Guided Vehicle Systems - A Simulation Analysis

by

Subir Dutt

Project submitted to the Faculty of

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Engineering

in

Industrial and Systems Engineering

Approved :

Dr. M. P. Deisenroth, Chairman

Dr. R. J. Reasor

Dr. O. K. Eyada

September, 1991

Guided Vehicle Systems: A Simulation Analysis

by

Subir Dutt

Committee Chairman: Dr. M. P. Deisenroth
Industrial and Systems Engineering

(ABSTRACT)

This project provides a tool for evaluating various guided vehicle systems. The tool is a discrete event simulation package which runs on entered parameters and simulates a variety of guided vehicle systems with different configurations. This provides a means of evaluating alternative system configurations and observing system performance for a particular set of restrictions. GVSim aids in testing and developing the feasibility of various concepts in the design of guided vehicle systems. Also, the user has the ability to drop down to the subroutine level and modify code in order to model the system using a different control logic. Hence the package provides some of the benefits of a simulator along with the ability to change the inherent assumptions within the package.

Acknowledgements

My thanks to Dr. Deisenroth for his help and advice, particularly towards the end of my research. Thanks also to Dr. Eyada and Dr. Reaser for serving on my committee.

For their friendship and support, I thank Shobha, Lal, Jibesh, Lata, Bart, Girish, Prateek, and Simon. My special thanks to Joni and LaVonda, two of the nicest people I have met. Your constant help and friendliness is much appreciated.

My deepest thanks go to my family to whom I would like to dedicate this work. To Jai and Pallabi, for constantly pushing me along. I really appreciated your concern even if there were times I didn't succeed in showing it. Finally, my thanks and appreciation to my mother, without whom all this would not have been possible.

Finally, I would also like to dedicate this project to 'Gorgeous', inspite of whose love and affection I managed to complete my work.

Table of Contents

List of Figures

Chapter I

1. INTRODUCTION

1.1 Statement of the Objective

The purpose of this research is to provide a development and analysis tool for modeling various guided vehicle systems. A generic, modular, flexible, 'C' based, discrete event simulation package was developed as a means for modeling various guided vehicle systems. The package serves as an aid for research by giving the users the ability to create their own particular model for system performance and evaluation. Alternative system configurations and system control methodologies are easily modeled by changes to the basic building blocks of the package.

1.2 Guided Vehicles

Over the last few decades a variety of advanced technologies have emerged to expand the capabilities of computer controls in the creation of automated factories. A guided vehicles system is one of the products brought about by the integration of computer control with material handling. Combining material transportation, storage, and production processes with central computer control operations can

provide a factory with smooth and efficient material flow. This provides a means to tightly govern material control policies and can be a cost effective alternative to labor intensive or floor space consuming methods of material handling [28]. The control system of a guided vehicle is responsible for both direct vehicle control as well as system control tasks, such as automated routing, task allocation, and conflict prevention.

Guided vehicles are driverless, guided by a variety of path regulation methods, and can effectively be interfaced with material handling subsystems or workcells. The automated guided vehicle product section of the Material Handling Institute defines an automatic guided vehicle (AGV) as "A vehicle equipped with automated guidance equipment, either electromagnetic or optical. Such a vehicle is capable of following prescribed guidepaths and may be equipped for vehicle programming and stop selection, blocking, and any other special functions required by the system [23]."

A self guided vehicle (SGV) is more advanced than an AGV. It provides the additional capability of having a guidance system, which does not need to follow a wire embedded path or paint stripes, but can provide its own path control. Benefits of guided vehicle systems usage include inventory

control, increased equipment and space utilization, manufacturing flexibility, higher productivity, and easier management control of operations [23].

1.3 Research Interests in AGVs and SGVs

Over the past few decades, guided vehicles have undergone substantial development and implementation. Various universities (Purdue University, Lehigh University, The University of Virginia, Cranfield Institute of Technology, and Imperial College of Science and Technology [23, 37]) are involved in research projects related to AGVs and SGVs. A number of vendors are also involved in creating/providing computer controlled guided vehicle systems. This includes such companies as Caterpillar, Cybermotion, SattControl AB (Sweden), Wagner Indumat Systems Ltd (U.K.), Seri Renault Automation (France) and Shinko Electric Co Ltd (Japan) [3, 23]. Major industrial users of AGV and SGV systems include the aircraft industry, the automotive industry and the electronics industry [23].

A major effort is underway at Virginia Tech to develop a laboratory facility for research and instruction in various areas of computer integrated manufacturing. A primary focus of the laboratory will be directed at the design, analysis,

and control of integrated manufacturing systems. The laboratory will utilize the feature of computer control over all aspects of manufacturing. It aims to demonstrate an integration of the different aspects of manufacturing and production. Starting with inventory maintenance, AS/RS, material flow, material processing and assembly, it will coordinate the different functions into one operation, under a central control. The task of the guided vehicles is simply one of moving material from storage to machining and back, but a simulator is needed to be able to model systems much more complex and vast.

Guided vehicle research issues are more often considered together than individually, since researchers and vendors tend to attack them as one problem. The main issues being addressed in guided vehicle research are guidance and path control, computer control and system integration, coordinating material handling with other manufacturing functions, simulation platforms to test AGV/SGV systems, safety, mechanical design of the vehicle, and docking station design.

Guidance and path control are explored from the points of view of both coarse and fine navigation. A number of systems use odometry for coarse navigation and pick one of

the many methods available for fine navigation. The computer control and system integration serves to provide for an accurate control of the system parameters and integrate it with other material handling functions. Coordinating material handling with other manufacturing functions overlaps with system integration. A number of vendors and researchers have and are developing simulation packages to test and demonstrate the applicability of their systems. Safety also plays an important part and various features are being included to prevent accidents among AGV/SGV users. The mechanical design of the vehicle plays an important part based on the application. Requirements of space, weight, and payload place different demands on vehicles. Also the role guided vehicles play in docking leads to different designs of docking stations.

1.4 Simulation in Guided Vehicle Systems Design

In the design and development of an AGV/SGV system, it is necessary to estimate system performance and behavior so as to evaluate different configurations or designs. Simulation is a technique which can be used in the modeling and evaluation of a proposed system.

Simulation is one of the most versatile and widely used analytical techniques in the design of material handling systems. It enables the user to build a model of a system on the computer that can be used to evaluate the effects of various changes in design and control configuration of the proposed system. With the execution of the simulation program, discrepancies and flaws can be observed and removed. The model can be conveniently manipulated until the desired results are achieved.

The main advantage of simulation is that the system being modeled can be specified with a great degree of accuracy. The simulation program produces an extremely accurate imitation of all hardware and software functions of the proposed system [29]. Also the particular or planned system may be tested or designed without existing at all.

Simulation of AGVs/SGVs provides an excellent tool for designing AGV/SGV systems, determining the path desired, fixing the number of vehicles, etc. Recently, with the increase in the number of AGV/SGV systems being installed, the simulation of guided vehicles has become a more visible issue. Current applications involve simulation packages for determining the optimum number of AGVs/SGVs, analyzing their

performance, and using graphics packages to demonstrate vehicle functionality [23].

One problem associated with the application of simulation to AGV/SGV system design is the validity of the performance of the model. Typically the actual AGV/SGV systems have not been built and verification of model results are difficult to make. One cause of this problem is associated with the simplifications of the dispatch and scheduling algorithms that are used in the actual systems [24]. System modelers must often rely on the fact that the logic of the model closely approximates the expected behavior of the system.

Computer simulation is becoming an effective aid for justification processes in the design of AGV/SGV systems. The computer model of AGVs/SGVs helps in the solution of a number of problems. Simulation provides the user with a tool to evaluate the consequences of a particular set of decisions. It helps to try alternatives and observe the results of picking a particular alternative. Also, it helps in picking another alternative and comparing the contrast and change between the two alternatives. It establishes the relationship between the different factors involved in the model. It provides the user with the ability to put in

fluctuation in load arrivals and estimate throughput quantities.

Finally, it enables the user to experiment with and question the various design decisions. "It is important to understand that simulation does not design the system. But when a design exists, simulation can test it and see how efficient the design is against the requirements. By using the simulation in an interactive and iterative way, the designer soon homes in on the most efficient solution to achieve the required materials flow [25]."

1.5 Simulation and GV Systems Research

In testing the design and formulation of an AGV/SGV system, the use of simulation is limited by the need to use/create a package capable of simulating the system. Invariably, the users needs to first obtain a base on which to build their model. Generic packages like SLAM II, even with its material handling extension, restrict the user to certain assumptions inherent in the package. The need is felt for a base on which to formulate a wide variety of models. Hence the need is felt for a vehicle for research.

Law and Haider [20] define the two major types of manufacturing analyses for which simulation is used as *high-level analysis* and *detailed analysis*. A high-level analysis is performed in the initial phases of design with the details of the operating or control logic not being included. "Typical objectives are determining the required numbers of machines and material handling equipment, evaluating the effect of a change in product volume or mix, and determining storage requirements for work-in-progress [20]". Manufacturing simulators are often used for high-level analyses, but a language could be used as well. A detailed analysis is performed to fine-tune or "optimize" the performance of a system, and the corresponding simulation models typically represent operating or control logic in considerable detail. Simulation languages are typically used for detailed analysis because of their ability to model complex decision logic. Simulators might be used in some cases, particularly if they have the ability to drop down to a lower-level language [20].

Chapter II

2. LITERATURE REVIEW

2.1 Introduction

The present literature shows a substantial level of research in the various aspects of AGV/SGV development and systems design. The first section focuses on guided vehicle development from a historic perspective and then moves to a discussion of the variety of tasks and features they incorporate. It concludes with an identification of ongoing research in these areas. The second section addresses the application of simulation with the AGV/SGV systems environment. It identifies the various aspects and features present in simulation of guided vehicle systems. The final section gives a description of some of the packages and simulators available for AGV/SGV system simulation.

2.2 Guided Vehicles

Automated material handling has been called the backbone of the automated factory [16]. Material handling is considered by many automation specialists as an important element in the entire scenario of automated manufacturing [16]. As a result, attention is being turned to automated material

handling systems and their efficient use. In recent years, this field has seen considerable activity in terms of research and development and a consequence of this has been an explosion of new technology. One of these new technologies is the self guided vehicle system or SGV system.

The first automated guided vehicles (AGVs as they were initially called) were developed in the U.S. by Barrett Electronics in the early 1950s. The first system was installed in 1954 at Mercury Motor Freight in Columbia, South Carolina. It was a tugger system, which followed a wire guidance path and had a controller based on vacuum tube technology [12]. The initial AGVs were used mostly to handle materials in the auto industry.

Even though the AGV was developed in the U.S., it did not gain widespread usage in the industry in the 50s and 60s. Its usage was hampered because vendors and users did not work together to improve the technology. Only on seeing the success of AGV systems in Europe and especially in the Volvo plant, did U.S. manufacturers start developing better AGVs. Foreign competition has also been a major factor for implementing AGVs in industry in the U.S., to improve the material handling task. Foreign competition focuses the

need on using advanced technology to improve operations and remain competitive in a developing market. The largest employers of guided vehicles are once again the automakers, but the vehicles can be found in electronics manufacturing, the aerospace industry, the postal service, hospitals, transportation systems, and newspaper publishers.

Initially, automated guided vehicles were considered to be simple replacements for other types of material transport systems, such as conveyors or forklift trucks. Guided vehicles combined well established electromagnetic technology and existing industrial truck equipment to create a more flexible self-steering vehicle. Then, with the inclusion of microprocessor technology, guided vehicles were transformed into a class of highly sophisticated and flexible material handling vehicles. They are now complex transportation systems requiring a substantial amount of time and effort in their design process. Their design is represented by aspects reaching from the installation of vehicles employed, the floor equipments and corresponding control included up to the central systems controller. In a number of countries in Europe, AGV/SGV systems are being planned and implemented to take care of material handling problems.

Guided vehicle technology is changing at a fast pace. This is evident from the words of Gene F. Schwind, executive editor, Material Handling Engineering. "If you plan to write a treatise on control systems for automatic guided vehicle systems (AGV systems), don't print more than a few copies. Technology is changing continuously, and new combinations are being tried all the time." Recently the trend in research and development has been more towards SGVs.

The difference between AGVs and SGVs is, basically, that AGVs use traditional methods of guidance, namely wire or optical guidance. Hence, the routes of AGVs are fixed and inflexible. SGVs do not have fixed routes but use radio frequency, sonar, and vision for guidance and have on board computers allowing for a high degree of flexibility in plant layout. This is provided by the inherent ability of SGVs to follow any path specified by the user. Hence, the plant layout could be changed and the SGV would travel along the new paths specified by the control program. There would be no need to install a wire or a reflective paint stripe for the SGV to follow. The flexibility of SGV systems, over AGV systems, was one of the main reasons for its installation in the Ford Motor Company in Indianapolis, Indiana.

Guided vehicles have different basic features with respect to guidance, vehicle steering, control, routing, vehicle dispatch, system monitoring, safety, vehicle design and load transfers. They are used for a variety of tasks, in many different industries, involving material handling and transport, as described in the following pages. Miller [23] identifies five basic AGV/SGV system features in automated manufacturing:

1) Guidance
2) Routing
3) Traffic Management
4) Load Transfer
5) System Management

Present research and application areas in guided vehicle systems include guidance, computer control and systems integration, mechanical design, safety, type of docking stations and simulation. A number of techniques are being employed in the guidance and control systems. These include:

1) Inductive guidepath (using a floor-embedded wire carrying alternating current for guidance),
2) Optical/chemical/magnetic guidepath (using a passive stripe on the floor that is detectable by sensors,
3) Odometry (using incremental sensors fixed at the

steering and wheels to measure rotation and travel of wheels),

4) Ultrasound distance measurement (accuracy in mm., can be disturbed by high frequency acoustic sources),
5) Laser triangulation (position of the vehicle is ascertained by measuring the angle from two points - the distance between which is known),
6) Inertial navigation system (measuring translational and rotational acceleration of the vehicle and calculating its position and heading relative to a fixed reference point by double integration of the respective accelerations),
7) Direct imaging systems (here optical or ultrasound images of the surrounding working area are used on-board the vehicle to deduce its position within that working area), and
8) Gyro-compass (the position deviation of a gyro-compass can be used for measuring the rotary position alteration of the vehicle).

Computer control and systems integration provide for the software control which runs the system and links it with the other material handling functions. It includes higher level coordination of many vehicles executing a continuous sequence of transportation demands and determination of

controlling principles. The shortest path algorithms, conflict prevention and timetable generation also form a part of the control software.

The design of the vehicle itself is another issue which plays an important role depending on the type of application. Factors like vehicle wheel configuration, vehicle hardware (weight capacity) and steering mechanism play an important role in the eventual guided vehicle.

Some of the features considered for safety precautions in guided vehicles are visual warnings (flashing lights), audible warnings (beeping), safety in operating environment (limited access) protection on the vehicle (bumper, ultrasonic or photo-electric sensors), and training. Tracey [35] provides a review of AGV safety features.

The design of the docking station comes into play varying with the role of the vehicle (active or passive) in docking. Considerations such as the accuracy of the vehicle in aligning with the docking station, method of load pick-up/delivery, and design of the pallet transfer mechanisms play important parts in docking.

In the computer applications and simulation of AGV/SGV systems, the following features play major roles [17]:

1) Planning of layouts and AGV/SGV routes,
2) Planning of the technical configuration of the vehicles,
3) Simulation of the material handling system, and
4) Evaluation and result presentation of material flow ascertainments (requirements).

2.3 Simulation and Guided Vehicle Systems

Presently, there is visible use of simulation for modeling systems before implementation. Current research involves simulation packages for determining the optimum number of AGV/SGVs, analyzing their performance, using graphics packages to demonstrate vehicle functionality, etc. They focus on features such as ease of entering and changing layout, ease of entering and changing various system parameters, animation, comprehensive statistics to show results, conflict prevention techniques, shortest route algorithms, and computer integration. A primary factor of concern is the difference created between the performance of the model and the real system. The differences are mainly caused by trying to simplify the model and by the accuracy of the dispatch and scheduling algorithms.

Akari, Takahasi, Suekane and Kawai [1] describe a simulator which allows flexible AGV path layout by arranging tiles representing 68 kinds of path patterns including station, entrance, exit, direct line, curve, T-cross, and cross on the personal computer display unit. The simulation of the system is performed according to conflict prevention rules and user defined conditions of number of machine tools/machines, the machining process, the required time, workpiece supply sequence, the number and speed of the AGV, and the limitation of buffer capacity at the station. The simulator consists of models called the path editor, numerical data input editor, shortest path calculator, simulation executor, animation and result output section.

Schulze and Rosenbach [30] describe a simulation system, MATSIM, with module libraries. The modules in MATSIM consist of 1) interactive input, i.e. modification of model parameters, 2) simulation execution, and 3) data concentration and result presentation. Each module is composed of four levels, namely: 1) the operative level (movements, loading and positioning), 2) the control level close to operations (control of blocks, priority selection at crossings & junctions and decisions of driving directions), 3) the dispositive level (task allocation and collection, location of vehicle w.r.t. a job request being

generated) and, 4) the administrative level (introduction of vehicles into the system, sequence of processing in the production and strategies of production control). MATSIM also employs an expert system with a shell and a knowledge base for the planning and configuration of AGV systems.

Schmidt [31] develops an algorithm to enable the user to execute a rational evaluation of the number of AGVs required in a system. He describes the factors involved in determining the number of vehicles and develops a simulation program for its calculation. The simulation program is based on input data of a matrix of distances, a transportation matrix, data about vehicle movements and pick-up/delivery times.

Norman, Norman and Farnsworth of AutoSimulations [24] describe some of the causes of inaccuracies in simulation models of AGV systems as faulty empty carrier management algorithms (20% - 30% error), faulty partitioning of guide paths (5% - 15% error), faulty blocking logic (5% - 10% error) and faulty vehicle speed (5% - 10% error).

Platts [25] presents a simulation package with a graphics facility for layout description. It shows the advantages of a graphics display in understanding the model. The package

uses decision points and ease of change of parameters to provide for flexible models. The package is quite similar to the one described in [1] on the control logic.

Jansson [14] and Schmidt [31] describe the use of a transport matrix for storing transport frequency, layout destinations, variations during operation, distances, AGV specifications, empty transportation and average speed.

Buda, Badida and Vrlik [8] describe the type of paths AGVs use as fundamental (consisting of straight lines or arcs) and derived (consisting of combinations of fundamental paths). The authors present a simulator called ANPRO 3D, which includes animation display of the model, for modeling the system. The user selects suitable objects from the database, which are then transferred to the animation display. The object can be moved around using a graphics cursor and it is possible to observe and to keep track of individual AGVs. They present a list of some of the problems solved, by the use of simulation, in the design of AGV systems such as:

- function and performance verification
- system behavior analysis under various operation requirements
- effects of failure in individual components

- transport device dimensions
- utilization of the transport system components
- transport path design
- illustration of interfaced and combined functions.

In Autosimulations Inc. [29], a CAD system is the platform used to enter the layout of the model. A high level simulation language called AutoMod is employed where the system being modeled is described in simple English. The layout is entered using CAD and the outputs of the simulation are used to run a graphics display. An emulator is used to interface the output, representing discrete control interface events, with the physical controller (requiring discrete signals). A simple formula is presented to calculate the approximate number of vehicles:

$$N = T/(t*l*v)$$

where N = number of vehicles

T = all load transit times

t = traffic factor

l = load factor

v = total vehicle time available

In [9] various simulation situations are presented and discussed by Duffau and Bardin. The authors evaluate

various AGV system circuits using analytical, statistical and simulation techniques. Also, a list of functions of a specialized simulator are presented. The authors stress the importance of cruising loops, in certain situations, to prevent bottlenecks.

2.4 Simulation Aides/Languages

Over the last decade a number of vendors have developed simulation aides/languages for various applications including modeling guided vehicle systems. Most of these packages follow similar lines of thought but differ in control logic, input parameters, algorithms, result analysis, and animation. A limited number of these packages are described here.

"There are two major categories of software for simulating manufacturing or warehousing systems [20]." A *general-purpose simulation language* can be used to model most manufacturing systems and generally consists of features for specific manufacturing characteristics. They normally require substantial programming time. "A *manufacturing simulator* is a computer package that allows one to simulate a system contained in a specific class of manufacturing systems with little or no programming [20]."

A large variety of simulation languages starting with general purpose languages like SIMSCRIPT II.5 (C.A.C.I.), GPSS (IBM's General Purpose System simulation), SLAM II (Pritsker & Associates), SIMAN (Systems Modeling Corporation), and PETRI networks (based on queue systems) to more specialized, manufacturing-oriented languages such as MAP/I (Pritsker & Associates), SPEED (Horizon Software), and MAST (CMS Research) are available.

