manner routing decisions are made. The server can only decide his next move the instant a service is completed. In other words, the server is committed to his decision, and once a service is started it cannot be preempted. The wide range of topics covered under preemptive queues are therefore not covered.

4.4 Comparison of Average Times in System of Strategies Tested

The following tables summarize the results of simulations run on each strategy. Each strategy is tested at five different speeds that proxy five different travel time. The speed "2E+07" is 20,000,000 distance units per time unit. The travel time of two nodes on opposite sides of the square 1000 by 1000 area would be 1000/20,000,000 = .00005. Similarly the speeds 40000, 20000, 10000 and 5000 would mean travel times of the order .025, .050, .100 and .500 time units respectively. The average service times are given by E(s) and average inter-arrival times can be found as the inverse of Lambda. Finally, the policy "45x" (not listed in the table) represents the no shortcut TSP policy where the circuit selected is non-optimal.[6] The second column of the table is identical to the first except the level of expected service time, variance of service times and arrival rates have been changed. Each strategy and speed combination was run on the same underlying network, arrival and service rates. In this case, each node had different service and inter-arrival means. This was done so that the priority disciplines could be tested. The nodes on the underlying network were placed randomly on a 1000 by 1000 plane. Distances were calculated assuming all nodes were directly connected.

6, This can be achieved by a user of the simulation by running TIMM.C with any of the TSP strategies and leaving out the route file (argument 3) of DIMTSP.C. The program finding the route file missing will supply an arbitrary route determined by the nodes' indices. See Appendix A.

4.4.1 Table: Comparison of Policies by Average Time in System

AVERAGE TIME IN SYSTEM

ORDER FROM BEST TO WORST OF 47 STRATEGIES UNDER VARIOUS TRAVEL TIMES E(s) , Var(s)

Nodes 24										Dist.		1000
Speeds:		2E+07		2E+07		40000		20000		10000		5000
lmbda 1		0.096		0.098		0.096		0.096		0.096		0.096
E(S)		0.387		0.365		0.387		0.387		0.387		0.387
Var(S)		0.020		0.039		0.020		0.020		0.020		0.020
Lambda		2.304		2.352		2.304		2.304		2.304		2.304
Rho		0.892		0.859		0.892		0.892		0.892		0.892
	17	1.683	35	1.352	36	2.229	4	2.604	4	3.257	4	4.730
	35	1.683	17	1.352	18	2.231	6	2.604	6	3.257	6	4.731
	23	1.722	19	1.373	17	2.248	5	2.629	9	3.322	11	4.887
	19	1.723	37	1.373	37	2.248	9	2.647	11	3.322	9	4.887
	37	1.723	18	1.376	19	2.249	11	2.647	5	3.324	5	4.960
	25	1.761	36	1.376	35	2.250	10	2.678	10	3.405	10	5.079
	18	1.763	23	1.376	4	2.277	2	2.706	2	3.554	44	5.945
	36	1.763	25	1.401	6	2.277	3	2.737	3	3.606	46	5.945
	24	1.767	24	1.405	25	2.294	7	2.874	7	3.932	2	6.009
	16	1.964	29	1.432	5	2.299	18	2.883	44	4.100	45	6.233
	42	1.964	31	1.458	23	2.301	36	2.883	46	4.100	3	6.375
	29	1.969	30	1.465	9	2.310	24	2.945	45	4.220	45x	9.999
	31	1.972	32	1.478	11	2.310	37	2.956	15	4.272	15	10.264
	30	1.977	34	1.482	10	2.319	19	2.960	13	4.279	13	10.316
	32	1.977	33	1.486	2	2.332	15	2.962	12	4.423	7	10.792

10	1.987	16	1.501	3	2.342	13	2.987	8	4.423	42	14.237
45	1.988	42	1.501	7	2.415	25	3.007	45x	4.702	16	14.237
13	1.989	7	1.502	15	2.439	17	3.034	16	4.968	8	14.895
3	1.990	8	1.505	13	2.455	35	3.035	42	4.968	12	14.895
15	1.990	12	1.505	8	2.476	8	3.044	24	5.437	34	29.724
2	1.991	43	1.506	12	2.476	12	3.045	18	5.562	43	33.918
9	1.992	1	1.509	42	2.501	23	3.078	36	5.593	28	39.200
11	1.992	10	1.509	16	2.502	44	3.114	25	6.092	35	39.274
46	1.993	5	1.511	45x	2.536	46	3.114	37	6.103	32	39.816
44	1.993	4	1.511	46	2.559	16	3.138	19	6.180	17	40.074
34	1.994	6	1.511	44	2.559	42	3.138	43	6.402	39	43.789
4	1.994	15	1.513	30	2.601	45x	3.155	33	6.518	29	45.782
6	1.994	9	1.514	45	2.603	45	3.182	34	6.919	23	51.707
33	1.996	11	1.514	33	2.606	33	3.397	30	6.926	20	51.814
45x	1.996	44	1.515	34	2.620	30	3.411	27	7.385	40	55.407
12	1.997	46	1.515	31	2.643	34	3.433	31	7.489	38	55.909
8	1.997	3	1.519	32	2.675	31	3.500	28	7.837	18	56.117
43	1.998	13	1.519	29	2.701	32	3.624	21	8.120	26	58.396
1	2.001	2	1.520	24	2.787	29	3.713	39	8.238	37	59.192
5	2.003	45	1.569	1	2.826	43	3.800	40	8.843	19	60.588
7	2.004	27	1.591	43	2.879	27	3.982	23	11.236	25	67.272
26	2.224	28	1.593	27	3.057	1	4.031	17	11.500	36	72.175
27	2.305	26	1.599	28	3.072	28	4.035	35	11.536	24	79.917
28	2.307	21	1.680	26	3.188	21	4.191	32	12.344	30	80.474
39	2.394	39	1.680	21	3.199	39	4.204	26	13.088	33	84.159
21	2.394	40	1.684	39	3.206	40	4.264	29	14.939	31	96.112
40	2.397	38	1.698	40	3.236	26	4.321	20	15.923	21	97.079
38	2.446	20	1.698	20	3.402	20	4.691	38	16.368	27	122.407
20	2.446			38	3.408	38	4.718	1	22.805	1	401.499

The first column is PTRP reduced to a simple non-spatial M/G/1 system. As predicted by the $c\mu$ results, the optimal policy selects the job to serve next by the class with the minimum expected service time. Adding in the travel times in calculating μ for the $c\mu$ rule improved the result in the second column as service time variance increased, but it is unclear whether this is significant. As predicted by theory, the optimal $c\mu$ policy is not exhaustive. The server is obligated to move on when a higher priority arrival enters the system. It is interesting to note that the policies that minimize E(s) times the arrival (rho) rate also perform well. The "control" priority strategies selecting priorities by the maximum expected service time performed badly as theory would expect: falling at the bottom of the list. In fact, the routing policies, though showing high average system times, were significantly better than these control strategies.

As travel times increase, (with speed equal to 40,000 distance units per time unit), the $c\mu$ policies continue to outperform all other policies, but by a smaller margin. Between 40,000 and 20,000 (maximum travel times .025 and .050), the routing policies and the $c\mu$ disciplines switch place but remain within the same magnitude of each other. Notice that at speed 20,000 the information based strategies (2,3) perform as well as the routing strategies. In fact these policies remain strong even as speeds decrease further. It is noteworthy that the vicinity policies, with the subsets set at 12 (one half the number of total nodes), performed worse at all speeds than both nearest neighbor and the longest queue strategies. Apparently, with so little differentiation between subsets, the policy offers no savings in travel time to the server.

As speeds decrease to 10,000 and 5,000 the original $c\mu$ regimes lose all efficacy. The strategies and the original strategies perform equally as poorly. The "wrong" policies outperform the "right" policies in a number of cases.

The superior policy at travel times of .1 and .5 (speeds 10,000 and 5,000) were the nearest neighbor followed by the TSP strategies. Of these strategies the exhaustive versions or the versions with high beta (the maximum number of jobs the server can take on at one node without moving on) had the lowest mean system times.[7] This result confirms the analysis in the previous chapters. Notice that the non-optimal TSP route strategy that uses an arbitrary circuit (policy 45x) still does much better than the TSP policies that use "one job gating" (8 and 43).

An inspection of gated and exhaustive policies (2,3), (9,10), (4,5),(18,19),(30,31) and (44,45) will show that as travel times become significant, the latter consistently have lower average times in system than the former. As discussed in relation to a SRPT disci-

7, The exhaustive and the beta gate policies with beta = 12 probably have close to identical paths. This is easy to see if one realizes that it is unlikely that an individual queue will remain a queue of size 12 but rarely. Therefore the server in the majority of cases is being sent on because the queue is exhausted not because the maximum number of jobs has been served.

plines, this result is natural. If travel times dominate individual job service requirements, a policy that disfavors travel would reduce delays. An exhaustive policy does just that. If this is indeed the reason behind the superiority of exhaustive regimes, it seems obvious that a policy that idles the server at a node after exhausting its jobs may be better yet. Could there exist an "idling" function that permits the server to stay at an empty node if he expects that an arrival is forthcoming nearby?

The most significant results of this comparison are:

- the simulation performs as predicted by theory at low travel times;
- the optimal priority discipline (using the $c\mu$ rule) loses all application at significant travel times. The use of this rule will lead to delays ten times greater than the NN policy at travel times roughly equivalent to the mean service time;
- the NN and TSP strategies manage to keep the increase in delays to a factor of two as travel times go from .025 to .5 for the maximum distance possible on the network. If one had to chose a dynamic strategy in a network with variable travel times, NN, TSP as well as the longest queue policies would provide the best performance. The $c\mu$ disciplines deteriorate too rapidly both absolutely and relatively.

4.5 More Evidence that Complete Exhaustive Policies are Superior

The following table shows a test of the NN policy as the maximum number of jobs the server can do at any one stay at a node is increased. It shows the best policy is to obligate the server to serve as many jobs as possible at a node without being idle. Notice that at a certain point, average times in system bottom out. This level is equivalent to the delay an exhaustive policy would give. The beta policy mandates that the server move on either when the queue is empty or when beta jobs have been served. As beta increases the queue must get longer for the latter clause to apply. At some point the probability the queue gets long enough becomes negligible so that the server will always exhaust the queue before serving beta jobs.

4.5.1 Effect of Beta on Times in System

Nodes	24
Strategy	6
lambda i	0.0960
E(s)	0.3872
Var(s)	0.0004
lambda	2.3037
rho	0.8919

Speed	Beta	Times in System
10000	1	3.9318
10000	2	3.3967
10000	3	3.3091
10000	4	3.2451
10000	5	3.2657
10000	6	3.2670
10000	7	3.2662
10000	8	3.2634
10000	9	3.2598
10000	10	3.2591
10000	12	3.2572
10000	14	3.2572
10000	16	3.2572
10000	18	3.2572
10000	20	3.2572

4.6 Comparison of Policies by Variance of Time in System

As predicted by theory system-wide FIFO (SFIFO) has the lowest variance when PTRP has effectively zero travel time. Yet as travel times increase, its variance increases to the highest on the list. This is intuitive when one realizes that at low travel speeds SFIFO essentially routes the server to a random node after each service. The variance of TSP is best or second best in all the speeds tested. In fact the exhaustive TSP policy creates the least variance except when SFIFO is applicable. Even in this case, the difference between SFIFO and exhaustive TSP are not large. The NN and the information based policies also outperform most of the priority disciplines across all travel times. This empirical result (if it holds under different E(s) and Var(s) makes the routing policies a good choice on networks with uncertain travel times. An interesting project would be to analyze these policies on a dynamic probabilistic network.

4.6.1 Table: Comparison of Strategies by Variance of Time in System

VARIANCE(TIME IN SYSTEM)

ORDER FROM BEST TO WORST OF 47 STRATEGIES UNDER VARIOUS TRAVEL TIMES E(s) Var(s)

Nodes 24												
Speeds:		2E+07		2E+07		40000		20000		10000	Dist	1000 5000
lambd i		0.096		0.098		0.096		0.096		0.096		0.096
E(S)		0.387		0.365		0.387		0.387		0.387		0.387
Var(S)		0.020		0.039		0.020		0.020		0.020		0.020
Lambda		2.304		2.352		2.304		2.304		2.304		2.304
Rho		0.892		0.859		0.892		0.892		0.892		0.892
	1	2.538	1	1.518	10	5.298	10	6.707	10	10.229	10	17.989
	10	3.840	10	2.261	11	5.434	11	6.887	11	10.459	11	18.503
	44	3.841	45	2.314	9	5.434	45	6.887	45	12.246	9	18.503
	11	4.047	11	2.314	1	5.897	44	8.097	44	12.539	45	21.013
	9	4.047	9	2.314	46	5.959	46	8.340	46	12.539	46	22.148
	45	4.048	43	2.314	44	5.959	4	8.340	6	18.317	44	22.148
	43	4.048	44	2.614	45	6.000	6	11.017	4	18.317	6	30.673
	12	4.841	12	2.691	45x	6.954	5	11.069	5	18.334	4	30.686
	8	4.841	8	2.691	4	8.020	45x	11.246	45x	20.763	5	45.667
	42	4.842	42	2.691	6	8.020	3	11.246	3	27.593	45x	55.686
	4	6.310	5	3.426	5	8.323	2	12.548	2	28.319	2	64.748
	6	6.310	6	3.681	8	8.692	9	14.200	9	29.237	3	70.550
	15	6.901	4	3.681	12	8.692	12	14.200	8	29.327	15	165.751
	5	6.984	15	3.898	43	11.193	8	15.369	12	29.328	13	165.893
	13	7.617	7	3.977	15	11.532	15	16.117	15	38.113	8	272.447
	7	8.051	13	4.008	3	11.753	13	16.644	13	38.237	12	272.447

3	8.125	3	4.303	2	12.259	7	18.119	7	41.728	16	351.248
2	8.474	2	4.385	13	12.477	43	18.836	43	51.380	42	351.248
16	8.870	14	4.520	16	14.735	16	19.092	42	54.907	7	529.173
41	8.870	16	4.941	42	14.735	42	21.246	16	54.909	41	1337.5
24	11.935	41	4.941	7	15.420	1	24.811	1	74.311	43	2652.7
25	12.090	18	5.648	41	25.313	41	24.811	41	149.32	34	11674.8
36	12.333	36	5.648	36	26.195	24	49.519	24	213.64	1	17272.4
18	12.333	19	5.724	18	26.215	18	49.757	18	234.14	39	18208.4
19	12.435	37	5.724	24	26.360	36	51.365	36	239.50	28	21069.9
37	12.435	24	5.738	19	28.030	25	54.807	25	249.47	40	36222.3
17	16.540	30	5.984	37	28.262	37	56.010	37	270.68	18	41677.7
35	16.541	31	5.987	25	28.670	19	56.047	19	274.08	37	45141.0
23	16.724	35	7.390	17	44.583	30	56.090	30	351.17	19	45565.6
33	18.687	23	7.402	23	44.640	33	88.167	33	356.98	25	57115.3
34	18.930	29	7.619	35	45.293	34	95.051	34	393.81	32	62370.4
30	19.368	33	7.779	33	46.622	31	100.67	31	422.05	36	63870.9
31	19.377	34	7.876	30	46.892	27	101.24	27	455.08	24	64573.3
32	24.630	17	7.899	34	47.073	28	103.19	28	513.95	30	70335.0
29	25.410	32	8.978	31	49.970	39	103.25	39	581.84	23	82513.8
28	28.200	28	10.502	32	68.778	21	104.17	21	595.07	35	88212.3
27	28.206	27	10.563	28	79.980	40	147.72	40	672.61	17	90004.4
39	29.611	21	11.278	27	80.567	23	148.45	23	2078.6	29	92109.0
21	29.611	39	11.278	21	89.517	17	156.5	17	2233	31	97659
40	29.645	40	11.336	39	89.868	35	159.0	35	2248	33	100174
26	33.660	26	11.834	40	91.856	26	160.2	26	2723	20	104478
38	41.089	38	12.597	26	101.36	32	166.1	32	3236	38	118822
20	41.089	20	12.597	20	119.69	29	194.9	29	3415	26	123527
		25	15.783	38	120.36	20	240.0	20	3639	21	128070
				29	783.37	38	245.5	38	3827	27	235358

4.7 Comparison of TSP and Nearest Neighbor Policies

The following tables present simulation results for NN and TSP under a larger selection of travel times and network sizes. These results are presented in graphical form at the end of the chapter.

4.7.1 Nearest Neighbor 24 Nodes

NEAREST NEIGHBOR Nodes 24					
Lambda i	0.098	0.096	0.099	0.097	0.102
E(S)	0.365	0.387	0.353	0.345	0.400
Var(S)	0.039	0.020	0.033	0.017	0.041
Lambda	2.352	2.304	2.385	2.333	2.452
Rho	0.859	0.892	0.841	0.804	0.980
Speed	Average Times In System				
1000	23.303	29.641	19.316	14.501	63.632
2000	9.195	11.708	7.780	5.773	31.048
4000	4.153	5.669	3.966	2.898	14.332
6000	2.950	4.217	2.953	2.163	11.094
8000	2.517	3.593	2.504	1.853	9.707
10000	2.280	3.257	2.265	1.669	8.384
12000	2.112	3.034	2.123	1.558	8.073
14000	2.002	2.886	2.021	1.480	7.326
16000	1.931	2.762	1.943	1.428	6.902
18000	1.873	2.672	1.881	1.388	6.628

4.7.2 TSP 24 Nodes

MODIFIED TSP POLICY Nodes 24					
Lambda i	0.098	0.096	0.099	0.097	0.102
E(S)	0.365	0.387	0.353	0.345	0.400
Var(S)	0.039	0.020	0.033	0.017	0.041
Lambda	2.352	2.304	2.385	2.333	2.452
Rho	0.859	0.892	0.841	0.804	0.980
Speed	Average Times In System				
1000	17.306	21.125	15.189	11.668	40.411
2000	8.458	10.743	7.751	5.748	23.678
4000	4.323	5.735	4.084	3.036	13.821
6000	3.162	4.275	3.046	2.268	11.026
8000	2.635	3.682	2.588	1.921	9.700
10000	2.378	3.322	2.325	1.720	8.817
12000	2.196	3.102	2.156	1.601	8.204
14000	2.077	2.953	2.044	1.507	7.755
16000	1.988	2.829	1.962	1.455	7.399
18000	1.924	2.723	1.904	1.408	7.216

4.7.3 Nearest Neighbor 48 Nodes

NEAREST NEIGHBOR Nodes 48					
Lambda i	0.102	0.096	0.095	0.102	0.101
E(S)	0.188	0.185	0.166	0.182	0.192
Var(S)	0.039	0.009	0.017	0.009	0.021
Lambda	4.886	4.618	4.569	4.902	4.828
Rho	0.918	0.857	0.761	0.890	0.929
Speed	Average Times In System				
1000	57.363	72.463	23.170	44.136	91.619
2000	27.314	30.609	8.812	15.057	45.953
4000	11.498	14.873	3.365	5.899	20.626
6000	6.737	7.924	2.047	3.520	13.869
8000	4.743	5.204	1.504	2.474	10.940
10000	3.830	3.882	1.237	2.016	8.764
12000	3.278	3.160	1.077	1.739	7.651
14000	2.929	2.743	0.986	1.552	6.711
16000	2.647	2.446	0.911	1.423	6.144
18000	2.447	2.269	0.861	1.344	5.667
20000000	1.367	1.223	0.542	0.790	3.286

4.7.4 TSP 48 Nodes

MODIFIED TSP POLICY Nodes 48					
Lambda i	0.102	0.096	0.095	0.102	0.101
E(S)	0.188	0.185	0.166	0.182	0.192
Var(S)	0.039	0.009	0.017	0.009	0.021
Lambda	4.886	4.618	4.569	4.902	4.828
Rho	0.918	0.857	0.761	0.890	0.929
Speed	Average Times In System				
1000	32.758	37.726	15.670	23.539	46.611
2000	17.570	19.598	7.615	11.492	26.640
4000	9.157	9.854	3.469	5.486	15.309
6000	6.122	6.492	2.237	3.541	11.273
8000	4.619	4.788	1.673	2.613	9.016
10000	3.817	3.825	1.364	2.147	7.714
12000	3.031	3.223	1.168	1.861	6.882
14000	2.947	2.872	1.054	1.670	6.281
16000	2.702	2.599	0.912	1.540	5.877
18000	2.521	2.378	23.539	1.433	5.491
20000000	1.353	1.225	0.537	0.784	3.268

These results betray the weakness of NN as travel times grow beyond expected service times. NN and TSP perform almost exactly as well with NN showing slightly lower system times above speed 5000. Yet as travel times approach about double the average service times, NN and TSP begin to blow up with NN performing significantly worse. The instability of NN may be explained by increasing probability that as travel times increase, NN will trap the server in a cluster of nodes ignoring the rest of the network.

In all, TSP with shortcuts is the most robust policy tested. It has the following advantageous features:

- it has either the lowest or second lowest variance across all travel times,
- it performs only slightly worse than NN in the higher speeds and much better as speeds drop off and travel times to mean service times,
- other than figuring out the route beforehand TSP is frugal with its information needs.

The TSP without shortcuts is also a robust policy and due to its reduced information needs may be the best choice for some probabilistic repairman management problems. The key variable will be the cost of obtaining information on whether the next node is empty or not, without physically visiting it.

4.8 Other Results

In Chapter 2.3.5, the Bertsekas Gallager result for a routing strategy on a network with uniform expected service requirements and arrival rates was discussed. The

TSP without Shortcut strategy analyzed in this paper is equivalent when applied to a network filling the B-G specifications. In order to investigate the effect of shortcuts on the applicability of their result the following results of simulations were tabulated. In the third column (Avg. Delay) the results of the simulation are given. In the fourth column (Predicted E(D)) the delay predicted by B-G are given using the expected service time (E(s)), arrival rate (lambda) and variance of service times (Var(s)). The next column presents the percent difference between the predicted and actual results. The first table presents the result for exhaustive strategies and the second for gated strategies.

4.8.1 Comparison of Predictions using Bertsekas and Gallager Exhaustive Formula with Simulation Results

.

Exhaustive Service Strategies

Strategy	Speed	Node	Avg. Delay	Predict. E(D)	% diff	Cycle Time	E(S)	lambd	var(S)
at finite speed compare with Bertsekas and Gallager									
44	2000	48	14.085	13.423	4.7%	2.312	0.189	4.80	0.0001
44	2000	48	14.419	13.548	6.0%	2.335	0.189	4.80	0.0001
44	2000	48	18.034	17.167	4.8%	3.006	0.189	4.80	0.0001
44	2000	48	74.601	79.040	-6.0%	14.475	0.189	4.80	0.0001
permitting shortcuts									
9	10000	18	6.436	7.095	-10.2%	0.8703	0.5050	1.80	0.0026
9	5000	36	16.048	19.335	-20.5%	3.3680	0.2525	3.60	0.0006
9	10000	12	6.793	6.431	5.3%	0.5124	0.7575	1.20	0.0057
9	10000	48	11.208	14.003	-24.9%	2.4184	0.1893	4.80	0.0004
9	5000	6	9.622	10.084	-4.8%	0.5212	1.5151	0.60	0.0230
9	10000	6	8.4	8.868	-5.6%	0.2606	1.5151	0.60	0.0230
9	10000	24	6.451	9.360	-45.1%	1.4073	0.3787	2.40	0.0014
9	5000	48	24.449	27.049	-10.6%	4.8367	0.1893	4.80	0.0004
9	5000	12	8.969	9.036	-0.7%	1.0248	0.7575	1.20	0.0057
9	5000	24	14.594	16.807	-15.2%	2.8145	0.3787	2.40	0.0014
9	5000	18	11.556	11.640	-0.7%	1.7406	0.5050	1.80	0.0026

4.8.2 Comparison of Predictions using Bertsekas and Gallager Gated Formula with Simulation Results

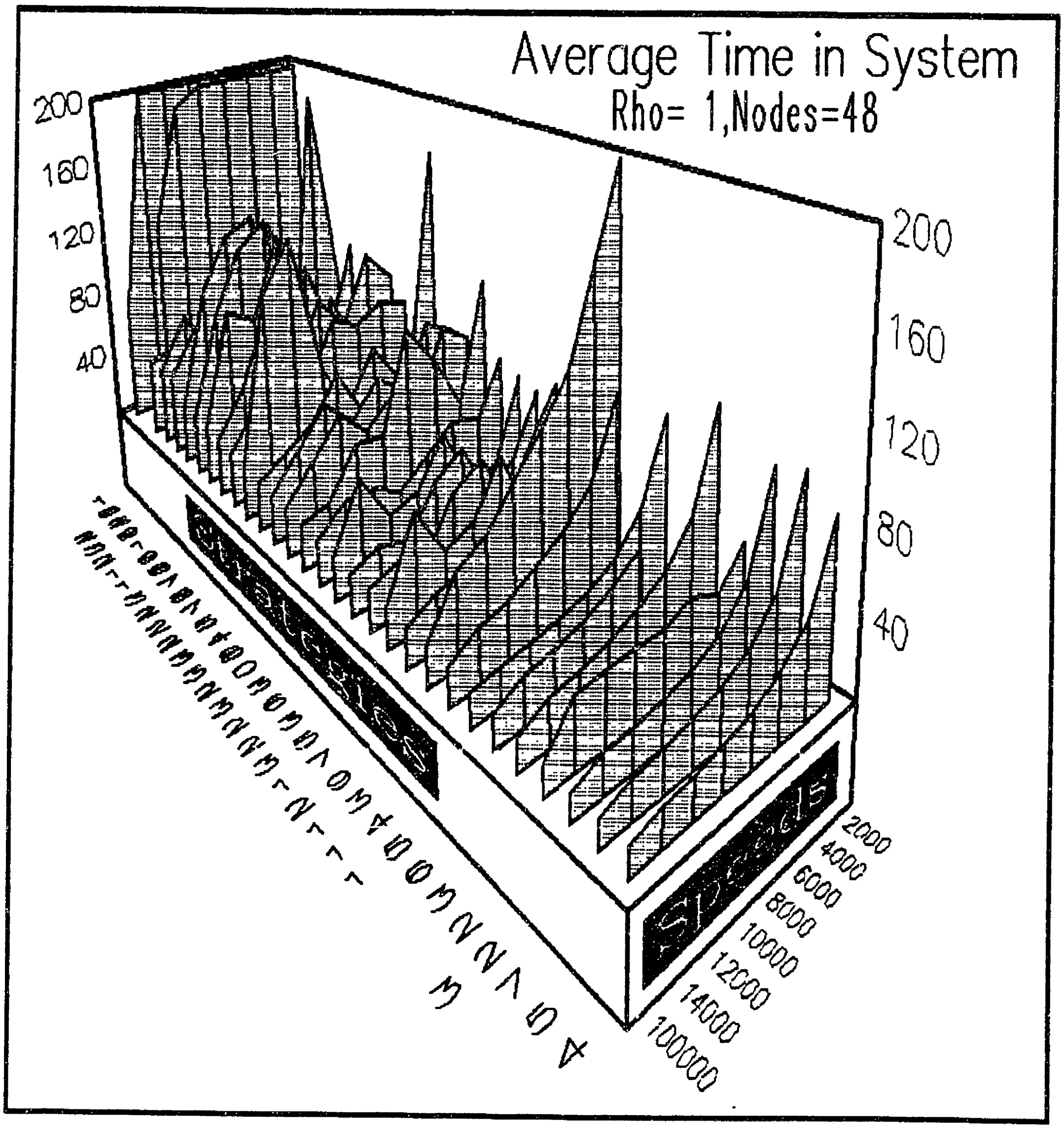
Gated Service Strategies
========================

Strategy	Speed	Node	Avg. Delay	Predicted E(D)	% diff	Cycle Time	E(S)	lambd	var(S)
compare with Bertsekas and Gallager Results									
45	2000	48	14.770	13.953	5.5%	2.312	0.189	4.80	0.0001
45	2000	48	14.855	14.083	5.2%	2.335	0.189	4.80	0.0001
45	2000	48	18.793	17.855	5.0%	3.006	0.189	4.80	0.0001
45	2000	48	79.265	82.357	-3.9%	14.475	0.189	4.80	0.0001
permitting shortcuts									
10	10000	24	6.847	10.005	-46.1%	1.4073	0.3787	2.40	0.0014
10	5000	12	9.654	9.975	-3.3%	1.0248	0.7575	1.20	0.0057
10	10000	48	11.782	14.557	-23.5%	2.4184	0.1893	4.80	0.0004
10	5000	36	17.089	20.365	-19.2%	3.3680	0.2525	3.60	0.0006
10	5000	18	12.667	12.704	-0.3%	1.7406	0.5050	1.80	0.0026
10	10000	12	7.106	6.901	2.9%	0.5124	0.7575	1.20	0.0057
10	5000	24	15.796	18.097	-14.6%	2.8145	0.3787	2.40	0.0014
10	10000	6	9.088	9.345	-2.8%	0.2606	1.5151	0.60	0.0230
10	10000	18	6.85	7.627	-11.3%	0.8703	0.5050	1.80	0.0026
10	5000	6	10.155	11.039	-8.7%	0.5212	1.5151	0.60	0.0230
10	5000	48	25.472	28.157	-10.5%	4.8367	0.1893	4.80	0.0004
10	10000	36	6.596	10.820	-64.0%	1.6840	0.2525	3.60	0.0006

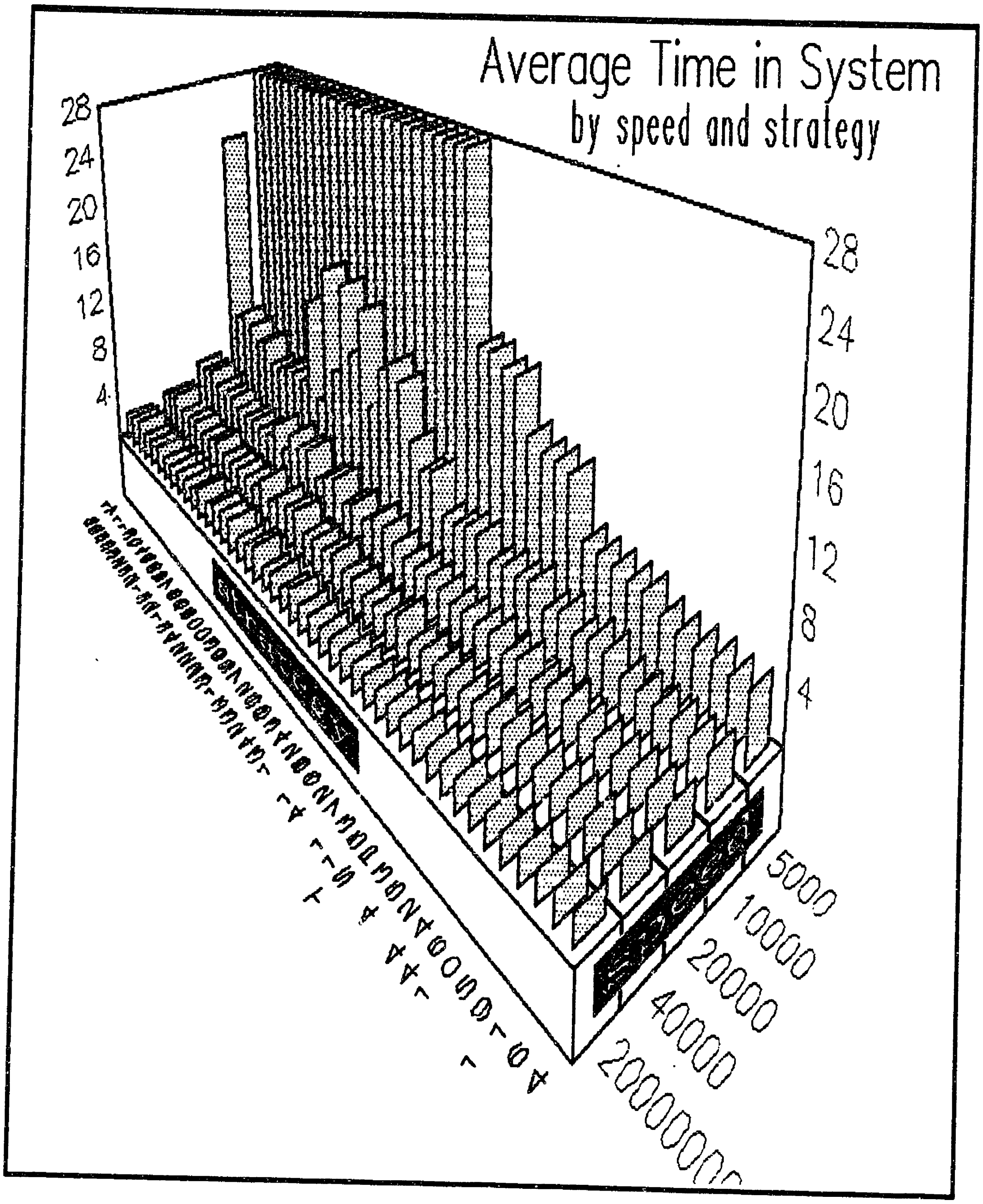
In all, the simulation performed close to the predicted values for the TSP policy without shortcuts. It is possible that longer simulations would eliminate the apparent bias that does exist. Yet it is clear that the formulas are inappropriate for approximating the TSP with shortcut policy.

4.9 Graphs

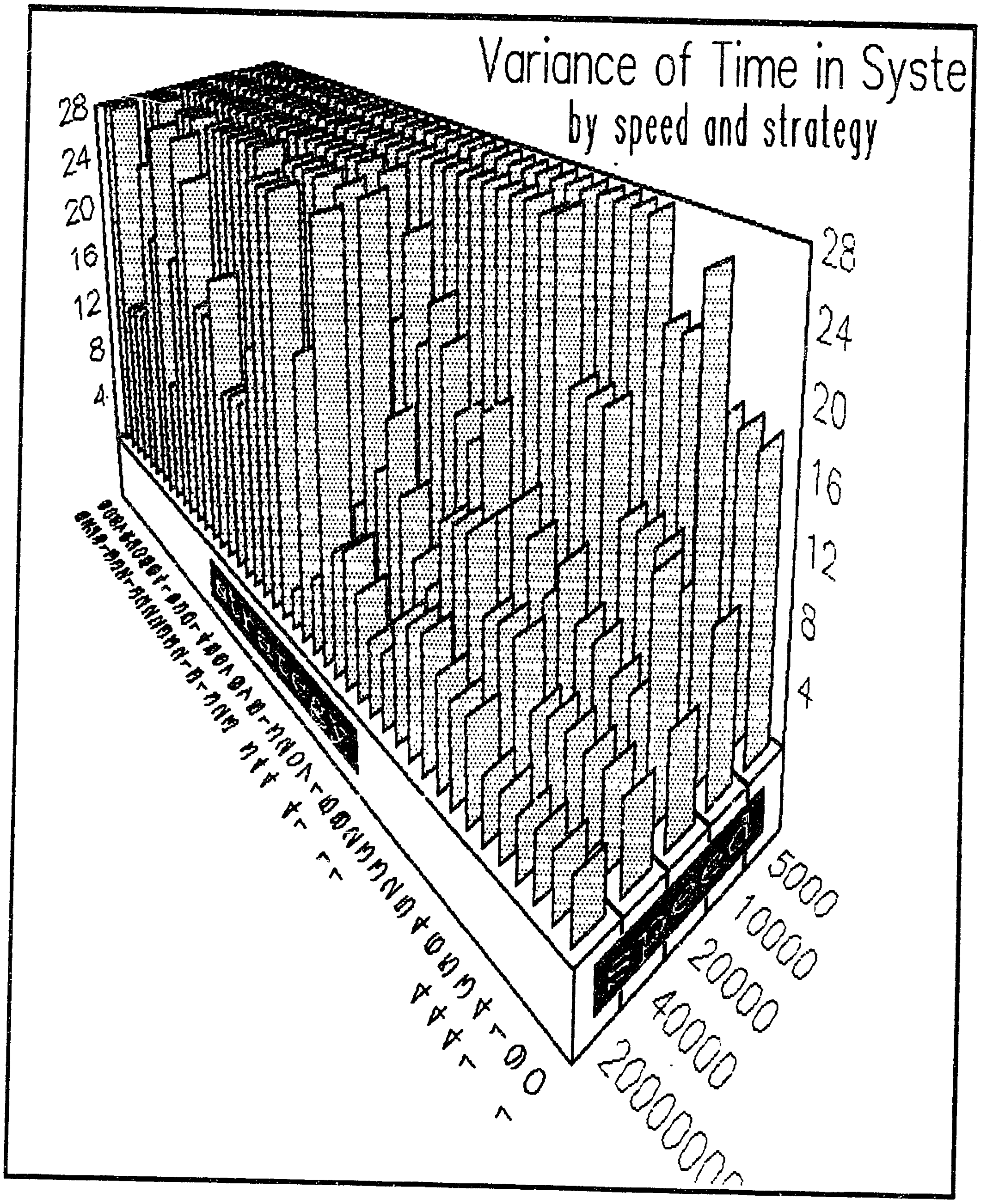
4.9.1 Effect of Increased Travel Times on Time in System (Rho = 1)



4.9.2 Effect of Increased Travel Times on Time in System (Rho < 1)

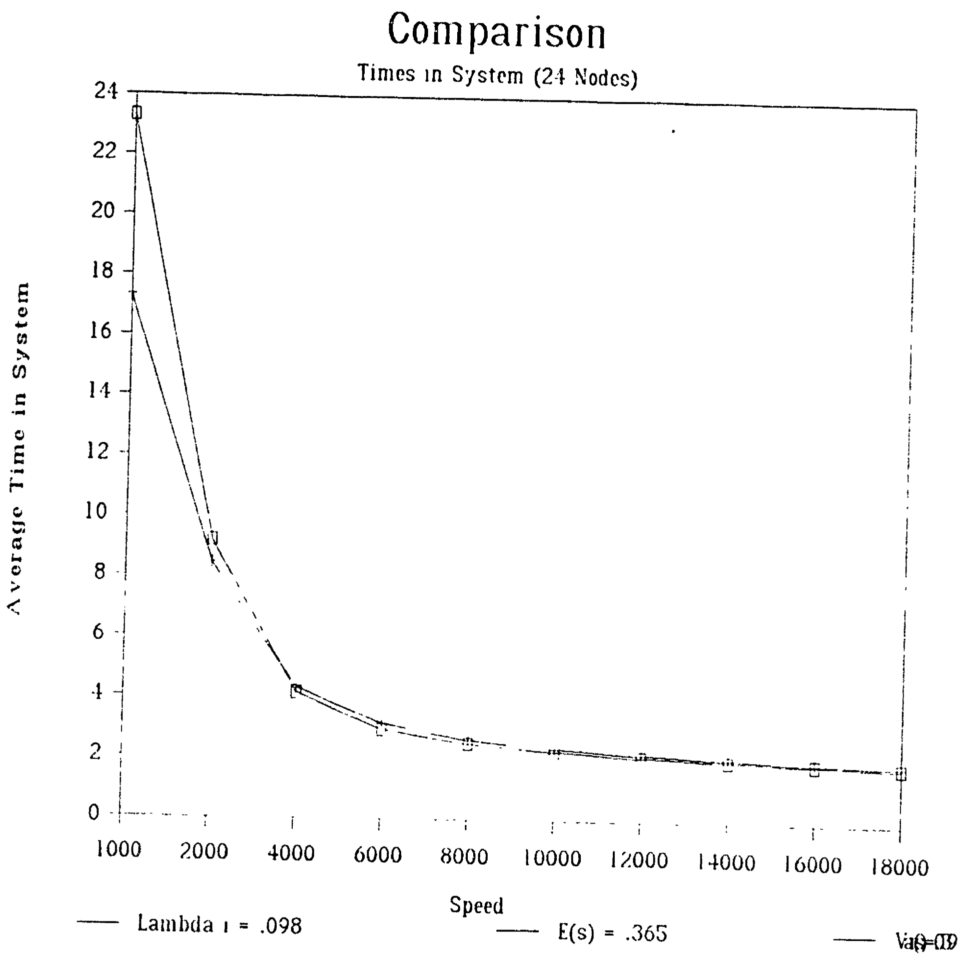


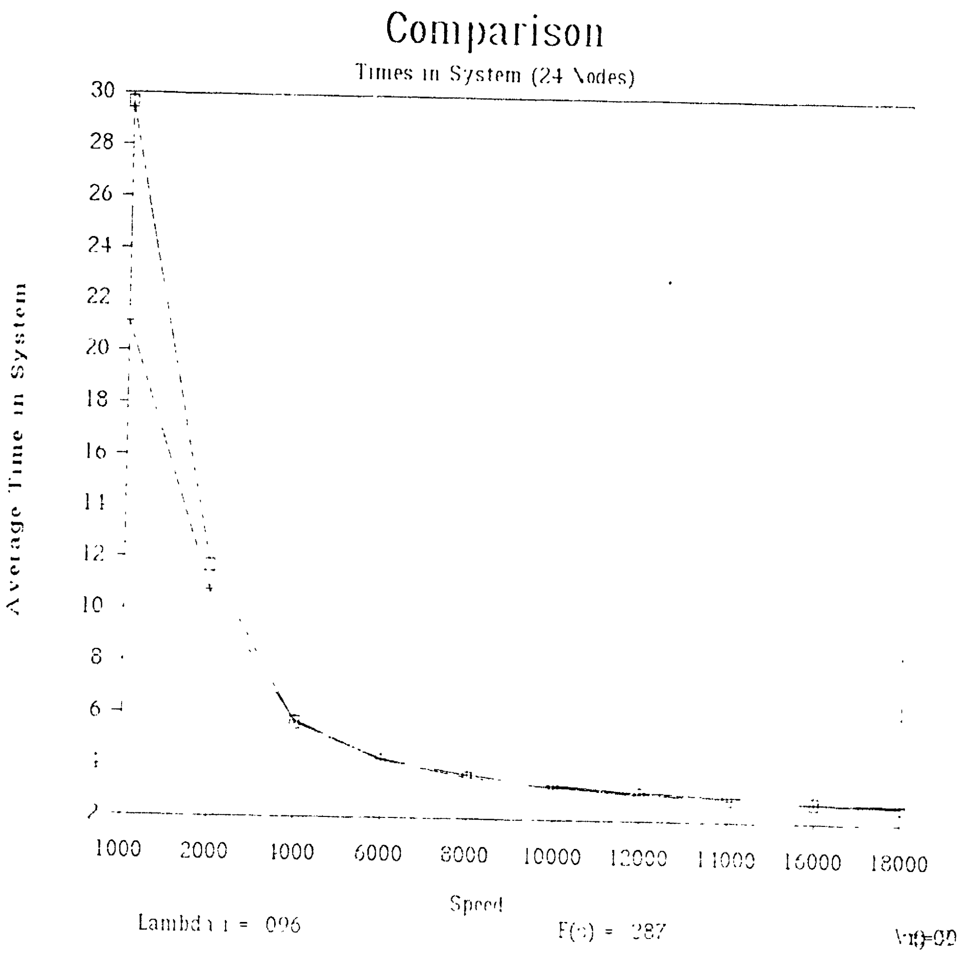
4.9.3 Effect of Increased Travel Times on Variance of Time in System (Rho < 1)



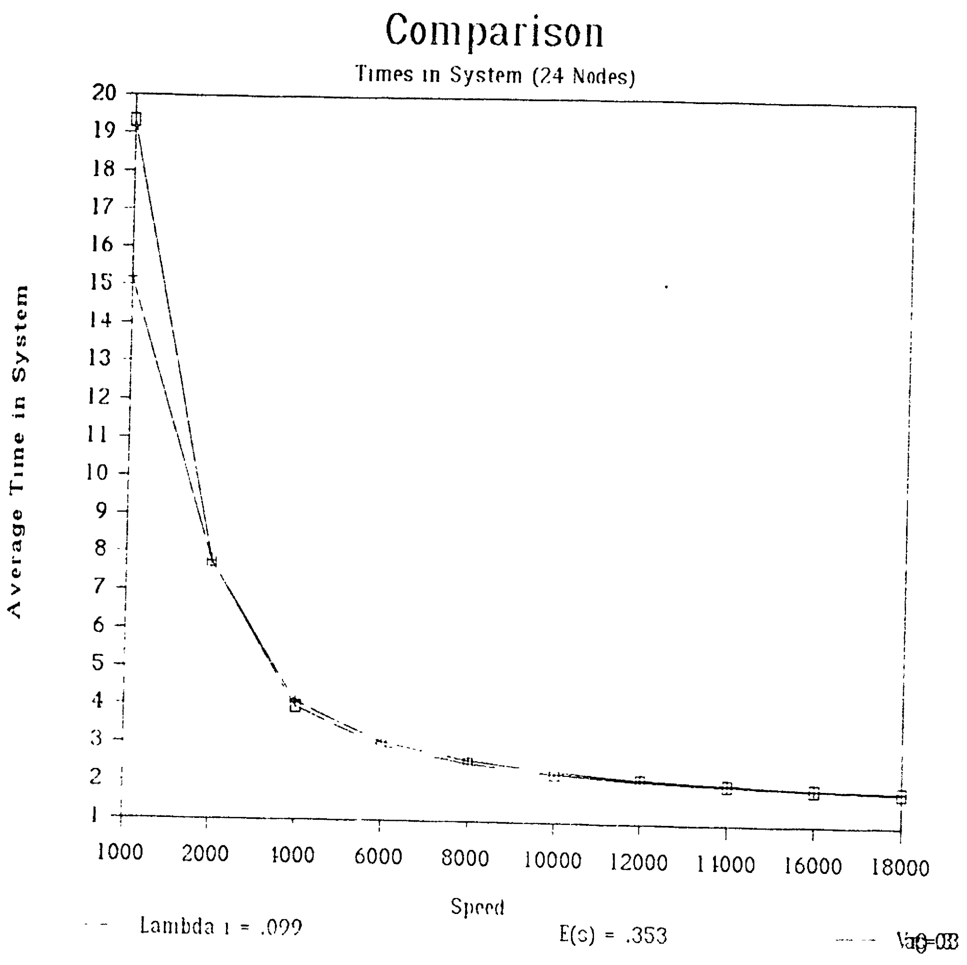
4.9.4 Comparison of NN and TSP: TSP Superior at Low Speeds:

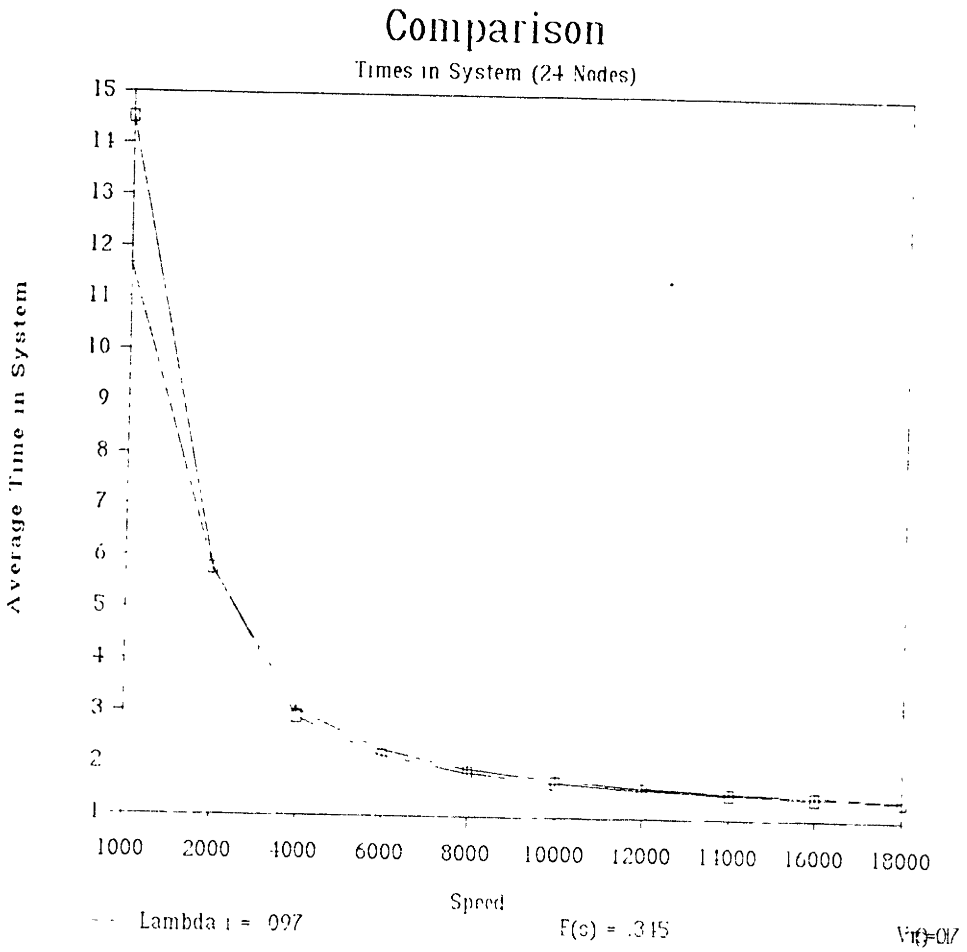
"+" for TSP, square for Nearest Neighbor.

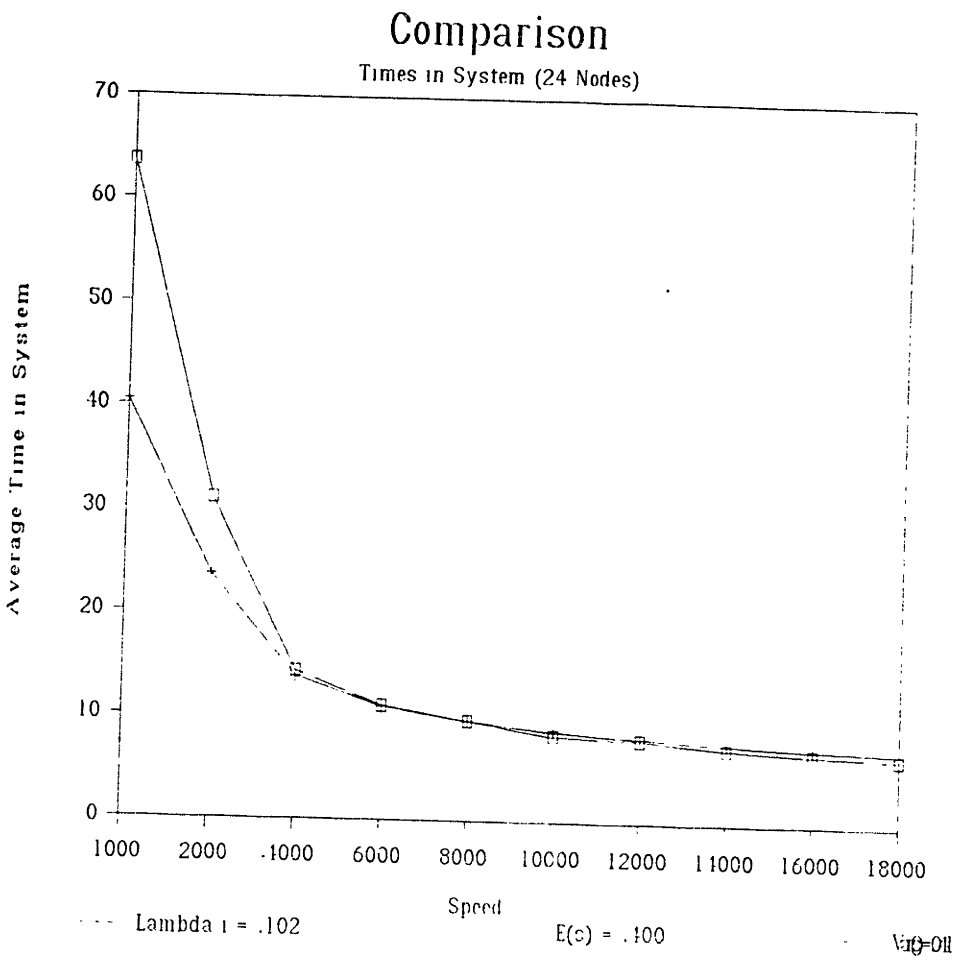


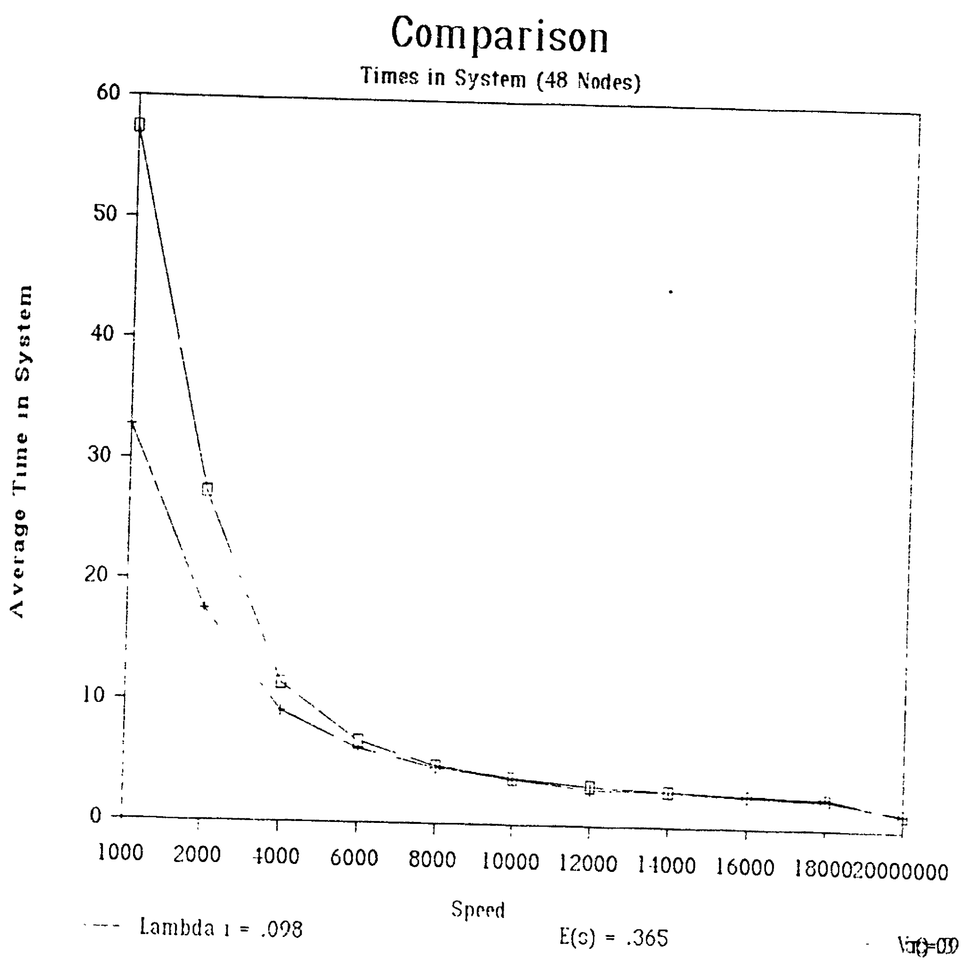


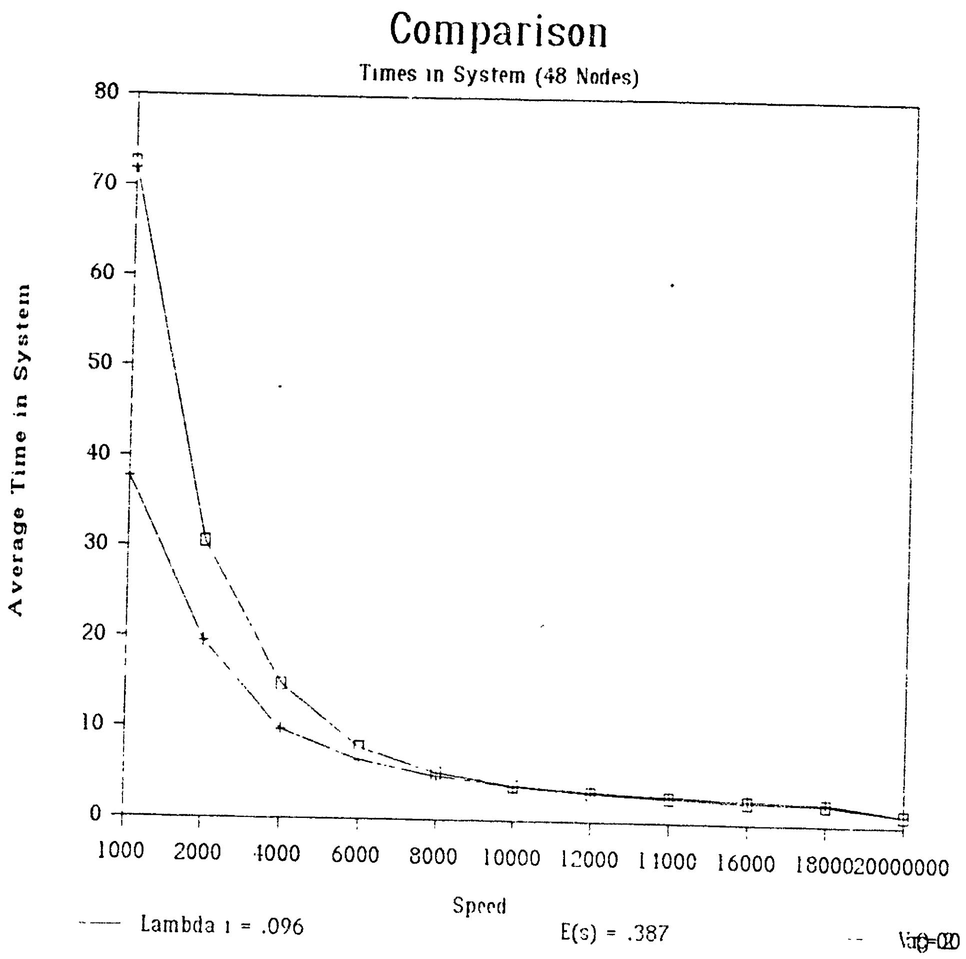
Comparison
Times in System (24 Nodes)
Average Time in System
30
28
26
24
22
20
18
16
14
12
10
8
6
4
2
1000
2000
4000
6000
8000
10000
12000
14000
16000
18000
Speed
Lambda = .096
F(s) = .287
Var=.00