SLAM II, for instance, has a material handling extension for AGVs, which provides modeling capabilities at a systems level. "The modeler controls vehicle movement patterns by specifying the logic rules to be used for path selection, precedence at intersections, idle vehicle control, and vehicle and load selection [27]." The user can also write FORTRAN subroutines, for emulating logic, which can be accessed by SLAM II. Therefore, if a particular situation or logic cannot be modeled by SLAM II, the user can use FORTRAN subroutines to model it. The material handling extension makes certain assumptions such as [27] i) similar vehicles, ii) one vehicle per job request, iii) vehicles have certain fixed states, iv) vehicles have physical location which may cause interference with other vehicles, v) acceleration and deceleration rates of vehicles do not vary with system or vehicle status, vi) breakdowns and

battery charging are modeled as type of job requests, vii) communication with controller limited to control points on guidepath, and viii) vehicles attempt to take shortest path.

TESS, The Extended Simulation System provides a framework for performing simulation projects. It integrates simulation, data management, and graphics capabilities. "Capabilities for graphically building SLAM II networks, animating simulation runs without programming, generating graphs of simulation runs without programming, and generating graphs of simulation results are provided [23]." It generates reports on the performance of the system and provides for analysis of simulation results. Additionally, it allows the user to input data and simulation-run controls.

MAP/1 is a user friendly manufacturing simulator with ease of change in the design parameters for operation of manufacturing systems. It allows for extensive modeling and analysis of the various components of batch systems and provides for an efficient means of understanding system behavior and improving system productivity [23].

MicroNET, a discrete event language, which is network-based, operates as a simulation system for microcomputers [23].

AutoSimulations, Inc. (Bountiful, UT) offers AutoMod and AutoGram software packages for simulation. In AutoMod the path is defined by control points and segments (path between control points). The vehicle receives all its instructions and communications at control paths. AutoMod is a comprehensive simulation software environment for the imitation of industrial systems such as factories, warehouses, and distribution centers. AutoMod consists of an English-like language for describing systems to be simulated and of proven techniques and algorithms implemented with General Purpose System Simulation (GPSS/H, Wolverine Software Corp.) as the simulator [23]. "Outputs from AutoMod can be used to drive an animated color, three dimensional graphic representation of the modeled system. The interface to the graphic system also includes a Computer Aided Design capability by which the physical system can be physically defined to the AutoMod Simulator [29]."

MODELMASTER aids in the design and analysis of automated factories. With the use of graphics, the layout is described and the various operational parameters are entered. The system is then simulated [23].

The *HEI* Simulator (HEI Corporation, Carol Stream, IL) is used for planning, designing, developing, installing, and

monitoring automated manufacturing, warehousing, and distribution systems. Also the system can be tested, at each step, without interrupting production. It works in three phases as a simulation package, emulator of a system and as diagnostic monitor [23].

In the *SEE WHY/WITNESS* system, by ISTEL, a simulation model is made by defining constituent elements of the AGV network to be modeled. This includes defining the various data about the system like number of machines, number of vehicles, and number of stations. The system is processed using the design on a screen layout. Then the control strategies are selected (like selection of vehicle, path selection, scheduling of events and junction priorities) [4]. Some of the benefits of simulation of AGV systems gained by Istel Ltd., U.K. in their various projects, are discussed in [4].

The major drawback of many simulators is that they are limited to modeling only those manufacturing configurations allowed by their standard features. This difficulty can be largely overcome if the simulator has the ability to "drop down" into a lower level language (e.g. FORTRAN or 'C') to program complicated decision logic [15]. Possible drawbacks of simulation languages are the need for programming

expertise and the possibly long coding and debugging time associated with modeling complex manufacturing systems. This project aims at reducing these drawbacks by providing for some of the benefits of both simulators and simulation languages. The user would be provided with the means to build a model using the standard features of the package and also manipulate the code for modeling features not included in the package. Essentially, the package is a modular simulator. 'C' provides an excellent programming background with its portability, flexibility, speed of execution, and ability to execute complex logic.

CHAPTER III

3. DESCRIPTION OF SIMULATION MODEL

3.1 Introduction

The simulation package developed in this research provides a development and analysis tool for modeling various guided vehicle systems. A generic, modular, discrete event simulation package is presented. Essentially, the objectives are twofold. The primary objective is for the user to be able to simulate a variety of guided vehicle systems by modifying input parameters. The system definitions (such as layout, number of vehicles, generation of vehicle requests, and location of workstations) are input by the user. Performance measures, such as vehicle utilization and machine utilization, are provided as standard output to facilitate system evaluation. The secondary objective is to provide the user the capability of changing the various control logic concepts, with only the restriction of modifying/creating the necessary subroutines. Thus, the user is able to test various control concepts and algorithms in the design of guided vehicle systems. This allows future users to transform the package into a truly generic and multi-functional simulation and modeling system.

This chapter presents a general description of the methodology and theoretical foundation on which the computer model is based. Also, it gives a broad overview of the structure of the model that has been developed during the course of this research. A guided vehicle system is considered and used as an example to describe the working of the simulator. Initially the input of the data to describe the layout is discussed. The concept of points and segments and how they compose the layout is introduced. Subsequently, a description is given as to how the guided vehicle system in Figure 3.1 is modeled using GVSim. The various types of points and their specific features are discussed. The next section deals with the working of the Path Planner, which plans and allocates routes within the layout. A description of the various modules used by the Path Planner is detailed. The following section then describes the vehicle characteristics and method of vehicle allocation. Finally, a listing of a complete data input and the resultant output are presented and discussed.

3.2 Layout Description

3.2.1 A Guided Vehicle System Layout Example

Consider the guided vehicle layout shown in Figure 3.1. It represents a FMS cell with a receiving cell, five machining

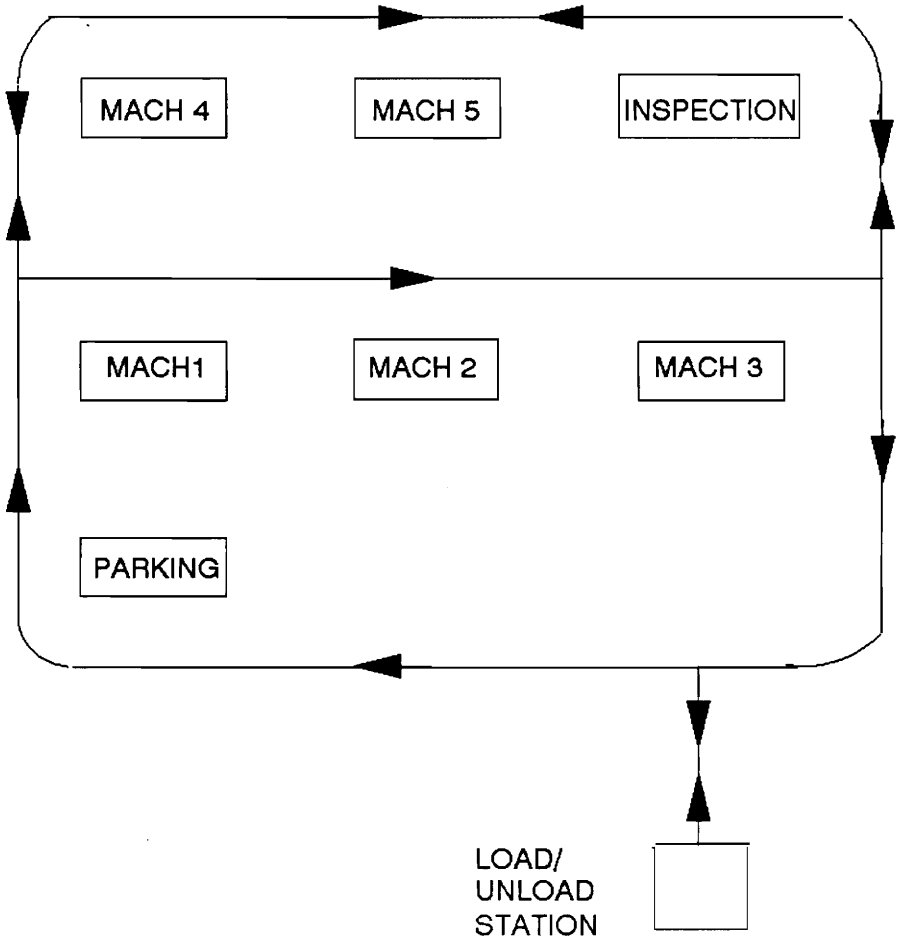


Figure 3.1: A Guided Vehicle System Example

workcells, an inspection station and a parking area. The various stations are placed as shown in the figure. This system will be used as an example to describe the functioning of the GVSim simulation package.

Loads enter the system at the receiving (loading/unloading) cell. Each load, on entering the system, needs to be machined at any one of the five machining cells. After machining, the part is taken to the inspection station. After inspection is complete, the parts are sent to the loading/unloading station to exit the system.

3.2.2 Layout Input

The layout is described as a list of interconnected points, entered by the user. These points describe the layout in a grid like fashion. The location of the points in the layout is specified by entering their x and y coordinates. As data for a specific point is entered, the points to which it is connected are identified as connecting points. The guided vehicles will then be permitted to travel from the specified point to one of the connecting points. Path segments occur naturally when specifying the connection between points, and are not separately defined. A flowchart for describing the layout is shown in Figure 3.2.

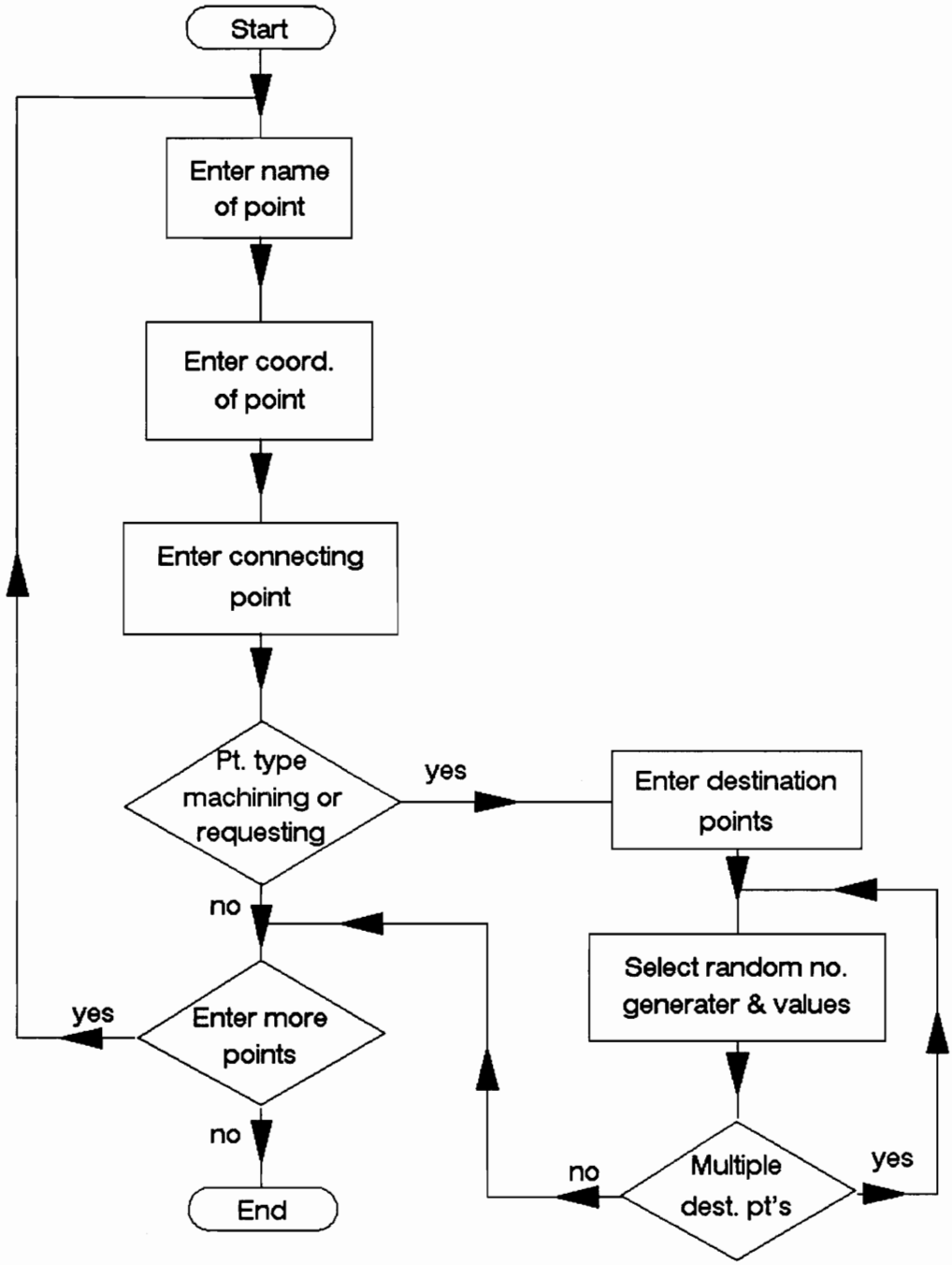


Figure 3.2: Conceptual Flow Chart for Inputting Layout

Points may be of different types and the type is specified when entering the data on each point. A 'normal' point assists in the flow of material within the system. Specifically, normal points are put in to specify corners of a given path or to break up long segments. Loads enter the system at a 'requesting' point. A 'machining' point indicates the location of a machining workcell. 'Parking' points are used to identify the initial location of the vehicles and location of idle vehicles. A detailed description of the type of points is given in Section 3.2.5.

Segments indicate the connection between points. If a point B1 is specified as connected to point A1, a segment exists between them. Flow of material is possible from A1 to B1. In this case segment A1-B1 is a unidirectional segment, with flow of material only possible from A1 to B1. Additionally, if point A1 is specified as connected to point B1, then flow of material is also possible from B1 to A1. Then segment A1-B1 is a bidirectional segment.

Points are entered in the format of letter-digit(s). The digit or digits may vary between 1 through 99. Typical point names are B3, C24, F79 and so on. The list of points can vary from A1 through Z99. The name of a point has no special significance besides assisting the user in

visualizing the layout as a grid. However, a reasonable convention is to let the letters represent the rows of a layout and the digits represent the columns.

Point Data is entered in the following format, in separate lines, when prompted by GVSim:

Point Data

- Name (A1 through Z99)
- Location (X coordinate)
- Location (Y coordinate)
- Type (normal, requesting, machining or parking)
- Names of connecting points (may require more than one line)
- Destination points, if the point type is 'requesting' or 'machining' (may require more than one line)
- Request generation/machining rate, if the type of point is 'requesting'/'machining' (may require more than one line)

Destination points and probability function data normally take up a number of lines for input, depending on the type of data. For instance a uniform probability distribution takes four lines of data to specify type of distribution, random number stream, low value and high value, while a Poisson distribution requires three lines of input data to

specify type of distribution, random number stream, and mean.

3.2.3 Probability Distribution Functions

Various probability simulation functions exist, which permit the following distributions. These distributions will be associated with inter-arrival times and machining times. The package provides for three random number streams, denoted by 'n', which are used by some of these functions. The functions available to the user are:

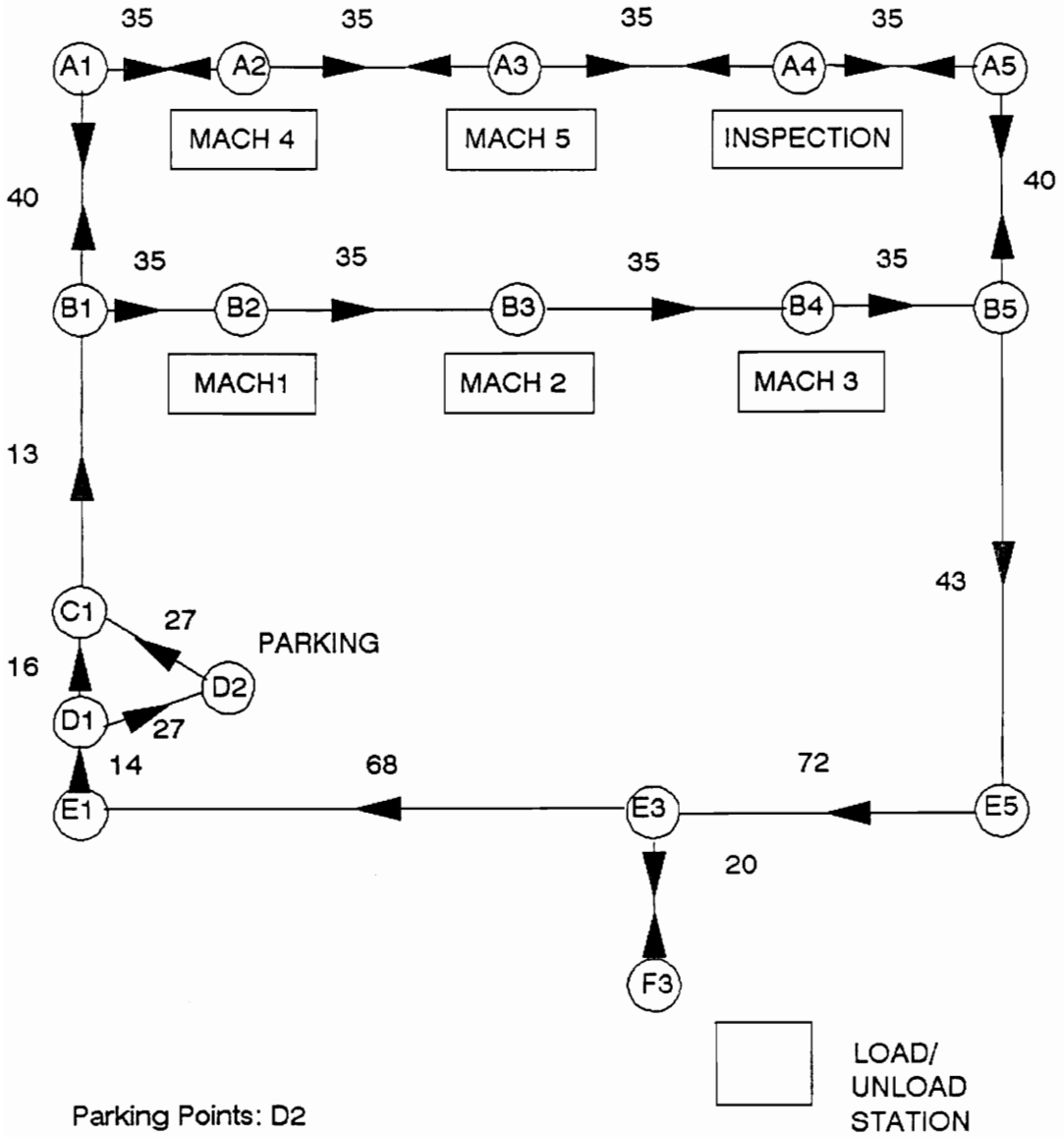
a) Poisson(n, mean) - returns a poisson distribution, about a mean value, using random number stream 'n'.
b) Uniform(n, low, high) - returns a uniform distribution, between a low and a high value, using random number stream 'n'.
c) Random(mean) - returns a uniform distribution between zero and twice the mean value.
d) Expon(n, beta) - returns an exponential distribution for beta, using random number stream 'n'.
e) Normal(n, mean, stdev) - returns a normal distribution, for a mean value and standard deviation, using random number stream 'n'.
f) Lognormal(n, mean, stdev) - returns a lognormal distribution, for a mean and standard distribution, using random number stream 'n'.

g) Triang(n, lo, hi, mid) - returns a triangular distribution, for a low, high and a middle value, using random number stream 'n'.

3.2.4 Guided Vehicle Example Expanded Using GVSim

Figure 3.3 shows how the example mentioned in Section 3.2.1 is modeled using GVSim. The loading/unloading station indicates the receiving and exiting points for the system. The location of the various machining points indicate the machining workcenters. The inspection station is also described as a machining point. The parking point indicates the initial location of the vehicles and also the location of idle vehicles. Arrows on various segments indicate their uni- or bi-directional nature. The coordinates of the points are used to specify their exact location within the layout. Distance values are not entered as part of the data input, but are derived from the coordinates of the points assuming a straight line distance.

Within the layout, the segments are defined by the two points they connected. Segments may be unidirectional or bidirectional. Consider the two points B1 and B2 in Figure 3.3. B2 is entered as a connecting point to B1, hence vehicles can flow from B1 to B2, but not from B2 to B1. Now consider points A2 and A3. A2 is entered as a connecting



Parking Points: D2

Requesting Points: F3

Machining Points: A2, A3, A4, B2, B3, B4

Normal Points: Rest of the points

Figure 3.3: A Guided Vehicle System Example

point to A3, and A3 is entered as a connecting point to A2. This makes the segment A2-A3 bidirectional.