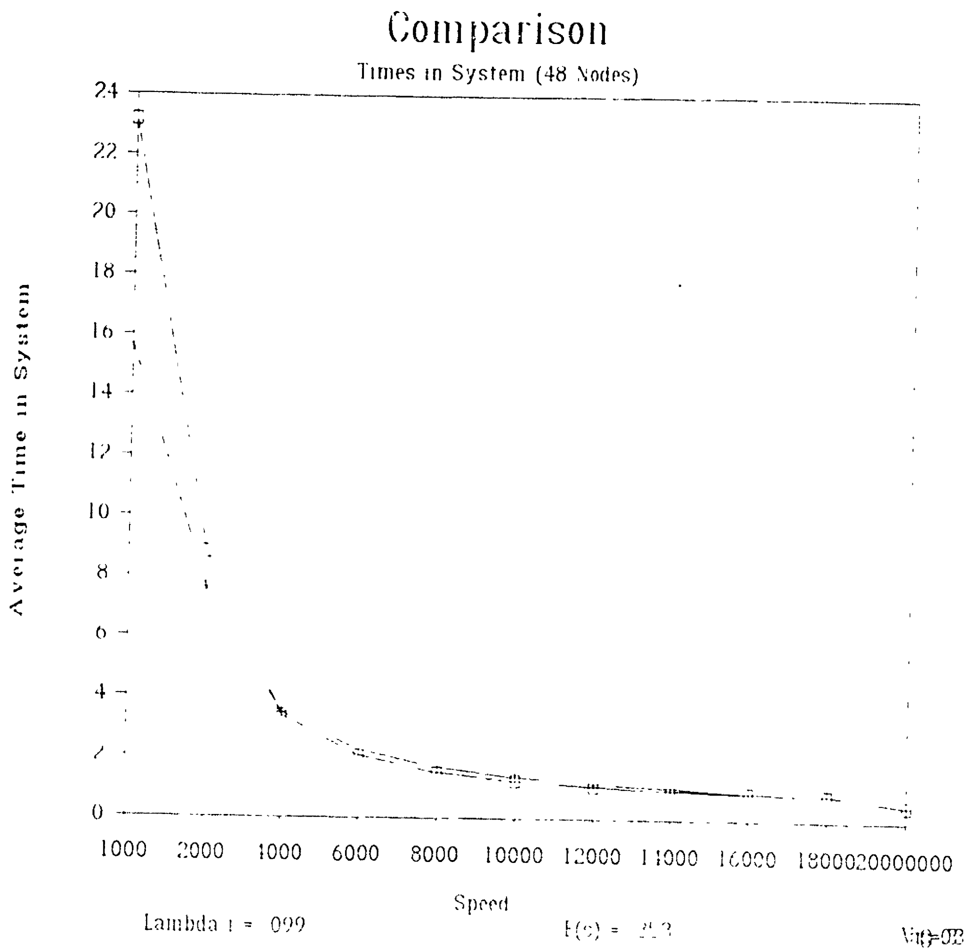

Comparison
Times in System (24 Nodes)
Average Time in System
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
1000
2000
4000
6000
8000
10000
12000
14000
16000
18000
Speed
Lambda i = .099
E(s) = .353
Var=.03


Comparison
Times in System (24 Nodes)
Average Time in System
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
1000
2000
4000
6000
8000
10000
12000
14000
16000
18000
Speed
Lambda i = .097
F(s) = .345

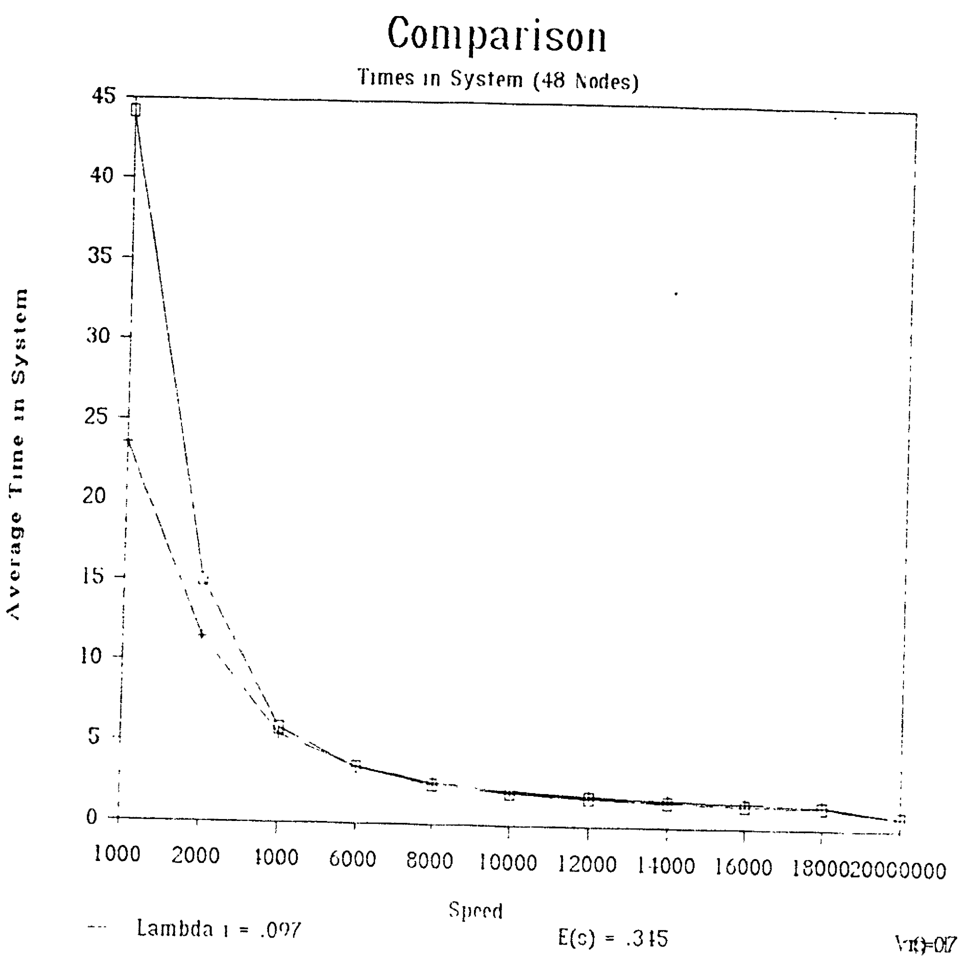

Comparison
Times in System (24 Nodes)
Average Time in System
70
60
50
40
30
20
10
0
1000
2000
4000
6000
8000
10000
12000
14000
16000
18000
Speed
Lambda i = .102
E(s) = .100
Var(s)=0


Comparison
Times in System (48 Nodes)
Average Time in System
60
50
40
30
20
10
0
1000
2000
4000
6000
8000
10000
12000
14000
16000
1800020000000
Speed
Lambda 1 = .098
E(s) = .365
Var()=.09

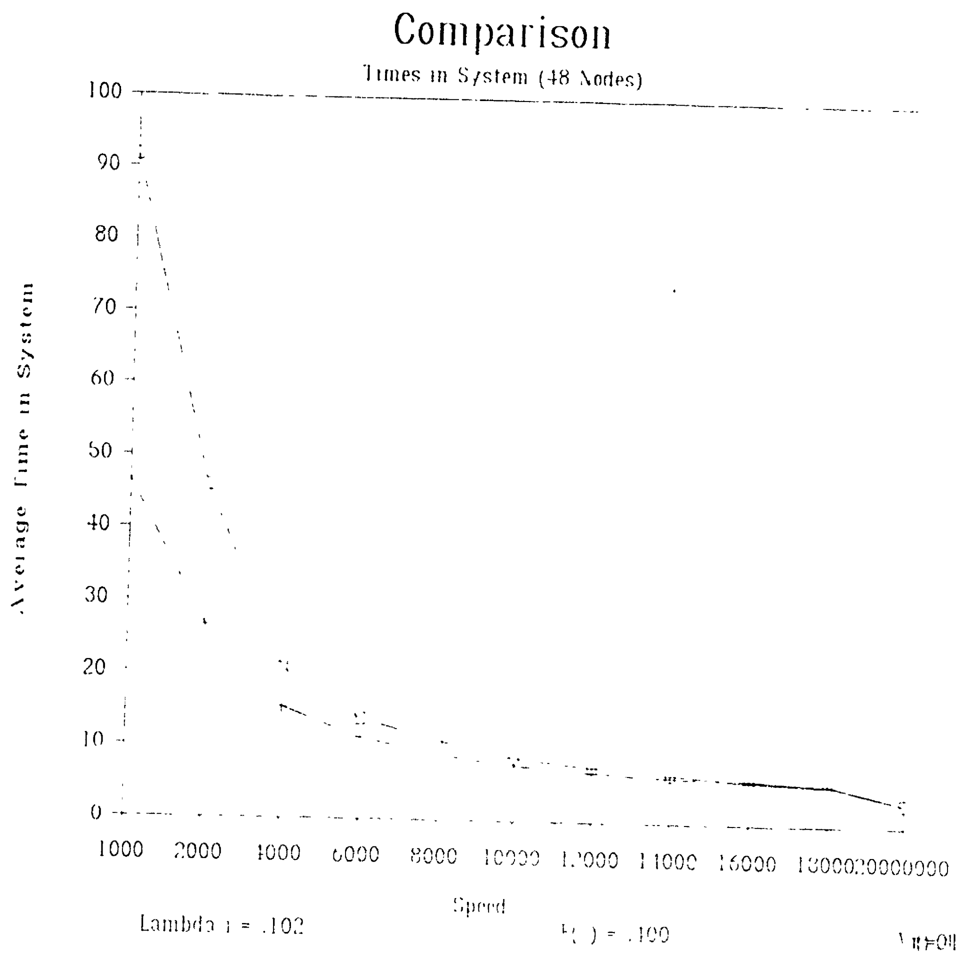

Comparison
Times in System (48 Nodes)
Average Time in System
80
70
60
50
40
30
20
10
0
1000
2000
4000
6000
8000
10000
12000
14000
16000
18000
20000
Speed
Lambda 1 = .096
E(s) = .387
Var=0



Comparison
Times in System (48 Nodes)
Average Time in System
24
22
20
18
16
14
12
10
8
6
4
2
0
1000
2000
4000
6000
8000
10000
12000
14000
16000
18000
20000000
Speed
Lambda i = .099
E(s) = 2.3
Var(s)=.033



Comparison
Times in System (48 Nodes)
Average Time in System
45
40
35
30
25
20
15
10
5
0
1000
2000
4000
6000
8000
10000
12000
14000
16000
18000
20000
00000
Speed
Lambda i = .097
E(s) = .345
V(s)=.07


Comparison
Times in System (48 Nodes)
Average Time in System
100
90
80
70
60
50
40
30
20
10
0
1000
2000
4000
6000
8000
10000
12000
14000
16000
18000
20000000
Speed
Lambda i = .102
E( ) = .100

5 Conclusions

The major result of this study has been to confirm the superiority of TSP and the Nearest Neighbor Policies. The coincident superiority of an exhaustive regime plays well into the conjecture that NN and TSP in fact behave like SRPT disciplines and therefore approach optimality. The divergence of NN and TSP at high travel times is more difficult to explain.

The simulations point out questions that need to be investigated:

- where do the $c\mu$ policies and the routing policies perform equally well?
- is there a predictive formula for the Nearest Neighbor policy?
- can NN be improved by the use of more system information?
- does the behavior of these strategies change when used on non-planar networks?
- and of greatest interest to the author, can NN or TSP be proven optimal, and, if so, at what relative service, arrival and travel times?

The greatest weakness of this study is the assumption that real world servers will be concerned about average time in system. Average time in system does not take into account that jobs that remain in queue longer become relatively more expensive, (for instance, as customers become increasingly upset). Moreover, in many cases the customers will be concerned with variance of service. In this case, the thesis would point to the TSP with shortcut policy as the best choice. But if customers are concerned with their total wait and their time costs are non-linear this thesis' results provide no help. The fact that SPRT results are strongly dependant on linear time costs make the applicability of Nearest Neighbor Strategies and especially exhaustive disciplines doubtful. Fortunately, the simulation package, included in the appendix, is easily modified to account for non-linear time costs. In this area I believe this thesis could be the most useful.

6 Appendix A: Description of Purpose and Use of Simulation Programs

6.1 Purpose of Programs Listed

The programs included in this appendix were written to simulate a single server queuing system with multiple classes of customers with state independent and dependant service requirements. The dependency of service requirements on the state of the system is modeled on customers arriving and being served on nodes of a graph. Since travel time to the node is added to each customers service requirement, and since travel times are dependant on the location of the previous service there is a strong system dependency.
The difficulty in deriving a probability distribution for the service requirement (and of course service rate) make calculation of expected service parameters, such as expected time in system for the most recent arrival, a daunting task. Therefore, a simulation of this queuing system would be helpful in getting data from which to compare proposed analytical descriptions.

This simulation program specifically attempts to derive data for the time in system for a randomly arriving customer. As described in the text, we can use this data to judge the optimality of dynamic routing policies, to compare the graph dependant service time distribution as derived empirically with other space dependant queuing results.

6.2 Description of Simulation Package

The mechanics of the simulation can be best understood as a three part process. First of all, the package produces a graph (network) either randomly or deterministically, in the case of a circular graph (network). Second the package generates a poisson arrival process for each node in the graph. The parameter λcan be set equal or different as specified by the user. Although this is a straightforward process, the program **SIMM.C** is complicated by the necessity of sorting the arrivals in a time line and creating a linked list to facilitate the later use of the same data runs by a wide range of strategy simulations. At the third step the actual simulation occurs.

DIMTSP.C derives the distance matrices used. It should be noted that node adjacency matrices are not created since either it is assumed that the graphs are totally connected or that the server always travels by the shortest path and ignores nodes he must pass through to reach his intended destination. Indeed this limits the type of dynamic strategies that can be simulated by this package but modification to include node incidences would not be difficult. Three choices of graphs are available: a planar graph with information on a shortest path circuit, a non-planar totally connected graph which supplies the simulation with the distance for all nodes shortest paths, and a circular equilateral non-connected graph with a simple distance matrix. The last graph is the only graph created that is not generated randomly. The SIMM.C program produces the arrival time line that includes in every record the time of arrival, the customers service requirement, the customer's node of arrival, his or her absolute order of arrival (i.e. 45th to arrive), and finally the order tag of the last customer to arrive before him at the queue and the order tag of the customer to follow him. The latter tag is not used as information by the strategies simulated. It merely serves as a convenient way to limit file searches. Finally TIMM.C takes the data created by SIMM.C and DIMTSP.C and runs dynamic server routing strategies. TIMM.C outputs mean time in system as well the variance of time in system. Other statistics are readily available since TIMM.C keeps track of queue lengths, accumulated service requirements as well as service time devoted to travel.

The utility of this set of programs is hopefully the wide selection of parameters that a user can specify to investigate various scenarios under which a strategy can be tested.

6.2.1 DIMTSP.C: Network Data Creation

DIMTSP.C has three sets of options. One, the user can select the graph type as discussed above: planar, non-planar and circular. Second the user can select the number of nodes in the graph. Finally, for the circular network, the user can select the length of the arcs. In the planar graph the user controls the dimensions of a the square in which the nodes will be randomly placed. The nodes are located on the x and y axes with uniform distributions. The graph is created by connecting all

nodes in the area. As mentioned previously, the program calculates a minimum distance TSP circuit on this graph for use by route specific strategies in TIMM.C. The code used is taken directly from Numerical Recipes in C.[8] In the non-planar graph, the user specifies the maximum distance between any two nodes. The program in turn randomly generates actual distances uniformly from zero to the specified maximum distance. Due to the computational difficulty in finding TSP circuits on graphs that violate the triangular inequality, the program only specifies the shortest path distance between nodes. For the circular graph, the user specified distance is used as the length of arcs between adjacent nodes. It is important to note that average distances between nodes in the three graphs will not be the same. For the circular graph the average distance will equal to $\left(\frac{(1+2+3+......N)}{N}\right)\ x\ Unit\ Distance$ for graphs with even number of nodes. For the planar graph, where nodes are distributed uniformly on the x and y axes, expected distance is equal to
$\frac{1}{3}*(\ 2\ x\ length\ of\ side\ of\ square\ area)$.[9] The average distance between two nodes in the non-planar graph is determined by initial distances with expected distance equal to half the maximum distance specified by the user. The expected shortest path distances on non-euclidean graphs is complex and will not be discussed here. Since average distances will be important in the study of certain simulations, DIMTSP.C outputs a text file with the TSP route and the distance matrices.

The command line parameters are as follows:

8, W. Press et al.,(1988) Numerical Recipes in C, Cambridge University Press, pages 345-352.

9, R. Larson, A. Odoni (1981), Urban Operations Research, Prentice-Hall, New Jersey pages 126-127.

TIMM <arg1>, <arg2>, <arg3>, <arg4>, <arg5> where the arguments are defined below:	
<arg1>	the number of nodes in the graph.
<arg2>	the name of the file to hold the distance matrix to be used by TIMM.
<arg3>	the name of the file to hold the order of the TSP route to be used in certain TIMM strategy simulations.
<arg4>	'n', 'p' or 'c' for non-planar, planar, or circular graph. Note that <arg3> will be ignored if 'n' or 'c' are chosen.
<arg5>	The maximum distance between nodes, the distance between adjacent nodes, and the dimensions of the square holding the graph for 'n', 'c' and 'p' respectively.

6.2.2 SIMM.C: Arrival and Service Data Creation

SIMM.C completely determines the arrival and service parameters. In brief the user selects the arrival rate, the service rate (not the expected service requirement), the inverse of the standard deviation of the service requirement, the service distribution (either Gaussian or Poisson), and finally whether these parameters will be equal for every node or will be used as the mean values from which random parameters will be derived. No facility is coded to allow the user to select parameters for every node exactly, but this would be a straight forward procedure. The generation of random parameters uses Gaussian deviates where the user supplied values are used as the mean and the standard deviation is fixed at one fifth of the mean. The arrival times for a customer at a node is generated by adding a first order interarrival time to the time of the last arrival. Service requirements are generated as Gaussian deviates with the mean and standard deviation supplied by the user or calculated by the system. The Gaussian generator is taken from code supplied in

Numerical Recipes in C.[10] The exponential generator follows the approach described in Urban Operations Research.[11]

The command line parameters are as follows:

SIMM <arg1>, <arg2>, <arg3>, <arg4>, <arg5>, <arg6>, <arg7>, <arg8>, <arg9>, <arg10>, where the arguments are defined as follows:	
<u><arg1></u>:	the total nodes in the network (an even number).
<u><arg2></u>:	a work file that will be overwritten in the next run,
<u><arg3></u>:	the file holding the time line linked list that will be used by
<u><arg4></u>:	TIMM.C as the basis for its simulations. The file holding the parameters used in creating the arrivals in <arg2>. This file is used by TIMM in calculating rho as control variable for certain simulations.
<u><arg5></u>:	either 'r' or 'd' for telling the program whether the arrival and service numbers supplied in the next arguments are to be used to generate <u>r</u>andom parameters or used as pre-<u>d</u>etermined parameters for all the nodes.
<u><arg6></u>:	either 'g' or 'm' for Gaussian generated service requirements or exponential.
<u><arg7></u>:	'lambda' the <u>arrival rate at each node</u>, not the system as a whole.
<u><arg8></u>:	the <u>inverse</u> of the expected service requirements. It should be equal to at least λ_i times the number of nodes in the network.

10, W. Pressman et al., (1989), Numerical Recipes in C, Cambridge University Press, pages 214-217.

11, R. Larson, A. Odoni (1981), Urban Operations Research, Prentice-Hall, pages 488-489.

<arg9>:	the inverse of the standard deviation of the service requirement. It should be at least 5 times the size of the seventh argument to limit the dispersion of service requirements below zero. The program will permit the user to enter an unreasonable σ, but it will convert any negative service requirements to zero. The user should be aware of the conditioning of the normal p.d.f. that this implies.
<arg10>	the total bytes that the temporary file will take up. The file created as <arg2> will be about twice as large. The number of observations implied by this number will equal approximately divided by the storage requirements of one floating point (single precision) and one integer value.

6.2.3 TIMM.C: Strategy Implementation

TIMM.C essentially takes the data created by the previous programs and inputs it into various dynamic strategy simulations. The output of the program is the mean time in the system of all customers served, the variance of their time in system and the number of customers served at end of time as well the number of total customers (to test whether the system is stable or not). The user also can get text output of the actual time-in-system data points accompanied by a running mean. This is particularly useful in convincing yourself that the results do or do not reflect a steady state. If the running mean is fluctuating greatly or steadily increasing one should suspect that equilibrium has not been reached. The user can control the simulations through the selection of strategies, the speed of travel, the beta (as will be explained later) and the choice of SIMM and TIMM input files.

The command line parameters are as follows:

TIMM <arg1>, <arg2>, <arg3>, <arg4>, <arg5>, <arg6>, <arg7>, <arg8>, <arg9>, <arg10>, where the arguments are defined as follows:	
<arg1>	the total nodes in the network (an even number).
<arg2>	the file holding the time line linked list created by SIMM
<arg3>	the file holding the parameters created by SIMM
<arg4>	the file holding the distance matrix created by DIMTSP.
<arg5>	a temporary work file that will be overwritten on the next run unless run under a new name
<arg6>	execution of TIMM.
<arg7>	the strategy selection 1- 47 (see below).
<arg8>	speed (integer). output file in text form of actual times in system of each
<arg9>	customer served and their running mean.
<arg10>	Beta (see below).
<arg11>	the TSP file if DIMTSP was run using 'p'. If this parameter is missing TIMM will run a default order of (0,1,2,3....).

The strategies are listed below with their identifying codes used as input in argument 6 of TIMM's argument list. The function **Read_Final_File** in the program list, and the comments in the simulation functions give more information if necessary.

Select	**Strategy**
	System First In First Out
Select 1	"Serve the oldest customer in the system". This strategy essentially serves customers on a FIFO basis regardless of the expense of reaching him. Computationally it reads the SIMM argument 3 file sequentially. The only information this policy needs is order of arrival.
	Longest Queue Criteria
Select 2	"Serve the node with the longest queue: serve until the queue is completely empty (exhaustive)". This system requires the server to proceed to the longest queue as soon as he empties the queue at the node in which he presides. Ties between longest queues are decided by favoring the longest queue nearest to the current node. This policy requires more dynamic information about the system than the "closest", "TSP", and "preference" policies which only require information on whether the queue is empty or not. The closest vicinity policies lie between these two sets.
Select 3	"Serve the node with the longest queue: serve until the queue is empty of the customers in the queue upon his departure from the last queue (gated)". This system requires the server to proceed to the longest queue as soon as he empties the queue at the node in which he presides of original members. Original members are defined as those who resided at the node at the moment the server completed his last service before moving. Ties are treated as before.
	Nearest Neighbor Policies

Select 4	"Go to the closest node: serve until the queue is completely empty (exhaustive)." This strategy ignores queue lengths and forces the server to travel to the nearest non-empty node queue as soon as the queue at his current node is completely empty. In fact this strategy could be classified as a preference ordering strategy, where preferences are defined locally not globally as in the preference strategies to follow. Ties are settled arbitrarily by selecting the node with the smallest index. Nevertheless, it is unlikely ties will occur if distances are generated randomly in 'p' or 'n' mode of DIMTSP. Ties occur constantly in the circular graph and the indexation forces the server to travel in only one direction in case of ties.
Select 5	"Go to the closest node: serve until the queue is empty of original members (gated)." This strategy is the same as above, but forces the server to depart to another node once all original members are served. The section below describes the definition of "original."
Select 6	"Go to the closest node: serve until the queue is empty or after serving beta customers whichever comes first." This strategy is the same as above, but forces the server to depart after serving a user specified number of customers. For instance if the user specifies beta = 10, then the server is not allowed to serve more than 10 customers at the current node without departing. The motivation behind this strategy is to allow the user to search for an optimal departure rule, be it beta = 1, beta = ∞.

Select 7	"Go to the closest node: serve until the queue is empty or after serving beta times rho customers whichever comes first." This strategy is the same as above, but forces the server to depart after devoting $B.x\rho$ time in serving customers at his current node. Since there is no preemption the server is forced to move immediately after he exceeds spending $B.x\rho$ time serving node i's customers. Service time is calculated as the total time serving customers at the current node, not total time (the sum of busy and idle periods).
	Traveling Salesman Policies
Select 8	"Go to the next node on the TSP route: move on immediately." This strategy also ignores queue lengths. The server is required to travel to the next node on a pre-specified route. If the next node is empty the server can travel directly to the next node on the route that is non-empty. The server is allowed to skip empty nodes. If the server were forced to travel to empty nodes this policy would assume that the server cannot perceive the state of the system outside his current node. Since all other strategies allow the server to differentiate between empty and non-empty nodes this limitation would unfairly handicap this policy.
Select 9	"Go to the next node on the TSP route: serve until the queue is completely empty." This strategy is the same as above, but allows the server to depart only after he completely clears the queue.
Select 10	"Go to the next node on the TSP route: serve until the queue is empty of original members." This strategy is the same as above, but forces the server to depart to another node once all original members are served. The section below describes the definition of "original."

Select 11	"Go to the next node on the TSP route: serve until the queue is completely empty or until beta customers have been served whichever comes first." This strategy is the same as above, but forces the server to depart after serving a user specified number of customers. The motivation behind this strategy is to allow the user to search for an optimal departure rule, be it beta = 1, beta = ∞.
Select 12	"Go to next node on the TSP route: serve until the queue is empty or after serving beta times rho customers whichever comes first." This strategy is the same as above, but forces the server to depart after devoting $B.xP$ time in serving customers at his current node. The server is forced to move immediately after he exceeds spending $B.x\rho$ time serving node i's customers. Service time is calculated as the total busy time serving.

	Closest Vicinity Policies
Select 13	"Go to vicinity with the most customers waiting to be served: stay until all nodes in the vicinity are completely empty." This strategy is a hybrid between the information hungry policies favoring the longest queues, (which need to see both queue lengths and distances), and the nearest neighbor policies that are fairly ignorant,(needing to know only distances and whether a queue is empty or not). A vicinity policy requires that the server know distances, know if a node is empty or not, and the queue lengths of nodes near his current position. Since it has been shown empirically that nearest neighbor policies perform well, vicinity i is defined as the set of β nodes closest to node i. Beta is defined by the user. Upon completion of service at one vicinity the server selects the next vicinity by the number of customers in all its nodes. Upon deciding on vicinity i, the server serves the node in the vicinity with the longest queue. Within a vicinity, (before it is emptied), the server serves his current node until it is empty and then travel to the nearest neighbor that belongs to the vicinity. If β is equal to one this policy is the nearest neighbor strategy, if β = total number of nodes then it is similar to the longest queue policies.
Select 14	"Go to vicinity with the most customers waiting to be served: stay until all nodes in the vicinity are empty of original members." This strategy is exactly like the previous policy, except that it requires the server travel to the "heaviest" vicinity as of the time he completes service on the vicinities last original member. Herein is an important distinction between this policy and the closest neighbor policy: **the server is allowed to stay at his current vicinity if it still remains the heaviest despite clearing out all its original members.**

Select 15	"Go to vicinity with the most accumulated service time plus travel time from current position: stay until all nodes in the vicinity are completely empty." This strategy is exactly the same as the previous two policies except that the weight of a queue is judged by the total accumulated service time. Theoretical results on non-spatial priority queues suggest that this policy may be the worst performing if B is set to one. If one considers each node as a customer with discrete packets of service requirements, the "cμ" rule would suggest that the server go to the queue with the least accumulated service requirement.
Select 16	"Go to vicinity with the most customers waiting to be served: stay until all nodes in the vicinity are empty of original members." This policy is exactly as above, except the server can leave (but is not required to) after serving all the original customers in the vicinity.
	Preference (Explicit Priority) Policies
	There are three variation of each preference policy
	"Serve nodes by order of preference: stay until a node with higher priority becomes non-empty (immediate gate)." This policy sends the server to a node if it is the non-empty node with the highest preference. Different preference schemes are defined below. Unlike any of the previous policies, this scheme allows the server to move to another node after the completion of any service. In other terms, the server is required to move to a node with higher preference regardless of the state of the queue he is presently in.
	"Serve nodes by order of preference: stay until the current node is completely empty (exhaustive)." This policy sends the server the highest preference node upon emptying his current queue.

	"Serve nodes by order of preference: stay until the current node is empty of original members (gated)." This policy sends the server the highest preference node upon emptying the current node of original customers. Analogous to the closest node strategy the server is required to leave the current node even if it remains the highest preference queue that is non-empty.
	The Following Are The Preference Schemes Implemented In The Program
Select 17 Select 18 Select 19	**"Give highest preference to nodes with the minimum expected service requirement."** This policy orders the nodes as most preferred the lower their expected service requirement, as suggested by the cμ for non-spatial poisson service priority queues. The user must use the random option on the SIMM program.
Select 20 Select 21 Select 22	**"Give highest preference to nodes with the maximum expected service requirement."** This policy orders the nodes as most preferred the higher the expected service requirement of their customers. This policy is implemented to contrast the previous policy and see exactly how effective it is.
Select 23 Select 24 Select 25	**"Give highest preference to nodes with the minimum Rho."** This policy orders the nodes as most preferred the lower the product of the expected service requirement and arrival rate for their customers.
Select 26 Select 27 Select 28	**"Give highest preference to nodes with the maximum Rho."** This policy orders the nodes as most preferred the higher the Rho

Select 29 Select 30 Select 31	**"Give highest preference to nodes with the minimum Arrival rate."** This policy orders the nodes as most preferred the lower the arrival rate. The purpose behind this implementation is to test whether arrival rates are relevant to optimization.
Select 32 Select 33 Select 34	**"Give highest preference to nodes with the maximum Arrival rate."** This policy orders the nodes as most preferred the higher the Arrival Rate.
Select 35 Select 36 Select 37	**"Give highest preference to nodes with the minimum sum of expected service requirement and average travel time to the node from all other nodes."** This policy orders the nodes as most preferred the lower the travel inclusive service requirement. Average travel times are the unweighted average of travel times to the node from all other .odes. The purpose behind this implementation is to improve on the smallest expected service time implementation.
Select 38 Select 39 Select 40	**"Give highest preference to nodes with the maximum sum of expected service requirement and average travel time to the node from all other nodes."** This policy orders the nodes as most preferred the higher the summed expected service requirement and average travel time.
	Miscellaneous Variations
Select 41	**"Go to nearest neighbor, as defined by travel time plus expected time to serve first customer"**. This Policy is the same as select 6 except near is redefined as travel time plus service time.
Select 42	**"Go to the Vicinity with the lightest work load".** This policy is just like select 14, except the server goes to the non-empty vicinity with the least accumulated service requirement.
	Pure TSP Policies (No Shortcuts Allowed)

Select 43	**<u>"Serve the shortest route".</u>** This is just like policy 8, except the server is required to pass through empty nodes on his way to the next non-empty node. This is an "ignorant" strategy: it assumes the server has to visit a node to verify whether it is empty or not.
Select 44	**<u>"Serve the shortest route".</u>** This is just like policy 9, except the server is required to pass through empty nodes on his way to the next service.
Select 45	**<u>"Serve the shortest route".</u>** This is just like policy 10, except the server is required to pass through empty nodes on his way to the next service.
Select 46	**<u>"Serve the shortest route".</u>** This is just like policy 11, except the server is required to pass through empty nodes on his way to the next service.
Select 47	**<u>"Serve the shortest route".</u>** This is just like policy 12, except the server is required to pass through empty nodes on his way to the next service.

6.2.4 How to Run the Package: Step by Step

Step	Instructions
	Hint: BIMM.C creates a convenient batch file to avoid the steps outlined below.
1	Select the Size of Network(s): the user can select any even sized network up to 70 nodes with the present code. This will be argument one of DIMTSP, SIMM and TIMM.
2	Select the type of network and the length of its arcs. It is suggested that sufficiently large integers be used. Otherwise, the parameters used in SIMM will be hard to match. Run DIMTSP with easy to understand mnemonics for the output files. These files will be used over and over again. A text output file called DLOG is created or updated in the current directory on very run. This file contains the distance matrix and the TSP route in text form. Do not confuse this file with the output file specified on the command line.
3	Select the arrival rates, service rates, service variances and distributions desired on SIMM. Recall: the command line parameters are divided by 100 by the program to give the user more decimal point control. It is important to note that SIMM is the slowest of the three parts of this package. It would be wise to save SIMM output for two reasons: 1) it is time consuming to recreate these files, 2) if comparing policies it may lead to more accurate results if they are run on the exact same data. SIMM's text output file is currently called SLOG. It will be written in the current directory. If the file already exists, new output will be appended to its end.
4	Having noted the names of the output files of DIMTSP and SIMM, put these on the command line of TIMM. Select the other parameters that are of interest. Each TIMM output will be appended to the end of the current output file. Currently, the output file is called TLOG. It will be written in the current directory.

The following is an example of a batch file:

First, the distance matrix files are created. It creates two networks one of 24 nodes and another of 48 nodes. Both are planar.

```
dimtsp 24 dpp.24 dop.24  p 1000
dimtsp 48 dpp.48 dop.48  p 1000
```

Second, four data files are created. (arrival rate = 10/100 arrivals per unit time per node (!), service rate 242/100 services per unit time system wide (!), inverse of σ_s^2 = 2424. Service distribution is Gaussian (g) and every node has the same service and arrival parameters (d). The number of arrivals will equal to 100000 divided by the sum of the byte sizes of one integer and one single precision float.

```
simm 24 temp adgl0110.24 sdgl1010.24 d g 10 242 2424
100000
simm 24 temp adgl0120.24 sdgl1020.24 d g 10 242 4848
100000
simm 48 temp adgl0110.48 sdgl1010.48 d g 10 484 4848
100000
simm 48 temp adgl0120.48 sdgl1020.48 d g 10 484 9696
100000
```

The next files run the data files "arg..." And "srg...." of SIMM and "dpp.." and "dop.." of DIMTSP. The strategies selected are 2 and 43. The speed selected is uniformly 12000 distance units per unit time. To get exact travel times one needs to see the output of DIMTSP: DLOG. Beta's are equal one except in the last run. Files m1, m2, m3, m4 contain the exact times in system as they were recorded.

```
timm 24 argl1010.24 srgl1010.24 dpp.24 t 2  12000 m1 1
dop.24
timm 48 argl1020.48 srgl1020.48 dpp.48 t 43 12000 m2 1
dop.48
timm 24 argl1010.24 srgl1010.24 dpp.24 t 2  12000 m3 1
dop.24
timm 48 argl1020.48 srgl1020.48 dpp.48 t 43 12000 m4 4
dop.48
```

6.2.5 Program Assumptions and Quirks

Several points need to be emphasized about all the strategies. First of all customers are served on a FIFO basis at each node queue. This limitation is a computational convenience: identifying the next customer to serve by other orders would require updating the successor and predecessor arrays which would slow the program considerably. Second, the program assumes servers make their decisions on their next service immediately after serving their last customer. In other words the server is not allowed to change his mind while in route. Again this assumption has been made to reduce computational effort. This same assumption also does not allow service preemption. Of the three limitations listed above, this last one would require the least modification of the code written. A number of strategies require the server to only serve original queue members and no customers that arrive after "he gets to the node." In fact, the program is written so as to exclude any customers that arrive after he decides to go to their queue. This quirk is consistent with the program rule of always waiting until the completion of a service to make the next decision. The travel time to a node is considered as the beginning of a service.

7 Appendix B: Listings of Simulation Programs

7.1 Note on Portability

The following programs are written in Microsoft QuickC(TM). Notable differences may exist between compilers in the following functions: rand(), size of arrays allowed, and the declaration in timeb_h structure. The limitations of memory in the P.C. Environment has necessitated that many temporary writes and reads to disk are used instead of creating arrays of structures in RAM. In another environment all the fseek, fwrite and freads could be eliminated by simple references to arrays. It would be worth the effort and time in converting these hard write and reads so as to increase the speed of this program.

7.2 TIMM.C

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <math.h>
#define CUTFRACTION 4     /* set for about 20,000 observations: 5000 cut                    */
#define TOTALSTRAT 21     /* increment this if new strategies are added                     */
#define START 1           /* arbitrarily start at node labeled 1                            */
#define TOTLNODES 70      /* MAX number of nodes allowed: may be larger depending on system */

FILE *fprn;                  /*file pointer to text output file*/
struct timestruct {       /* structure used to store data created by SIMM.C                 */
     float time;           /*.time of arrival           */
     float remaining;     /* service requirement       */
     int node;            /* node id                   */
     int pred;            /* predecessor of arrival at node i*/
     int possys;          /* position in system as awhole    */
     int succ;            /* successor of arrival in queue   */
          } timeline;
struct nodestruct {       /* structure used to store parameter data created by SIMM.C       */
     float servicereq_X;
     float servicereq_V;
     float arrivalrate;   } static node[TOTLNODES];
struct  queuestruct {     /* structure used to store state of queue as seen by server       */
     int qlength;
     float qtime;
          } static q[TOTLNODES];
struct statstruct {       /* structure used to store final statistics                       */
     float mean;
     float variance;
          }acc[TOTALSTRAT];
static int svic[TOTLNODES];                  /*store node preference order */
static int ntest[TOTLNODES];                 /*check first arrival at node */
static int latest[TOTLNODES];                        /*      used to maintain     */
static int newtime[TOTLNODES];                       /*      information next     */
static long recent[TOTLNODES];                       /*      customer to serve    */
static int tsporder[TOTLNODES];                      /* tsp  order from DIMTSP    */
static float minnodedist[TOTLNODES][TOTLNODES];/* distance from   DIMTSP    */
static float temp[TOTLNODES][TOTLNODES];            /*used in assorted  sorts   */
float maxnodedistance;       /*  used as seed for determining closest nodes*/
float speed;                 /*  store node preference order               */
int beta;                    /*  beta used in routines6,7,11,12,13,14,15,16*/
float rhoestimate;           /*  also used in conjunction with beat        */
int totalnodes = TOTLNODES;  /*  number of nodes in network                */
char tlog[12] = "tlog.c";    /*  output file for statistics                */
int statnumserved;
int id;                                               /* code for strategy on printout*/
int startlag;                                         /* number of beg. obsv. to cut  */
float quartermean,halfmean,threefourthsmean;  /* samples of running means     */
*/
float quartermean,halfmean,threefourthsmean;  /* samples of running means     */
float printsum;


/*=========================================================================*/
/*=========================================================================*/
/*     MAIN MAIN MAIN                                                      */
/*=========================================================================*/
/*=========================================================================*/


main(argc, argv)
     int argc;
     char *argv[];
     {
     int i,j,test,end,tt;
     int select;
     long sspeed;
     float sumarrival,sumservice,avgservice;
     FILE *fprn;                          /*file pointer to text output file*/
     if  (argc > 11 || argc < 10) {fprintf(stderr,"\nWrong number of parameters");exit(-1);}
     tt = totalnodes;

     /*================================================================*/
     /*  Read use supplied parameters, convert to integer data      ===*/
     /*================================================================*/
     /*     argv[2] argv[3],argv[4],argv[5]                            */
     /*     arriv   para    dist     durr                              */
     /*     argv[6] argv[7]  argv[8] argv[9] argv[10]                  */
     /*      select   speed  mean     Beta    TSP                      */
     /*================================================================*/

     if( (fprn = fopen(tlog,"a")) == NULL) fprintf(fprn,"Error 1\n"); /* */
     if(1 != sscanf(argv[1],"%ld",&totalnodes)) {fprintf(fprn,"\n\nErr:Bad Speed\n\n");
exit(-1);}
     if ( totalnodes > TOTLNODES)
          {
```

```
        fprintf(fprn, "\n # nodes may cause memory overflow: check source code and hardware
");
        exit(0);
        }
    if(1 != sscanf(argv[7],"%ld",&sspeed)) {fprintf(fprn,"\n\nErr:Bad Speed\n\n"); exit(-1);}
    if(1 != sscanf(argv[9],"%d",&beta)) {fprintf(fprn,"\n\nErr:Bad beta\n\n"); exit(-1);}
    speed = (float) sspeed;
    if(1 != sscanf(argv[6],"%d",&select)) {fprintf(fprn,"\n\nErr:Bad beta\n\n"); exit(-1);}
    fprintf(fprn,"--------------------%d",select);

    /*===================================================================*/
    /*= Read Data Supplied by SIMM.C and DIMTSP.C               ========*/
    /*===================================================================*/

    Read_Parameter_File(argv[3]);                   /*argument 3 of SIMM.C */
    Read_Distance_File(argv[4]);                    /*argument 1 of DIMTSP */
    for(i=0;i<totalnodes;i++) tsporder[i]=i;
    if (argc == 11) { Read_TSP_File(argv[10]); test = 1;}     /*argument 2 of DIMTSP */
    if (test == 0) fprintf(fprn,"\nCertain Strategies require TSP file");
    end = Read_Final_File(argv[2],argv[5],select,argv[8]);

    /*===================================================================*/
    /*= Calculate Rho estimate to be used by strategies 7 & 12   ========*/
    /*===================================================================*/


    sumarrival = 0.0;
    for(i=0;i<totalnodes;i++)
        {sumarrival=node[i].arrivalrate+sumarrival;}
    sumservice = 0.0;
    for(i=0;i<totalnodes;i++)
         {sumservice=(node[i].servicereq_X+sumservice);}
    avgservice = sumservice/totalnodes;
    rhoestimate = sumarrival*avgservice;

    /*===================================================================*/
    /*= Print Results of Simulation to File                     ========*/
    /*===================================================================*/

    fprintf(fprn,"\nBeginning Hardcopy Print");
    fprintf(fprn,"\n*********************************************\n");
    for ( i = 0; i < argc; i++) fprintf(fprn,"\t%s",argv[i]);
    fprintf(fprn,"\n%f = Steady State Average Time in System", acc[id].mean);
    fprintf(fprn,"\n     at 1/4: %.1f || at 1/2: %.1f || at 3/4: %.1f",
              quartermean,      halfmean,     threefourthsmean);
    fprintf(fprn,"\n%f = Steady State Variance in System\n", acc[id].variance);
    fprintf(fprn,"\n%d = Number of Arrivals Served in System", statnumserved);
    fprintf(fprn,"\n%d = Number of Arrivals Total in System", end);
    fprintf(fprn,"\n%d = Number of beginning observations cut/length of route %f",start-
lag,printsum);
    fprintf(fprn,"\n*********************************************\n");
    if (fclose(fprn) == EOF) fprintf(fprn,"Error 3.04\n");
    fprintf(stderr,"\nFinished Hardcopy Update : in current directory");
    fprintf(stderr,"\nProgram Finished: No Error Exit");
    exit(0);
    }

/*==================================================================*/
/*==================================================================*/
/*  Read Arrival File and Direct Performance of Simulations          */
/*==================================================================*/
/*==================================================================*/


int Read_Final_File(yfile,vfile,tst,meanfile)
    char *yfile;   /* name of arrival file created by SIMM supplied by user      */
    char *vfile;   /* name of temporary file for length of time in system stats */
    int  tst;      /* pass value of user selected strategy                       */
    char *meanfile;/* name of file in text form of mean and time in system data */
    {
    char ch;
    int end=0,i=0;
    FILE *fpy,*fpv;
    struct timestruct endof;
    long int p;
    float Serve_the_oldest();
    float Serve_LongestQ();
    float Serve_Preference();
    float Serve_the_Closest();
    float Serve_the_Route() ;
    float Serve_the_Vicinity();
    fprintf(stderr,"============%d",tst);

    /*======================================================*/
    /*= Open SIMM's arrival file (argument 2)      ========*/
    /*======================================================*/

    p =  (sizeof(struct timestruct)); /*size of offset for file reads*/
    if ( (fpy = fopen(yfile,"rb")) == NULL )
        { fprintf(fprn,"\nError 1.11\n"); exit(-1); }

    /*======================================================*/
    /*= Find Last entry of file to serve as EOF marker  ========*/
    /*======================================================*/

    if (fseek(fpy, -p , SEEK_END) != 0)
        {fprintf(fprn,"\nError 1.10\n");}
    if(fread( &endof, sizeof(struct timestruct), 1, fpy ) != 1)
        {fprintf(fprn,"\nError 1.12\n");exit(1);}
    end = endof.possys-totalnodes;
    fprintf(stderr,"End of file marker  %d",end);
```

```
/*=======================================================*/
/*= Select Strategy Requested by user 1-40        ========*/
/*=======================================================*/

            /*case (user selection argument 6)              */
id = tst;        /* function performing simulation and subselection*/
switch(tst)      /* function to calculate stats                   */
   {
   case 1:
     Serve_the_oldest(vfile,fpy,end);  /*Serve the oldest*/
     Calculate_Stats(vfile,meanfile,0);
     break;
   case 2:
     Serve_LongestQ(vfile,fpy,end,2); /*Stay 'til empty then move on*/
     Calculate_Stats(vfile,meanfile,0);
     break;
   case 3:
     Serve_LongestQ(vfile,fpy,end,3); /*Stay 'til present pop is empty then move on*/
     Calculate_Stats(vfile,meanfile,0);
     break;
   case 4:
     Serve_the_Closest(vfile,fpy,end,2); /*Stay 'til empty*/
     Calculate_Stats(vfile,meanfile,0);
     break;
   case 5:
     Serve_the_Closest(vfile,fpy,end,3); /*Stay 'til those present are served*/
     Calculate_Stats(vfile,meanfile,0);
     break;
   case 6:
     Serve_the_Closest(vfile,fpy,end,4); /*Stay 'til static beta are served*/
     Calculate_Stats(vfile,meanfile,0);
     break;
   case 7:
     Serve_the_Closest(vfile,fpy,end,5); /*Stay 'til beta*rho are served*/

     Calculate_Stats(vfile,meanfile,0);
     break;
   case 8:
     Serve_the_Route(vfile,fpy,end,1);  /*Move on immediately*/
     Calculate_Stats(vfile,meanfile,0);
     break;
   case 9:
     Serve_the_Route(vfile,fpy,end,2);   /*Stay 'til empty*/
     Calculate_Stats(vfile,meanfile,0);
     break;
   case 10:
     Serve_the_Route(vfile,fpy,end,3);   /*Stay 'til those present are served*/
     Calculate_Stats(vfile,meanfile,0);
     break;
   case 11:
     Serve_the_Route(vfile,fpy,end,4);   /*Stay 'til static beta are served*/
     Calculate_Stats(vfile,meanfile,0);
     break;
```

```
      case 12:
        Serve_the_Route(vfile,fpy,end,5);    /*Stay 'til beta*rho are served*/
        Calculate_Stats(vfile,meanfile,0);
        break;
      case 13:
        Serve_the_Vicinity(vfile,fpy,end,2);  /* Vicinity by most service and travel time */
        Calculate_Stats(vfile,meanfile,0);      /*Stay 'til vicinity is empty*/
        break;
      case 14:
        Serve_the_Vicinity(vfile,fpy,end,4);  /* Vicinity by most service and travel time */
        Calculate_Stats(vfile,meanfile,0);      /*Stay 'til vicinity is empty of originals*/
        break;
      case 15:
        Serve_the_Vicinity(vfile,fpy,end,1);  /* Vicinity by longest queue */

        Calculate_Stats(vfile,meanfile,0);      /*Stay 'til  vicinity is empty*/
  break;
      case 16:
        Serve_the_Vicinity(vfile,fpy,end,3);  /* Vicinity by longest queue */
        Calculate_Stats(vfile,meanfile,0);      /*Stay 'til vicinity is empty of originals*/
        break;
      case 17:
        Serve_Preference(vfile,fpy,end,40);/*Stay 'til present pop is empty then move on*/
        Calculate_Stats(vfile,meanfile,0); /*      Smallest E(service)                   */
        break;
      case 18:
        Serve_Preference(vfile,fpy,end,50); /*Stay 'til empty then move on*/
        Calculate_Stats(vfile,meanfile,0); /*      Smallest E(service)                   */
        break;
      case 19:
        Serve_Preference(vfile,fpy,end,60); /*Stay 'til present pop is empty then move on*/
        Calculate_Stats(vfile,meanfile,0); /*      Smallest E(service)                   */
        break;
      case 20:
        Serve_Preference(vfile,fpy,end,10);/*Stay 'til present pop is empty then move on*/
        Calculate_Stats(vfile,meanfile,0); /*      Largest E(service) Mu                 */
        break;
      case 21:
        Serve_Preference(vfile,fpy,end,20); /*Stay 'til empty then move on*/
        Calculate_Stats(vfile,meanfile,0); /*      Largest E(service)                    */
        break;
      case 22:

Serve_Preference(vfile,fpy,end,30); /*Stay 'til present pop is empty then  move on*/
        Calculate_Stats(vfile,meanfile,0); /*      Largest E(service)                 */
        break;
      case 23:
        Serve_Preference(vfile,fpy,end,41);/*Stay 'til present pop is empty then move on*/
        Calculate_Stats(vfile,meanfile,0); /*      Smallest Rho                          */
        break;
      case 24:
        Serve_Preference(vfile,fpy,end,51); /*Stay 'til empty then move on*/
        Calculate_Stats(vfile,meanfile,0); /*      Smallest Rho                          */
```

```
   break;
  case 25:
   Serve_Preference(vfile,fpy,end,61); /*Stay 'til present pop is empty then move on*/
   Calculate_Stats(vfile,meanfile,0); /*      Smallest Rho                           */
   break;
  case 26:
   Serve_Preference(vfile,fpy,end,11);/*Stay 'til present pop is empty then move on*/
   Calculate_Stats(vfile,meanfile,0); /*      Largest Rho                            */
   break;
  case 27:
   Serve_Preference(vfile,fpy,end,21); /*Stay 'til empty then move on*/
   Calculate_Stats(vfile,meanfile,0); /*      Largest Rho                            */
   break;
  case 28:
   Serve_Preference(vfile,fpy,end,31); /*Stay 'til present pop is empty then move on*/
   Calculate_Stats(vfile,meanfile,0); /*      Largest Rho                            */
   break;
  case 29:

   Serve_Preference(vfile,fpy,end,42);/*Stay 'til present pop is empty then  move on*/
   Calculate_Stats(vfile,meanfile,0); /*     Smallest Arrival                        */
   break;
  case 30:
   Serve_Preference(vfile,fpy,end,52); /*Stay 'til empty then move on*/
   Calculate_Stats(vfile,meanfile,0); /*      Smallest Arrival                       */
   break;
  case 31:
   Serve_Preference(vfile,fpy,end,62); /*Stay 'til present pop is empty then move on*/
   Calculate_Stats(vfile,meanfile,0); /*      Smallest Arrival                     */
   break;
  case 32:
   Serve_Preference(vfile,fpy,end,12); /*Stay 'til present pop is empty then move on*/
   Calculate_Stats(vfile,meanfile,0); /*      Largest Arrival                    */
   break;
  case 33:
   Serve_Preference(vfile,fpy,end,22); /*Stay 'til empty then move on*/
   Calculate_Stats(vfile,meanfile,0); /*      Largest Arrival                        */
   break;
  case 34:
   Serve_Preference(vfile,fpy,end,32); /*Stay 'til present pop is empty then move on*/
   Calculate_Stats(vfile,meanfile,0); /*      Largest Arrival                        */
   break;
  case 35:
   Serve_Preference(vfile,fpy,end,43);/*Stay 'til more preferred has a customer     */
   Calculate_Stats(vfile,meanfile,0); /*     Smallest E(s) + Avg. Travel Time       */
   break;
  case 36:
   Serve_Preference(vfile,fpy,end,53); /*Stay 'til empty then move on*/
   Calculate_Stats(vfile,meanfile,0); /*     Smallest E(s) + Avg. Travel Time       */
   break;
  case 37:
   Serve_Preference(vfile,fpy,end,63); /*Stay 'til present pop is empty then move on*/
   Calculate_Stats(vfile,meanfile,0); /*     Smallest E(s) + Avg. Travel Time       */
```