3.2.5 Description of Points

A normal point assists the path planner in generating paths between two points. It also assists in creating buffer zones, which only one vehicle may occupy at a given instant, for preventing collisions. Point E5, for instance, is a normal point. It serves to define the path a vehicle will follow in going from B5 to E3. In the absence of E5, vehicles would move from B5 to E3 in a straight line. Hence normal points assist in defining the exact flow patterns for vehicles within the layout.

'Requesting' points indicate the locations where loads enter the system, i.e. where loads are generated. When entering data on a 'requesting' point, the user needs to specify the probability distribution data for generation of loads. The user is also required to specify the various destination points loads are sent to, from this point. If the load is sent to a machining point, the load will be machined. If the load is sent to a point which is not a machining point, then it exits the system at that point.

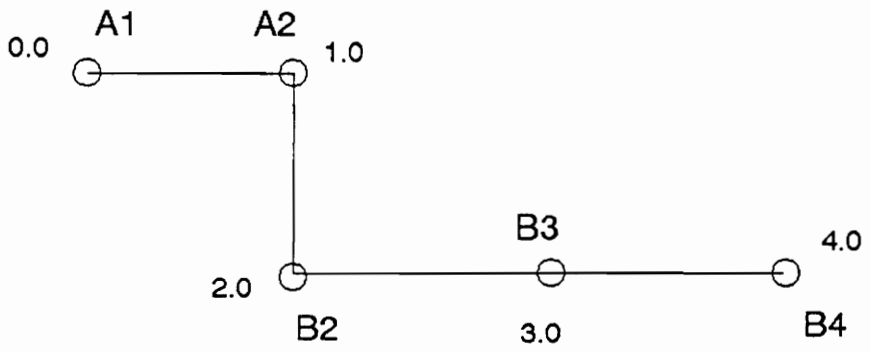
At 'machining' points, operations are performed on the part for a time duration based on the probability distribution data selected for that point. The user also needs to specify the destination point or points to which loads are sent from this machining point. If a specified destination point is a machining point, then the load will be further machined. Otherwise, if the destination point is not a machining point, the load will exit the system at that point. Hence, loads may exit the system from any point which is not a machining point.

A 'parking' point can be used to indicate the initial location of vehicles, as well as specify parking spaces for vehicles when no requests need to be answered. If the number of parking points for the system is less than the number of vehicles, the vehicles are evenly distributed among the parking points. In case the number of parking points equals the number of vehicles, one vehicle is stored at each parking point. Otherwise, if the number of parking points is more than the number of vehicles, then vehicles are stored arbitrarily among the parking points available, until no more vehicles exist to be stored.

Four common options are provided to the user for specifying destination points from any 'requesting' or 'machining'

point. Loads may travel to the same point, or a number of points alternatively, without machining. Loads may travel to some specified machining points or any machining point. A fifth option, for 'requesting' points only, allows loads to enter the system at the same point using separate probability distribution data and exit the system at a single destination point.

Paths are defined as sequences of one or more path segments from one point to another. Vehicles travel along these segments when going from some initial point to a desired destination point. Along a path, points are booked by the vehicle for the time duration a vehicle is present on any segment on which the point lies. For instance, let points A2, B2, and B3 be on a path (see Figure 3.4). Point B2 is booked for the duration a vehicle is on segment A2-B2 and B2-B3. This is done to prevent any other vehicles from accessing point B2 during the booked time period, and hence prevent conflicts. That is why, in case of long segments, normal points may need to be specified within the segment. This prevents vehicles from booking a point for a long period of time.



The numbers indicate the time a vehicle arrives and departs at a point

Point B2 is booked from time t = 1.0 to t = 3.0

Figure 3.4: Booking of Points

3.2.6 Routing, Scheduling, and Sequencing

Within a normal production environment, the user might have information on the layout and on the process flow of the parts moving through the system. Besides a diagram to aid in describing the layout, the user needs a means for planning the sequence of part movement within the layout. The user may use a from/to chart to aid in specifying part flow. The from/to chart would help in describing the various stations parts visit on their way through the system.

Jobs are put in the system by specifying where they enter the system and where they are going. The user specifies where the job goes first and, if the destination point is a machine, specifies at that machine where it goes next. When a job reaches a destination point which is not a machining point, it exits the system. A detailed description of how parts actually plan their paths within the system is given in the following section (Section 3.3).

3.3 Path Planning

Path planning consists of a cyclic process which uses the shortest path software module, the timing function, and the

conflict resolver software module to arrive at a path between two points (Figure 3.5). The task of path planning is performed by the Path Planner, which dynamically allocates routes between requesting and destination points. Hence, the path allocated between two points may not be the same each time. In selecting the path of the vehicle, the main criteria is the earliest arrival time in reaching the destination.

There are four possible types of paths possible within the package. Vehicles may travel from requesting points to exiting points. They may also travel from requesting points to machining points. From a machining point, vehicles may travel to other machining points or exiting points.

The Path Planner requires the starting and ending points of the path as input. By invoking the shortest path algorithm, the shortest path between the two points is obtained. The shortest path between the two points is the ideal path. If there is some conflict along the shortest path, then in the next iteration, the shortest path module returns the next longer path. A detailed description of the shortest path algorithm and the conflict resolver follow this general introduction.

The timing function is called to attach the time values the vehicle occupies various points along the path. The timing function superimposes timing values for the arrival times at points along the path. This is based on the distance between points.

The path, with attached arrival time values, is sent to the conflict resolver module. The conflict resolver checks the path to determine conflicts. Conflicts may exist with already existing paths allocated to vehicles. In case of conflicts, the conflict resolver removes the conflict by lengthening the time it takes the vehicle to traverse the specified path, and returns this information.

On obtaining the path and final travel time from the conflict resolver, the path planner checks to see if any conflict had been present and removed from the path. In the absence of any conflict, it returns the path as the optimal. In case some conflict had been present and removed, it goes through the next iteration of calling the shortest path, timing and conflict resolver routines. It now obtains the next longer path and compares the paths to see which leads to a quicker arrival time.

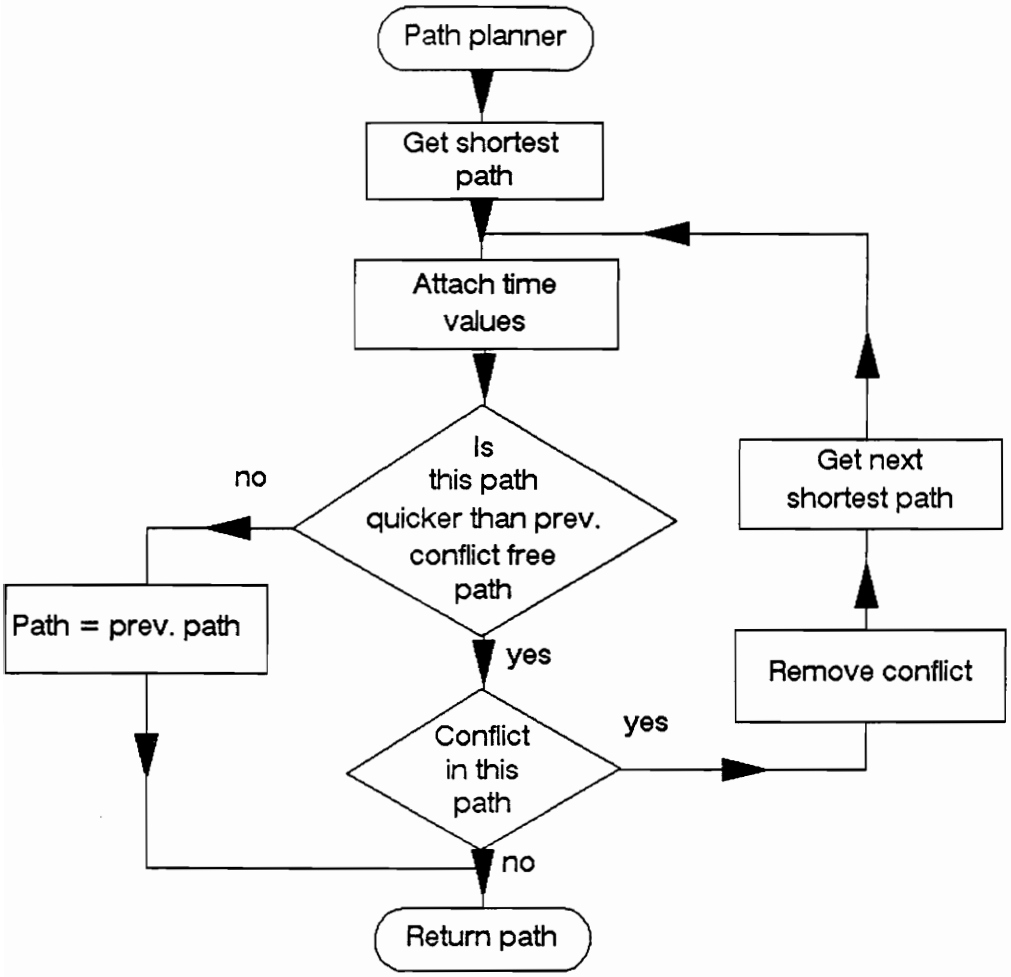


Figure 3.5: Conceptual Flow Chart for Path Planner

Suppose Path 1 has an ideal arrival time of 19.0. However, due to point A2 being booked by another vehicle at an overlapping time, Path 1 has a estimated arrival time of 25.0 after conflict removal. Since the estimated time is different (greater) from the ideal time, Path 2 (the next shortest path) is calculated. Obviously Path 2 is equal to or longer than Path 1. Now, Path 2 can have a ideal time less than and equal to, or greater than the new estimated time of Path 1. If Path 2 has an ideal time greater than the estimated time of Path 1, then the estimated time of Path 1 provides the quickest path. Otherwise, if Path 2 has an ideal time, say 21.0, which is less than the estimated time of Path 1, then it is sent to the conflict resolver to obtain its estimated time (time after conflict removal). If the estimated time of Path 2 is less than the estimated time of Path 1, then Path 2 is returned as the quickest path. Otherwise the iterations are continued until a quickest path is obtained.

The existing procedure involves allocating the shortest path and removing conflicts as and when they occur, based on different conflict removal principles. This leads to a limitation when paths between two sets of points may overlap, creating bottlenecks. In other words, a longer path may be better if there is less congestion.

Additionally, in the presence of conflict, vehicles attempting to follow the shortest path may lead to high waiting times along the path. This leads to increased contention of certain segments of the layout. Alternate path consideration reduces this limitation in transporting material.

After determining the quickest path, the occupation times of the vehicle along the path are booked to prevent other vehicles from occupying the points at the same time. Starting with the second point, the point is booked for the time period the vehicle is present at the point, or on the segments on which the point lies. In Figure 3.4, point B2 is booked from time 1.0 to 3.0. The conflict resolver prevents the occupation of these points by other vehicles for the time periods they are booked.

3.3.1 Shortest Path Algorithm

The shortest path algorithm used is the one described by Horowitz and Sahni [13]. An iterative search is performed to arrive at the shortest path between two points in a network of connected points. The algorithm requires a layout of points, some of which are connected, and their Cartesian coordinates. The inputs to this algorithm are the starting and ending points of the path being calculated.

The algorithm uses the shortest straight line distance between connected points in its calculations.

Initially, the distance of every point from the starting point is set as infinity. Also, the starting point is put as the parent for every point in the layout. From the starting point, the closest connected point is reached. As a closest point is visited, it is marked as 'visited'. In subsequent iterations, the algorithm travels to the non-visited point which is closest to the origin via a path which only traverses points that have already been visited. Now, say the algorithm visits a closest point B33. The points connected to B33 are scanned to determine whether the distance covered in reaching them via B33 is shorter than the already existing distance. If the distance via B33 is shorter, then B33 is put as the parent of that point, else the point is left alone. This, however, does not mark the point as visited. The algorithm then returns to scanning the non-visited points and finds the one with the shortest path to the origin. This cycle continues until the algorithm visits the destination point. The route to arrive at the destination is then output to the Path Planner.

Additionally, a second iteration of the shortest path algorithm is required to find the next longer path. If the

path planner has determined that the travel time on the shortest path may not be optimal, the shortest path module follows the same procedure as above, with one difference. When the connected points to a newly 'visited' point are being scanned, information on two parents may be stored. If the path via the 'visited' point is shorter than an already existing path, the 'visited' point is put as the parent of that point and the already existing parent of that point is then put as the second parent or vice versa. Of course, if there is no already existing path, the procedure is as in the first iteration. When the destination point is reached, the iterations continue until a second path to the destination point is determined. The same process is performed for finding subsequent longer paths.

3.3.2 Timing Function

The timing function determines the actual time necessary to travel each segment of the path, from the starting point to the destination. Taking each adjacent pair of points along the path, the timing function determines the straight line distance between them. Taking into account the velocity and acceleration of vehicle, it determines the time taken for the vehicle to travel between the two points. If a vehicle is just starting to move, it accelerates the vehicle (to its

maximum velocity, if possible). On nearing the destination, it decelerates the vehicle to stop at its destination.

3.3.3 Conflict Resolver

The conflict resolver performs the task of checking the paths for possible conflicts (Figure 3.6). The conflicts along the path are determined and removed. One thing the conflict resolver does not do is dynamic rescheduling of vehicles, i.e. the paths of already allocated vehicles are not altered. All the iterations and feasibility testing are of vehicles which have just received job requests and are trying to plan the route to the requesting or destination points. There are three types of conflicts possible, namely head on collision, intersection collision, and faster moving vehicles colliding with slower moving vehicles in front. All of these collisions are removed by the following procedure.

Going point by point along the path, the occupation times of the point are compared with the time periods (if any) the point is booked by already allocated paths. On observing the occupancy of any point, at an overlapping time with a previously booked time, the vehicle waits at a previous point for the point to become idle. Then, the conflict

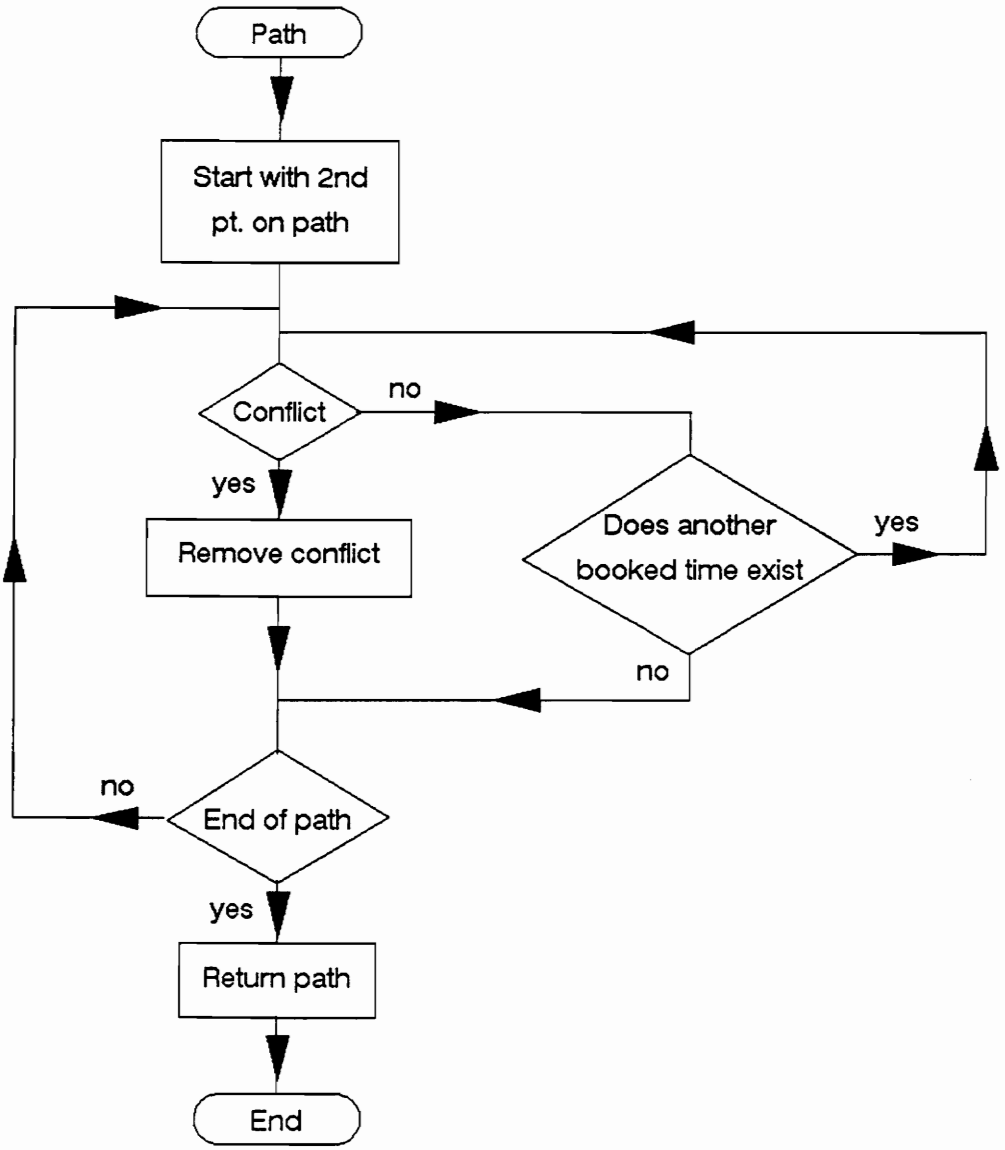


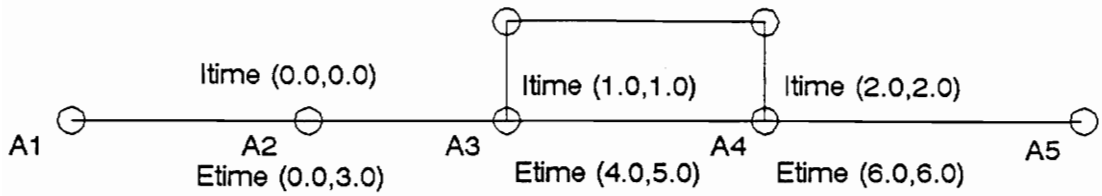
Figure 3.6: Conceptual Flow Chart for Conflict Resolver Module

resolver checks to see if waiting creates a conflict at the previous point.

For example, consider a vehicle arriving from point A2 to point A3 at time 1.0 (Figure 3.7). Point A3 is not booked by any other vehicle till time 2.0. On arriving at point A3 it observes that point A4 is booked from time 1.0 till 5.0. Therefore the vehicle cannot reach point A4 till after time 5.0, requiring it to wait at point A3 until time 5.0 (which is after time 2.0). This creates a conflict at point A3, since A3 is booked from 2.0 to 3.0. The conflict resolver then backs up to point A2. Essentially, on observance of conflict, the vehicle waits at the previous point, and even backtracks if the waiting creates another conflict. The conflict resolver continues this iteration, making the vehicle wait at previous points on observance of conflict, moving forward in the absence of conflict, until it reaches its destination. The conflict free path, with an estimated arrival time, is then returned to the path planner.

3.4 Vehicle Description and Allocation

The number of vehicles present in the system are specified by the user in the beginning of the simulation. All vehicles are similar and carry one load at one time. They



Pt. A1 = Not booked

Pt. A2 = Not booked

Pt. A3 = Booked from 2.0 to 3.0

Pt. A4 = Booked from 1.0 to 5.0

Itime (t1, t2)

Itime = Ideal time

t1 = arrival time

t2 = departure time

Etime (t1, t2)

Etime = Estimated time

t1 = arrival time

t2 = departure time

Travel time between points is 1 time unit

Figure 3.7: Ideal and Estimated Times

all have the same velocity, acceleration, and deceleration, as specified in an 'include' file.

Initially all vehicles are located at parking points in the layout. They are symmetrically spread among the parking points. Idle vehicles are also sent to the parking points to wait for job requests.

Vehicle Allocation is not based on first in first out, but on distance and time taken to satisfy the request (Figure 3.8). The first vehicle becoming idle is not necessarily allocated. On generation of a vehicle request, the nearest vehicles are scanned for vehicle allocation. The main criteria is the arrival time of the vehicles at the vehicle requesting point. Loaded vehicles that are currently busy and vehicles heading for parking are also considered, taking into account the time they will take to become available. Priority is given to vehicles which are scheduled to drop off a part at the requesting point. Hence, the quickest response time in answering the request is the primary criteria for allocating vehicles for a particular request.