```
     break;
    case 38:
     Serve_Preference(vfile,fpy,end,13);/*Stay 'til more preferred has a customer    */
     Calculate_Stats(vfile,meanfile,0); /*     Smallest E(s) + Avg. Travel Time       */
     break;
    case 39:
     Serve_Preference(vfile,fpy,end,23); /*Stay 'til is empty then move on*/
     Calculate_Stats(vfile,meanfile,0); /*     Smallest E(s) + Avg. Travel Time       */
     break;
    case 40:
     Serve_Preference(vfile,fpy,end,33); /*Stay 'til present pop is empty then move on*/
     Calculate_Stats(vfile,meanfile,0); /*     Largest Rho                            */
     break;

    /*** Last Minute Policies*/
    case 41:
     Serve_the_Closest(vfile,fpy,end,7); /*Stay 'til all are served*/
     Calculate_Stats(vfile,meanfile,0);  /*closest = traveltime+E(s)*/
     break;
    case 42:
     Serve_the_Vicinity(vfile,fpy,end,3);  /*lightest non zero vicinity */
     Calculate_Stats(vfile,meanfile,0);    /*Stay 'til vicinity is empty*/
     break;
    case 43:
     Serve_the_Route(vfile,fpy,end,11);  /*Move on immediately*/
     Calculate_Stats(vfile,meanfile,0);  /*No Short Cuts Allowed*/
     break;

    case 44:
     Serve_the_Route(vfile,fpy,end,12);   /*Stay 'til empty*/
     Calculate_Stats(vfile,meanfile,0);   /*No Short Cuts Allowed*/
     break;
    case 45:
     Serve_the_Route(vfile,fpy,end,13);   /*Stay 'til those present are served*/
     Calculate_Stats(vfile,meanfile,0);   /*No Short Cuts Allowed*/
     break;
    case 46:
     Serve_the_Route(vfile,fpy,end,14);   /*Stay 'til static beta are served*/
     Calculate_Stats(vfile,meanfile,0);   /*No Short Cuts Allowed*/
     break;
    case 47:
     Serve_the_Route(vfile,fpy,end,15);   /*Stay 'til beta*rho are served*/
     Calculate_Stats(vfile,meanfile,0);   /*No Short Cuts Allowed*/
     break;
    default:
     fprintf(fprn,"\nNo Such Choice of Q-System");
     break;
    }

/*=========================================================*/
/*=  Close Arrival File and Return to Main        ========*/
/*=========================================================*/
```

```
     if (fclose(fpy) == EOF)
        { fprintf(fprn,"\nError 3.01"); exit(-1);}


     fprintf(stderr,"\n\n Finished Reading : Arrival File %s \n\n",yfile);
     return(end);
     }

/*****************************************************************************/
/*****************************************************************************/
/*    STRATEGIES as Selected by Read_Final_File as Directed by User          */
/*****************************************************************************/
/*****************************************************************************/


/*=========================================================================*/
/*=========================================================================*/

/* Strategy I: Go to longest in system as measured beginning service       */
/*=========================================================================*/
/*=========================================================================*/

float Serve_the_oldest(vfile,fpy0,end)
     char *vfile;
     FILE *fpy0;
     int end;
     {
     int presentnode = START;          /*server is located at node START */
     int mm = 0;                       /* initially */
     float freetime=0.0,duration;
     FILE *fpv;
     fprintf(stderr,"\nServe the Longest being Calculated\n");

     /*========================================================*/
     /*= Open Arrival and Service File Created by SIMM.C =======*/
     /*========================================================*/

     if ( (fpv = fopen(vfile,"wb")) == NULL ) { fprintf(fprn,"\nError 8.11\n"); exit(-1); }

     /*========================================================*/
     /*= Begin Loop: Each itreation = 1 Service, 1 Arrival  ======*/
     /*========================================================*/


     if (fseek(fpy0,0L,SEEK_SET)!=0) fprintf(fprn,"\nError 1.016");
     do
          {
          if(fread( &timeline, sizeof(struct timestruct), 1, fpy0 ) != 1) {fprintf(fprn,"\nError
2.050\n");exit(-1);}
          if ( timeline.time < freetime )     /*freetime is "clock" time */
               {                              /*duration is time to get to node+time to be
served+wait time til server arrives*/
```

```
              duration = ( minnodedist[presentnode][timeline.node]/speed)+
                     timeline.remaining+
                     (freetime-timeline.time);
              freetime = freetime +( minnodedist[presentnode][timeline.node]/speed)+
                     timeline.remaining;
              }
           else
              {
              duration =( minnodedist[presentnode][timeline.node]/speed)+
                     timeline.remaining;
              freetime = timeline.time + duration;
              }

           ++mm;

     /*=========================================================*/
     /*= write time in system statistic to temp file      ========*/
     /*=========================================================*/

         if(fwrite( &duration, sizeof(float), 1, fpv ) != 1) {fprintf(fprn,"\nError
 2.221\n");exit(-1);}


     /*=========================================================*/
     /*= End loop if no more customers or loop for next customer */
     /*=========================================================*/

         presentnode = timeline.node;
         if (timeline.possys == end ) break;
         } while (1);

     /*=========================================================*/
     /*=  Close Arrival file                              ========*/
     /*=========================================================*/

     statnumserved = mm;
     if (fclose(fpv) == EOF) { fprintf(fprn,"\nError 3.05"); exit(-1);}
     return(0);
     }

/*====================================================================*/
/*====================================================================*/
/* go to longest queue in system with no preemption FIFO              */
/*           closest distance choice in tie situations               */
/*====================================================================*/
/*====================================================================*

float Serve_LongestQ(vfile,fpy05,end,sub)
     char *vfile;
     FILE *fpy05;
     int end;
```

```
      int sub;
      {
      int last_in=0,lastserved=0;
      int presentnode = START;
      int i, qq, mm, idum, first, set, test, kk, longest;
      int test1,test0,test5;
      float closest;
      float duration,freetime,subduration;
      long ldum,offset,lastread,startread;
      FILE *fpv;

      fprintf(stderr,"\nServe the Longest Queue\n");

      /*=========================================================*/
      /*=  Initiate Variables                           ========*/
      /*=========================================================*/

      for ( i=0; i < totalnodes; i++ )
          {
          ntest[i] = 0;
          q[i].qlength = 0;   /* queue length at queue i at current time */
          recent[i] = 0L;     /* records most recent arrival at node i for use by predesessor
search*/
          newtime[i] = 0;     /* records identity of next arrival in queue i by his position in
system*/
          latest[TOTLNODES];  /*identify latest arrival in queue i by his system position*/
          }
     offset = sizeof(struct timestruct);
     freetime=0.0; first = 29999; test1=0;
     test0=0; startread=0L; kk=0; longest=0;
     mm=0;test5 = 1;

     /*=========================================================*/
     /*= Open SIMM arrival file                         ========*/
     /*=========================================================*/

       if ( (fpv = fopen(vfile,"wb")) == NULL ) { fprintf(fprn,"\nError 9.11\n"); exit(-1); }

     /*=========================================================*/
     /*= Begin loop: each iteration = 1 service 0,1,2+ Arrivals ==*/
     /*=========================================================*/


      do
      {


   /*=========================================================*/
      /*= Find arrivals that ocurred since during last service  ===*/
      /*=========================================================*/
      do
     {
      /*** Break if last served customer indicates he has no  *********/
      /*** following arrivals: i.e. newtime[kk]=timelime[kk].succ = ****/
      /*** 29998 the indicator that no successor exists        *******/
     if ( newtime[kk] >= 29998 ) break;
      /***  Set file seek to startread which in last loop recorded  ***/
      /***  the file location of node i's next arrival using the   ****/
      /***  successor array   timeline.succ                     *******/
     if (fseek(fpy05,startread,SEEK_SET)!=0) fprintf(fprn,"\nError 1.017\n");
      do
        {
         if(fread( &timeline, sizeof(struct timestruct), 1, fpy05 ) != 1) {fprintf(fprn,"\nEr-
ror 2.1052\n");exit(-1);}
         if((lastread = ftell(fpy05)) == -1L){fprintf(fprn,"Error 0.01");exit(-1);}
                                             /*make sure file location is correct*/
         if( timeline.time >  freetime ){break;} /* Stay in loop til all current arrival have
been recorded*/
         idum = timeline.node;   /* Record identity of last current arrival*/
         q[idum].qlength = q[idum].qlength + 1;          /*record his arrival in the queue*/
         recent[idum] = lastread - offset;               /*record where arrival was located
in file*/
         latest[idum] = timeline.possys;                 /*for some strategies we */
                                             /*record arrival sysytem position*/
       }    while(1);
      startread = lastread-offset;                              /*used in loop to verify
crrect read*/

      /**** If system is empty: set freetime to time of next    *********/
      /****    Arrival:                                          *********/
      qq=0;
      for (i=0; i < totalnodes; i++) qq = qq + q[i].qlength;
      if ( qq == 0 ){ freetime = timeline.time; test1 = 1;}
      else test1 = 0;

      /**** Continue to loop until read of arrival above indicates *****/
      /****    that there is no more data: test1=0.             *********/
      } while (test1);

      /*=======================================================*/
      /*= Select one of Various "longest queue" strategies      ===*/
      /*=======================================================*/

      switch (sub)
    {
      case 2:

      /**************************************************************/
      /*** STRATEGY: Serve Queue determined in past to be longest **/
      /************** Until completely empty ***********************/
```

```
 if ( q[presentnode].qlength == 0  || test5 )  /* Test if Queue is empty */
  {                                            /* if not continue to serve current*/
  if (test5) test5 = 0;                        /* test5: always determine longest on*/
  longest = 0;                                 /* first arrival*/
  for ( i = 0; i < totalnodes; i++)
      {
      if ( q[i].qlength > longest )            /* simple move up or out sort */
        {
        longest = q[i].qlength;
        kk = i;
        }
      }
  closest = maxnodedistance;                   /* go to closest queue if */
  for ( i = 0; i < totalnodes; i++)            /* there is a tie        */
     {
     if ( q[i].qlength == longest )
       {
       if ( minnodedist[presentnode][i] < closest )
        {
        closest = minnodedist[presentnode][i];
        kk = i;
        }
       }
     }
  }
break;

  case 3:
  /*************************************************************/
  /*** STRATEGY: Serve Queue determined in past to be longest **/
  /******* until empty of customers originally in Q. ***********/

                        /* move to longest queue after last customer*/
                        /* originally in Q has been served*/
  if ( q[presentnode].qlength == 0 ||  last_in == lastserved)
  {                                /*no test5 neededsince first condition*/
  longest = 0;                     /* always holds on first arrival      */
  for ( i = 0; i < totalnodes; i++)
     {
     if ( q[i].qlength > longest )
       {
       longest = q[i].qlength;
       kk = i;
       }
     }
 closest = maxnodedistance;       /* go to closest if there is a tie */
 for ( i = 0; i < totalnodes; i++)
    {
    if ( q[i].qlength == longest )
      {
      if ( minnodedist[presentnode][i] < closest )
       {
       closest = minnodedist[presentnode][i];
```

```
            kk = i;
            }
           }
         }
      last_in = latest[kk];  /* Update program to whom the last customer is*/
      }                      /* identified by his system position         */
      break;
       default:
      fprintf(fprn,"\n No Such Choice");
      exit(-1);
       }

     /*========================================================*/
     /*= For Selected Queue: Find Oldest (FIFO) member   ========*/
     /*========================================================*/

      if ( ntest[kk] == 0)
      {
      ldum = recent[kk]; ntest[kk] = 1;            /* Use predecessor link  */
      while(1)                                     /* For Initial FIFO seek */
        {                                          /* ntest[kk] indicates if queue has been
initiated*/
                                    /* seek earliest member by descending Pred array*/
        if (fseek(fpy05,ldum,SEEK_SET)!=0) fprintf(fprn,"\nError 1.013\n");
        if (fread( &timeline, sizeof(struct timestruct), 1, fpy05 ) != 1) {fprintf(fprn,"\nEr-
ror 2.106\n");exit(-1);}
        ldum = offset*(timeline.pred-1);
        /*DEBUGfprintf(fprn,"\n%d\t%d\t%d\t%d", timeline.node,timeline.possys,timeline.pred,ti-
meline.succ);*/
        if ( timeline.possys == newtime[kk]) break;  /* stop seeking if find earliest member*/
        if ( timeline.pred == 29999 ) break;         /* stop! don't proceed if at beginning of
file*/
        }
      }
      else                                         /* Use successor link for */
        {                                          /* following FIFO seeks   */
        ldum = offset*(newtime[kk]-1); /* since much more efficient */
        if (fseek(fpy05,ldum,SEEK_SET)!=0) fprintf(fprn,"\nError 1.013\n");
        if (fread( &timeline, sizeof(struct timestruct), 1, fpy05 ) != 1) {fprintf(fprn,"\nEr-
ror 2.106\n");exit(-1);}
        /*DEBUG:fprintf(fprn,"\n-->%d\t%d\t%d\t%d",
timeline.node,timeline.possys,timeline.pred,timeline.succ);*/
        }
       newtime[kk] = timeline.succ;               /*update next eleigible customer*/
       q[kk].qlength = q[kk].qlength - 1;         /* successor array, show that a */
       lastserved = timeline.possys;              /* been served, tell strategy who last*/
                                     /* customer was                              */
       /*DEBUGfprintf(fprn,"\n -------------------FINISH PREDECESSOR");*/

    /*=========================================================*/
    /*= Calculate time in system statistics as well as update   */
    /*    Clock: i.e. freetime , write statistic to temp file    */
    /*=========================================================*/
```

```
        subduration = ( minnodedist[presentnode][timeline.node]/speed)+ timeline.remaining;
        duration = subduration + (freetime-timeline.time);
        freetime = freetime + subduration;
        ++mm;
        if(fwrite( &duration, sizeof(float), 1, fpv ) != 1) {fprintf(fprn,"\nError
   2.222\n");exit(-1);}

       /*=======================================================*/
       /*= Loop again if not at end of file, tell loop where    ==*/
       /*                     current queue is                  ==*/
       /*=======================================================*/


       presentnode = timeline.node;
        if (newtime[kk] >= 29998 ) break;  /* Quit if sucessor of last service*/
                             /* indicates end of file          */
      } while(1);

      /*=======================================================*/
      /*= Close file and return to Main                        ==*/
      /*                     current queue is                  ==*/
      /*=======================================================*/

      if (fclose(fpv) == EOF) { fprintf(fprn,"\nError 3.06"); exit(-1);}
      statnumserved = mm;
      return(0);
      }

/*=================================================================*/
/*=================================================================*/
/* Go to preferred queue in system with no preemption FIFO, Preferences   */
/*=   are  calculated prior to system initialization and are left unchanges*/
/*=================================================================*/
/*=================================================================*/


float Serve_Preference(vfile,fpy05,end,sub)  /*temp file, pointer to arrival */
                             /* file and end of file marker  */
    char *vfile;
    FILE *fpy05;
    int end;
    int sub;
                          /*****************************/
                          /* Documentation for        */
                          /* most ststements following*/
                          /* In SERVE_THE_LONGEST     */
                          /* Function since routines  */
                          /* are essentially the same */
                          /* with the exception of    */
                          /* code in switch statement*/
                          /****************************/
    {
```

```
int last_in=0,lastserved=0;
int presentnode = START;
int i=0, qq, mm, idum, first, set, test, kk, longest;
int test1,test0,test5;
float closest;
float duration,freetime,subduration;
long ldum,offset,lastread,startread;
FILE *fpv;
int j=0,k=0,dm=0;
float max=0.0,lowest=0.0;
static float tmp[TOTLNODES];
fprintf(stderr,"\nServe the Preferred Queue\n");

/*=====================================================*/
/*=  Initialize Variables                      ========*/
/*=====================================================*/

for ( i=0; i < totalnodes; i++ )
     {
     ntest[i] = 0;
     q[i].qlength = 0;
     recent[i] = 0L;
     newtime[i] = 0;
     latest[TOTLNODES];
     }
for ( i = 0;i < totalnodes; i++)  tmp[i] = 0.0;
offset = sizeof(struct timestruct);
freetime=0.0; first = 29999; test1=0;
test0=0; startread=0L; kk=0; longest=0;
mm=0;test5 = 1;


/*=====================================================*/
/*= Find order of nodes to be given priority in service ====*/
/*=====================================================*/

for ( k = 0; k < totalnodes; k++ ){ tmp[k] =0.0; }
switch(sub)
    {
    case 10:
    case 20:
    case 30:
    case 40:      /* Assign values to a temporary array for up or out sort*/
    case 50:      /* Calculate maximum seed value for sort: max         */
    case 60:
            /* Preferences by expected service requirement*/
      for ( k = 0; k < totalnodes; k++ ){ max = max + node[k].servicereq_X; }
      for ( k = 0; k < totalnodes; k++ ){ tmp[k] = node[k].servicereq_X; }
      break;
    case 11:
    case 21:
    case 31:
        case 41:
        case 51:
        case 61:
                /*Preferences by Rho if each queue were independent*/
          for ( k = 0; k < totalnodes; k++ ){ max = max + node[k].servicereq_X*node[k].arrival-
rate; }
          for ( k = 0; k < totalnodes; k++ ){ tmp[k] =
node[k].servicereq_X*node[k].arrivalrate; }
          break;
        case 12:
        case 22:
        case 32:
        case 42:
        case 52:
        case 62:
                /*Preferences by Lambda for Exponential Interarrival times*/
          for ( k = 0; k < totalnodes; k++ ){ max = max + node[k].arrivalrate; }
          for ( k = 0; k < totalnodes; k++ ){ tmp[k] = node[k].arrivalrate; }
          break;
        case 13:
        case 23:
        case 33:
        case 43:
        case 53:
        case 63:
              /*Preferences by Expected service requirement plus approximation*/
              /* Expected travel times to get to a node, i.e. if p(server at   */
              /* node i) = 1/total nodes.                                      */
          for ( i = 0; i < totalnodes; i++)
          {
          for ( j = 0; j < totalnodes; j++)
          tmp[i] = tmp[i] + minnodedist[i][j]/speed;
          }
          for ( i = 0; i < totalnodes; i++) tmp[i] = tmp[i]/totalnodes;
          for ( k = 0; k < totalnodes; k++ ){ max = max + node[k].servicereq_X+tmp[k]; }
          for ( k = 0; k < totalnodes; k++ ){ tmp[k] = node[k].servicereq_X+ tmp[k]; }
          break;
        default:
          break;
        }


    /*=========================================================*/
    /*= Create Index Variable to identify 1st,2nd,...preferred node*/
    /*=========================================================*/

    lowest = max;
    for ( k= 0; k < totalnodes; k++)
        {
        lowest = max;
        for ( j= 0; j < totalnodes; j++ )
         {
         if ( tmp[j] < lowest )
```

```
            {
             lowest = tmp[j];
             svic[k] = j;
             dm = j;
            }
        }
       tmp[dm] = max*(k+1);
       }

   /*========================================================*/
   /*=  Open SIMM arrival file for analysis          ========*/
   /*========================================================*/

     if ( (fpv = fopen(vfile,"wb")) == NULL ) { fprintf(fprn,"\nError 9.11\n"); exit(-1); }

   /*==========================================================*/
   /*= Loop: each iteration = one service, any number of arrivals*/
   /*==========================================================*/


    do
    {

   /*========================================================*/
   /*= Find arrivals that occured during last service if any ==*/
   /*========================================================*/

     do
   {
   if ( newtime[kk] >= 29998 ) break;
   if (fseek(fpy05,startread,SEEK_SET)!=0) fprintf(fprn,"\nError 1.017\n");
     do
       {
        if(fread( &timeline, sizeof(struct timestruct), 1, fpy05 ) != 1) {fprintf(fprn,"\nEr-
ror 2.1052\n");exit(-1);}
        if((lastread = ftell(fpy05)) == -1L){fprintf(fprn,"Error 0.01");exit(-1);}
        if( timeline.time >  freetime ){break;}
        idum = timeline.node;
        q[idum].qlength = q[idum].qlength + 1;
        recent[idum] = lastread - offset;
        latest[idum] = timeline.possys;
       }    while(1);
    startread = lastread-offset;
    qq=0;
    for (i=0; i < totalnodes; i++) qq = qq + q[i].qlength;
    if ( qq == 0 ){ freetime = timeline.time; test1 = 1;}
    else test1 = 0;
    } while (test1);
```

```
/*=========================================================*/
/*= Selection of Strategies based on going to non-empty queue*/
/*= with highest preference or priority                    */
/*=========================================================*/

  sub = sub/10;   /* a convenient trick to save on switch statements*/
  switch (sub)
 {

  case 1:
  case 4:
  /**********************************************************/
  /*** STRATEGY: Go to higher  priority queue as soon as it is */
  /*   non empty                                             */
  /**********************************************************/

 for ( i = 0; i < totalnodes; i++)
   {
   if ( q[svic[i]].qlength != 0 )
    {
    kk = svic[i];      /*svic[1]=2 means most preferred node is 2    */
    if (sub==4) break; /*this statement to select minimum based pref.*/
    }
   }
  break;

  case 2:
  case 5:
  /**********************************************************/
  /*** STRATEGY:                                   *********/
  /**********************************************************/

if ( q[presentnode].qlength == 0  || test5 )
 {                                  /*Same as above but now the*/
 if (test5) test5 = 0;              /*server is required to stay*/
 for ( i = 0; i < totalnodes; i++) /*until the queue is empty  */
   {
   if ( q[svic[i]].qlength != 0 )
    {
    kk = svic[i];
    if (sub==5) break;
    }
   }
}
 break;

 case 3:
 case 6:
 /**********************************************************/
 /*** STRATEGY:                                   *********/
 /**********************************************************/
```

```
        if ( q[presentnode].qlength == 0 || last_in == lastserved)
        {
         for ( i = 0; i < totalnodes; i++) /*Same as above but server    */
            {                             /*is required to move on after*/
            if ( q[svic[i]].qlength != 0 && svic[i] != kk )
             {                        /* all original members of queue are served*/
             kk = svic[i];
             if (sub ==6) break;
             }
            }
       last_in = latest[kk];
       }
        break;
        default:
       fprintf(fprn,"\n No Such Choice");
       exit(-1);
        }


     /*=========================================================*/
     /*=  Find FIFO member of currently selected node          */
     /*=========================================================*/

       if ( ntest[kk] == 0)
       {
       ldum = recent[kk]; ntest[kk] = 1;
       while(1)
          {
          if (fseek(fpy05,ldum,SEEK_SET)!=0) fprintf(fprn,"\nError 1.013\n");
          if (fread( &timeline, sizeof(struct timestruct), 1, fpy05 ) != 1) {fprintf(fprn,"\nEr-
ror [illegible] 106\n");exit(-1);}
          ldum = offset*(timeline.pred-1);
          /*DEBUGfprintf(fprn,"\n%d\t%d\t%d\t%d", timeline.node,timeline.possys,timeline.pred,ti-
meline.succ);*/
          if ( timeline.possys == newtime[kk]) break;
          if ( timeline.pred == 29999 ) break;
          }
       }
       else
          {
          ldum = offset*(newtime[kk]-1);
          if (fseek(fpy05,ldum,SEEK_SET)!=0) fprintf(fprn,"\nError 1.013\n");
          if (fread( &timeline, sizeof(struct timestruct), 1, fpy05 ) != 1) {fprintf(fprn,"\nEr-
ror 2.106\n");exit(-1);}
          /*DEBUGfprintf(fprn,"\n-->%d\t%d\t%d\t%d",
timeline.node,timeline.possys,timeline.pred,timeline.succ);*/
          }
        newtime[kk] = timeline.succ;
        q[kk].qlength = q[kk].qlength - 1;
        lastserved = timeline.possys;
        /*DEBUGfprintf(fprn,"\n -------------------FINISH PREDECESSOR");*/
     /*======================================================*/
     /*=   find service duration stats              ========*/
     /*======================================================*/


      subduration = ( minnodedist[presentnode][timeline.node]/speed)+ timeline.remaining;
      duration = subduration + (freetime-timeline.time);
      freetime = freetime + subduration;
      ++mm;
      if(fwrite( &duration, sizeof(float), 1, fpv ) != 1) {fprintf(fprn,"\nError
 2.222\n");exit(-1);}

      presentnode = timeline.node;
       if (newtime[kk] >= 29998 ) break;
     } while(1);
     if (fclose(fpv) == EOF) { fprintf(fprn,"\nError 3.06"); exit(-1);}
     statnumserved = mm;
     return(0);
     }

/*=====================================================================*/
/*=====================================================================*/
/* Strategy III : Go to queue in system closest to present position   */
/*=====================================================================*/
/*=====================================================================*/

float Serve_the_Closest(vfile,fpy2,end,sub)
    char *vfile;
    int end;
    FILE *fpy2;
    int sub;
                    /****************************/
                    /* Documentation for        */
                    /* most ststements following*/
                    /* In SERVE_THE_LONGEST     */
                    /* Function since routines  */
                    /* are essentially the same */
                    /* with the exception of    */
                    /* code in switch statement*/
                    /***************************/
    {
    static last_in, lastserved;
    int numserved;
    float timeserved;
    int presentnode = START;
    int i, qq, mm, idum, first, set, test, kk, longest;
    int test1,test0;
    float closest;
    float savetime,duration,freetime,subduration;
    long ldum,offset,lastread,startread;
    FILE *fpv;
    fprintf(stderr,"\nServe the Closest Queue\n");
```

```
/*=========================================================*/
/*=  Initialize Variables                          ========*/
/*=========================================================*/

for ( i=0; i < totalnodes; i++ )
     {
     ntest[i] = 0;
     q[i].qlength = 0;
     recent[i] = 0L;
     newtime[i] = 0;
     latest[TOTLNODES];
     }
offset = sizeof(struct timestruct);
freetime=0.0; first = 29999; test1=0;
test0=0; startread=0L; kk=0; longest=0;
mm=0;
 if ( (fpv = fopen(vfile,"wb")) == NULL ) { fprintf(fprn,"\nError 9.11\n"); exit(-1); }

/*=========================================================*/
/*= Begin Loop: One Iteration = One Service, any # arrivals */
/*=========================================================*/

 do
 {

/*==========================================================*/
/*= Find who arrived durring last service, update queue stats=*/
/*==========================================================*/

  do
{
if (newtime[kk] >= 29998 ) break;
if (fseek(fpy2,startread,SEEK_SET)!=0) fprintf(fprn,"\nError 1.017\n");
  do
    {
     if(fread( &timeline, sizeof(struct timestruct), 1, fpy2 ) != 1) {fprintf(fprn,"\nError
2.1052\n");exit(-1);}
     if((lastread = ftell(fpy2)) == -1L){fprintf(fprn,"Error 0.01");exit(-1);}
     if( timeline.time >  freetime ){break;}
     idum = timeline.node;
     q[idum].qlength = q[idum].qlength + 1;
     recent[idum] = lastread - offset;
     latest[idum] = timeline.possys;
    }   while(1);
 startread = lastread-offset;
 qq=0;
 for (i=0; i < totalnodes; i++) qq = qq + q[i].qlength;
 if ( qq == 0 ){ freetime = timeline.time; test1 = 1;}
 else test1 = 0;
 } while (test1);
```