The user is provided the option of having only certain vehicles answer requests from particular points or having vehicles free to reach any point on the layout. This

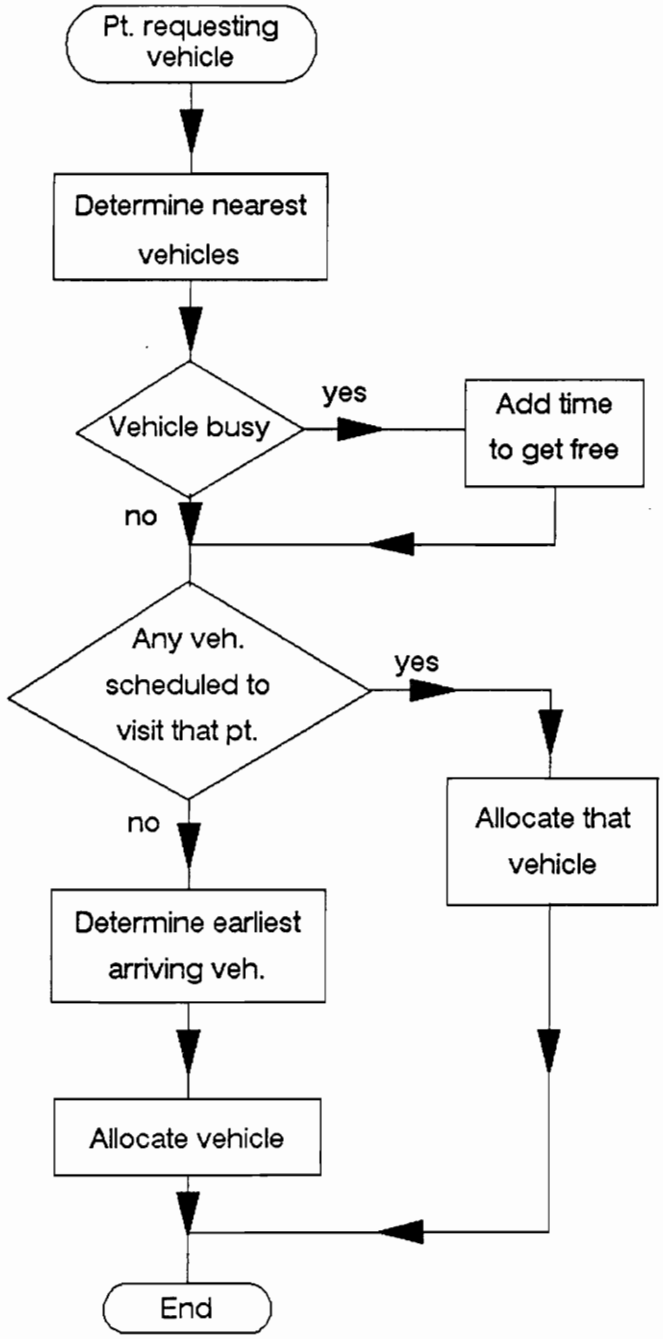


Figure 3.8: Conceptual Flow Chart for Vehicle Allocator

feature is particularly helpful in large layouts, with a large number of vehicles. In that case, if vehicles are free to move to any part of the layout, simulation might lead to congestion of vehicles in a particular area. Congestion leads to blocking and further congestion. For this, only certain vehicles may answer requests from certain points, and in job requests from them, only those vehicles are considered.

3.5 Data Input

Before simulating a system, the system parameters need to be defined by the user. The data input described is composed of three parts. The first part is the description of the layout. The other two parts are specification of vehicle data and options, as described below. A sample data input for the system shown in Figure 3.3 is presented in Appendix 1.

Each parameter of data is individually prompted for, and is entered in separate lines. Initially, the time of simulation is entered by the user. Then the user defines various points within the layout in the format described in Section 3.2.2. Blank lines are used to separate data on different points. On completion of layout description, the

user enters 'FINISH' to indicate completion of layout input. The next stage requires defining the various vehicle parameters and options. The number of vehicles and load/unload times of parts from the vehicle are then entered, on being prompted by GVSim. Finally, the user is asked to specify dedication of vehicles to certain points and also select the option of having idle vehicles wait at the parking or drop off point.

3.6 Data Output

Data is output on the various program constructs to allow for system evaluation. Data is collected on 1) machine utilization, 2) average waiting time before a job request for each decision point is satisfied, 3) percentage of simulation time each point causes conflict, 4) vehicle idle times, 5) loaded vehicle busy times, and 6) empty vehicle busy times.

A sample of the data output is presented in Figure 3.9. As can be seen, individual as well as collective performance measures are output for various vehicles and points. The output data is divided into two parts, namely point data and vehicle data. In point data the various columns represent the following data:

name = name of point

type = type of point

blocked = percentage simulation time a vehicle is blocked at that point

part waiting = percentage simulation time a part is waiting to be picked up by a vehicle at that point

busy = percentage simulation time a machining point is busy

In vehicle data the various columns represent the following data:

Veh. no. = identification number of vehicle

Ded = indicates whether the vehicle is dedicated or not (0 = not dedicated, 1 = dedicated)

Idle = percentage simulation time vehicle is idle

Busy empty = percentage simulation time vehicle is traveling to pick up load

Busy loaded = percentage simulation time vehicle is travelling with load

Parking = percentage simulation time vehicle is travelling to parking destination

Loading = percentage simulation time vehicle is loading a part

Unloading = percentage simulation time vehicle is unloading a part

POINT DATA

NAME	TYPE	BLOCKED	PART WAITING	BUSY
A1	normal	0.000		
B1	normal	0.000		
C1	normal	0.000		
D1	normal	0.000		
E1	normal	0.000		
F1	request	0.051	0.001	
A2	machining	0.008	0.001	0.868
B2	machining	0.010	0.000	0.902
D2	parking	0.000		
E2	normal	0.000		
A3	machining	0.007	0.001	0.704
B3	machining	0.007	0.000	0.867
E3	normal	0.006		
A4	machining	0.007	0.001	0.618
B4	machining	0.003	0.000	0.826
A5	normal	0.002		
B5	normal	0.003		

Av. utilization of machines = 0.834

VEHICLE DATA

VEH NUMBER	DED	IDLE	BUSY EMPTY	BUSY LOADED	PARKING	LOADING	UNLOADING
0	0	0.407	0.115	0.127	0.118	0.117	0.116
1	0	0.567	0.085	0.094	0.080	0.087	0.087
2	0	0.822	0.037	0.044	0.028	0.035	0.034
MEAN		0.599	0.079	0.088	0.075	0.080	0.079

Figure 3.9: Sample Data Output

Chapter IV

4. THE GVSim COMPUTER MODEL

4.1 Introduction

GVSim was developed in Unix, on the SUN 386i system. It is written in the 'C' programming language. GVSim employs a discrete event simulation methodology. In discrete simulation, the values of dependent variables can only change discreetly, at specified points in simulated time. These points are referred to as event times. In the systems modeled using GVSim, the state of the system remains constant between event times. A dynamic representation of the system is obtained by advancing simulated time from one event to the next. This is referred to as the next event approach and is used in most discrete simulation languages.

This chapter discusses in detail the computer model developed. Flowcharts depicting the overall model as well as some of the individual modules are presented. The chapter starts with a description of the overall structure of the package. First, the sequence of operations of the package is presented. Then, the major data structures used within the package are listed and detailed. The next

section provides a more detailed explanation of the sequence of operation of the code. Finally, the probability distribution functions provided within the package are detailed. The code for the computer model is contained in the Appendix.

4.2 Overall View

The 'main' file within the GVSim is called 'sim_run()'. 'Sim_run()' incorporates the primary file for executing the simulation. A conceptual flow chart of 'sim_run()' is shown in Figure 4.1. 'Sim_run()' starts by calling the function 'entcord', for inputting data. 'Entcord' first prompts the user to specify the layout as described in Chapter III. The user is then prompted for the remaining system parameters, such as number of vehicles, load transfer times, etc. After the description of the system parameters, 'sim_run()' calls the function 'intevlst'. 'Intevlst' initializes the event list by loading the first events for each point generating requests. After initializing the event list, the first event is removed. Then, depending on the type of event, the required function is called and the logic is executed. Events are loaded and removed from the event list by 'sim_run()' until the end of simulation time.

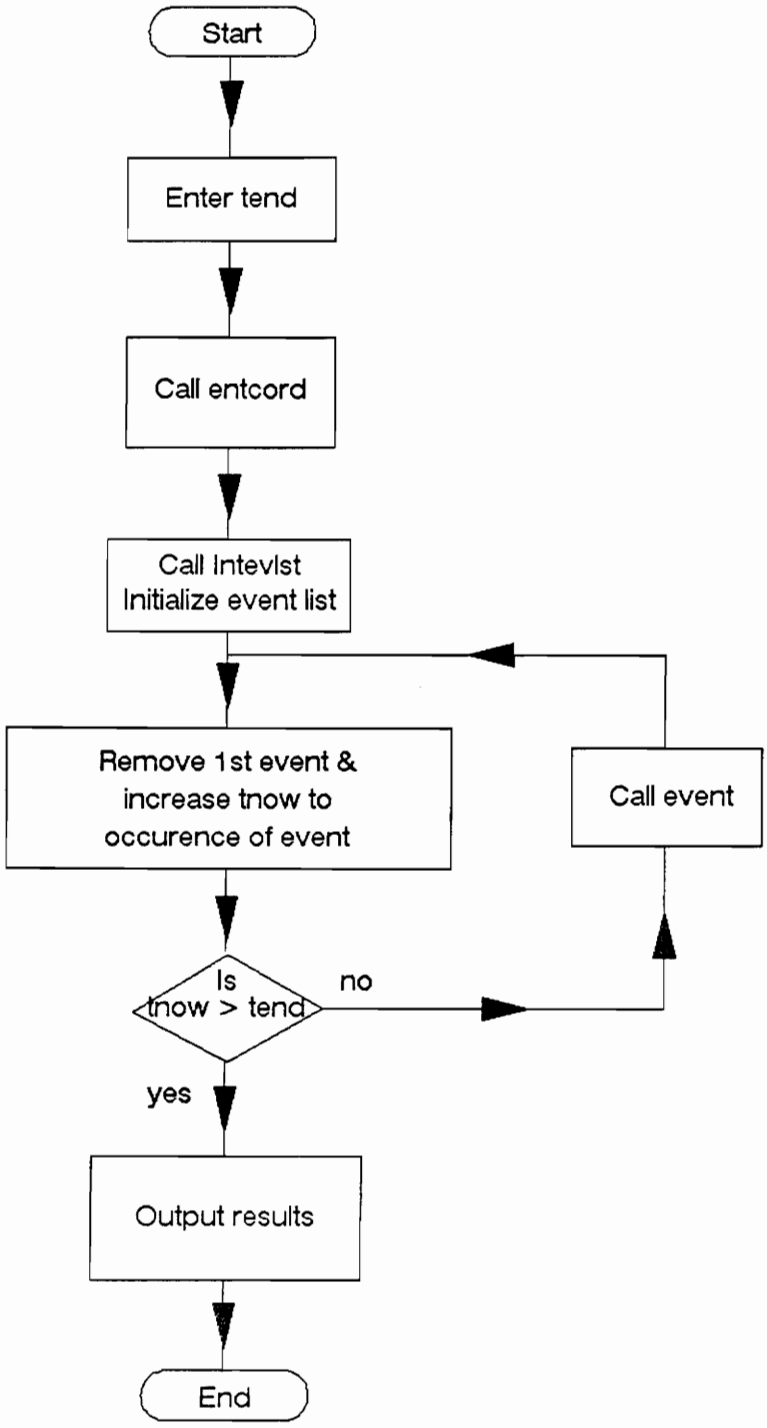


Figure 4.1: Conceptual Flow Chart for Sim_run

The program incorporates four types of events, namely request, pickup, release and parking. The logic modeled for each event is contained in separate functions, named after the type of event. These functions are called by 'sim_run()' on occurrence of that event. Event 'request' indicates a request for a vehicle to pickup a part. In this event, the vehicle allocation module is called and a vehicle is allocated to answer the request. The path of the vehicle is determined and an event notice of type 'pickup' is loaded on to the event list. Event 'pickup' indicates the arrival of a vehicle at a point to pickup a load. Here, the path of the vehicle to the destination point is determined and an event of type 'release' is scheduled. Event 'release' indicates the arrival of a vehicle at a destination point. The last type of event is 'parking', which indicates the arrival of a vehicle at a parking location. The vehicle waits at the parking location until it is allocated. A more detailed description of the various events is presented in Section 4.5.3.

4.3 Major Data Structures

This section provides a detailed explanation of the major data structures used by GVSim for storing and using information on the program constructs. A listing of these

structures along with other fixed parameters (such as speed of vehicles, maximum number of parking locations) are provided in the file 'struct.h'.

The data structure 'matrix' is the major structure used within the program. It acts as a warehouse for storing information on the layout. It contains the name, coordinates and type of each point. Information on the interconnection between various points is also stored. The variable 'dest[]' stores the various points loads may travel to, from this point. The variable 'rand' is a pointer to a data structure 'random', which stores the probability distribution function data. Finally, the variable 'occup_time', which is a pointer to a linked list 'time', stores the data on the various time periods a point is booked by a vehicle.

'Ev_lst' is another major data structure used within the program. It is a doubly linked list, containing events in their sequence of occurrence. It contains information on the type and time of occurrence of each event. The location of occurrence of each event and the destination point the vehicle travels to from that location are also stored. The identification number of the vehicle servicing this event is also stored depending on the type of event.

The data structure 'vehicle' contains the variable describing each vehicle at any instant in time. The variable 'status' can have six values namely: 1 (unloaded, answering a job request), 2 (loaded, transporting a part), 3 (travelling to parking location), 4 (unspecified), 5 (idle, at parking point), and 6 (idle, waiting at last release point). The variable 'dest_pt' indicates the point where the vehicle is presently located or is headed towards. 'Dest_time' is used to store the time the vehicle reached or will reach 'dest_pt'.

'Mapname' is the data structure used within successive path calculations. For each point, it stores information on the name of that point and whether the point has been visited. The variable 'sdist[]' stores the successive shortest distances to that point from the 'from_no' point (in the function 'shortp').

'Nodeparent' is used to store the ancestors (previous point) of a point, in path calculations within the shortest path module. For example, in a path A-B-C, B is the parent of C and A is the parent of B. Hence when the program is at point C, it can retrace its steps to describe the path taken to reach C.

The data structure 'path' contains data on successive paths between two points. It is used for quickest path calculation by the path planner. It contains the arrival and departure times of a vehicle for points along a particular path.

The data structure 'statistics' contains the variables which store information on system characteristics. The element 'veh_idle[][2]' contains the last time each vehicle was idle and total time the vehicle has been idle. Element 'veh_blocked[]' contains the total time a vehicle has been blocked by some other vehicle. 'Veh_busy_empty[]' and 'veh_busy_loaded[]' store the total time each vehicle was answering a job request. 'Veh_parking' indicates the time a vehicle spends in travelling to a parking location.

'Raw_part_waiting[][2]'/'fin_part_waiting[][2]' indicates the total time and number of times a requesting/machining point waits for a vehicle to pick up a load. Finally, 'pt_blocked' indicates the time period for which a point is under contention by two or more vehicles.

4.4 GVSim Input/Output

A sample data input for GVSim is shown in Appendix I. Each set of data is entered on separate lines, on being prompted by GVSim. The sequence for input of data is as follows:

Specify simulation time.

Specify layout (as shown in Section 3.2.2).

Specify number of vehicles.

Specify load transfer times from and to the vehicle.

Specify whether idle vehicles wait at drop off point or parking.

Specify if any vehicles are dedicated to a point.

Other parameters such as vehicle speed, maximum number of machining points, maximum number of paths calculated by the path planner, etc. are defined in the include file "struct.h". The users may modify various such parameters defined within the file, if they so desire.

Data is output on the various program constructs to allow for system evaluation (Figure 3.9). Data is collected on 1) number of times all vehicles are busy, 2) average waiting time before a vehicle is allocated for a job request for each decision point generating job requests, 3) percentage of simulation time each decision point causes conflict, 4)

vehicle idle times, 5) loaded vehicle busy times, 6) empty vehicle busy times, and 7) machine utilization.

4.5 Detailed Code Explanation

4.5.1 Entcord()

'Entcord()' is the function which contains the logic for specifying the system being modeled. A conceptual flow chart for layout input part of 'entcord()' is given in Fig. 3.2. 'Entcord()' prompts the user to input the layout and define other system characteristics. Besides the description of the layout, the user is prompted to specify the number of vehicles, waiting location of idle vehicles, load transfer times to and from vehicle, and dedication of vehicles to specific areas.

In specification of the layout, the user is prompted to define data on the various points comprising the layout. When the type of point being defined is 'requesting' or 'machining', the function 'demand' is called. 'Demand' incorporates the code for specifying the various destinations and probability distribution data for these points. From 'requesting' and 'machining' points, loads may leave for one of several destination points, as specified by the user. The destination points may be non-

'machining' points (where loads exit the system) or 'machining' points. However, the possible destination points may not be a combination of non-'machining' and 'machining' points. In case of 'machining' points, the user may specify some or all 'machining' points as destinations from that point. During allocation, the search for an idle machine will be limited to the machining points specified by the user as destination points. Single or multiple non-'machining' points may also be specified, with loads travelling alternatively in case of multiple non-'machining' points. Additionally, from 'requesting' points, the user may specify loads entering the system with separate probability distribution data and travelling to a single destination point.

A number of random number functions are provided to the user to generate load requests for vehicles from workcells (Section 4.6). The user selects the random number functions for each 'requesting' and 'machining' point. The user is prompted to enter the values (mean value, low value etc.) of the variables used by the random number generator selected. For example, the triangular distribution random number generator would require specification of the random number stream and the low, middle and high values of the random number. A detailed description of the various probability

distributions available to the user is given in Sections 3.2.3 and 4.6.

4.5.2 Intevlst

On completion of the definition of the system parameters, the 'intevlst' routine is called. This routine visits all the 'requesting' points (where loads enter the system) and loads the occurrence of the first event for that point. In scheduling events, the probability distribution data for that point are used.

4.5.3 Event Logic Description

After the event list is initialized, simulation is started. The first event is removed from the event list and, depending on the type of event, the logic is performed. Events continue to be loaded and removed from the event list until the end of simulation time.

The flow chart depicting the logic for event type 'request' is depicted in Figure 4.2. Event 'request' indicates a request for a vehicle by a load, at either a 'requesting' or a 'machining' point. If the point generating a request is of type 'requesting', then the next 'request' event is scheduled, based on the probability distribution data for

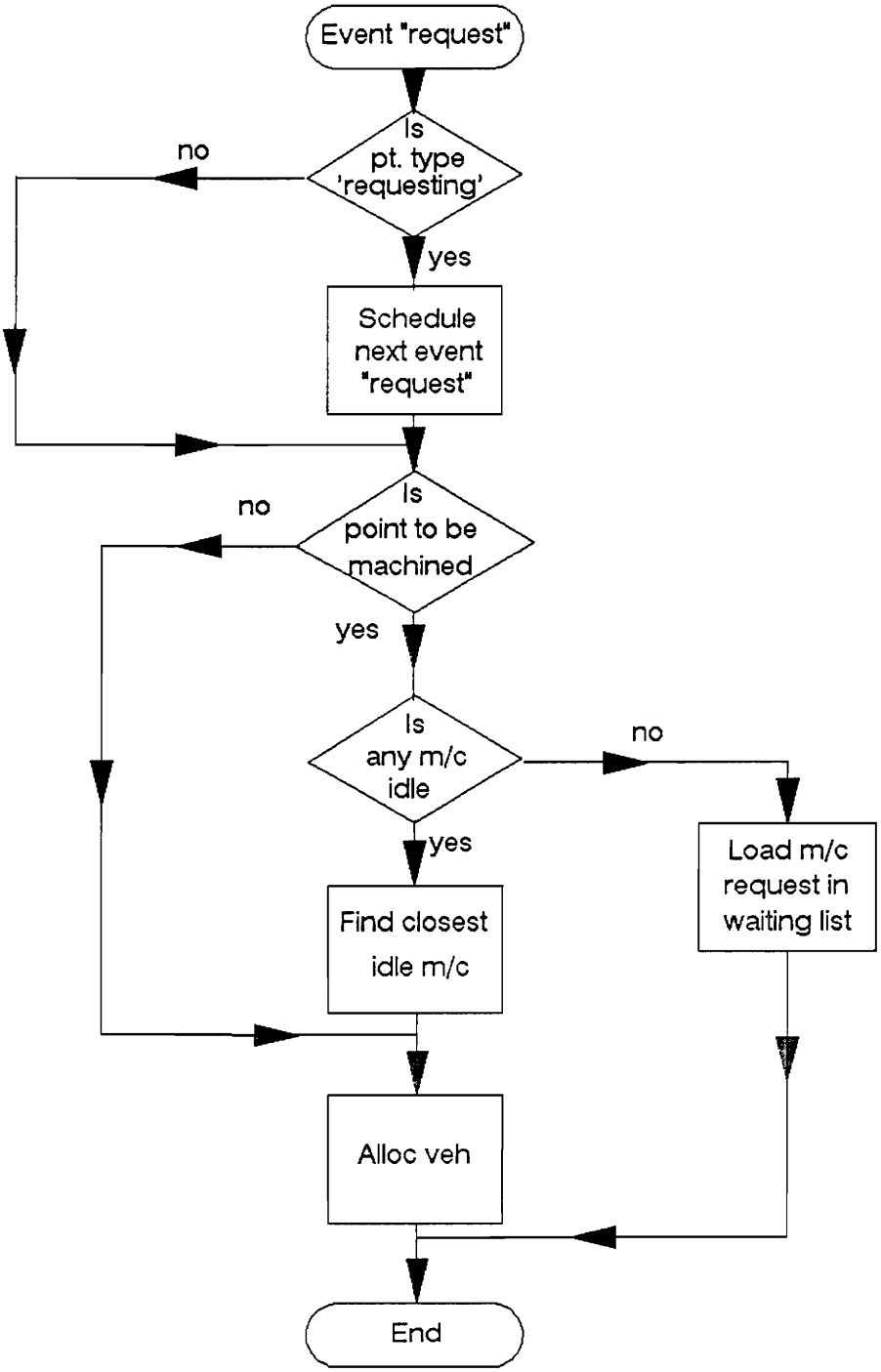


Figure 4.2: Event 'Request'

that point. For both type of points, the program checks to see if the load needs to be machined. Whether a load needs to be machined or not is determined by the destination points loads may travel to from that point. If the load needs to be machined, the 'machining' points are checked to see if they are idle. At this time, the closest idle 'machining' point, to which loads may travel, is selected. In case an idle 'machining' point is not available, the request is stored in the event list 'top_mach'.

'Top_mach' is a doubly linked list for storing requests waiting for an idle machine. If a machine is idle or if the load does not need to be machined, the vehicle allocation module is called. The vehicle allocation module scans available vehicles for determining which vehicle can answer the request fastest. If no vehicles are available, the vehicle request is placed in the event list 'top_veh'. 'Top_veh' is a doubly linked list for storing requests for vehicles. If vehicles are available, the path of the vehicle is determined and an event of type 'pickup' is loaded on to the event list. The logic for event type 'pickup' is depicted in the flow chart in Figure 4.3. If an event 'pickup' occurs at a 'machining' point, the machine is set to idle. The event list 'top_mach' is then checked to see if any requests for idle machines are present. If a

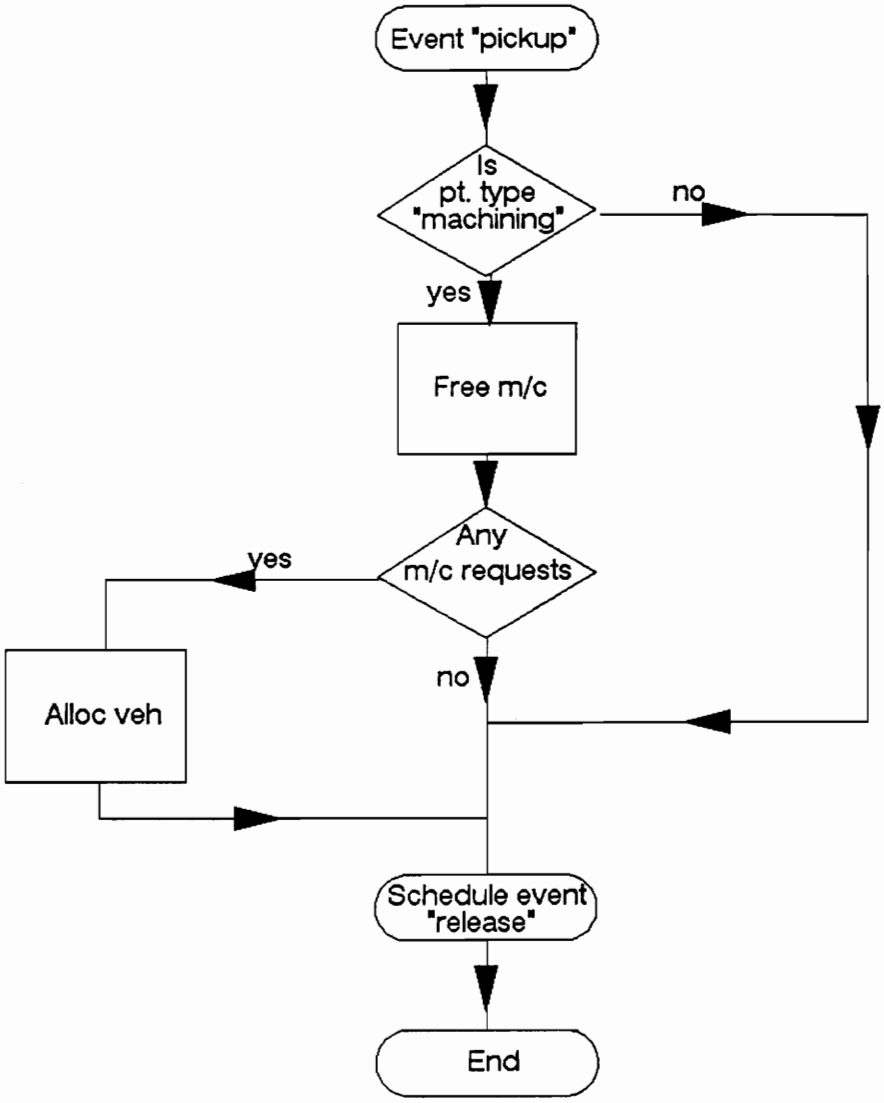


Figure 4.3: Event 'Pickup'

request is waiting, the vehicle allocation module is called to allocate a vehicle to satisfy that request. The next step in the logic is to scan the destination points that loads may travel to from this point. Similar to event 'request', a destination is selected and a event of type 'release' is scheduled.

Event type 'release', as depicted in Figure 4.4, indicates the release of a load at a point. If the vehicle releasing the load has not already been allocated for a task, then the event list 'top_veh' is checked. If no vehicle requests are waiting, the vehicle is either sent to parking or waits at the release point (depending on user specification at startup).

Finally, the event 'parking' indicates the arrival of a vehicle at a parking point. The vehicle is then checked to see if it has already been allocated. If the vehicle has not already been allocated, it is set as idle. The logic for this is shown in the flowchart in Figure 4.5.

4.6 Probability Distribution Functions

A number of random number functions [26] are provided for scheduling vehicle demand. These random number functions

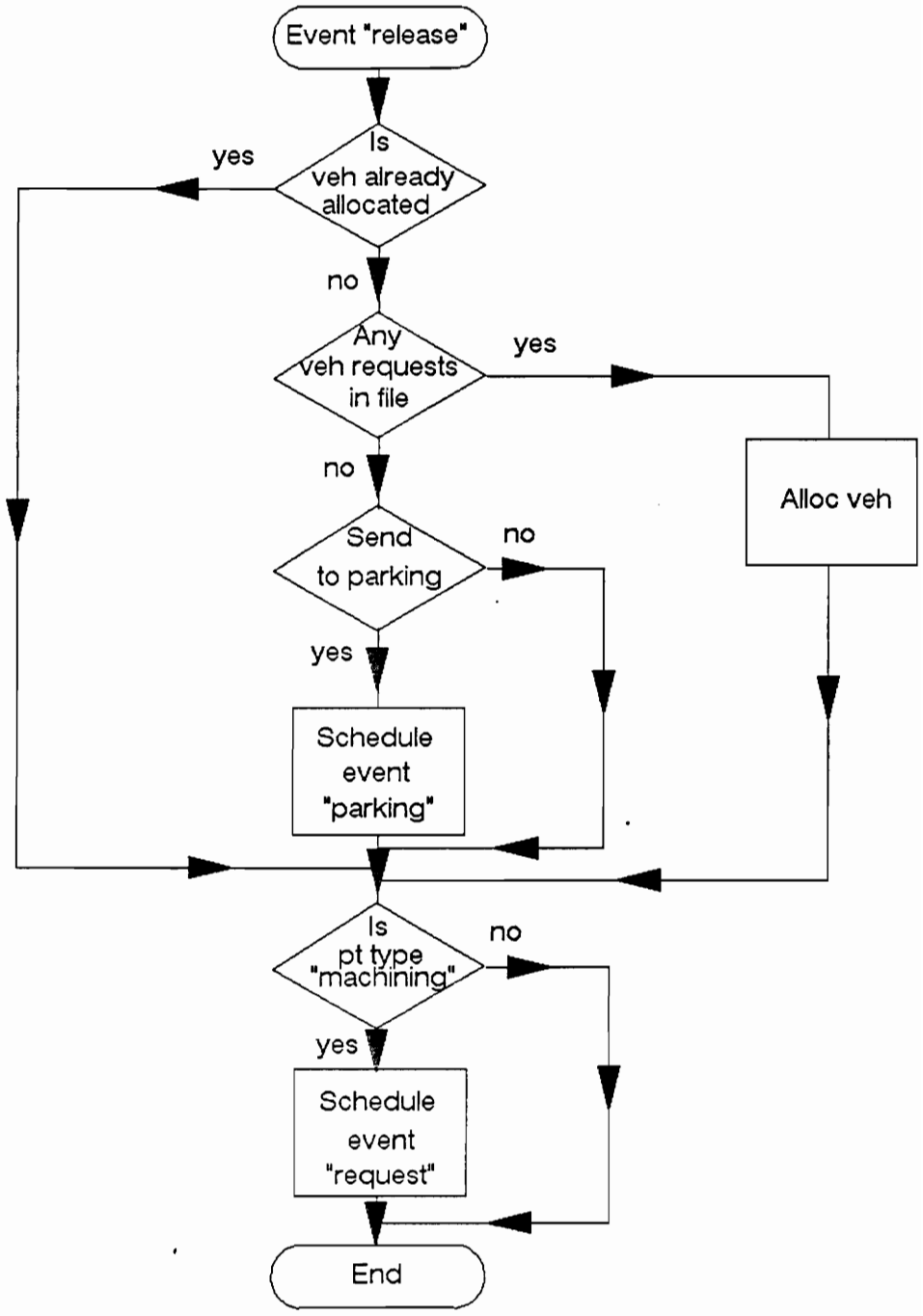


Figure 4.4: Event 'Release'

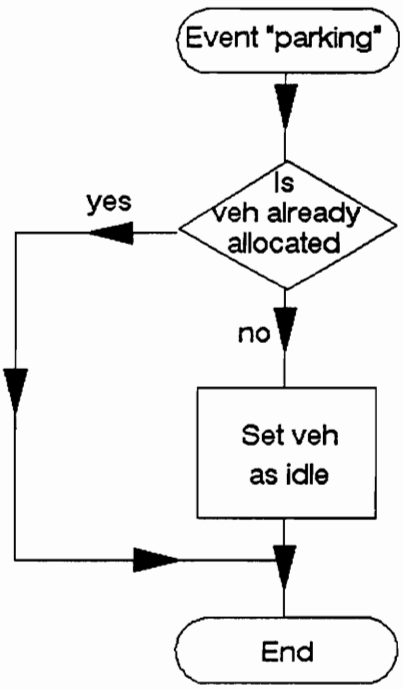


Figure 4.5: Event 'Parking'

are used to schedule events, as specified by the user. Initially, the user specifies the random number function for different 'requesting' points. These functions are then used to generate vehicle requests from these points. The package provides for three random number streams, denoted by 'n', which are used by some of these functions. The functions available to the user are:

a) Poisson(n, mean) - returns a poisson distribution, about a mean value, using random number stream 'n'.

b) Uniform(n, low, high) - returns a uniform distribution, between a low and a high value, using random number stream 'n'.

c) Random(mean) - returns a random distribution about a mean value.

d) Expon(n, beta) - returns a exponential distribution for beta, using random number stream 'n'.

e) Normal(n, mean, stdev) - returns a normal distribution, for a mean value and standard distribution, using random number stream 'n'.

f) Lognormal(n, mean, stdev) - returns a lognormal distribution, for a mean and standard distribution, using random number stream 'n'.

g) Triang(n, lo, hi, mid) - returns a triangular distribution, for a low, high and a middle value, using random number stream 'n'.

4.7 Include file "struct.h()"

The file 'struct.h()' contains the definition of the fixed parameters used within the package. Data such as the velocity and acceleration of the vehicles, maximum number of points possible, maximum number of machining points, maximum number of parking locations, maximum number of paths which may be calculated by the path planner, etc. are stored here. The various structures and linked lists used within the computer model are also stored here. The users can go in and redefine these parameters, if they so desire.

Chapter V

5. Model Validation

A means is required to provide confidence in model simulation by GVSim. Reliability of data output by GVSim on system performance needs to be demonstrated. One means of doing so is to run a model using both GVSim and some established software package. Comparison of results will lead to determination of accuracy of GVSim in depicting the system being modeled. Hence a model of an industrial layout is considered. The model is similar to the one given in Example 16.2 [27] by Pritsker. The system is simulated using both SLAM and GVSim. The resultant output data of both are compared to provide confidence in the data output by GVSim.

5.1 SLAM Model Description

The SLAM model used in the validation is similar to the one in [27], with a few modifications. The layout consists of six machining workcells, one parking location, and a vehicle arrival station. There are two vehicles available to the system, for transporting the parts between the various stations. Vehicle flow is clockwise along uni-directional

segments, except on the spur in front of the vehicle arrival station. The spur allows for vehicle flow in both directions. Raw cast parts arrive exponentially, with a mean of 436 seconds, at the vehicle arrival station. The raw parts need to be machined at one of the machining cells. If a machining cell is available, a request is made for transport of the part to the machining cell by an AGV. Otherwise, the request is placed in a queue until a machine becomes available.

Vehicles are allocated and parts are picked up using FIFO. Idle vehicles wait at segment 4, the parking location. The two vehicles are initially idle at control point 4. On allocation of a vehicle, the part is picked up and delivered to the appropriate machining cell. It is unloaded from the AGV and processing by the machine begins. The time to put a part on or to take it off the AGV is 45 seconds. Machine processing time for a part is triangularly distributed with a minimum value of 960, a modal value of 1860, and a maximum value of 2720 seconds. After delivery, the AGV is released to perform other jobs. If no jobs are outstanding, the vehicle returns to the parking station. Upon completion of the machining process, an AGV is requested to return the part to the vehicle arrival point, where it exits the system.

Vehicle control points on the AGV guidepath are marked as circles with the identification number of the control point in the circle. Vehicle path segments are identified with a number on the line between control points. Arrows show the direction of movement allowed on a particular segment. The network consists of 19 nodes which model the flow of loads from the fixture station to the manufacturing cells over the AGVS. The layout is described as a series of segments, with the nodes it connects, in VSGMENT. The length and type (uni/bi-directional) of the segments are also specified.

The vehicle characteristics are defined in VFLEET. They are specified as NVEH = number of vehicles, ESPD, LSPD = empty and loaded speed of vehicle, ACC, DEC = acceleration/deceleration of vehicle, LEN = length of vehicle, DBUF = minimum space between vehicles, CHKZ = check zone at intersections, IFL/RJREQ = file list containing job requests for the vehicle/rule for task selection by vehicle, RIDL = logic rule applied for the positioning of idle vehicles, and ICPNUM(NOV, SGNUM) = initial location of vehicles.

The system was simulated for 89,400 seconds using SLAM and the data on system characteristics was output.

The model was then simulated using GVSim. While differences exist in the logical structure between GVSim and SLAM, restrictions are placed on GVSim so that it closely approximates the same control logic as SLAM. The model is entered as a series of points/nodes with unidirectional flow within segments except for the spur segment. The same input data such as speed and number of vehicles, location and length of segments, rate of part arrival, rate of machining operation, and location of vehicles in idle status are provided to GVSim. The startup conditions are kept the same as the SLAM example and simulation started at time 0 seconds. GVSim then simulates the model for the 89,400 seconds and outputs data on system performance.

5.2 Comparison

Finally, a comparison is made of the data on system performance output by both SLAM and GVSim. It is assumed that a difference of less than 1% between system statistics would be ideal in showing confidence in the model building capability of GVSim. Also a difference of more than 5% would indicate some discrepancy in the logic of GVSim.

Initially, a sample run is executed to obtain an indication of the performance of the model using SLAM and GVSim.

First, the vehicle utilization statistics of the two packages are compared. Vehicle utilization provides an indicator to the capability of modeling the system, primarily because vehicles are involved in all of the event notices. Any variation in model simulation would be instantly obvious on comparison with SLAM. However, some variation of the data is to be expected, because of the inherent differences in the control logic of the two packages. A substantial difference would lead to questioning the reliability of GVSim.

The average total vehicle utilization in carrying a load or travelling to pickup a load are identical for both models. This indicates, assuming reliability of the statistics output by SLAM, an accurate execution of simulation of the proposed model by GVSim.

Next, utilization of the machining cells in the model, run on the two packages, are compared. Both packages use a similar method for selection of machine for task performance. Hence, both the utilizations should be approximately equal. Here a difference of less than 1% is observed, which is within credibility limits.

These two key indicators provide the essential data for evaluation of the model. Their similarity for both the packages leads to confidence in the model simulation capability of GVSim. Finally, the model is run using different random generators to obtain an estimate of the variance and standard deviation of the values obtained. The results are presented in Figure 5.2 and Figure 5.3. The results obtained show that GVSim demonstrates approximately 1% higher machine utilization and 4% higher vehicle utilization. These values fall within credibility limits and may be attributed to GVSim's further optimization of vehicle travel.

	SLAM	GVSIM
Veh. travelling to load (empty)	.109	.103
Veh. travelling to unload (full)	.113	.119
Total Vehicle travelling to pick load/with load	.222	.222
Mach 1 average utilization	.91	.928
Mach 2 average utilization	.89	.887
Mach 3 average utilization	.87	.838
Mach 4 average utilization	.82	.894
Mach 5 average utilization	.78	.773
Mach 6 average utilization	.70	.687
Mean	.828	.834

Figure 5.1: Comparison of Model in a Sample Run

SLAM		GVSim	
Av. Machine Utilization	Veh. Travel to/with Load	Av. Machine Utilization	Veh. Travel to/with Load
.811	.211	.834	.222
.813	.220	.795	.235
.643	.174	.789	.238
.953	.255	.808	.244
.725	.198	.793	.229
.831	.223	.836	.220
.845	.227	.817	.236
.825	.216	.845	.204
.828	.223	.842	.202

Figure 5.2: Comparison of Mean Values for Different Runs

	SLAM		GVSim	
	Average Machine Utiliza-tion	Vehicle Travel to/with Load	Average Machine Utiliza-tion.	Vehicle Travel to/with Load
Mean	.809	.216	.817	.225
St. Dev.	.08497	.02199	.02216	.01488
Variance	.00722	.00048	.00049	.00022

Figure 5.3: Resultant Data for Model

Chapter VI

6. CONCLUSIONS AND RECOMMENDATIONS

The package developed as part of this research provides a means of analyzing various guided vehicle systems. It provides flexibility in modeling various systems and incorporates various standard features common to most simulators for AGVs. Additionally, it provides increased flexibility by modeling a larger class of system. It also provides a variation of the path routing logic present in most simulators. It extends the shortest path concept to develop the quickest path concept for better material flow and reduction of bottlenecks.

The most significant contribution presented in this program is found by the quickest path concept. The quickest path concept incorporates principles of selection of alternate paths and pre-removal of conflict. While the benefit of this may not be evident in the modeling of smaller models, it becomes apparent in the modeling of large systems. GVSim provides a different approach to flow of materials within a layout.

In context of the computer program, the model can be made more sophisticated. Due to its highly interactive yet

simplistic nature, the model is not able to account for incorrect or questionable values entered at many places. To be able to do that, the programming effort would have been increased significantly. So, keeping in mind the research objective of building a basic package to model various guided vehicle system, the ability of the model to handle such errors was restricted. However, the computer program has been developed in such a way as to allow easy incorporation of such enhancements. Also, GVSim does not attempt to provide a variety of modules for selecting different logic concepts. For instance, vehicle allocation is based on quickest arrival time. It might be desirable to provide different vehicle allocation protocols to evaluate their effect on system performance. A different protocol may be provided for conflict prevention. Another enhancement to the program may be the use of computer graphics to depict results.