```
/*=======================================================*/
/*= Stategy based on always traveling to the closest non- ==*/
/*    queue from current position                        ==*/
/*=======================================================*/

 switch(sub)
  {

  case 2:
  /************************************************************/
  /*** STRATEGY: Serve current node until empty of all customers*/
  /************************************************************/
   kk = presentnode;
   if ( q[kk].qlength == 0 )
     {
     closest = maxnodedistance;        /*Determine nearest neighbor              */
     for ( i = 0; i < totalnodes; i++) /*Note: this is not an efficient          */
          {                         /*implementation, see Vicinity strategy      */
          if ( presentnode != i )
               {                   /*Do not consider current location as nearest*/
               if ( q[i].qlength != 0 )
                    {
                    if ( minnodedist[presentnode][i] <= closest )
                         {
                         closest = minnodedist[presentnode][i];
                         kk = i;
                         }
                    }
               }
          }
       /* fprintf(fprn,"\nClosest node is %d", kk);*/
     }
  break;

 case 3:
 /**************************************************************/
 /*** STRATEGY: Same as above  but server is now required to     */
 /***  move on after all original members of the queue are served*/
 /**************************************************************/

  kk = presentnode;
  if ( q[kk].qlength == 0 || lastserved == last_in )
    {
    closest = maxnodedistance;
    for ( i = 0; i < totalnodes; i++)
         {
         if ( presentnode != i )
              {
              if ( q[i].qlength != 0 )
                   {
                   if ( minnodedist[presentnode][i] <= closest )
                        {
                            closest = minnodedist[presentnode][i];
                            kk = i;
                            }
                        }
                    }
                }
        last_in = latest[kk];
          /* fprintf(fprn,"\nClosest node is %d", kk);*/
        }
    break;

   case 4:
     /************************************************************/
     /*** STRATEGY: Same as above but required to move on after   */
     /***  Beta(supplied by user) customers arte served           */
     /************************************************************/

       numserved=numserved+1;
       kk = presentnode;
       if ( q[kk].qlength == 0 || numserved == beta )
        {
        numserved = 0;
        closest = maxnodedistance;
        for ( i = 0; i < totalnodes; i++)
             {
             if ( presentnode != i )
                  {
                  if ( q[i].qlength != 0 )
                       {
                       if ( minnodedist[presentnode][i] <= closest )
                            {
                            closest = minnodedist[presentnode][i];
                            kk = i;
                            }
                       }
                  }
             }
          /* fprintf(fprn,"\nClosest node is %d", kk);*/
        }
    break;

    case 5:

/*******************************************************************/
    /*** STRATEGY: As Above but server is required to move on after     */
    /***  Beta( as supplied by user) times Rho( average arrival rate    */
    /***  times expected service requirement )                         */
    /*******************************************************************/
```

```
          kk = presentnode;
          if ( q[kk].qlength == 0 || timeserved >= beta*rhoestimate )
           {
           timeserved = 0.0;                /*timeserved == sum of service  */
           closest = maxnodedistance;       /* times of queue members served*/
           for ( i = 0; i < totalnodes; i++) /* since servers arrival at node*/
                {
                if ( presentnode != i )
                     {
                     if ( q[i].qlength != 0 )
                          {
                          if ( minnodedist[presentnode][i] <= closest )
                               {
                               closest = minnodedist[presentnode][i];
                               kk = i;
                               }
                          }
                     }
                }
            /* fprintf(fprn,"\nClosest node is %d", kk);*/
          }
        break;
      default:
        fprintf(fprn,"\nNo Such Choice");
        exit(-1);
       }

     /*=========================================================*/
     /*= Find First-in memeber of current nodes queue    ========*/
     /*=========================================================*/

      if ( ntest[kk] == 0)
      {
      ldum = recent[kk]; ntest[kk] = 1;
      while(1)
         {
         if (fseek(fpy2,ldum,SEEK_SET)!=0) fprintf(fprn,"\nError 1.013\n");
         if (fread( &timeline, sizeof(struct timestruct), 1, fpy2 ) != 1) {fprintf(fprn,"\nError
2.106\n");exit(-1);}
         ldum = offset*(timeline.pred-1);
         /*DEBUGfprintf(fprn,"\n%d\t%d\t%d\t%d", timeline.node,timeline.possys,timeline.pred,ti-
meline.succ);*/
         if ( timeline.possys == newtime[kk]) break;
         if ( timeline.pred == 29999 ) break;
         }
       }
       else
         {
         ldum = offset*(newtime[kk]-1);
         if (fseek(fpy2,ldum,SEEK_SET)!=0) fprintf(fprn,"\nError 1.013\n");
         if (fread( &timeline, sizeof(struct timestruct), 1, fpy2 ) != 1) {fprintf(fprn,"\nError
2.106\n");exit(-1);}
         /*DEBUGfprintf(fprn,"\n-->%d\t%d\t%d\t%d", timeline.node,timeline.possys,timeline.pred-
,timeline.succ);*/
         }
        newtime[kk] = timeline.succ;
        q[kk].qlength = q[kk].qlength - 1;
        lastserved = timeline.possys;
        timeserved = timeserved + timeline.remaining;
      /*DEBUGfprintf(fprn,"\n -------------------FINISH PREDECESSOR");*/

      /*========================================================*/
      /*= Calculate Statistics                        ==========*/
      /*========================================================*/

       subduration = ( minnodedist[presentnode][timeline.node]/speed)+ timeline.remaining;
       duration = subduration + (freetime-timeline.time);
       freetime = freetime + subduration;
       ++mm;

      /*========================================================*/
      /*= Write Statistics to file and continue with new iteration*/
      /*========================================================*/

       if(fwrite( &duration, sizeof(float), 1, fpv ) != 1) {fprintf(fprn,"\nError
 2.223\n");exit(-1);}

       presentnode = timeline.node;
        if (newtime[kk] >= 29998 ) break;
      } while(1);
      if (fclose(fpv) == EOF) { fprintf(fprn,"\nError 3.06"); exit(-1);}
      statnumserved = mm;
      return(0);
      }

/*=====================================================================*/
/*=====================================================================*/
/* Strategy IV       Go to next node on TSP  route                    */
/*=====================================================================*/
/*=====================================================================*/


float Serve_the_Route(vfile,fpy3,end,sub) /*temp file, pointer to arrival */
                          /* file and end of file marker  */
    char *vfile;
    int end;
    FILE *fpy3;
    int sub;
                         /*****************************/
                         /* Documentation for         */
                         /* most ststements following*/
                         /* In SERVE_THE_LONGEST      */
                         /* Function since routines   */
                         /* are essentially the same */
                         /* with the exception of     */
```

```
                    /* code in switch statement*/
                    /****************************/
{
int jj = 0;
int last_in, lastserved;
int numserved;
float timeserved;
int presentnode = START;
int i, qq, mm, idum, first, set, test, kk, longest;
int test1,test0;
float closest;
float duration,freetime,subduration;
long ldum,offset,lastread,startread;
FILE *fpv;
fprintf(stderr,"\nServe the Next in Route\n");
int k,j,ff,gg;
float dsum;

/*=====================================================*/
/*= Initialize Variables                       ========*/
/*=====================================================*/

for ( i=0; i < totalnodes; i++ )
    {
    ntest[i] = 0;
    q[i].qlength = 0;
    recent[i] = 0L;
    newtime[i] = 0;
    latest[TOTLNODES];
    }
offset = sizeof(struct timestruct);
freetime=0.0; first = 29999; test1=0;
test0=0; startread=0L; kk=0; longest=0;
mm=0;
if ( (fpv = fopen(vfile,"wb")) == NULL ) { fprintf(fprn,"\nError 9.11\n"); exit(-1); }

/*=====================================================*/
/*=  Revised Distances: TSP no shortcuts allowed : must  =*/
/*=   follow path regardless of node state ahead        =*/
/*=====================================================*/
if( sub == 11 || sub == 12 || sub == 13 || sub == 14 || sub == 15 )
 {
 for ( k = 0; k < totalnodes; k++)
    {
    for ( j = 0; j < totalnodes; j++)
         {
         for(ff=0; ff < totalnodes;ff++)
              {
              if ( tsporder[ff]==k)break;
              }
         for(gg=0; gg < totalnodes;gg++)
              {
              if ( tsporder[gg]==j)break;
```

```
                }
            dsum = 0.0;
            do
                {
                dsum += minnodedist[tsporder[ff]][tsporder[(ff+1)%totalnodes]];
                ff= (ff+1)%totalnodes;
                } while ( ff != gg );
            temp[k][j]=dsum;
            fprintf(stderr,"\n%d\t%d\t%f",k,j,temp[k][j]);
            }
        }
    printsum = temp[1][1];
    for ( k = 0; k < totalnodes; k++)
        {
        for ( j = 0; j < totalnodes; j++)
              {
              minnodedist[k][j]=temp[k][j];
              }
        minnodedist[k][k]=0.0;
        }
    }


  /*=========================================================*/
  /*= Begin Loop: One iteration = one service performed     =*/
  /*=========================================================*/

   do
   {

  /*=========================================================*/
  /*= Find arrivals to system during last service if any    =*/
  /*=========================================================*/

    do
  {
  if ( newtime[kk] >= 29998 ) break;
  if (fseek(fpy3,startread,SEEK_SET)!=0) fprintf(fprn,"\nError 1.017\n");
    do
      {
       if(fread( &timeline, sizeof(struct timestruct), 1, fpy3 ) != 1) {fprintf(fprn,"\nError
2.1052\n");exit(-1);}
       if((lastread = ftell(fpy3)) == -1L){fprintf(fprn,"Error 0.01");exit(-1);}
       if( timeline.time >  freetime ){break;}
       idum = timeline.node;
       q[idum].qlength = q[idum].qlength + 1;
       recent[idum] = lastread - offset;
       latest[idum] = timeline.possys;
      }  while(1);
   startread = lastread-offset;
   qq=0;
   for (i=0; i < totalnodes; i++) qq = qq + q[i].qlength;
   if ( qq == 0 ){ freetime = timeline.time; test1 = 1;}
   else test1 = 0;
   } while (test1);

/*==========================================================*/
/*= Strategies based on following a TSP path               */
/*= Note: this strategy allows the server to skip the next  */
/*= if it has an empty queue. Only if all nodes have customers*/
/*= will the TSP path be followed exactly                   */
/*==========================================================*/

  switch(sub)
{

case 1:case 11:
  /*************************************************************/
  /*** STRATEGY: Serve one customer at moston each node of path*/
  /*************************************************************/

        qq = 0;  /*Test Variable to prevent infinite loops*/
        while(1)
         {
         qq++;
         if ( jj == totalnodes - 1)
             {
             jj = 0; /*loop to beginning i.e. go from 9 to 0*/
             }
         else { jj = jj + 1;}
         if ( q[tsporder[jj]].qlength != 0 || qq == totalnodes) break;
         }
        kk = tsporder[jj];
        /*DEBUGfprintf(fprn,"\nNext node is %d", kk);*/
        break;

case 2:case 12:
  /*************************************************************/
  /*** STRATEGY: Serve all customers at current node before going*/
  /***  to next node on TSP tour                               */
  /*************************************************************/

        kk = presentnode;
        if ( q[kk].qlength == 0 )
         {
         qq = 0;
         while(1)
              {
              qq++;
              if ( jj == totalnodes - 1)
                 {
                 jj = 0;
                 }
              else { jj = jj + 1;}
```

```
            if ( q[tsporder[jj]].qlength != 0 || qq == totalnodes break;
            }
        kk = tsporder[jj];
         /*  fprintf(fprn,"\nNext node is %d", kk);*/
        }
       break;

case 3:case 13:
  /****************************************************************/
  /*** STRATEGY: Same as above but server must move after       */
  /*** all original customers or when queue is empty            */
  /****************************************************************/

       qq=0;
       kk = presentnode;
       if ( q[kk].qlength == 0 || lastserved == last_in )
        {
        while(1)
            {
            qq++;
            if ( jj == totalnodes - 1)
               {
               jj = 0;
               }
            else { jj = jj + 1;}
            if ( q[tsporder[jj]].qlength != 0 || qq == totalnodes) break;
            }
        kk = tsporder[jj];
        last_in = latest[kk];
          /* fprintf(fprn,"\nNext node is %d", kk);*/
        }
       break;

case 4:case 14:
  /*******************************************************************/
  /*** STRATEGY: same as above but must move on after beta customers*/
  /***  are served                                                  */
  /*******************************************************************/

       numserved = 1+numserved;
       kk = presentnode;
       if ( q[kk].qlength == 0 ||  numserved == beta )
         {
        qq = 0;
        numserved = 0;
        while(1)
            {
            qq++;
            if ( jj == totalnodes - 1)
               {
               jj = 0;
               }
               else { jj = jj + 1;}
               if ( q[tsporder[jj]].qlength != 0 || qq == totalnodes) break;
               }
           kk = tsporder[jj];
             /*DEBUGfprintf(fprn,"\nNext node is %d", kk);*/
            }
          break;

      case 5:case 15:
        /**************************************************************/
        /*** STRATEGY: Serve until server has spent Rho*Beta at node */
        /**************************************************************/

          kk = presentnode;
          if ( q[kk].qlength == 0 ||  timeserved >= beta*rhoestimate)
            {
           qq=0;
           timeserved = 0;
           while(1)
                {
                qq++;
                if ( jj == totalnodes - 1)
                   {
                   jj = 0;
                   }
                else { jj = jj + 1;}
                if ( q[tsporder[jj]].qlength != 0 || qq == totalnodes) break;
                }
           kk = tsporder[jj];
           /*DEBUGfprintf(fprn,"\nNext node is %d", kk);*/
           }
          break;
       default:
          fprintf(fprn,"\nNo such Selection");
          exit(-1);
    }

    /*=========================================================*/
    /*= Find First-In member of current queue             ===*/
    /*=========================================================*/

    if ( ntest[kk] == 0)
    {
    ldum = recent[kk]; ntest[kk] = 1;
    while(1)
       {
       if (fseek(fpy3,ldum,SEEK_SET)!=0) fprintf(fprn,"\nError 1.013\n");
       if (fread( &timeline, sizeof(struct timestruct), 1, fpy3 ) != 1) {fprintf(fprn,"\nError
2.106\n");exit(-1);}
       ldum = offset*(timeline.pred-1);
       /*DEBUGfprintf(fprn,"\n%d\t%d\t%d\t%d", timeline.node,timeline.possys,timeline.pred,ti-
meline.succ);*/
```

```
          if ( timeline.possys == newtime[kk]) break;
          if ( timeline.pred == 29999 ) break;
          }
        }
        else
          {
          ldum = offset*(newtime[kk]-1);
          if (fseek(fpy3,ldum,SEEK_SET)!=0) fprintf(fprn,"\nError 1.013\n");
          if (fread( &timeline, sizeof(struct timestruct), 1, fpy3 ) != 1) {fprintf(fprn,"\nError
2.106\n");exit(-1);}
          /*DEBUGfprintf(fprn,"\n-->%d\t%d\t%d\t%d", timeline.node,timeline.possys,timeline.pred-
,timeline.succ);*/
          }
         newtime[kk] = timeline.succ;
         q[kk].qlength = q[kk].qlength - 1;
         lastserved = timeline.possys;
         timeserved = timeserved + timeline.remaining;
      /*DEBUGfprintf(fprn,"\n -------------------FINISH PREDECESSOR");*/

      /*========================================================*/
      /*= Find First-In member of current queue             ===*/
      /*========================================================*/

       subduration = ( minnodedist[presentnode][timeline.node]/speed)+ timeline.remaining;
       duration = subduration + (freetime-timeline.time);
       freetime = freetime + subduration;
       /*DEBUGfprintf(fprn,"\nsubduration: %f", subduration);
       fprintf(fprn,"\nduration: %f", duration);
       fprintf(fprn,"\nfreetime: %f", freetime);*/
       ++mm;

      /*========================================================*/
      /*= Write time in system for current customer to file =====*/
      /*========================================================*/

       if(fwrite( &duration, sizeof(float), 1, fpv ) != 1) {fprintf(fprn,"\nError
2.222\n");exit(-1);}

       presentnode = timeline.node;
        if (newtime[kk] >= 29998 ) break;
      } while(1);
      if (fclose(fpv) == EOF) { fprintf(fprn,"\nError 3.06"); exit(-1);}
      statnumserved = mm;
      return(0);
      }

/*==================================================================*/
/*==================================================================*/
/* Strategy V: Go to queue in Beta vicinity of the heaviest concentration */
/*  of customers or of service time required                       */
/*==================================================================*/
/*==================================================================*
```

```
float Serve_the_Vicinity(vfile,fpy4,end,sub) /*temp file, pointer to arrival */
                              /* file and end of file marker  */
     char *vfile;
     int end;
     FILE *fpy4;
     int sub;
                         /*****************************/
                         /* Documentation for        */
                         /* most ststements following*/
                         /* In SERVE_THE_LONGEST     */
                         /* Function since routines  */
                         /* are essentially the same */
                         /* with the exception of    */
                         /* code in switch statement*/
                         /****************************/
     {
     float vclosest=0;
     static int vic[TOTLNODES][TOTLNODES];     /*nearest neighbor array     */
     int j=0,k=0,dm=0,qs=0;
     static int vicq[TOTLNODES];               /*concentration of customers at vicinity centered
at i*/
     static float vicqt[TOTLNODES];            /*concentration of service times at vicinity
centered at i*/
     float tlongest=0;
     int vv = START;
     int lt=0;
     static int lastservice[TOTLNODES],lastin[TOTLNODES];
                              /*arrays holding system positions of last served and*/
                              /*last arrived at each node                          */

     /*========================================================*/
     /*= Initialize varaibles                          ========*/
     /*========================================================*/

     int last_in=0, lastserved=0;
     int numserved=0;
     float timeserved=0.0;
     int presentnode = START;
     int i=0, qq=0, mm=0, idum=0, first=0, set=0, test=0, kk=0, longest=0;
     int test1=0,test0=0;
     float closest=0.0;
     float savetime=0.0,duration=0.0,freetime=0.0,subduration=0.0;
     long ldum=0L,offset=0L,lastread=0L,startread=0L;
     FILE *fpv;
     fprintf(stderr,"\nServe the Heaviest Vicinity\n");

     for ( i = 0 ; i < totalnodes; i++ )
          {
          lastservice[i]=0;
          lastin[i]= 0;
          vicq[i] = 0;
          vicqt[i] = 0.0;
          for ( j = 0; j < totalnodes; j++)
          {
          vic[i][j]=0;
          temp[i][j]=0.0;
          }
     }
 for ( i=0; i < totalnodes; i++ )
     {
     ntest[i] = 0;
     q[i].qlength = 0;
     recent[i] = 0L;
     newtime[i] = 0;
     latest[TOTLNODES];
     }
offset = sizeof(struct timestruct);
freetime=0.0; first = 29999; test1=0;
test0=0; startread=0L; kk=0; longest=0;
mm=0; tlongest=0.0;
if ( (fpv = fopen(vfile,"wb")) == NULL ) { fprintf(fprn,"\nError 9.11\n"); exit(-1); }

/*======================================================*/
/*= Find Beta Vicinities: a vicinity is defined by how the  */
/*= set of beta nodes closest to node i                 */
/*======================================================*/

if (beta > totalnodes) {fprintf(fprn,"\n Beta > totalnodes"); exit(-1);}
for ( i = 0; i < totalnodes; i++ )
     {
     for ( k = 0; k < totalnodes; k++ )
          {
          temp[i][k] = minnodedist[i][k];
          }
     }
 for ( i= 0; i < totalnodes; i++)
{
for ( k= 0; k < totalnodes; k++)
    {
    vclosest = maxnodedistance;
    for ( j= 0; j < totalnodes; j++ )
     {
     if ( temp[i][j] < vclosest )
          {
          vclosest  = temp[i][j];
          vic[i][k] = j;      /*vic[2][3]=j means fourth   */
          dm = j;             /*closest node to i is j,    */
          }                   /*it also says that j belongs*/
     }                            /*to any beta vicinity of i  */
                         /*if beta is greater than 4  */
    temp[i][dm] = maxnodedistance*(k+1);
    }
}
```

```
/*======================================================*/
/*= Implement Strategy of  Serving Beta Vicinities as    ==*/
/*= individual nodes were served in Closest strategy     ==*/
/*======================================================*/

 do
 {

/*======================================================*/
/*=  Find Arrivals during last service            ========*/
/*======================================================*/

  do
{
if ( newtime[kk] >= 29998 ) {break;}
if (fseek(fpy4,startread,SEEK_SET)!=0) fprintf(fprn,"\nError 1.017\n");
 do
   {
    if(fread( &timeline, sizeof(struct timestruct), 1, fpy4 ) != 1) {fprintf(fprn,"\nError 
2.1052\n");exit(-1);}
    if((lastread = ftell(fpy4)) == -1L){fprintf(fprn,"Error 0.01");exit(-1);}
    if( timeline.time >  freetime ){break;}
    idum = timeline.node;
    q[idum].qtime   = q[idum].qtime + timeline.remaining;
    q[idum].qlength = q[idum].qlength + 1;
    recent[idum] = lastread - offset;
    latest[idum] = timeline.possys;
   }  while(1);
 startread = lastread-offset;
 qq=0;
 for (i=0; i < totalnodes; i++) qq = qq + q[i].qlength;
 if ( qq == 0 ){ freetime = timeline.time; test1 = 1;}
 else test1 = 0;
 } while (test1);

/*======================================================*/
/*= Beta Vicinity Strategies                      ========*/
/*======================================================*/

/*======================================================*/
/*= Measure Vicinity Statistics                   ========*/
/*======================================================*/

qs = 0.0;
for ( j = 0; j < beta; j++)
    {
    qs = qs + q[ vic[vv][j] ].qlength;
    }

/*======================================================*/
/*= Start Strategies                                          ========*/
/*======================================================*/
```

```
switch(sub)
{

case 2:
/*****************************************************************/
/** STRATEGY: Find Vicinity with heaviest amount of service    */
/** requirement, go to it if servers current vicinity is empty*/
/*****************************************************************/

if ( qs == 0.0 )
 {
  for ( i = 0; i < totalnodes; i++ )
    {
    vicqt[i] = 0.0;  /*Calculate weights at each vicinity*/
    for ( j = 0; j < beta; j++)
       {
       vicqt[i] = vicqt[i] + q[ vic[i][j] ].qtime;
       }
    vicqt[i] = vicqt[i] + minnodedist[presentnode][i]/speed;
    }
  tlongest = 0.0;        /*Select Vicinity with highest weight*/
  for ( i = 0; i < totalnodes; i++ )
     {
     if ( vicqt[i] > tlongest )
     {
     /*DEBUGfprintf(fprn,"\nVicinity %d weight = %f",i,vicqt[i]);*/
     tlongest = vicqt[i];
     vv = i;
     }
     }
  }
 break;

 case 4:
 /*****************************************************************/
 /*** STRATEGY: Same as above, but serve only original customers*/
 /*** before checking which vicinity has heaviest weight, server*/
 /*** may stay at current vicinity if reamins the heaviest      */
 /*****************************************************************/

lt = 0;
for ( j = 0; j < beta; j++)
   {
   if ( lastservice[vic[vv][j]] = lastin[vic[vv][j]] ) {++lt;}
   }
if ( qs == 0.0 || lt == beta )
   {
   for ( i = 0; i < totalnodes; i++ )
     {
     vicqt[i] = 0.0;
     for ( j = 0; j < beta; j++)
        {
```

```
        vicqt[i] = vicqt[i] + q[ vic[i][j] ].qtime;
        }
      vicqt[i] = vicqt[i] + minnodedist[presentnode][i]/speed;
      }
    tlongest = 0.0;
    for ( i = 0; i < totalnodes; i++ )
      {
      if ( vicqt[i] > tlongest )
      {
      tlongest = vicqt[i];
      vv = i;
      }
      }
    for ( j = 0; j < beta; j++)
      {
      lastin[vic[vv][j]] = latest[vic[vv][j]];
      }
  }
break;

 case 1:
 /************************************************************************/
 /*** STRATEGY: Same as first strategy but now weights are            **/
 /*** determined by queue lengths. Leave only when vicinity is empty**/
 /************************************************************************/

 if ( qs == 0.0 )
  {
   for ( i = 0; i < totalnodes; i++ )
     {
     vicq[i] = 0;
     for ( j = 0; j < beta; j++)
        {
        vicq[i] = vicq[i] + q[ vic[i][j] ].qlength;
        }
     }
   longest = 0;
   for ( i = 0; i < totalnodes; i++ )
      {
      if ( vicq[i] > longest )
      {
      /*DEBUGfprintf(fprn,"\nVicinity %d length = %d",i,vicq[i]);*/
      longest = vicq[i];
      vv = i;
      }
      }
   }
 break;
 case 3:
 /*******************************************************************/
 /*** STRATEGY:Same as above but server is allowed to move after*/
 /*** current customers are served. May remain at current if it */
 /*** is still the vicinity with the most customers in queue    */
 /*******************************************************************/

lt = 0;
for ( j = 0; j < beta; j++)
   {
   if ( lastservice[vic[vv][j]] = lastin[vic[vv][j]] ) {++lt;}
   }
if ( qs == 0.0 || lt == beta )
   {
   for ( i = 0; i < totalnodes; i++ )
     {
     vicq[i] = 0;
     for ( j = 0; j < beta; j++)
        {
        vicq[i] = vicq[i] + q[ vic[i][j] ].qlength;
        }
     }
   longest = 0.0;
   for ( i = 0; i < totalnodes; i++ )
      {
      if ( vicq[i] > longest )
      {
      longest = vicq[i];
      vv = i;
      }
      }
   for ( j = 0; j < beta; j++)
     {
     lastin[vic[vv][j]] = latest[vic[vv][j]];
     }
  }
 break;

 default:
 fprintf(fprn,"\nNo Such Choice");
 exit(-1);
 }

/*=======================================================*/
/*==     node vv is the heaviest node                    */
/*==     vic[vv][i] are eligible for visit (i < beta )   */
/*==   Go to closest non-empty in the vicinity           */
/*=======================================================*/
```

```
        if ( q[presentnode].qlength == 0)
          {
          for ( j = 0; j < beta; j++)
             {
             closest = maxnodedistance;
             if ( minnodedist[presentnode][ vic[vv][j] ] < closest )
              {
              if ( q[ vic[vv][j]  ].qlength != 0 )
                 {
                 closest = minnodedist[presentnode][ vic[vv][j] ];
                 kk = vic[vv][j];
                 }
              }
             }
          }
       else { kk = presentnode; }


       /*  fprintf(fprn,"\npresent node> %d vicinity> %d",kk,vv);
           fprintf(fprn,"\n   qt   node> %d qt vicin> %d",q[kk].qlength,q[vv].qlength);
           fprintf(fprn,"\n nodes in this vicinity");
           for ( j = 0; j < beta; j++)
           {
           fprintf(fprn,"\t%d",vic[vv][j]);
           }  */


      /*=======================================================*/
      /*=  Find First-In Member of Queue              ========*/
      /*=======================================================*/

       if (ntest[kk] == 0)
       {
       ldum = recent[kk]; ntest[kk] = 1;
       while(1)
          {
          if (fseek(fpy4,ldum,SEEK_SET)!=0) fprintf(fprn,"\nError 1.013\n");
          if (fread( &timeline, sizeof(struct timestruct), 1, fpy4 ) != 1) {fprintf(fprn,"\nError
2.106\n");exit(-1);}
          ldum = offset*(timeline.pred-1);
          /*DEBUGfprintf(fprn,"\n%d\t%d\t%d\t%d", timeline.node,timeline.possys,timeline.pred,ti-
meline.succ);*/
          if ( timeline.possys == newtime[kk]) {break;}
          if ( timeline.pred == 29999 ) {break;}
          }
        }
       else
          {
          ldum = offset*(newtime[kk]-1);
          if (fseek(fpy4,ldum,SEEK_SET)!=0) fprintf(fprn,"\nError 1.013\n");
          if (fread( &timeline, sizeof(struct timestruct), 1, fpy4 ) != 1) {fprintf(fprn,"\nError
2.106\n");exit(-1);}
```