Finally, GVSim is not intended to optimize the structure of the model. For a given set of parameters it attempts to produce near optimum system performance. It does not suggest improvements or modifications to the user. Instead the user must analyze the data output on system performance in leading to a better system. One feature which neither GVSim nor existing simulation platforms provide for is

dynamic rescheduling of vehicles. This might be addressed for future research to further optimize material flow within a layout.

Bibliography

1) Akari, T., T. Takahasi, M. Suekane and M. Kawai, "Flexible AGV system simulator", Proceedings of the 5th International Conference on AGVS, pp 77-86, 1987.

2) Almodovar, A. R., "Manufacturing Simulators Save Time, Provide Good Data for Layout Evaluation", Industrial Engineering, pp 28-33, June, 1988.

3) Andersson, M., "AGV system simulation-a planning tool for AGV route layout", Proceedings of the 3rd International Conference on AGVS, pp 291-295, 1985.

4) Beadle, R. B. and G. M. Hook, "Why you need to simulate your AGV networks", Proceedings of an Executive briefing on Automated Guided Vehicles, pp 65-69, 1986.

5) Bozer, Y. A. and Srinivasan, M. M., "Tandem Configurations For Automated Guided Vehicle Systems Offer Simplicity And Flexibility", Industrial Engineering, pp 23-27, February, 1989.

6) Brown, Evelyn, "Case Study: Using Different Types of Simulation to Design an IBM Facility", Industrial Engineering, pp 22-26, June, 1988.

7) Buda, J. and M. Badida, "Simulation of mobile systems for avoiding obstacles", Proceedings of the 3rd International Conference on AGVS, pp 279-290, 1985.

8) Buda, J., M. Badida and J. Vrlik, "Anticipation and simulation in workshop transport", Proceedings of the 4th International Conference on AGVS, pp 103-113, 1986.

9) Duffau, B. and C. Bardin, "Evaluating AGVS circuits by simulation", Proceedings of the 3rd International Conference on AGVS, pp 229-245, 1985.

10) Eade, Robert, "AGVs Make Their Move", Manufacturing Engineering, pp 53-55, September 1989.

11) Gunsser, P., "Control techniques in automatic guided vehicle systems. Examples of application and various control concepts", Proceedings of the 2nd International Conference on AGV and 16th IPA Conference, pp 85-113, 1983.

12) Hedman, A., "Automatic transport for heavy loads", Proceedings of the 3rd International Conference on AGVS, pp 363-368, 1985.

13) Horowitz, E. and S. Sahni, "Fundamentals of computer algorithms", Computer Science Press, 1984.

14) Jansson, U., "Design parameters and specification of an AGV system", Proceedings of an Executive briefing on Automated Guided Vehicles, pp 43-49, 1986.

15) Jonsson, S., "New AGV with revolutionary movement", Proceedings of the 3rd International Conference on AGVS, pp 135-144, 1985.

16) Klug, H., "Control Structure in Complex AGVS", Proceedings of the 2nd International Conference on AGV and 16th IPA Conference, pp 41-60, 1983.

17) Kuhn, A. and F. Schmidt, "General EDP-aided planning and realization of AGV-systems", Proceedings of the 3rd International Conference on AGVS, pp 247-257, 1985.

18) Kuhn, A. and U. Mienberg, "AGVS control concepts", Proceedings of the 4th International Conference on AGVS, pp 129-139, 1986.

19) Lasecki, R. R., "AGVs: The Latest in Material Handling Technology", CIM Technology, pp 90-94, Winter, 1986.

20) Law, Averill M. and S. Wali Haider, "Selecting Simulation Software for Manufacturing Applications: Practical Guidelines and Software Survey", Industrial Engineering, pp 33-46, May 1989.

21) Law, A. M. and McComas, M. G., "Pitfalls to Avoid in the Simulation of Manufacturing Sustems", Industrial Engineering, pp 28-31, May, 1989.

22) Lindsay, Bruce, "Warehouse Business System and 60 AGVs Track and Route Products at Kodak Distribution Center", Industrial Engineering, pp 50-54, May, 1988.

23) Miller, Richard K., Automated Guided Vehicles and Automated Manufacturing, Society of Manufacturing Engineers, 1987.

24) Norman, V. B., T. A. Norman and K. Farnsworth, "Rule-based simulation of AGV systems", Proceedings of the 5th International Conference on AGVS, pp 113-120, 1987.

25) Platts, R., "AGV system simulation-a tool for better design", Proceedings of an Executive briefing on Automated Guided Vehicles, pp 37-41, 1986.

26) Press, W. H., Flannery, B. P., Teukolsky, S. A. and W. T. Vetterling, Numerical Recipes in 'C': The Art of Scientific Programming, Cambridge University Press, Cambridge, 1988.

27) Pritsker, A. Alan B., Introduction to Simulation and SLAM II, John Wiley & Sons, Inc., 1986.

28) Putrus, R. S., "Layout design: key to advanced assembly/manufacturing success", Proceedings of the 4th International Conference on AGVS, pp 141-154, 1986.

29) Quinn, E. B., "A simulation based system for automatic development and testing of AGV control software," Proceedings of the 3rd International Conference on AGVS, pp 219-227, 1985.

30) Schulze, L. and K. D. Rosenbach, "Computer applications for the planning of AGV systems", Proceedings of the 5th International Conference on AGVS, pp 87-101, 1987.

31) Schmidt, F, "Rational approach for evaluating the number of AGVs", Proceedings of the 5th International Conference on AGVS, pp 103-112, 1987.

32) Schwind, Gene, "AGVS deliver more flexibility, easier programming", Material Handling Engineering, pp 57-64, October, 1987.

33) Schwind, G. F., "AGV'S Creative solutions go looking for problems", Material Handling Engineering, pp 44-47, September, 1988.

34) Takahashi, T. and T. Araki, "AGVS design using computer in factory automation", Proceedings of the 3rd International Conference on AGVS, pp 13-24, 1985.

35) Tracey, P. M., "AGV safety, new developments to meet changing needs", Proceedings of the 3rd International Conference on AGVS, pp 47-55, 1985.

36) Turner, D. H., "Manufacturing Simulation Comes of Age", CIM Technology, pp 16-19, Fall, 1986.

37) Walker, S. P., S. K. Premi, C. B. Besant, and A. J. Broadbent, "The Imperial College free-ranging AGV(ICAGV) and scheduling system", Proceedings of the 3rd International Conference on AGVS, pp 189-198, 1985.

38) Rygh, O. B., "New AGVS applications", Proceedings of the 3rd International Conference on AGVS, pp 357-361, 1985.

39) Witt, C. E., "Kodak Automates Worldwide Distribution Center", Material Handling Engineering, pp 66-74, November, 1988.

Appendix I: Sample Data Input

```
89400           simulation time

a1          name of point
0           X coordinate of point
0           Y coordinate of point
normal          type of point
a2          name of connecting point
DONE        indicates completion of data entry for this point

a2
35
0
machining
a3
DONE
0           indicates single non-machining destination point
f1          name of destination point
DONE
0
7           probability distribution function identifier
1
960         |
2720        |probability distribution function data
1860        |

a3
70
0
machining
a4
DONE
0
f1
DONE
0
7
1
960
2720
1860

a4
105
0
machining
a5
DONE
```

```
0
f1
DONE
0
7
1
960
2720
1860

a5
140
0
normal
b5
DONE

b1
0
40
normal
a1
b2
DONE

b2
35
40
machining
b3
DONE
0
f1
DONE
0
7
1
960
2720
1860

b3
70
40
machining
b4
DONE
0
f1
DONE
0
```

```
7
1
960
2720
1860

b4
105
40
machining
b5
DONE
0
f1
DONE
0
7
1
960
2720
1860

b5
140
40
normal
e3
DONE

c1
0
53
normal
b1
DONE

d1
0
69
normal
c1
d2
DONE

d2
25
61
parking
c1
DONE
```

```
e1
0
83
normal
d1
DONE

e2
68
83
normal
e1
f1
DONE

e3
140
83
normal
e2
DONE

f1
68
103
request
e2
DONE
1          indicates any machining point as destination
4          probability distribution function identifier
1          |
436        |probability distribution function data
FINISH     indicates end of point data
2          number of vehicles
45         load/unload times from/to vehicles
0          idle vehicles go to parking
0          no dedication of vehicles
```

Appendix II: Comparison of GVSim with Existing Simulators

A comparison between GVSim and some existing simulators available for modeling guided vehicle simulations is provided below. The features and limitations of these packages are also discussed.

AGVSim

AGVSim is a simulation package adapted for the design of AGVS developed by Dr. P.N. Egbleu as a Ph. D. thesis at Virginia Polytechnic Institute, in conjunction with J.M.A. Tanchoco. It is a FORTRAN based simulator that provides an evaluation tool for analyzing, planning, and designing Automatic Guided Vehicle Systems. It consists of two routines A and B, for executing the simulation of the proposed model. Routine A establishes the shortest paths and distances between every pair of points in the network and passes it on to Routine B. Only the case of unidirectional flow of vehicles is incorporated due to their predominance in the industry and their simplicity. A network is modeled as a collection of nodes and arcs with every node requiring at least one entering and one departing arc with a maximum of upto four arcs. A safety zone is constructed around each node which only one vehicle can

access. This is the primary method of conflict prevention between two vehicles.

Routine B, the main simulator, incorporates the following subsystems:

(a) the unit load assignment model which involves grouping of parts into transferable sizes or unit loads,

(b) the machining center model where, depending on the number of machines entered as contained in a center, it is single or multiple server queueing system. Unit loads are delivered and picked up from machining centers on job completion,

(c) the vehicle-unit load transport model which involves vehicle allocation and transportation within the system,

(d) the facility layout model which requires definition of the model, and

(e) the system operating policy w.r.t. decisions to enter and cross an intersection, holding to give right of way to other passing vehicles, or holding for a space to become available in the next succeeding arc.

Flexible AGV System Simulator

A flexible AGV system simulator was developed by Akari, Takahashi, Suekane and Kawai [1] which is representative of most simulators available in the market today. It is programmed in BASIC and FORTRAN. The simulator starts with the definition of the physical layout (by an arrangement of tiles consisting of lines and arcs) and entering of numerical data such as number of machines, process sequence, number and speed of AGVs, transfer time between the AGV and the station. The package then calculates the shortest path between the points between which the vehicle may travel, assuming unidirectional flow. The simulator then initializes the variables and increments time until end of execution. Vehicles are dispatched using FIFO and are managed using various principles such as prevention of interference between vehicles and only one vehicle may enter a single tile or stop at a particular workstation. Loads are generated from machining workcenters and modeled with increments of time. The results are then output for evaluation.

Matsim

Matsim is a software-tool with a module library consisting of three fields of functions namely, interactive input of

model parameters, simulation execution and data output, and result presentation. Within each function, modules exist at four levels. First, the operative level consists of determining vehicle movements, loading functions and positioning/location functions. This requires defining parameters such as vehicle velocity, running time, and location of machining centers. The next level is the operations control level consists of strategies of priority at crossings and route decisions. The dispositive level relates to the strategy for allocating tasks to vehicles. Finally the administrative level controls the production sequence, generation of tasks, and priorities of task.

AutoSimulations

Autosimulations USA provides a simulation modeling simulator for guided vehicle systems. AutoMod is the simulation language employed to model the system using GPSS/h as the simulator. It concentrates on building a model as close to the real system as possible to lend credence to the model developed. For blocking it applies a generic scheme which requires physical detection methods for avoiding collision between vehicles. The model produces a table consisting of the shortest courses available between each point. A major benefit of the package focuses on the vehicle dispatch and

scheduling algorithms. Unlike other simulators which use a FIFO dispatch scheme, Autosimulations uses priority of task and routing of vehicles to heavy activity areas to increase vehicle response time in answering pickup requests. A vehicle first scans the nearest points for a pending load. Finally the simulator provides for model validation by outputting three sets of tables to enable the user to evaluate the data and observe its correlation with existing or manually derived data.

Limitations of Existing Packages

Other simulators are in existence; however, their limited use or similarity to the above simulators has excluded them from this review. All of the above simulators provide a tool to guide the user toward a solution to their problem, but limit the design of the system. Two major drawbacks of the above simulators is the inability to select alternate paths and provide the option of bidirectional flow of vehicles. This in turn leads to a compromise in the design options available to the user, namely by necessating unidirectional flow, extended layouts and creating bottlenecks. This leads to a model, which, in some cases, is unable to accurately depict the system being modeled. One of the reasons for these limiting design features the

rigidity of the design of earlier existing systems present while developing these simulators.

GVSim - Comparison and Benefits

GVSim attempts to provide a package which does not incorporate the above limitations, and therefore provides the user flexibility in modeling a more generic class of guided vehicle systems. It presents a new approach in optimizing the path taken by a vehicle, called the quickest_path, which bases the path selection on the quickest arrival time. This is in direct contrast to the shortest path method used by simulators, which, with conflict, may result in a longer arrival time. (Shortest paths between points may overlap creating bottlenecks, and creating unnecessary blockage of vehicles.) GVSim also provides a means for pre-removal of conflict. It doesn't follow the rule of allocating paths and then removing conflicts as and when they occur but predetermines conflict occurrence and based on that selects the quickest_path. This results in a more efficient method of material transportation.

GVSim also considers a different approach for vehicle allocation. In contrast to the usual approach, vehicle allocation is not limited to idle vehicles. Busy vehicles are also considered for answering requests in anticipation of their becoming idle.

Appendix III : GVSim Code

```
#include <stdio.h>
#include <math.h>
#include "struct3.h"
#include "entcord3.c"
#include "sshortp3.c"
#include "intevls3.c"
#include "veh_all3.c"
#include "machal13.c"
#include "erequest.c"
#include "epickup.c"
#include "erelease.c"

struct ev_lst *dest_alloc(), *intevlst();
struct ev_lst *top, *last, *top_veh, *last_veh, *top_mach, *last_mach;
float tnow, tbeg, tend;
int veh_wait, no_veh, load_no, transfer_time;
extern int all_veh_busy;
float mach_busy[MACHINING][2]; /* [0] = total time busy, [1] = last time
busy */
float loading[VEHICLE_NO], unloading[VEHICLE_NO], tloading[VEHICLE_NO];
int all_mach_busy, load_no = 0, unload_no = 0;



main()
{
FILE *in;
struct ev_lst *i, *j, *temp1, *temp2;
int storepath;
float a, no;
int c, d, e, f, type, choice;
float empty_time, park_time;
float finpart_wait;
int k, from_no, to_no, y;
int x, dest, z;
float time, ttime;
char b[9];
int doit = NULL;
float total_busy_empty=0.0, total_busy_loaded=0.0, total_idle=0.0;
float total_parking=0.0, total_loading=0.0, total_unloading=0.0;
float av_total = 0.0;
init_number();
init_stat();
all_veh_busy = 0;
top = last = top_veh = last_veh = top_mach = last_mach = NULL;
tnow = 0.0;
for(x = 0; x < VEHICLE_NO; x++)
   {
   loading[x] = NULL; /* stores time each veh. is loading */
   unloading[x] = NULL; /* stores time each veh. is unloading */
   tloading[x] = NULL;
   }
for(x = 0; x < MACHINING; x++)
   mach_busy[x][0] = NULL;
printf("ENTER TEND: ENDING TIME OF SIMULATION\n");
scanf("%f", &tend);
while(getchar() != '\n');
```

```
entercord();
top = (struct ev_lst *) malloc(sizeof(struct ev_lst));
if (!top)
   {
   printf("OUT OF MEMORY FOR EVENT LIST\n");
   exit(1);
   }
top = intevlst(top);
/*printf("Returned from intevlst()\n");*/
tnow = top->time;
/* goto all decision points
   identify pickup points
   generate random initial random no. for each pickup point */
while(tnow < tend) /* While there is time remaining to perform
simulation. */
    {
    i = (struct ev_lst *) malloc(sizeof(struct ev_lst));
    if (!i)
       {
       printf("OUT OF MEMORY FOR EVENT LIST\n");
       exit(1);
       }
    a = top->time; /* time event notice will occur. */
    strcpy(b, top->type); /* type of event notice. */
    c = top->which; /* point at/for which event notice occurs. */
    d = top->from; /* where the vehicle is coming from */
    e = top->veh; /* which vehicle */
    f = top->dest;
/*printf("a = %f, b = %s, c = %d, d = %d, e = %d, f = %d\n", a, b, c, d,
e, f);*/
printf("tnow = %f, top->time = %f\n", tnow, top->time);
    if (!strcmp(b, "request"))
       type = 1;
    else if (!strcmp(b, "pickup"))
       type = 2;
    else if (!strcmp(b, "release"))
       type = 3;
    else if (!strcmp(b, "parking"))
       type = 4;
    else if (!strcmp(b, "charging"))
       type = 5;
    else
       printf("Error in type of point\n");
    switch (type)
          {       /* event type request */
          case 1: ev_request(top);
                  break;

                  /* event type pickup */
          case 2: ev_pickup(top);
                  break;


                  /* event type release */
          case 3: ev_release(top);
                  break;


                  /* event type parking ie idle */
          case 4: if (vehicle_no[e].status != 1)
```

```
                        {
                        vehicle_no[e].dest_time = INFINITY;
                        vehicle_no[e].status = 5;
                        stat.veh_idle[e][0] = tnow;
                        }
                    break; /* parking */

                    /* event type charging */
            case 5: vehicle_no[e].status = 4;
                    break; /* charging; */

            }
/*printf("Simrun: Deleting last event\n");*/
    temp1 = top;
    top = dldelete(top); /* returns new top after deleting old top */
    free (temp1);
    tnow = top->time;
/*printf("tnow = %f, top->time = %f\n", tnow, top->time);*/
    }

for(y = 0; y < MAX; y++)
   if (number[y]/* && strcmp(number[y]->type, "request")*/)
      printf("stat.pt_blocked[%d] = %f, Percentage = %f\n", y,
stat.pt_blocked[y], (stat.pt_blocked[y]/tend));

    y = 0;
    while(y < no_veh)
         {
         printf("stat.veh_busy_empty[%d] = %f, percentage = %f\n", y,
stat.veh_busy_empty[y], (stat.veh_busy_empty[y]/tend));

         printf("stat.veh_busy_loaded[%d] = %f, percentage = %f\n", y,
stat.veh_busy_loaded[y], (stat.veh_busy_loaded[y]/tend));

         if (vehicle_no[y].status == 5 || vehicle_no[y].status == 6)
            stat.veh_idle[y][1] += (tend - stat.veh_idle[y][0]);
         printf("stat.veh_idle[%d][1] = %f, percentage = %f\n", y,
stat.veh_idle[y][1], (stat.veh_idle[y][1]/tend));

printf("stat.veh_blocked[%d] = %f, percentage = %f\n", y,
stat.veh_blocked[y], (stat.veh_blocked[y]/tend));

printf("stat.veh_parking[%d] = %f, percentage = %f\n", y,
stat.veh_parking[y], stat.veh_parking[y]/tend);

printf("Loading[%d] = %f\n", y, loading[y]/tend);

printf("Unloading[%d] = %f\n", y, unloading[y]/tend);

         printf("total(%d) =
%f\n",y,stat.veh_busy_empty[y]+stat.veh_busy_loaded[y]+stat.veh_idle[y][
1]+stat.veh_parking[y]+loading[y]+unloading[y]);

total_busy_empty += stat.veh_busy_empty[y];
total_busy_loaded += stat.veh_busy_loaded[y];
total_idle += stat.veh_idle[y][1];
total_parking += stat.veh_parking[y];
total_loading += loading[y];
total_unloading+= unloading[y];
```

```
          y++;
          }

printf("Average_busy_empty = %f\n", total_busy_empty/(no_veh*tend));
printf("Average_busy_loaded = %f\n", total_busy_loaded/(no_veh*tend));
printf("Average_idle = %f\n", total_idle/(no_veh*tend));

printf("total_busy_empty = %f\n", total_busy_empty/(tend));
printf("total_busy_loaded = %f\n", total_busy_loaded/(tend));
printf("Loading = %f\n", loading[0]+loading[1]);
printf("Unloading = %f\n", unloading[0]+unloading[1]);
printf("total =
%f\n",total_busy_empty/(tend)+total_busy_loaded/(tend)+loading[0]+loadin
g[1]+unloading[0]+unloading[1]);

     y = 0;
     while(request_pt[y])
          {
          printf("Av. stat.raw_part_waiting = %f, For pt. = %d\n",
stat.raw_part_waiting[y][0]/stat.raw_part_waiting[y][1], request_pt[y]);
          y++;
          }

     y = 0;
     while(machining_pt[y][0])
          {
          printf("Av. stat.fin_part_waiting = %f, For pt. = %d\n",
stat.fin_part_waiting[y][0]/stat.fin_part_waiting[y][1],
machining_pt[y][0]);
          y++;
          }

     y = 0;
     av_total = 0.0;
     while(machining_pt[y][0])
          {
          if (machining_pt[y][1] == 1)
             mach_busy[y][0] += tend - mach_busy[y][1];
          printf("Av. time machine[%d]_busy = %f\n", machining_pt[y][0],
mach_busy[y][0]/tend);
          av_total += mach_busy[y][0]/tend;
          y++;
          }
          av_total = av_total/((float) y);
printf("Av. utilization of machines = %f\n", av_total);

printf("No. of loads = %d\n", load_no);
printf("No. of unloads = %d\n", unload_no);
printf("No. of times all_veh_busy = %d\n", all_veh_busy);
printf("No. of times all_mach_busy = %d\n", all_mach_busy);


if ((in = fopen("results", "w")) != NULL)
   {
   fprintf(in,"POINT DATA\n");

fprintf(in,"_____________________________________________________________
\n");
   fprintf(in,"NAME      TYPE       BLOCKED       PART       BUSY\n");
   fprintf(in,"                                  WAITING\n");
```

```
fprintf(in,"_____
\n");
   for(y = 0; y < MAX; y++)
   if (number[y])
      {
      fprintf(in,"%s    %9s      %2.3f          ",number[y]->name,number[y]-
>type,(stat.pt_blocked[y]/tend));
      if (!strcmp(number[y]->type, "request"))
         {
         z = 0;
         while(request_pt[z] != y)
              z++;

fprintf(in,"%2.3f\n",stat.raw_part_waiting[z][0]/(stat.raw_part_waiting[
z][1]*tend));
         }

      else if (!strcmp(number[y]->type, "machining"))
         {
         z = 0;
         while(machining_pt[z][0] != y)
              z++;
         fprintf(in,"%2.3f
%2.3f\n",stat.fin_part_waiting[z][0]/(stat.fin_part_waiting[z][1]*tend),
mach_busy[z][0]/tend);
         }
      else
         fprintf(in,"\n");


      }

fprintf(in,"_____
\n\n");

      fprintf(in,"Av. utilization of machines =             %f\n\n\n\n",
av_total);

      fprintf(in,"VEHICLE DATA\n");

fprintf(in,"_____
\n");
      fprintf(in,"VEH   DED  IDLE    BUSY    BUSY     PARKING   LOADING
UNLOADING\n");
      fprintf(in,"NO.                EMPTY   LOADED\n");

fprintf(in,"_____
\n");
      y = 0;
      while(y < no_veh)
             {
             fprintf(in,"%2d     %d   %2.3f   %2.3f   %2.3f    %2.3f
%2.3f
%2.3f\n",y,vehicle_no[y].ded[0],stat.veh_idle[y][1]/tend,stat.veh_busy_e
mpty[y]/tend,stat.veh_busy_loaded[y]/tend,stat.veh_parking[y]/tend,loadi
ng[y]/tend,unloading[y]/tend);
            y++;
            }
```

```
fprintf(in,"_____
\n");
                fprintf(in,"MEAN        %2.3f    %2.3f    %2.3f     %2.3f
%2.3f
%2.3f\n",total_idle/(tend*no_veh),total_busy_empty/(tend*no_veh),total_b
usy_loaded/(tend*no_veh),total_parking/(tend*no_veh),total_loading/(tend
*no_veh),total_unloading/(tend*no_veh));

                fprintf(in,"No. of loads = %d\n", load_no);
                fprintf(in,"No. of unloads = %d\n", unload_no);

                fclose(in);
    }
}
```

```
/* Assumption #1 Name of points in form of char_digit(s). */
/* Success of a simulation is based primarily upon how well */
/* the programmer understands the event being simulated.    */
/* This function contains the entercord function used to enter */
/* coordinates of decision points.                          */

#include <stdio.h>
#include <math.h>
#include <ctype.h>
#include "demand31.c"


double distance();
double dist3();
void store();  /* Stores name & (after cal. it) index no. in matrix. */
void store1();


/*int reverseflow;*/
extern int veh_wait, no_veh, transfer_time;
extern float tnow;

entercord()
{
int charge, chargetime, row = 0, q = 0, r = 0, s = 0, m = 0, x, y;
char response[9];
struct matrix *i;
char destname[3];
int j, ddest, cindex, vehicle;
int req, req_veh, ded_veh;
printf("HIT RETURN TO START INPUT\n");
gets(response);
while(strcmp(response, "FINISH"))
    {
    i = (struct matrix *) malloc(sizeof(struct matrix));
    if (!i) /* allocates memory for point being stored */
       {
       printf("Allocation failure\n");
       exit(1);
       }
       printf("ENTER NAME OF POINT\n");
       gets(i->name);
       if (*i->name)          /* *i->name instead of i->name since name
is */
          {                   /* a pointer and value at name is *i->name.
*/
          *i->name = toupper(*i->name); /* Converts to upper case. */
          }
       printf("ENTER X COORDINATE OF POINT\n");
       scanf("%f", &i->coord[0]);
       printf("ENTER Y COORDINATE OF POINT\n");
       scanf("%f", &i->coord[1]);
       while(getchar() != '\n')
          ;
       printf("ENTER TYPE OF POINT: OPTIONS NORMAL:, REQUEST:,
MACHINING:, CHARGING: & PARKING:\n");
       gets(i->type);
       while(strcmp(i->type, "normal") && strcmp(i->type, "request") &&
strcmp(i->type, "machining") && strcmp(i->type, "charging") && strcmp(i-
>type, "parking"))
```

```
            {
            printf("IMPROPER OPTION PLEASE TRY AGAIN\n");
            gets(i->type);
            }
       printf("ENTER THE NAMES OF CONNECTING POINTS, TYPE \"DONE\" TO
EXIT\n");
       gets(response);
       while(strcmp(response, "DONE"))
            {
            *response = toupper(*response); /* Converts to upper case. */
            strcpy(i->c_name[row], response);
            /*printf("ENTER X COORDINATE OF CONNECTING POINT\n");
            scanf("%f", &i->c_coord[row][0]);
            printf("ENTER Y COORDINATE OF CONNECTING POINT\n");
            scanf("%f", &i->c_coord[row][1]);
            while(getchar() != '\n')
                 ; *//* To clear buffer of newline characters. */
            i->c_dist[row] = distance(i, row);
            i->c_name[row + 1][0] = NULL; /* Puts next connecting pt. as
NULL */
            i->c_dist[row + 1] = NULL;    /* to mark last connec pt.
entered. */
            store(i, row);
            row += 1;
            printf("ENTER NAME, NEXT CONNECTING POINT, TYPE \"DONE\" TO
EXIT\n");
            gets(response);
            if (row == CONNECT)
               {
               printf("SORRY MAX NO OF CONNECTING PTS POSSIBLE
REACHED\n");
               strcpy(response, "DONE");
               }
            }
            row = 0;
            if (!strcmp(i->type, "request"))
               {
               request_pt[q] = index1(i->name);
/*printf("request_pt[q] = %d, request_pt[q+1] = %d\n", request_pt[q],
request_pt[q+1]);*/
               request_pt[q + 1] = NULL;
               demand(i);
               q++;
               }
            else if (!strcmp(i->type, "machining"))
               {
               machining_pt[m][0] = index1(i->name);
               machining_pt[m][1] = 0;
               machining_pt[m+1][0] = NULL;
               demand(i);
               /*printf("ENTER DESTINATION POINTS\n");
               gets(destname);
               *destname = toupper(*destname);
               ddest = index1(destname);
               i->dest[0] = ddest;*/
               m++;
               }
            else if (!strcmp(i->type, "charging"))
               {
               charging[r] = index1(i->name);
```

```
                charging[r + 1] = NULL;
                r++;
                }
            else if (!strcmp(i->type, "parking"))
                {
                parking[s] = index1(i->name);
                parking[s + 1] = NULL;
                s++;
                }
            else if (!strcmp(i->type, "normal"))
                ;
            else
                {
                printf("ERROR IN TYPE OF POINT STORAGE\n");
                }
                store1(i); /* don't need this, remove it later */
                printf("ENTER \"FINISH\" IF NO MORE POINTS, ELSE PRESS
RETURN\n");
                gets(response);
    }
    for(j = 0; j < MAX; j++)
    if (number[j])
        {
        row = 0;
        while(number[j]->c_index[row] && row < CONNECT)
             {
             cindex = number[j]->c_index[row];
             number[j]->c_dist[row] = dist3(j, cindex);
             row++;
             }
        }
        row = 0;
    printf("SPECIFY NO. OF VEHICLES: MAX = %d\n", VEHICLE_NO);
    scanf("%d", &no_veh);
    if (s <= no_veh) /* stores initial loc. of vehicles at parking pt's
*/
        for(x = 0, y = 0; x < no_veh; x++)
            {
            vehicle_no[x].dest_pt = parking[y];
            vehicle_no[x].status = 5;
            stat.veh_idle[x][0] = tnow;
/*printf("vehicle_no[%d].dest_pt = %d\n", x, parking[y]);*/
            if (parking[y + 1] == NULL)
                y = 0;
            else
                y++;
            }
    else /* if s > vehicles */
        for(x = 0, y = 0; x < no_veh; x++, y++)
            {
            vehicle_no[x].dest_pt = parking[y];
            vehicle_no[x].status = 5;
            stat.veh_idle[x][0] = tnow;
            }

    printf("DO YOU WANT VEHICLE TO WAIT AT DROP OFF POINT IF NO OTHER
REQUESTS\n");
    printf("OR GO TO PARKING 0 = GO TO PARKING, 1 = WAIT AT DROP OFF
POINT\n");
    scanf("%d", &veh_wait);
```

```
    while(getchar() != '\n')
         ;

    printf("ENTER LOAD TRANSFER TIME FROM AND TO VEHICLE\n");
    scanf("%d", &transfer_time);

    printf("DO YOU WISH TO HAVE DEDICATE VEH'S FOR PT'S: YES = 1, NO =
0\n");
    scanf("%d", &ded_veh);
    while(getchar() != '\n')
         ;
    if (!ded_veh)
       for(x=0; x<VEHICLE_NO; x++)
          vehicle_no[x].ded[0] = 0;
    while(ded_veh)
       {
       x = 0;
       printf("Enter number of vehicle which is to be dedicated\n");
       scanf("%d", vehicle);
       while(getchar() != '\n')
            ;
       vehicle_no[vehicle].ded[x] = 1; /* indicates this veh. is
dedicated */
       while(vehicle)
            {
            printf("Enter name of pt. which will have dedicated
vehicles\n");
            gets(destname);
            *destname = toupper(*destname);
            ddest = index1(destname);
            vehicle_no[vehicle].ded[x] = ddest;
            vehicle_no[vehicle].ded[x + 1] = NULL;
            printf("Enter next point this veh. is dedicated to else
enter 0\n");
            scanf("%d", vehicle);
            x++;
            while(getchar() != '\n')
            ;
            }
       }

       /*printf("Enter name of pt. which will have dedicated
vehicles\n");
       gets(destname);
       *destname = toupper(*destname);
       ddest = index1(destname);
       number[ddest]->ded = 1;
       req_veh = 0;
       x = 0;
       printf("Enter veh's which services pt %s\n", number[ddest]-
>name);
       if (!strcmp(number[ddest]->type, "request"))
          {
          while(ddest != request_pt[x] && x < REQUEST)
               x++;
          scanf("%d", &request_veh[x][req_veh]);
          req_veh++;
          while(req_veh < no_veh)
               {
```

```
                printf("Enter next veh which services pt
%s\n",number[ddest]->name);

                printf("Else enter 9999\n");
                scanf("%d", &request_veh[ddest][req_veh]);
                if (request_veh[ddest][req_veh] == 9999)
                   break;
                else
                   req_veh++;
                }
             }
          else
             {
             while(ddest != machining_pt[x][0] && x < MACHINING)
                x++;
             scanf("%d", &machining_pt[x][req_veh]);
             req_veh++;
             while(req_veh < no_veh)
                {
                printf("Enter next veh which services pt
%s\n",number[ddest]->name);

                printf("Else enter 9999\n");
                scanf("%d", &machining_pt[ddest][req_veh]);
                if (request_veh[ddest][req_veh] == 9999)
                   break;
                else
                   req_veh++;
                }
             }
          printf("DO YOU WISH TO HAVE DEDICATE VEH'S FOR PT'S: YES = 1, NO
= 0\n");
          scanf("%d", &ded_veh);
          }*/


      printf("DO YOU WISH TO PLACE CHARGING REQUESTS, 1: YES & 0: NO\n");
      scanf("%d", &charge);
      while(getchar() != '\n')
           ;
      if (charge == 1)
         {
         printf("ENTER TIME BETWEEN CHARGING OF VEHICLES\n");
         scanf("%d", &chargetime);
         while(getchar() != '\n')
              ;
         }
}


double distance(i, row)
int row;
struct matrix *i;
{
double dist1, dist;
double x1, x2, y1, y2, z;
x1 = i->coord[0];
y1 = i->coord[1];
x2 = i->c_coord[row][0];
y2 = i->c_coord[row][1];
```

```
if (x1 < x2)
   {
   z = x1;
   x1 = x2;
   x2 = z;
   }
if (y1 < y2)
   {
   z = y1;
   y1 = y2;
   y2 = z;
   }
/*printf("x1 = %f, y1 = %f, x2 = %f, y2 = %f\n", x1,y1,x2,y2);*/
dist1 = pow((x1 - x2), 2.0) + pow((y1 - y2), 2.0);
dist = sqrt(dist1);
/*printf("dist = %f\n", dist);*/
return dist;
}

double dist3(a, b)
int a, b;
{
double d;
double x1, x2, y1, y2, z;
x1 = number[a]->coord[0];
y1 = number[a]->coord[1];
x2 = number[b]->coord[0];
y2 = number[b]->coord[1];
if (x1 < x2)
   {
   z = x1;
   x1 = x2;
   x2 = z;
   }
if (y1 < y2)
   {
   z = y1;
   y1 = y2;
   y2 = z;
   }

d = sqrt(pow((x1 - x2), 2.0) + pow((y1 - y2), 2.0));
return(d);
}

/* Finds the index of number[] corresponding to name of element and
stores */
void store(i, row) /* 'row' indicates which connecting pt. */
struct matrix *i;
int row;
{
int loc, loc1;
char *p, *p1;

/* rows * 26 */
loc = (*(i->name) - 'A'); /* name is stored as character & the atoi fn.
*/
p = &(i->name[1]); /* converts alphabatical string part into integer
value */
```

```
loc += (atoi(p) - 1) * 26 + 1; /* Plus rows, alphabet part indicates
columns. */
if (loc > 2600) /* 'name' is type char *; hence i->name is an address;
*/
   {             /* hence *(i->name) is char value at that address.
*/
   printf("Cell out of bounds\n");
   return;
   }
loc1 = (*(i->c_name[row]) - 'A');
p1 = &(i->c_name[row][1]);
loc1 += (atoi(p1) - 1) * 26 + 1;
i->c_index[row] = loc1;
/*printf("i->name = %s, index no. of entered point = %d\n", i->name,
loc);
printf("i->c_name[row] = %s, i->c_index[row] = %d\n", i->c_name[row], i-
>c_index[row]);*/
number[loc] = i;
number[loc]->occup_time = NULL;
/*printf("number[loc]->name = %s\n", number[loc]->name);*/
}


index1(name) /* index() also defined in shortp4.c, demand1.c */
char *name;
{
int loc;
char *p;
loc = *name - 'A';
p = &name[1];
loc += (atoi(p) - 1) * 26 + 1;
return loc;
}


/* Finds the index of number[] corresponding to name of element and
stores */
void store1(i)
struct matrix *i;
{
int loc;
char *p;

/* rows * 26 */
loc = (*(i->name) - 'A'); /* name is stored as character & the atoi fn.
*/
p = &(i->name[1]); /* converts numerical string part into integer value
*/
loc += (atoi(p) - 1) * 26 + 1; /* Plus rows, alphabet part indicates
columns. */if (loc > 2600) /* 'name' is type char *; hence i->name is an
address; */
   {             /* hence *(i->name) is char value at that address.
*/
   printf("Cell out of bounds\n");
   return;
   }
number[loc] = i;
/*if (!strcmp(i->type, "request"))
```

```
printf("loc = %d, number[loc]->name = %s, number[loc]->rand->multiple =
%d, number[loc]->rand->ran_no[0] = %d\n", loc, number[loc]->name,
number[loc]->rand->multiple, number[loc]->rand->ran_no[0]);  */

}
```