```
          /*DEBUGfprintf(fprn,"\n-->%d\t%d\t%d\t%d", timeline.node,timeline.possys,timeline.pred-
,timeline.succ);*/
          }

       if (timeline.node != kk) {fprintf(fprn,"FATAL");exit(0);}
       newtime[kk] = timeline.succ;
       q[kk].qtime = q[kk].qtime - timeline.remaining;
       q[kk].qlength = q[kk].qlength - 1;
       lastservice[kk] = timeline.possys;
       lastserved = timeline.possys;
       timeserved = timeserved + timeline.remaining;

         /*DEBUGfprintf(fprn,"\n ------------------FINISH PREDECESSOR");*/

      /*=====================================================*/
      /*=  Calculate Statistics                      ========*/
      /*=====================================================*/


       subduration = ( minnodedist[presentnode][timeline.node]/speed)+ timeline.remaining;
       duration = subduration + (freetime-timeline.time);
       freetime = freetime + subduration;
       ++mm;

      /*=====================================================*/
      /*=  Write Stats to temporary file and continue loop ======*/
      /*=====================================================*/

       if(fwrite( &duration, sizeof(float), 1, fpv ) != 1) {fprintf(fprn,"\nError
2.223\n");exit(-1);}

       presentnode = timeline.node;
        if (newtime[kk] >= 29998 ){break;}
      } while(1);
      if (fclose(fpv) == EOF) { fprintf(fprn,"\nError 3.06"); exit(-1);}
      statnumserved = mm;
      return(0);
      }


/*=================================================================*/
/*=================================================================*/
/*==  Utility and Data Read Files       ===========================*/
/*=================================================================*/
/*=================================================================*/

Read_Distance_File(xfile)
    char *xfile;
    {
    int i=0,j=0;
    FILE *fp;
```

```
     /*======================================================*/
     /*= Reads data from argument 2 file of DINTSP.C    ========*/
     /*======================================================*/

     maxnodedistance = 0.0;
     if( (fp = fopen(xfile,"rb")) == NULL) {fprintf(fprn,"Error 1: Distance File Missing?\n");
exit(0);}
     for (j =0; j < totalnodes; j++)
          {
          for ( i = 0; i < totalnodes; i++ )
               {
               if ( fread( &minnodedist[i][j], sizeof(float), 1, fp ) != 1 ) {fprintf(fprn,"Er-
ror 2\n");exit(0);}
               maxnodedistance = maxnodedistance + minnodedist[i][j]/2.0;
               }
          }
     if (fclose(fp) == EOF) fprintf(fprn,"Error 3.04\n");
     fprintf(stderr,"\n\nFinished Reading Distance File %s \n", xfile);
     return(0);
     }

Read_TSP_File(tspfile)
     char *tspfile;
     {
     FILE *fptsp;
     int j=0;

     /*======================================================*/
     /*= Reads data from argument 3 file of DINTSP.C    ========*/
     /*======================================================*/

       if( (fptsp = fopen(tspfile,"rb")) == NULL) {fprintf(fprn,"Error 9.03: Shortest Circuit
File Missing?\n"); exit(0);}
       fprintf(fprn,"\nTsp Order:");
       for (j =0; j < totalnodes; j++)
          {
          if ( fread( &tsporder[j], sizeof(int), 1, fptsp ) != 1 ) fprintf(fprn,"Error 2\n");
          fprintf(fprn,"\t%d",tsporder[j]);
          }
       fprintf(fprn,"\n");
       if (fclose(fptsp) == EOF) fprintf(fprn,"Error 9.04\n");
       fprintf(stderr,"\n\nFinished Reading TSP File %s \n", tspfile);
     return(0);
     }

Read_Parameter_File(pfile)
     char *pfile;
     {
     FILE *fp;
     int i=0;
     /*======================================================*/
     /*= Reads Parameter File from SIMM.C argument 3    ========*/
     /*======================================================*/


     if ( (fp = fopen(pfile,"rb")) == NULL ){ fprintf(fprn,"\nError 6.11: Parameter File Mis-
sing?\n"); exit(-1); }
     for (i = 0; i < totalnodes; i++)
          {
          if(fread( &node[i], sizeof(struct nodestruct), 1, fp ) != 1){fprintf(fprn,"\nError
6.12\n");exit(1);}
          }
     if (fclose(fp) == EOF) { fprintf(fprn,"\nError 6.13"); exit(-1);}


     fprintf(stderr,"\n\nFinished Reading : Parameter File \n\n");
     return(0);
     }

/*===================================================================*/
/*===================================================================*/
/*== CALCULATE STATISTICS                        ====================*/
/*===================================================================*/
/*===================================================================*/

Calculate_Stats(durfile,meanfile,sub)
     char *durfile;      /*temporary file created by strategy files*/
     char *meanfile;     /*temporary file created by strategy files*/
     int sub;            /*dummy parameter*/
     {
     FILE *fpp;
     FILE *fpm;
     float end;
     float  Find_End();
     float Calculate_Mean();
     float Calculate_Variance();

     /*======================================================*/
     /*= Open temp file for read and arg[8] file for text write=*/
     /*======================================================*/

     if ( (fpp = fopen(durfile,"rb")) == NULL )
          { fprintf(fprn,"\nError 1.11: Arrival File Missing?\n"); exit(-1); }
     if ( (fpm = fopen(meanfile,"w")) == NULL )
               { fprintf(fprn,"\nError 1.11\n"); exit(-1); }

     /*======================================================*/
     /*= Calculate Mean and Variance                 ========*/
     /*======================================================*/
```

```
        end =  Find_End(fpp);
        acc[id].mean = Calculate_Mean(fpp,fpm,end,sub);
        acc[id].variance = Calculate_Variance(fpp,acc[id].mean,end);

    /*================================================= =======*/
    /*= Close both files                                ========*/
    /*=========================================================*/

     if (fclose(fpm) == EOF)
             { fprintf(fprn,"\nError 1.12"); exit(-1);}
    if (fclose(fpp) == EOF)
       { fprintf(fprn,"\nError 1.12"); exit(-1);}
    return(0);
    }

float Find_End(fpp)
    FILE *fpp;
    {
    float endd=0.0,lendd=0.0,fendd=0.0,flendd=0.0;
    int nn=0;
    int mm =1;

    /*=========================================================*/
    /*= find last two entries of temp file as EOF marker      */
    /*=========================================================*/

    while(1)
    {
        if (fseek(fpp, -sizeof(float)*((long) mm) , SEEK_END) != 0)
         {fprintf(fprn,"\nError 1.10\n");}
        if(fread( &endd, sizeof(float), 1, fpp ) != 1)
         {fprintf(fprn,"\nError 1.12\n");exit(1);}
        if( endd <= 0.0 ) ++mm;
        else break;
    }
        ++mm;
        if (fseek(fpp, -sizeof(float)*((long) mm) , SEEK_END) != 0)
         {fprintf(fprn,"\nError 1.10\n");}
        if(fread( &fendd, sizeof(float), 1, fpp ) != 1)
         {fprintf(fprn,"\nError 1.12\n");exit(1);}

    fprintf(fprn,"\n\nEnd of File Marker: %f\n\n",endd);

    /*=========================================================*/
    /*= Count number of data points in file             ========*/
    /*=========================================================*/

    if (fseek(fpp, OL , SEEK_SET) != 0)
         {fprintf(fprn,"\nError 1.10\n");}
    while(1)
        {
        flendd = lendd;
        if(fread( &lendd, sizeof(float), 1, fpp ) != 1)
         {fprintf(fprn,"\nEOF\n");break;}
        ++nn;
        if ( lendd == endd &&  flendd == fendd ) break;
        }
    startlag = nn/CUTFRACTION;
    fprintf(fprn,"\nNumber of beginning observations cut %d\n",startlag);
    return(endd);
    }

float Calculate_Mean(fpp,fpm,ende,sub)
    FILE *fpp;
    FILE *fpm;
    float ende;
    int sub;
    {
    int n = 0,j=0;
    float mean=0.0,accumulate=0.0,durati=0.0;
    float quarter=0.0,half=0.0,threefourths=0.0;

    /*=========================================================*/
    /*=  Fraction where mean durations are sampled to check   ==*/
    /*=   for stability                                       ==*/
    /*=========================================================*/

    quarter      = (int) (startlag*CUTFRACTION/4);
    half         = (int) (startlag*CUTFRACTION/2);
    threefourths = (int) (startlag*CUTFRACTION*(3/4));

    /*=========================================================*/
    /*=  Calculate Mean after cutoff determined in Find_End   ==*/
    /*=  Write results to arg[8] text file                    ==*/
    /*=========================================================*/

    if (fseek(fpp,sizeof(float)*((float)startlag), SEEK_SET) != 0) {fprintf(fprn,"\nError
1.10\n");}
    fprintf(fpm,"\n\tObsv.\tDurat.\tMean Duration");
    while(1)
        {
        n=n+1;
        if ( fread( &durati, sizeof(float), 1, fpp ) != 1 ) fprintf(fprn,"Error 2.31\n");
        accumulate = accumulate + durati;
        mean = accumulate/n;
        /**** Keep Samples of Mean to test Stabil.***/
        if ( n == quarter     ) quartermean      = mean;
        if ( n == half        ) halfmean         = mean;
        if ( n == threefourths) threefourthsmean = mean;
        /**** Print Running Mean to File ************/
        j = n + startlag;
        fprintf(fpm,"\n%d\t%f\t%f",j,durati,mean);
        if( durati == ende ) break;
        }
```

```
     fprintf(stderr,"\n\nMean Read Complete\n\n");
     return(mean);
     }

 float Calculate_Variance(fpp,mean,endf)
     FILE *fpp;
     float mean;
     float endf;
     {
     float variance=0.0,duration=0.0;
     float addvalue=0.0;
     int n=0;
     if (fseek(fpp,sizeof(float)*((long)startlag), SEEK_SET) != 0) {fprintf(fprn,"\nError
5.01\n");}
     do
        {
        if(fread( &duration, sizeof(float), 1, fpp) != 1)
              {fprintf(fprn,"\nError 5.10\n");exit(1);}
        addvalue = addvalue + (duration - mean)*(duration - mean);
        n=n+1;
        if (duration == endf) break;
        } while(1);
     variance = addvalue/n;
     fprintf(stderr,"\n\n Variance read complete.\n\n");
     return(variance);
     }
```

7.3 SIMM.C

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <fcntl.h>
#include <math.h>
#include <float.h>
#include <sys\timeb.h>

/*              The organization of this program is as follows:              */
/* Main ----> Create_Arrival_File                                             */
/* Create_Arrival_File  --->Assign_Arr_Parameters--> Calculate_ArrSer_ParmetersR */
/*                                              --> Calculate_ArrSer_Parmetersd */
/* Create_Arrival_File  ---> Create_Parameter_File                            */
/* Create_Arrival_File  ---> Calculate_Arrival_Times                          */
/* Create_Arrival_File  ---> Create_Final_File---->   Create Successor Array  */
/*-------------------------------------------------------------------------*/
/*Create_Arrival_File : Only Directs Execution                                */
/*Assign_Arr_Parameters: Switch between two methods of creating parameters    */
/*                     Randomly or Deterministically                          */
/*Calculate_Arrival_Times: Creates Unsorted file of arrival times             */
/*Create_Parameter_File: writes file to be used by TIMM as arg <3>            */
/*Create_Final_File and Create_Successor_Array create sorted doubly linked list */
/*  of arrivals with time of arrival, individual service requirement and
arr. node */
```

```
#define M1 259200        /*integer values used in random generating routines */
#define IA1 7141         /* as derived fron "Numerical Recipes in C" by      */
#define IC1 54773        /*Press,Flannery et. al (see Ran1,Ran0 and Gasdev   */
#define RM1 (1.0/M1)
#define M2 134456
#define IA2 8121
#define IC2 28411
#define RM2 (1.0/M2)
#define M3 243000
#define IA3 4561
#define IC3 51349
#define TOTLNODES 70  /* May be greater depending on Hardware static......*/
int totalnodes;       /*Graph size*/
long int totalbytes;  /*Maximum bytes for temporary file, final file will 2x as large
*/
float randl= 32767.0; /*Microsoft C rand() ranges in int, 0 to 32767, randl acts as unit
divisor*/
int jarrive;         /* variables to hold use inputted lambda,mu and std*/
int jservex;
int jservev;
             /*structure to hold parameter data, TIMM receives data in same format*/
struct nodestruct {
     float servicereq_X;
     float servicereq_V;
     float arrivalrate;   }
struct nodestruct node[100];
             /*structure to hold temporary unsorted and linked arrival data*/
struct arrstruct    {
                         int node;
                         float time;
                         }
struct arrstruct arrival;
            /*structure to hold doubly linked list that will be paseed to TIMM in the final
file*/
            /* argument 3 in SIMM, argument 2 in TIMM
*/
struct timestruct  {
                       float time;
                       float remaining;
                       int node;
                       int pred;
                       int possys;
                       int succ;
                       }
struct timestruct timeline;
            /* text output file */
char logfile[] = "slog.c";
FILE *fprn;

main(argc, argv)
    int argc;
    char *argv[];
    {
    char ch;
    int i;
```

```
/*-----------------------------------------------------*/
/*- Open Text output file and integrate user input from--*/
/*- command line.                                      -*/
/*-----------------------------------------------------*/


if ( (fprn = fopen(logfile,"a")) == NULL )
     { fprintf(stderr,"\nError 1.11\n"); exit(0); }

if(argc !=10+1)
     {fprintf(stderr,"Wrong Number of Parameters");
      exit(0);}
if(1 != sscanf(argv[1],"%d",&totalnodes))
     {fprintf(stderr,"\nErr:Bad Arrival Rate\n");
     exit(0);}
if ( totalnodes > TOTLNODES )
     {
     fprintf(stderr, "\n +70 nodes may cause memory overflow: check source code and hard-
ware ");
     exit(0);
     }
if(1 != sscanf(argv[7],"%d",&jarrive))
     {fprintf(stderr,"\nErr:Bad Arrival Rate\n");
     exit(0);}
if(1 != sscanf(argv[8],"%d",&jservex))
     {fprintf(stderr,"\nErr:Bad Service Mean over Arrival Rate\n");
     exit(0);}
if(1 != sscanf(argv[9],"%d",&jservev))
     {fprintf(stderr,"\nErr:Bad Service Variance over Arrival Rate\n");
     exit(0);}
if(1 != sscanf(argv[10],"%ld",&totalbytes))
     {fprintf(stderr,"\nErr:Bad #Bytes\n");
     exit(0);}
fprintf(stderr,"\nOutput sent to: %s on this directory",logfile);


totalbytes = (totalbytes/(12*totalnodes))*(12*totalnodes); /*make sure data file is divis-
ible by datastructures*/

/*-----------------------------------------------------*/
/*- Write Parameters Generated and Status comments     --*/
/*-  to text output file                               -*/
/*-----------------------------------------------------*/

fprintf(fprn,"\n***************************************************");
fprintf(fprn,"\n**********    NEXT RUN    **********************");
fprintf(fprn,"\n***************************************************");
for ( i=0; i < argc; i++)
     {
     fprintf(fprn,"\nargu: %d %s",i,argv[i]);
     }

/*-----------------------------------------------------*/
/*-    Begin Execution                                --*/
/*-----------------------------------------------------*/
```

```
     Create_Arrival_File(argv[2],argv[3],argv[4],argv[5],argv[6]);
     if (fclose(fprn) == EOF)
        { fprintf(stderr,"\nError 3"); exit(0);}

     return(0);
     }


Create_Arrival_File(xfile,yfile,sfile,tst,tst2)
     char *xfile;
     char *yfile;
     char *sfile;
     char *tst;
     char *tst2;
     {
     int t;

     /*--------------------------------------------------*/
     /*-     Direct Execution                          --*/
     /*--------------------------------------------------*/


     Assign_Arr_Parameters(tst);
     Create_Parameter_File(sfile);
     Calculate_Arrival_Times(xfile);
     Create_Final_File(xfile,yfile,tst2);
     return(0);
     }

Assign_Arr_Parameters(tst)
     char *tst;
     {
     char ch;
     ch = *tst;

     /*--------------------------------------------------*/
     /*-     Direct Execution                          --*/
     /*--------------------------------------------------*/

     switch(ch)
         {
         case 'r':
         case 'R':
         Calculate_ArrSer_ParametersR();
         break;             /* Use randomly generated parameters*/
         case 'd':
         case 'D':
         Calculate_ArrSer_ParametersD();
         break;             /* Use User supplied parameters for all*/
         default:
         fprintf(stderr,"\n Failure: Choice only R or D\n");
         exit(0);
         }
     return(0);
     }
                        /* Parameters are generated with a Gaussian */
Calculate_ArrSer_ParametersR()  /* distribtion with mean supplied by user    */
     {                          /* Gasdev is a Gaussian random number gener-*/
```

```
   int i,j,l=0;          /* as supplied by "Numerical Recipes for "C"*/
   float arrivetot=0.0,servetot=0.0;
   float start;
   float Gasdev(i);
   struct timeb tval;
   ftime(&tval);
   start = Gasdev(&tval);

   fprintf(stderr,"\nPlease Wait: Random Arrival/Service Parameters being Generated\n");
   fprintf(fprn," Node  Arrival     Expected     Variance of\n");
   fprintf(fprn," #     Rate        Service      Service   \n");
   fprintf(fprn,"------------------------------------------\n");
   for ( i = 0; i < totalnodes; i++)
       {
       node[i].arrivalrate = fabs(Gasdev(&tval)*((float) 1.0/5.0)+ 1.0001)*
                     ( (float) jarrive/100.0);
       node[i].servicereq_X =fabs(Gasdev(&tval)*((float) 1.0/5.0)+ 1.0001)*
                       (1.0/( (float) jservex/100.0));
       while(1)
           {
           node[i].servicereq_V = fabs(Gasdev(&tval)*((float) 1.0/5.0)+ 1.0001)*
                         (1.0/( (float) jservev/100.0));
           if ( node[i].servicereq_X > 3.7*node[i].servicereq_V) break;
           }
       fprintf(fprn, "  %d\t%.5f\t\t%.5f\t\t%.5f\t\n", i,
           node[i].arrivalrate,
           node[i].servicereq_X,
           node[i].servicereq_V);
       /*Make sure that the queues do not explode*/
       arrivetot = arrivetot + node[i].arrivalrate;
       servetot  = servetot  + node[i].servicereq_X;
       }
   fprintf(fprn, "\nAverage Service Rate %f\nAverage Arrival Rate %f\n",
          servetot/totalnodes,
          arrivetot/totalnodes);
   return(0);
   }


Calculate_ArrSer_ParametersD(j1,j2,j3)/*Takes the user data and applies it to all*/
  int *j1,*j2,*j3;                 /* nodes equally                            */
  {
  int i,j,l;
  fprintf(stderr,"\nPlease Wait: Determined Arrival/Service Parameters being Generated\n");
  fprintf(fprn,"\nNode  Arrival     Expected     Variance of");
  fprintf(fprn,"\n #     Rate        Service      Service   ");
  fprintf(fprn,"\n------------------------------------------\n");
  for ( i = 0; i < totalnodes; i++)
      {
      node[i].arrivalrate  =  (float) jarrive/100.0;
      node[i].servicereq_X =  1.0/( (float) jservex/100.0);
      node[i].servicereq_V =  1.0/( (float) jservev/100.0);
      fprintf(fprn, "  %d\t%.5f\t\t%.5f\t\t%.5f\t\n", i,
          node[i].arrivalrate,
          node[i].servicereq_X,
          node[i].servicereq_V);
      }
  return(0);
  }
```

```
Calculate_Arrival_Times(xfile)
        char *xfile;
     {
     int i,t=0,l=0;
     float Ranl(idum);
     float rnd, randomtime, start;
     static float passvalue[100]; /*Record last arrival time and calculate        */
     FILE *fp;                    /* next arrival time by adding an exponential     */
     struct timeb tval;           /* distributed random variable with parameter lambda*/
     ftime(&tval);
     fprintf(stderr,"\nPlease Wait: Temporary (Unsorted) Arrival File Being Created\n");
     if ( (fp = fopen(xfile,"wb")) == NULL )
         { fprintf(fprn,"\nError 1\n"); exit(0); }


     /*---------------------------------------------------*/
     /*-Initialize point of deaparture for arrival time calcu.*/
     /*---------------------------------------------------*/

     for(i = 0; i < totalnodes; i++ )
     {
     passvalue[i] = 0.0;
     }
     start = Ranl(&tval);

     /*---------------------------------------------------*/
     /*- Loop creating temporary arrival file: arrival times  */
     /*--for each node are entered sequentially in parallel  -*/
     /*--stacks (i.e. (1:0,2:0,3:0,.........n:0),           -*/
     /*--            (1:1,2:1,3:1,.........n:1),....        -*/
     /*--where i:j is ariival j at node i.                   -*/
     /*---------------------------------------------------*/

     while( t <  (totalbytes/sizeof(struct arrstruct)) )
       {
         for(i = 0; i < totalnodes; i++ )
           {
           ftime(&tval);
           rnd =  Ranl(&tval);
           randomtime = -log(1-rnd+.0000001)/node[i].arrivalrate;  /*prevent generation of
zero value          */
           passvalue[i] = passvalue[i] + randomtime;             /* randomtime is
exponential interarrival time */
           arrival.time = passvalue[i];                          /* time of arrival in
system              */
           arrival.node = i;                                     /* identity of arrival by
location          */
           if ( t > totalbytes/sizeof(struct arrstruct) )        /* if exceeds then final
sorter will fail     */
               {fprintf(fprn,"\nBytes Exceeded at % d arrivals\n",t);break;exit(0);}
           if(fwrite( &arrival, sizeof(struct arrstruct), 1, fp ) != 1)
               {fprintf(fprn,"/nError 2/n");exit(0);}
           t++;
           /*printf("%d %f %d \n",t,arrival.time,arrival.node);*/
           }
       }
    if (fclose(fp) == EOF)
```

```
        { fprintf(fprn,"\nError 3"); exit(0);}
     fprintf(stderr,"\nFinished: Writing Temporary Arrival File %s\n", xfile);
     return(0);
     }


Create_Final_File(xfile,yfile,tst2)
char *xfile;
char *yfile;
char *tst2;
     {
     int i,j,tt,kk,l;
     static int last[100],posi[100];
     FILE *fpx, *fpy;
     char ch;
     static struct arrstruct endof[100],reader[100];
     static long int offset[100];
     long int offsetsize, o,p;
     float keep=0.0,start,diff=0.0,save=0.0,sumdiff=0.0,sumremaining=0.0;
     struct timeb tval;
     float Gasdev(idum);
     ftime(&tval);
     start = Ran1(&tval);
     offsetsize = sizeof(struct arrstruct)*totalnodes;
     for ( i = 0; i < totalnodes; i++)
        { offset[i] = 0;posi[i] = 0;last[i] = 29999;}
     kk=0;
     tt=0;
     l=0;
     ch = *tst2;

     /*-----------------------------------------------*/
     /*- Open Temp.File to read and Final file to write ------*/
     /*-----------------------------------------------*/

     if ( (fpx = fopen(xfile,"rb")) == NULL )
          { fprintf(fprn,"\nError 1.10\n"); exit(0); }
     if ( (fpy = fopen(yfile,"wb")) == NULL )
          { fprintf(fprn,"\nError 1.11\n"); exit(0); }


     /*-----------------------------------------------*/
     /*- Loop sorting arrivals from temporary file: the       */
     /*--method used exploits the fact the parallel stacks   -*/
     /*--are internally ordered. The next arrival is pulled  -*/
     /*--from the minimum value at the tops of each stack.The-*/
     /*--top is repositioned downward whenerver a top memeber-*/
     /*--is selected.                                        -*/
     /*-----------------------------------------------*/
     /*-----------------------------------------------*/
     /*- Find end of temporary file to serve as EOF indicator-*/
     /*- This is especially important since we want to stop   */
     /*  recording arrival data as soon as a customer arrives */
     /*  at a time exceeding our timehorizon, if not we will  */
     /*  essntially freeze one queue until the others catch up*/
     /* ,creating a conditioning error                        */
     /*-----------------------------------------------*/
     for (i = 0; i < totalnodes; i++)
          {
          p =  -(sizeof(struct arrstruct))*(totalnodes - i);
          fseek(fpx, p, SEEK_END);
          if(fread( &endof[i], sizeof(struct arrstruct), 1, fpx ) != 1) {fprintf(fprn,"\nError
2.1\n");exit(1);}
          }
     ftime(&tval);
     start = Gasdev(&tval);

     /*-----------------------------------------------------------*/
     /* Service requirements are generated now to to min. memory use*/
     /*-----------------------------------------------------------*/
     /* offset[i] identifies the top of stack i        */
     /* i identifies the stack position                */
     /* reader structure keeps the value of top member*/

     do
     {
     for (i = 0; i < totalnodes; i++)
          {
          o = (offsetsize * offset[i])+(i*sizeof(struct arrstruct));
          fseek( fpx, o , SEEK_SET);
          if(fread( &reader[i], sizeof(struct arrstruct), 1, fpx ) != 1) {fprintf(fprn,"\nError
2.201\n");exit(0);}
          }
     /* find minimum arrival time of top members of each stack       */
     save = keep;
     keep = reader[0].time;
     for (i = 0; i < totalnodes; i++)
          {
          if ( reader[i].time <=  keep )
             {
             kk = i;
             keep = reader[i].time;
             }
          }
     /*calculate useful information about each selected min. arrival*/
     diff = keep - save;
     offset[kk] = offset[kk]+1;    /*if selected move stack down one*/
     tt++;
     posi[kk]++;                   /*residual if one wants to add order of arrival at individual
nodes*/
     timeline.time = keep;         /*time of arrival                */
     timeline.node = kk;           /*node that arrived              */
     timeline.possys = tt;         /*position in system as a whole  */
     timeline.pred = last[kk];     /*who customers predecessor is   */
     timeline.succ = 29998;        /*fill in successor array to calculated later*/
     /*Switch allowing user or updater to determine the pdf used to derive the length*/
     /* of service requirements                                                       */
     switch(ch)
          {
          case 'G':
          case 'g':
             ftime(&tval);
             timeline.remaining = (Gasdev(&tval)*node[kk].servicereq_V)+
                        node[kk].servicereq_X;
             if (timeline.remaining < 0.0 )
```

```
            {
            fprintf(fprn,"Gaussian Mean too small: NEGATIVE SERVICE TIMES PRODUCED\n");
            fprintf(fprn,"   DATA TRUNCATED TO .0000001");
            timeline.remaining = 0.0000001;
            }
          break;
       case 'M':
       case 'm':
          ftime(&tval);
          timeline.remaining =
               -log(1-Ranl(&tval)+.0000001/1000)*node[kk].servicereq_X;
          break;
       default:
          fprintf(stderr,"\n Failure: No Such Distribution\n");
          exit(0);
       }
   last[kk] = tt;
   /*printf("%f\t%f\t%f\t%d\t%d\t%d\n",
           timeline.time,
           diff,
           timeline.remaining,
           timeline.node,
           timeline.possys,
           timeline.pred);*/
   /*This data is calculated to let user know whether the use has */
   /* created an unstable system or not                            */
   sumdiff = sumdiff + diff;
   sumremaining = sumremaining + timeline.remaining;

   /*-----------------------------------------------------*/
   /*-Write sorted entry to file                      ----*/
   /*-----------------------------------------------------*/

   if (fwrite( &timeline, sizeof(struct timestruct), 1, fpy ) != 1) {fprintf(fprn,"/nError
2.3/n");exit(0);}

   /*-----------------------------------------------------*/
   /*- Break out as soon as first customer reaches end of time*/
   /*-----------------------------------------------------*/

   if (endof[kk].time == timeline.time) break;


   } while (1);
   fprintf(fprn,"\nTotal Service Time %f / Total Arrival Times %f\n",
        sumremaining,sumdiff);


   /*-----------------------------------------------------*/
   /*- Close  Files                                   ----*/
   /*-----------------------------------------------------*/
```

```
     if (fclose(fpy) == EOF)
        { fprintf(fprn,"\nError 3"); exit(0);}
     if (fclose(fpx) == EOF)
        { fprintf(fprn,"\nError 3"); exit(0);}
     fprintf(stderr,"\nFinished: Creating Timeline File %s\n", yfile);
     Create_Successor_Array(yfile);
     return(0);
     }

Create_Successor_Array(yfile)
     char *yfile;
     {
     long p;
     int i;
     int j = 0;
     FILE *fpy;
     static int pred[100+1];
     struct timestruct update;
     struct timestruct timer;

     /*-----------------------------------------------------*/
     /*- Use predecessor array and sytem position to create -*/
     /*  successor array                                   -*/
     /*-----------------------------------------------------*/

     for(i = 0; i < totalnodes; i++) pred[i] = 29998;
     if ( (fpy = fopen(yfile,"r+b")) == NULL ) {fprintf(fprn,"\nError 1.11\n"); exit(0); }
     while(1)
         {
         j = j + 1;
     /* read file backwards deriving successors from  predecessors      */
         p = ( (long) (sizeof(struct timestruct))) *( (long) j);
         if (fseek(fpy, -p ,SEEK_END)!=0) fprintf(fprn,"\nError 1.016");
         if (fread( &update, sizeof(struct timestruct), 1, fpy ) != 1) {fprintf(fprn,"\nError
2.14: Successor Array Failed: File No Good \n");exit(1);}
           update.succ = pred[update.node];
           pred[update.node] = update.possys;


     /*-----------------------------------------------------*/
     /*- write successor values into final arrival file    -*/
     /*-----------------------------------------------------*/



         if (fseek(fpy,-p,SEEK_END)!=0) fprintf(fprn,"\nError 1.016");
         if(fwrite( &update, sizeof(struct timestruct), 1, fpy ) != 1) {fprintf(fprn,"\nError
2.1\n");exit(0);}
     /* stop as soon as you get to start of file              */
         if (update.possys == 1) break;
         }
     if (fclose(fpy) == EOF)
        { fprintf(fprn,"\nError 3"); exit(0);}
     fprintf(stderr,"\nSuccessor Array Created\n");
     return(0);
     }

 Create_Parameter_File(sfile)
 char *sfile;
     {
     int i;
     FILE *fps;

     if ( (fps = fopen(sfile,"wb")) == NULL )
         { fprintf(fprn,"\nError 3.10\n"); exit(0); }

     for (i = 0; i < totalnodes; i++)
         {
         if(fwrite( &node[i], sizeof(struct nodestruct), 1, fps ) != 1)
         {fprintf(fprn,"\nError 3.20\n");exit(1);}
         }

     if (fclose(fps) == EOF)
        { fprintf(fprn,"\nError 3"); exit(0);}

     fprintf(stderr,"\nFinished: Creating Parameter File %s\n", sfile);
     return(0);
     }

/*-----------------------------------------------------------------------*/
/*--   Routines Copied from Nunmerical Recipes in "C"--------------------*/
/*-----------------------------------------------------------------------*/


float Ran0(idum)          /* Random number generator that improves on system */
--------------------------/*   supplied rand(0)                                */
float Ran1(int *idum)
float Gasdev(idum)
     int *idum;
```

7.4 DIMTSP.C

```
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <fcntl.h>
#include <math.h>
#include <float.h>
#include <sys\timeb.h>
#define M1 259200      /* Seed Value for Random Number Generator      */
#define IA1 7141       /* Generator derived from "Numerical Recipes  */
#define IC1 54773      /* for C                                       */
#define RM1 (1.0/M1)
#define M2 134456
#define IA2 8121
#define IC2 28411
#define RM2 (1.0/M2)
#define M3 243000
```

```
#define IA3 4561
#define IC3 51349
#define IB1 1
#define IB2 2
#define IB5 16
#define IB18 131072
#define TOTLNODES 50  /* Must set within comiler and hardware limitations*/
              /* Microsoft C on Dos is limited to TOTLNIDES < 70  */

        TSP Algorithm is derived from Numeriacl Recipes for C
/************************ TSP Algorithm controls ******************************/
#define MAXPATHS_TRIED_PER_NODE 50 /* 100 is ideal but slow                  */
#define MAXCHANGE_PATH_PER_NODE 10  /* 10 is ideal but has min effect on speed */
#define MAX_INVERSE_SPEED 30        /* 100 is ideal but slow                  */
#define TFACTR 0.7;                 /*  .9 is ideal but slow                  */
#define ALEN(a,b,c,d) sqrt( ((b)-(a))*((b)-(a)) + ((d)-(c))*((d)-(c)) )      /**/
/******************************************************************************/

int totalnodes;          /* specified by uder in argument 1           */
int  maxnodedistance;    /* specified by use in argument  2           */
float randl= 32767.0;    /* divisor to get system supplied  rand()=(1,0)*/

static float nodedistance[TOTLNODES][TOTLNODES];
static float minnodedist[TOTLNODES][TOTLNODES];
static int list[TOTLNODES];   /* will hold special array to help */
static int iorder[TOTLNODES]; /*   distance matrix for circle    */
char logfile[] = "dlog.c";    /* output text file                */
FILE *fprn;                   /* pointer to output file for text */

main(argc, argv)
     int argc;
     char *argv[];
     {
     char ch;
     int i,j;
     if(argc != 5+1)
          {fprintf(stderr,"\nWrong Number of Parameters");
           exit(0);}
     fprintf(stderr,"\nOutput sent to %s on this directory",logfile);

      /*-----------------------------------------------------*/
      /*-   Open text file for output and status reports to users- --*/
      /*-----------------------------------------------------*/

     if ( (fprn = fopen(logfile,"a")) == NULL )
          { fprintf(stderr,"\nError 1.11\n"); exit(0); }
     fprintf(fprn,"\n**************************************************");
     if(argc != 5+1)
          {fprintf(stderr,"\nWrong Number of Parameters");
           exit(0);}


      /*-----------------------------------------------------*/
      /*-  Read and and input user supplied data           --*/
      /*-----------------------------------------------------*/
```

```
  if(1 != sscanf(argv[1],"%d",&totalnodes))
      {fprintf(stderr,"\n\nErr nodes\n\n");
       exit(0);}
  if( (totalnodes%2) != 0 ) fprintf(stderr,"Only Even Number of Nodes Allowed");
  if(1 != sscanf(argv[5],"%d",&maxnodedistance))
      {fprintf(stderr,"\n\nErr Distance\n\n");
       exit(0);}


   /*-------------------------------------------------------*/
   /*-  Initialize arrays  for circular graph distances      --*/
   /*-------------------------------------------------------*/

  for ( i = 0; i < totalnodes; i++) iorder[i]=i;
  for ( i = 0; i <= totalnodes/2; i++) list[i]=i;
  for ( i = 0; i <  totalnodes/2; i++) list[totalnodes-i] = i;
  for ( i=0; i < argc; i++)
      {
      fprintf(fprn,"\nargu: %d %s",i,argv[i]);
      }
  fprintf(fprn,"\n");


   /*-------------------------------------------------------*/
   /*-  Select TYPE  OF GRAPH                                --*/
   /*-------------------------------------------------------*/

  ch = *argv[4];
  switch(ch)
   {
   case 'n':
   case 'N':
    Create_Distance_File(argv[2]);       /*Non-Planar*/
    break;
   case 'p':
   case 'P':
    Create_Planar_File(argv[2],argv[3]);  /*Planar*/
    break;
   case 'c':
   case 'C':
    Create_Circle_File(argv[2]);         /*Circular */
    break;
   default:
    fprintf(stderr,"\nNon-Valid Parameter: ('n' or 'p' or 'c')");
    exit(0);
   }
  fprintf(fprn,"\n ************************************************");
  if (fclose(fprn) == EOF)
     { printf("\nError 3"); exit(0);}
  return(0);
  }

/*--------------------------------------------------------------*/
/*-   Functions Used to Create Planar Distance MAtrix   ----------------*/
/*-   And TSP path                                      ----------------*/
/*--------------------------------------------------------------*/
```

```
Create_Planar_File(pfile,tfile)
     char *pfile;
     char *tfile;
     {
     int i,j,n;
     FILE *fpl;

     /*-------------------------------------------------------------*/
     /*-  Find Straightline Distances                             --*/
     /*-  TSP path is also calculated                             --*/
     /*-------------------------------------------------------------*/

     Calculate_Planar_DistancesR(tfile);


     /*-------------------------------------------------------------*/
     /*-  Create File for input into TIMM as argument 4: Input Data */
     /*-------------------------------------------------------------*/

     if( (fpl = fopen(pfile,"wb")) == NULL) fprintf(fprn,"Error 1\n");
     for (j =0; j < totalnodes; j++)
          {
          for ( i = 0; i < totalnodes; i++ )
               {
               if ( fwrite( &minnodedist[i][j], sizeof(float), 1, fpl ) != 1 ) fprintf(fprn,"Er-
ror 2\n");
               }
          }
     if (fclose(fpl) == EOF) fprintf(fprn,"Error 3\n");
     /** Write Results to Status File ****************************/
     fprintf(fprn,"\n");
     for ( n = 0; n < totalnodes; n++ )
          {
          fprintf(fprn,"%6d",n);
          }
     for( i = 0; i < totalnodes; i++)
          {
          fprintf(fprn,"\n%d ",i);
          for ( n = 0; n < totalnodes; n++ )
               {
               fprintf(fprn,"%5.1f ",minnodedist[i][n]);
               }
          }
     fprintf(stderr,"\n\nFinished: Writing File %s \n", pfile);
     return(0);
     }

Calculate_Planar_DistancesR(tfile)
     char *tfile;
     {
     FILE *fpi;
     int i,j,l=0;
     float start;
     static float xm[TOTLNODES],ym[TOTLNODES]; /* Location Coordinates */
     struct timeb tval;                        /* for nodes on plane   */
     void Anneal();                            /* Numerical Recipes in  */
```

```
                         /* C code for finding TSP*/
ftime(&tval);                          /* Seeds for Random number*/
srand(tval.time+tval.millitm);         /* generators             */
fprintf(stderr,"\n\nPlease Wait: Random Location Generator\n\n");

/*-------------------------------------------------------------*/
/*-  Locations are generated with uniform pdf 0 to Maxnodedistance*/
/*-------------------------------------------------------------*/

for ( i = 0; i < totalnodes; i++)
    {
    ym[i] =(rand()/randl)*((float) maxnodedistance);
    }
for ( i = 0; i < totalnodes; i++)
    {
    xm[i] = (rand()/randl)*((float) maxnodedistance);
    }

/*-------------------------------------------------------------*/
/*-  Use locations to calculate distances                      */
/*-------------------------------------------------------------*/

fprintf(stderr,"\n\nPlease Wait: Distance Calculator\n\n");
for ( i = 0; i < totalnodes; i++)
    {
    for ( j = i; j < totalnodes; j++)
        {
        minnodedist[i][j] = sqrt (
            ( (ym[i] - ym[j])*(ym[i] - ym[j]) ) +
            ( (xm[i] - xm[j])*(xm[i] - xm[j]) )
                          );
         minnodedist[j][i] = minnodedist[i][j];
        }
    minnodedist[i][i] = 0.0;
    }

/*-------------------------------------------------------------*/
/*-  Use locations to find TSP path: Routine supplied by Numerical*/
/*-  Recipes in C pages 347-359
/*-------------------------------------------------------------*/

Anneal(xm,ym,iorder);
/*print order to status file **********************/
fprintf(fprn,"\n The TSP order is :\n");
for ( i = 0; i < totalnodes; i++)
    {
    fprintf(fprn,"%4d",iorder[i]);
    }
fprintf(fprn,"\n");
/* create input file for argument 10 of TIMM**********/

if( (fpi = fopen(tfile,"wb")) == NULL) fprintf(fprn,"Error 9.1\n");
for ( i = 1; i <= totalnodes; i++ )
    {
    if ( fwrite( &iorder[i], sizeof(int), 1, fpi ) != 1)
            fprintf(fprn,"Error 9.2\n");
```

```
          }
     if (fclose(fpi) == EOF) fprintf(fprn,"Error 9.3\n");
     return(0);
     }

void Anneal(x,y,iorder){} /*Code is in Numerucal Recipes in "C'*/

Create_Circle_File(xfile)
     char *xfile;
     {
     int i,j;
     FILE *fp;

     /** Find all node to node distance                         **/
     /** NOTE: this implementation only uses even number of nodes */
     Calculate_DistancesC();
     /** create output file for argument 3 of TIMM                */

     if( (fp = fopen(xfile,"wb")) == NULL) fprintf(fprn,"Error 1\n");
     /** Update this file                                          */
     for (j =0; j < totalnodes; j++)
          {
          for ( i = 0; i < totalnodes; i++ )
               {
               if ( fwrite( &minnodedist[i][j], sizeof(float), 1, fp ) != 1 ) fprintf(fprn,"Er-
ror 2\n");
               }
          }
     if (fclose(fp) == EOF) fprintf(fprn,"Error 3\n");
     fprintf(fprn,"\n\nFinished: Writing File %s \n", xfile);
     return(0);
     }

Calculate_DistancesC()
     {
     int i,j,n,l=0;
     static int list2[TOTLNODES];

     /*Distances are calculated by a peaking step function    */
     for ( i = 0; i < totalnodes; i++)
          {
          for (j = i; j < totalnodes; j++)
               {
               minnodedist[i][j]= list[j]*maxnodedistance;
               minnodedist[j][i]= minnodedist[i][j];
               }
          for (j = 1; j < totalnodes-1; j++) list2[j+1] = list[j];
          for (j = 0; j < totalnodes; j++)   list[j] = list2[j];
          minnodedist[i][i]=0.0;
          }
     /** Output result to text file ****************************/
     for ( n = 0; n < totalnodes; n++ )
               {
               fprintf(fprn,"%6d",n);
               }
     for( i = 0; i < totalnodes; i++)
```

```
                {
                 fprintf(fprn,"\n%d ",i);
                for ( n = 0; n < totalnodes; n++ )
                    {
                    fprintf(fprn,"%5.1f ",minnodedist[i][n]);
                    }
                }

        return(0);
        }

/*--------------------------------------------------------------------------*/
/*--Routines for Creating Non-Planar Graph Distances and Finding ----------*/
/*--Corresponding Shortest Paths                                  ----------*/
/*--------------------------------------------------------------------------*/

Create_Distance_File(xfile)
     char *xfile;
     {
     int i,j;
     FILE *fp;

     /*------------------------------------------------------------*/
     /*-  Generate Random Distances  and then Find Shortest Paths    */
     /*------------------------------------------------------------*/

     Calculate_DistancesR();
     Dikstra();

     /*------------------------------------------------------------*/
     /*- Create Ouput File and give it shortest path data           */
     /*------------------------------------------------------------*/

     if( (fp = fopen(xfile,"wb")) == NULL) fprintf(fprn,"Error 1\n");
     for (j =0; j < totalnodes; j++)
         {
         for ( i = 0; i < totalnodes; i++ )
             {
             if ( fwrite( &minnodedist[i][j], sizeof(float), 1, fp ) != 1 ) fprintf(fprn,"Er-
ror 2\n");
             }
         }
     if (fclose(fp) == EOF) fprintf(fprn,"Error 3\n");
     fprintf(fprn,"\n\nFinished: Writing File %s \n", xfile);
     return(0);
     }

Calculate_DistancesR()
     {
     int i,j;
     float start;
     struct timeb tval;
     ftime(&tval);
     srand(tval.time+tval.millitm);
     fprintf(stderr,"\n\nPlease Wait: Random Distance Generator\n\n");
```

```
/*-------------------------------------------------------------------*/   /*- Distances
are generated with a Uniform Distribution (1,Maxnodedistance*/
/*-------------------------------------------------------------------*/

     for ( i = 0; i < totalnodes; i++)
          {
          for ( j = i; j < totalnodes; j++)
               {
               nodedistance[i][j]=(rand()/randl)*( (float) maxnodedistance);
               nodedistance[j][i] = nodedistance[i][j];
               }
               nodedistance[i][i] = 0;
          }
     return(0);
     }

Dikstra()
          {
          int source,i,f,n,m;
          static int pred[TOTLNODES]; /*predecessor array                    */
          static int p_set[TOTLNODES];/*holds nodes with assured min. dist to source*/
          static t_set[TOTLNODES];    /*holds the rest of the nodes           */
          fprintf(stderr,"\n\nPlease wait: Dikstra\n\n");
          /*----------------------------------------------------------------*/

          /*- All-Path Shortest Paths is calculated by Repeated Implementations of   */
          /*- Dikstra's Generic Shortest Path Algorithm                              */
/*-------------------------------------------------------------------*/

          for (source = 0; source < totalnodes; source++)
               {
               /*INITIALIZE ARRAYS FOR NEW SOURCE*/
               for( i = 0; i < totalnodes; i++) p_set[i] = 0;
               for( i = 0; i < totalnodes; i++) t_set[i] = 1;
               for( i = 0; i < totalnodes; i++)
               minnodedist[source][i] = nodedistance[source][i];
               minnodedist[source][source]=0;
               p_set[source] = 1;
               t_set[source] = 0;
               /*FIND MINIMUM DISTANCE UPDATE    */
               while ( Set_Summ(p_set) - totalnodes < 0 )
                    {
                    f = Find_Min(source,t_set);
                    p_set[f]=1;
                    t_set[f]=0;
                    for ( n = 0; n < totalnodes; n++ )
                         {
                         if ( minnodedist[source][n] > ( minnodedist[source][f]+
                         nodedistance[f][n] )     )
                                   {
                                   minnodedist[source][n] = minnodedist[source][f]+
                                   nodedistance[f][n];
                                   }
                         }
                    }
               }
          fprintf(stderr,"\n\nDikstra completed\n\n");
          /* Send Results to Text Output File */
          /* 1: Straightline Matrix: Recall Triangular Inequality*/
          /*    Does not hold                                    */
          for ( n = 0; n < totalnodes; n++ )
               {
               fprintf(fprn,"%6d",n);
               }
          for( i = 0; i < totalnodes; i++)
               {
                fprintf(fprn,"\n%d ",i);
                for ( n = 0; n < totalnodes; n++ )
                     {
                    fprintf(fprn,"%5.1f ",nodedistance[i][n]);
                    }
               }
          /* 2: Minimum Distance MAtrix to be used by TIMM */
          fprintf(fprn,"\n");
          for ( n = 0; n < totalnodes; n++ )
               {
               fprintf(fprn,"%6d",n);
               }
          for( i = 0; i < totalnodes; i++)
               {
                fprintf(fprn,"\n%d ",i);
                for ( n = 0; n < totalnodes; n++ )
                     {
                    fprintf(fprn,"%5.1f ",minnodedist[i][n]);
                    }
               }


          return(0);
          }

int Set_Summ(set) /*Generic Set Function*/
     static int set[TOTLNODES]
          {
          int i;
          int sum=0;
          for( i = 0; i < totalnodes; i++ )
               {
               sum = sum + set[i];
               }
          return(sum);
          }

int Find_Min(source,set[TOTLNODES])
     int source;
     static int set[TOTLNODES];/*Simple Sort*/
          {
          int i;
          float shortest_so_far = (sqrt(2)*maxnodedistance)+20.0;
          int ret_value;
          for ( i = 0; i < totalnodes; i++)
               {
               if ( (minnodedist[source][i] <= shortest_so_far)&&
                    (set[i] != 0)                                 )
                 {
```

```
                shortest_so_far = minnodedist[source][i];
                ret_value = i;
                }
            }
        return(ret_value);
        }
```

7.5 BIMM.C: Simple Batch Creation File

```
#include <stdio.h>
char *sub;
char *outafile = "imma.bat";   /*First Batch File for Data Creation*/
char *outbfile = "immb.bat";   /*Second Batch File for Simulation*/

/***************************************************************************/
/*            NOTE: This File Must be Run Twice                            */
/*  < bimm a > for first data creation batch  <bimm b > for analysis bch.*/
/***********  Caution Exponential Number of Choice Combinations  *********/

/***************************************************************************/
/*********** Distance File Parameters ************************************/
/***************************************************************************/
                                    /*p- 2D graph w/ TSP          */
char dm2[] = {'c','n','p'};                /*c- circular equilateral graph*/
int dd2 = 3;                               /*n- non-planar w/o TSP       */
int dm3[]  = {6,12,18,24,30,36,42,48,54}; /*number nodes in graph MAX 60 */
int dd3 = 9;                               /*******************************/
int distance =1000;

/***************************************************************************/
/************ Arrival and Service Parameters ******************************/
/***************************************************************************/
                                    /*******************************/
char sm5[]  = {'r','d'};                   /*r- random parameters         */
int sd5 =  2;                              /*d- deterministic parameters  */
char sm6[]  = {'g','m'};                   /*g- gaussian service parameter*/
int sd6 =  2;                              /*m- exponential service para. */
int arrival = 10;                          /*******************************/
int sm8[]  = { 92, 96,100,101,102,103,     /*E(Service Rate) as percent of*/
            104,105,016,107,108,109,       /*     arrivalrate*#of nodes   */
            110,112,114,116,118,120,       /*******************************/
            124,128,132,140,148,156,
            166,176};
int sd8  = 26;                             /**Standard Deviation as *******/
int sm9[] = { 10,12,14,16,18,20 };         /* Multiple of service rate    */
int sd9  = 6;                              /*******************************/
unsigned long int totalbytes = 100000;

/***************************************************************************/
/******** Selection of Strategies and Speeds ******************************/
/***************************************************************************/
int tm7[] = {1, 2, 3, 4, 5, 6, 7, 8, 9,10, /*******************************/
```

```
        11,12,13,14,15,16,17,18,19,20, /*  Selection of Strategies    */
        21,22,23,24,25,26,27,28,29,30, /********************************/
        31,32,33,34,35,36,37,38,39,40 };
 int td7 = 40;                               /********************************/
                              /* Different Speeds Selected  */
                              /******************************/
 unsigned long int tm8[] =   { 1000,1100,1200,1300,1400,1500,
                         10000,20000,30000,40000,50000,
                         100000,200000,300000,400000,500000,
                         1000000,2000000,3000000,4000000,5000000,
                         100000000};
 int td8 = 21;                               /********************************/
 int tm9[] = {1,2,4,6,8,10,15,20,25,50,75};/* beta as % of totalnodes    */
 int td9 = 11;                               /********************************/
 /*****************************************************************************/
 /*****************************************************************************/
 /*****************************************************************************/

 main(argc,argv)
  int argc;
  char * argv[];
  {
   int run;
   int n2 = 0;
   if(1 != sscanf(argv[1],"%ld",&run)) {fprintf(stderr,"\n\nErr:Bad Arg.\n\n"); exit(-1);}
   if (run == 2) batch2();
   if (run == 1) batch1();
   else {fprintf(stderr," 1 --> Create DATA batch  2---> Create ANALYSIS batch");}
   return(0);
  }

 batch1()
 {
 FILE *fprn;
 int n2,n3;
 int m5,m6,m7,m8,m9;
 int o7,o8,o9;
 if( (fprn = fopen(outafile,"w")) == NULL) fprintf(stderr,"Error 1\n");
  for ( n2 = 0 ; n2 < dd2; n2++ )
   {
   for ( n3 = 0 ; n3 < dd3; n3++ )
    {
    fprintf(fprn,"dimtsp\t%d\tdp%c.%d\tdo%c.%d\t%c\t%d\n",
       dm3[n3],
       dm2[n2],dm3[n3] ,
       dm2[n2],dm3[n3],
       dm2[n2],
       distance);
    for ( m5 = 0 ; m5 < sd5; m5++ )
     {
     for ( m6 = 0 ; m6 < sd6; m6++ )
      {
      for ( m8 = 0 ; m8 < sd8; m8++ )
      {
      for ( m9 = 0 ; m9 < sd9; m9++ )
       {
fprintf(fprn,"simm\t%d\ttemp\ta%c%c%d%d.%d\ts%c%c%d%d.%d\t%c\t%c\t%d\t%d\t%d\t%lu\n",
       dm3[n3],
       sm5[m5],sm6[m6],sm8[m8],sm9[m9],dm3[n3],
       sm5[m5],sm6[m6],sm8[m8],sm9[m9],dm3[n3],
       sm5[m5],
       sm6[m6],
       arrival,
       (sm8[m8]/100)*arrival*dm3[n3],
       sm9[m9]*(sm8[m8]/100)*arrival*dm3[n3],
       totalbytes);
       }
      }
     }
    }
   }
  }
if (fclose(fprn) == EOF) fprintf(stderr,"Error 3.04\n");

}
batch2()
{
FILE *fprd;
int n2 = 0;
int n3;
int m5,m6,m7,m8,m9;
int o7,o8,o9;

if( (fprd = fopen(outbfile,"w")) == NULL) fprintf(stderr,"Error 1\n");
 {
  for ( n3 = 0 ; n3 < dd3; n3++,n2++ )
   {
   if (n2 > dd2) break;
   for ( m5 = 0 ; m5 < sd5; m5++ )
    {
    for ( m6 = 0 ; m6 < sd6; m6++ )
     {
     for ( m8 = 0 ; m8 < sd8; m8++ )
      {
      for ( m9 = 0 ; m9 < sd9; m9++ )
       {
       for ( o7 = 0 ; o7 < td7; o7++ )
     {
     for ( o8 = 0 ; o8 < td8; o8++ )
      {
      for ( o9 = 0 ; o9 < td9; o9++ )
       {
       fprintf(fprd,"timm %d a%s%s%d%d.%d s%s%s%d%d.%d dp%c.%d t %d %lu m %d do%c.%d\n",
          dm3[n3],
          sm5[m5],sm6[m6],sm8[m8],sm9[m9],dm3[n3],
          sm5[m5],sm6[m6],sm8[m8],sm9[m9],dm3[n3],
            dm2[n2],dm3[n3],
            tm7[o7],
            tm8[o8],
            tm9[o9],
            dm2[n2],dm3[n3]);
         }
        }
       }
         }
        }
       }
      }
     }
if (fclose(fprd) == EOF) fprintf(stderr,"Error 3.04\n");
}
```

8 Bibliography

Bertsekas, D., Gallager, R. (1987), Data Networks, Prentice HAll, N.J. Chapter 3.5.

Bertsimas, D., van Ryzin, G. "A Stochasic and Dynamic Vehicle Routing Problem in the Euclidean PLane",(M.I.T. technical report) February 7, 1990.

Heyman, D.P., Sobel, M.J. (1982), Stochastic Models in Operations Research, Volume 1, McGraw Hill, U.S.A. Chapter 11.5,11.6.

Hogg, R.V., Ledolter, J. (1987),Engineering Statistics, MacMillan, New York. Chapter 6.3.

Kernighan,B.W.,Ritchie,D.M. (1988), The C Programming Language (2nd Edition), Prentice Hall, New Jersey.

Larsen, R.J., Marx,M.L. (1986), An Introduction to Mathematical Statistics and its Applications, Prentice Hall, New Jersey.

Press, W.H.,Flannery, B.P. Teukolski, S.A., Vetterling, W.T. (1989), Numerical Recipes in C: The Art of Scientific Computing, Cambridge University Press, New York. Chapters 7,10.

Walrand, J.,(1988), An Introduction to Queuing Networks,Prentice Hall, New Jersey. Chapter 8.

Wolff, R.W, (1989), Stochastic Modelling and the Theory of Queues, Prentice Hall, New Jersey. Chapters 8,10.