Demand31.c

```
 /* demand() */
/* This program is called whenever a pickup point is entered.  It
prompts
the user for the load generator function for that point & destinations
load would have to go to from that point. */

#include <stdio.h>

demand(ematrix)
struct matrix *ematrix;
{
int part_exit;
ematrix->rand = (struct random *) malloc(sizeof(struct random));
if (!ematrix->rand)
       {
       printf("Allocation failure\n");
       exit(1);
       }
destination(ematrix);
}



destination(ematrix)
struct matrix *ematrix;
{
struct matrix *ttop;
int x, y, choice, ddest;
int all_mach, part_mach, separate;
char destname[3];
ttop = (struct matrix *) malloc(sizeof(struct matrix));
if (!ttop)
       {
       printf("Allocation failure\n");
       exit(1);
       }
x = 0;
printf("Do parts travel to any machining pt for machining, 0 = no; 1 =
yes\n");
scanf("%d", &all_mach);
while(getchar() != '\n')
     ;
if (all_mach)
   {
   ematrix->rand->multiple = 3;
   ddemand(ematrix);
   }
else
   {
   printf("ENTER DESTINATION POINTS\n");
   gets(destname);
   while(strcmp(destname, "DONE"))
        {
        *destname = toupper(*destname);
        ddest = index2(destname);
        ematrix->dest[x] = ddest;
        x++;
```

```
        printf("ENTER NEXT DESTINATION POINT, TYPE DONE IF NO MORE\n");
        gets(destname);
        }
        ematrix->dest[x + 1] = NULL;
        printf("Are dest pt's machining, 0 = no; 1 = yes\n");
        scanf("%d", &part_mach);
        if (!part_mach)
           {
           if (x == 1) /* if single exiting dest pt */
              {
              ddemand(ematrix);
              ematrix->rand->multiple = 0;
              }
           else
              {
              if (!strcmp(ematrix->type, "request"))
                 {
                 printf("Do you want parts to go alternatively = 0\n");
                 printf("or do parts have separate random no. generators
= 1\n");
                 scanf("%d", &separate);
                 while(getchar() != '\n')
                      ;
                 }
              else
                 separate = 0;
              if (separate)
                 {
                 ematrix->rand->multiple = 4;
                 ttop->rand = ematrix->rand;
                 for(y = 0; y < x; y++)/*sch asif from diff pt's but
pt's same*/                    {
                    ematrix->rand = ematrix->rand->next;
                    ematrix->rand = (struct random *)
malloc(sizeof(struct random));                if (!ematrix->rand)
                       {
                       printf("Allocation failure for ematrix->rand\n");
                       exit(1);
                       }
                    ddemand(ematrix);
                    }
                    ematrix->rand = ttop->rand; /* return to top of list
*/
                    free(ttop);
                 }
              else
                 {
                 ddemand(ematrix);
                 ematrix->rand->multiple = 1;
                 }
              }
           }
        else
           {
           ematrix->rand->multiple = 2;
           }
     }
}
```

```
ddemand(ematrix) /* user specifies random no generater and parameter
values */
struct matrix *ematrix;
{
int choice;
printf("SELECT RANDOM NO GENERATOR.\n");
printf("YOUR CHOICES ARE:\n");
printf("1: POISSON(N, MEAN), 2: UNIFORM(N, LO, HI), 3: RANDOM(MEAN)\n");
printf("4: EXPON(N, BETA), 5: NORMAL(N, MEAN, STDEV) 6: LOGNORMAL(N,
MEAN, STDEV)\n");
printf("7: TRIANG(N, LO, HI, MID)\n");
scanf("%d", &choice);
while(getchar() != '\n')
     ;
switch(choice)
      {
      case 1:  r_no1(ematrix);
               break;
      case 2:  r_no2(ematrix);
               break;
      case 3:  r_no3(ematrix);
               break;
      case 4:  r_no4(ematrix);
               break;
      case 5:  r_no5(ematrix);
               break;
      case 6:  r_no6(ematrix);
               break;
      case 7:  r_no7(ematrix);
               break;
      }
}



r_no1(ematrix) /* poisson(n, mean) */
/* returns as a floating-point number an integer value that is a
   random deviate drawn from a Poisson distribution of mean 'mean' */
struct matrix *ematrix;
{
int n, mean;
printf("ENTER N: RANDOM NUMBER STREAM\n");
scanf("%d", &n);
printf("ENTER MEAN: MEAN VALUE OF NUMBER\n");
scanf("%d", &mean);
while(getchar() != '\n')
     ;
ematrix->rand->ran_no[0] = 1;
ematrix->rand->ran_no[1] = n;
ematrix->rand->ran_no[2] = mean;
}



r_no2(ematrix) /* uniform(n, lo, hi) */
struct matrix *ematrix;
{
int n, lo, hi;
printf("ENTER N: RANDOM NUMBER STREAM\n");
scanf("%d", &n);
```

```
printf("ENTER LO: LOW VALUE OF NUMBER\n");
scanf("%d", &lo);
printf("ENTER HI: HIGH VALUE OF NUMBER\n");
scanf("%d", &hi);
while(getchar() != '\n')
     ;
ematrix->rand->ran_no[0] = 2;
ematrix->rand->ran_no[1] = n;
ematrix->rand->ran_no[2] = lo;
ematrix->rand->ran_no[3] = hi;
}


r_no3(ematrix)
struct matrix *ematrix;
{
int mean;
printf("ENTER MEAN VALUE OF LOAD GENERATION\n");
scanf("%d", &mean);
while(getchar() != '\n')
     ;
ematrix->rand->ran_no[0] = 3;
ematrix->rand->ran_no[1] = mean;
printf("ematrix->rand->ran_no[0] = %d\n", ematrix->rand->ran_no[0]);
printf("ematrix->rand->ran_no[1] = %d\n", ematrix->rand->ran_no[1]);
}


r_no4(ematrix)  /* expon(n, beta) */
struct matrix *ematrix;
/* returns an exponentially distributed, positive random deviate of unit
mean */{
int n, beta;
printf("ENTER N: RANDOM NUMBER STREAM\n");
scanf("%d", &n);
printf("ENTER BETA: BETA VALUE OF NUMBER\n");
scanf("%d", &beta);
while(getchar() != '\n')
     ;
ematrix->rand->ran_no[0] = 4;
ematrix->rand->ran_no[1] = n;
ematrix->rand->ran_no[2] = beta;
}


r_no5(ematrix) /* normal(n, mean, stdev) */
struct matrix *ematrix;
{
int n, mean, stdev;
printf("ENTER N: RANDOM NUMBER STREAM\n");
scanf("%d", &n);
printf("ENTER MEAN: MEAN VALUE OF NUMBER\n");
scanf("%d", &mean);
printf("ENTER STDEV: STANDARD DEVIATION\n");
scanf("%d", &stdev);
while(getchar() != '\n')
     ;
ematrix->rand->ran_no[0] = 5;
ematrix->rand->ran_no[1] = n;
ematrix->rand->ran_no[2] = mean;
```

```
ematrix->rand->ran_no[3] = stdev;
}



r_no6(ematrix) /* lognormal(n, mean, stdev) */
struct matrix *ematrix;
{
int n, mean, stdev;
printf("ENTER N: RANDOM NUMBER STREAM\n");
scanf("%d", &n);
printf("ENTER MEAN: MEAN VALUE OF NUMBER\n");
scanf("%d", &mean);
printf("ENTER STDEV: STANDARD DEVIATION\n");
scanf("%d", &stdev);
while(getchar() != '\n')
     ;
ematrix->rand->ran_no[0] = 6;
ematrix->rand->ran_no[1] = n;
ematrix->rand->ran_no[2] = mean;
ematrix->rand->ran_no[3] = stdev;
}



r_no7(ematrix) /* triang(n, lo, hi, mid) */
struct matrix *ematrix;
{
int n, lo, hi, mid;
printf("ENTER N: RANDOM NUMBER STREAM\n");
scanf("%d", &n);
printf("ENTER LO: LOW VALUE OF NUMBER\n");
scanf("%d", &lo);
printf("ENTER HI: HIGH VALUE OF NUMBER\n");
scanf("%d", &hi);
printf("ENTER MID: MID VALUE OF NUMBER\n");
scanf("%d", &mid);
while(getchar() != '\n')
     ;
ematrix->rand->ran_no[0] = 7;
ematrix->rand->ran_no[1] = n;
ematrix->rand->ran_no[2] = lo;
ematrix->rand->ran_no[3] = hi;
ematrix->rand->ran_no[4] = mid;
}



index2(name)
char *name;
{
int loc;
char *p;
loc = *name - 'A';
p = &name[1];
loc += (atoi(p) - 1) * 26 + 1;
return loc;
}
```

Intevls3.c

```
/* int_ev_lst() */
/* int_ev_lst() will initially load the first event notices for
   each pickup point and charging request for each vehicle.     */

#include <stdio.h>
#include "rand3.c"
#include "ev_lst3.c"

struct ev_lst *int_pick();
struct ev_lst *int_charge();

extern float tnow;

struct ev_lst *intevlst(top)
struct ev_lst *top;
{
/*printf("reached intevlst\n");*/
top = int_pick(top);
/*top = int_charge(top);*/
return top;
}


float select();


struct ev_lst *int_pick(top)
struct ev_lst *top;
{
struct ev_lst *i;
int x, j, index, choice, flag; /* no needs to be type float */
float no;
for(j = 0; j < REQUEST; j++)
if (request_pt[j])
   {
   index = request_pt[j];
   if (number[index]->rand->multiple <= 3)  /* go to idle m/c or single
dest */
      {
      choice = number[index]->rand->ran_no[0];
      no = select(choice, index);
      i = (struct ev_lst *) malloc(sizeof(struct ev_lst));
      if (!i) {
         printf("OUT OF MEMORY FOR EVENT LIST\n");
         exit(1);
         }
      i->time = tnow + no;
      strcpy(i->type, "request");
      i->which = index;
      i->dest = number[index]->dest[0];
      i->from = INFINITY; /* since it's not coming from anywhere */
      i->veh = INFINITY; /* since veh. is only allocated when event
occurs */
      number[index]->dest[MAX_DEST] = i->dest; /* stores dest.*/
      top = load(i, top);
      }
   else /* (number[c]->rand->multiple == 4) */
      {
```

```
      x = 0;
      flag = 1;
      while(number[index]->dest[x] && flag)
           {
           choice = number[index]->rand->ran_no[0];
           no = select(choice, index);
           i = (struct ev_lst *) malloc(sizeof(struct ev_lst));
           if (!i)
              {
              printf("OUT OF MEMORY FOR EVENT LIST\n");
              exit(1);
              }
           i->time = tnow + no;
           strcpy(i->type, "request");
           i->which = index;
           i->dest = number[index]->dest[x];
           i->from = NULL; /* since it's not coming from anywhere */
           i->veh = NULL; /* since vehicle allocated only when event
occurs */              top = load(i, top);
           if (number[index]->rand->next)
              number[index]->rand = number[index]->rand->next;
           else
              flag = 0;
           x++;
           }
      }
   }
   return top;
}

struct ev_lst *int_charge(top) /* this logic to be modified according to
*/
/* charging interval request user places and then program selects
closest &available */
struct ev_lst *top, *last;
{
struct ev_lst *i; /* charging point */
int j, index, choice;
float no;
for(j = 0; j < CHARGING && charging[j] != NULL; j++)
     {
     index = charging[j];
     choice = number[index]->rand->ran_no[0];
/*printf("choice = %d\n", choice);*/
     no = select(choice, index);
/*printf("no = %f\n", no);*/
     i = (struct ev_lst *) malloc(sizeof(struct ev_lst));
     if (!i)
        {
        printf("OUT OF MEMORY FOR EVENT LIST\n");
        exit(1);
        }
     i->time = tnow + no;
     strcpy(i->type, "charge");
     i->veh = 0; /* totally wrong */
     i->from = 0; /* totally wrong */
     i->which = index;
     /*number[index]->dest[MAX_DEST] = number[index]->dest[0];*/
     top = load(i, top);
     }
```

```
     return top;
}

float select(choice, index)
int choice, index;
{
int n, mean, lo, hi, mid, beta, m, stdev;
float no, value;
switch (choice)
       {
case 1: n = number[index]->rand->ran_no[1];
                mean = number[index]->rand->ran_no[2];
                no = poisson(n, mean);
/*printf("1: no = %d\n", (int) no);*/
                break;

       case 2: n = number[index]->rand->ran_no[1];
                lo = number[index]->rand->ran_no[2];
                hi = number[index]->rand->ran_no[3];
                no = uniform(n, lo, hi);
/*printf("2: no = %d\n", (int) no);*/
                break;

       case 3: mean = number[index]->rand->ran_no[1];
                value = random(); /* varies between 0 and 1 */
                value += .5;
                /* to make value fluctuate bet. (1.0+x) & (1.0-x) * mean
do */
                /* while(!((value) > (1.0-x)) && !((value) < (1.0+x)))
*/
                /*      value = random();
*/
                /* where x is a floating point no.
*/
/*printf("value = %f, mean = %d\n", value, mean);*/
                value *= (float) mean;
                no = value;
/*printf("no = %f\n", no);*/
                break;

       case 4: n = number[index]->rand->ran_no[1];
                beta = number[index]->rand->ran_no[2];
                no = expon(n, beta);
                while((int) no < 100 || (int) no > 800)
                     no = expon(n, beta);
/*printf("4: no = %f\n", no);*/
                break;


       case 5: n = number[index]->rand->ran_no[1];
                mean = number[index]->rand->ran_no[2];
                stdev = number[index]->rand->ran_no[3];
                no = normal(n, mean, stdev);
/*printf("5: no = %f\n", no);*/
                break;

       case 6: n = number[index]->rand->ran_no[1];
                mean = number[index]->rand->ran_no[2];
                stdev = number[index]->rand->ran_no[3];
                no = lognormal(n, mean, stdev);
```

```
/*printf("6: no = %f\n", no);*/
                break;

        case 7: n = number[index]->rand->ran_no[1];
                lo = number[index]->rand->ran_no[2];
                hi = number[index]->rand->ran_no[3];
                mid = number[index]->rand->ran_no[4];
                no = triang(n, lo, hi, mid);
/*printf("7: no = %f\n", no);*/
                break;
        }
        return no;
}
```

Erequest.c

```
#include <stdio.h>
/*#include "intevls3.c"
#include "veh_all3.c"
#include "machall3.c"*/


ev_request(top)
struct ev_lst *top;
{
int choice, dest, doit, x, k, c, d, e, f;
float no, a;
char b[9];
struct ev_lst *i;

i = (struct ev_lst *) malloc(sizeof(struct ev_lst));
if (!i)
   {
   printf("OUT OF MEMORY FOR EVENT LIST\n");
   exit(1);
   }
a = top->time; /* time event notice will occur. */
strcpy(b, top->type); /* type of event notice. */
c = top->which; /* point at/for which event notice occurs. */
d = top->from; /* where the vehicle is coming from */
e = top->veh; /* which vehicle */
f = top->dest;

if (!strcmp(number[c]->type, "request"))
   {                  /* loads next occur. of same event*/
   if (number[c]->rand->multiple <= 3) /* if same rand gen.*/
{
      choice = number[c]->rand->ran_no[0];/*which rand gen.*/
no = select(choice, c); /* gets random no. */
      }
   else /*if multiple dest pts with seperate random no gen.*/
{
      k = 0;
      while(f != number[c]->dest[k]) /* k < x)*/
           {          /* to allign dest pt & random no data */
number[c]->rand = number[c]->rand->next;
           k++;
           }
           choice = number[c]->rand->ran_no[0];
           while(k) /* reset list to top */
                {
                number[c]->rand = number[c]->rand->prior;
                k--;
                }
                no = select(choice, c);
      }
      i->time = tnow + no;
      strcpy(i->type, "request");
      i->which =  c;
      i->dest = INFINITY;
      i->from = INFINITY;
      i->veh = INFINITY;
      top = load(i, top);
   }
```

```
   doit = 1;
   if (number[c]->rand->multiple == 2 || number[c]->rand->multiple == 3)
/* if dest pt. is a machining pt. */
      {
      dest = mach_alloc(c);
      if (dest == INFINITY) /* if unable to alloc m/c */
         doit = NULL;
/*printf("checkdest = %d\n", dest);*/
         }
      else if (number[c]->rand->multiple != 1)
         {
         dest = f;
         }
      else /* number[c]->rand->multiple == 1 */
         {
         x = number[c]->dest[MAX_DEST];
         if (x == MAX_DEST - 1)
            {
            i->dest = number[d]->dest[0];
            number[c]->dest[MAX_DEST] = 0;
            }
         else
            {
            i->dest = number[d]->dest[x + 1];
            number[c]->dest[MAX_DEST] = x + 1;
            }
         }

         if (doit)
            {
            veh_alloc(c, dest); /* allocs arrival of veh. at req pt */
            /*printf("Returned from veh_alloc()\n");*/
            }
}
```

Epickup.c

```
#include <stdio.h>

ev_pickup(top)
struct ev_lst *top;
{
int x, c, d, e, f, from_no, to_no, storepath;
char b[9];
float a, loaded_time;
struct ev_lst *i;
extern int load_no;
extern float tloading[], unloading[];

i = (struct ev_lst *) malloc(sizeof(struct ev_lst));
if (!i)
   {
   printf("OUT OF MEMORY FOR EVENT LIST\n");
   exit(1);
   }
a = top->time; /* time event notice will occur. */
strcpy(b, top->type); /* type of event notice. */
c = top->which; /* point at/for which event notice occurs. */
d = top->from; /* where the vehicle is coming from */
e = top->veh; /* which vehicle */
f = top->dest;

load_no += 1;
tloading[e] += transfer_time;
if (!strcmp(number[c]->type, "request"))
   {
   i->which = f;
   }
else /* if type of pt. == machining */
   {
   x = 0;
   while(machining_pt[x][0] != c)
        x++;
   machining_pt[x][1] = 0; /* m/c is now free */
   mach_busy[x][0] += tnow - mach_busy[x][1];
/*printf("mach_busy[%d][0] = %f, mach_busy[%d][1] = %f, tnow = %f\n", x,
mach_busy[x][0], x, mach_busy[x][1], tnow);*/
   if (top_mach) /* if veh waiting for idle m/c */
      {
/*printf("Allocating free machine\n");*/
      machining_pt[x][1] = 1; /* m/c is now busy */
      mach_busy[x][1] = tnow;
      veh_alloc(top_mach->which, machining_pt[x][0]);
      top_mach = delete_mach(top_mach);
      }
      i->which = number[c]->dest[0]; /* allocs dest pt */
   }

   /* schedule event "release" for this veh. picking up part */
   vehicle_no[e].status = 2;
   vehicle_no[e].dest_pt = i->which;
   strcpy(i->type, "release");
   i->from = c;
   i->veh = e;
   i->dest = INFINITY;
```

```
   from_no = c;
   to_no = i->which;
   storepath = 1;
/*printf("event pickup: from_no = %d, to_no = %d\n", from_no, to_no);*/
   i->time = sshortp(from_no, to_no, storepath, e);
/*printf("pickup: returned from sshortp\n");*/
   loaded_time = (i->time - transfer_time) - tnow;
   if ((i->time - transfer_time) > tend)/*minus excess time*/
loaded_time -= ((i->time - transfer_time) - tend);
   else
      {
      unloading[e] += transfer_time;
      if (i->time > tend)
         unloading[e] -= i->time - tend;
      }
      stat.veh_busy_loaded[e] += loaded_time;
   /*printf("stat.veh_busy_loaded[%d] = %f\n", e,
stat.veh_busy_loaded[e]);*/
      vehicle_no[e].dest_time = i->time;
      top = load(i, top);
}
```

Erelease.c

```
#include <stdio.h>

ev_release(top)
struct ev_lst *top;
{
int c, d, e, f, k, y, choice, from_no, to_no, doit, dest, storepath;
char b[9];
float a, no, park_time, finpart_wait;
extern int unload_no;
struct ev_lst *i, *j;

i = (struct ev_lst *) malloc(sizeof(struct ev_lst));
if (!i)
   {
   printf("OUT OF MEMORY FOR EVENT LIST\n");
   exit(1);
   }
a = top->time; /* time event notice will occur. */
strcpy(b, top->type); /* type of event notice. */
c = top->which; /* point at/for which event notice occurs. */
d = top->from; /* where the vehicle is coming from */
e = top->veh; /* which vehicle */
f = top->dest;

unload_no += 1;
from_no = c;
if (vehicle_no[e].status != 1) /* if not already allocated */
{
   if (top_veh)
      {
      doit = 1;
      if (!strcmp(number[top_veh->which]->type, "request"))
         {
         dest = mach_alloc(top_veh->which);
         if (dest == INFINITY)
            doit = NULL;
         }

/*printf("Allocating free vehicle\n");*/
         vehicle_no[e].status = 5;
         stat.veh_idle[e][0] = tnow;
         if (doit)
            {
            veh_alloc(top_veh->which, dest);
            finpart_wait = tnow - top_veh->time;
            y = 0;
            while(machining_pt[y][0] != c)
                 y++;
            stat.fin_part_waiting[y][0] += (finpart_wait -
stat.fin_part_waiting[y][0])/(stat.fin_part_waiting[y][1] + 1.0);/* this
looks fucked up */
            stat.fin_part_waiting[y][1] += 1.0;
            top_veh = delete_veh(top_veh);
            }
         }
      else if (veh_wait == 0) /* if go to parking option */
         {
```

```
         to_no = parking[0]; /* need to code parking selection*/
vehicle_no[e].status = 3;
         storepath = 1;
         j = (struct ev_lst *) malloc(sizeof(struct ev_lst));
         if (!j)
            {
            printf("OUT OF MEMORY FOR EVENT LIST\n");
            exit(1);
            }
         j->time = sshortp(from_no, to_no, storepath, e);
         vehicle_no[e].dest_time = j->time;
         vehicle_no[e].dest_pt = to_no;
         j->which = to_no;
         strcpy(j->type, "parking");
         j->from = c;
         j->veh = e;
         j->dest = INFINITY;
         top = load(j, top);
/*printf("j->time = %f, tnow = %f\n", j->time, tnow);*/
         park_time = (j->time - tnow);
         if (j->time > tend)
            park_time -= (j->time - tend);
         stat.veh_parking[e] += park_time;
/*printf("stat.veh_parking[%d] = %f\n", e, stat.veh_parking[e]);*/
         }
      else /* wait at drop off pt.*/
         {
         vehicle_no[e].dest_time = INFINITY;
         vehicle_no[e].status = 6;
         vehicle_no[e].dest_pt = c;
         stat.veh_idle[e][0] = tnow;
         }
      }
   else /* if vehicle_no[e].status = 1 */
      ;

/* schedule a request event */
      if (!strcmp(number[c]->type, "machining"))
         {
         if (number[c]->rand->multiple <= 1) /* if same rand gen.*/
{
            choice = number[c]->rand->ran_no[0];/*which rand gen.*/
no = select(choice, c); /* gets random no. */
            }
         else /*if multiple dest pts with seperate random no gen.*/
{
            k = 0;
            while(f != number[c]->dest[k]) /* k < x)*/
                 {         /* to allign dest pt & random no data */
number[c]->rand = number[c]->rand->next;
                 k++;
                 }
                 choice = number[c]->rand->ran_no[0];
                 while(k) /* reset list to top */
                      {
                      number[c]->rand = number[c]->rand->prior;
                      k--;
                      }
                      no = select(choice, c);
            }
```

```
            strcpy(i->type, "request");
            i->time = tnow + no;
            i->which = c;
            i->from = INFINITY;
            i->veh = INFINITY;
            i->dest = INFINITY; /*number[c]->dest[0];*/
            top = load(i, top);
        }
}
```

Sshortp3.c

```
/* sshortp() ties in shortp(), time() & conflict() to give the earliest
   conflict free, short path. */

#include <stdio.h>
#include "shortp3.c"
#include "time3.c"
#include "conflct3.c"


/* whatever is declared below is not accessible to above files */

float sshortp(from, to, storepath, veh_i_d)
int from, to, storepath, veh_i_d;
{
float time, delay;
int i, j, k, path_no, flag;
float t_time[2]; /* t_time[0] == path_no, t_time[1] == time reaching
dest. */
float dest_time[PATH_NO][2]; /* [i][0] == time before removing conflict
*/
                             /* [i][1] == arrival time after removing
conflict */
/*printf("reached sshortp.c\n");*/
if (from == to)
   {
   return tnow; /* return time to arrive as instantaneous */
   }
t_time[0] = 0.0;
t_time[1] = 0.0;
for(i = 1; i < PATH_NO; i++) /* 'i' indicates path number */
   {
/*if (i > 1)
printf("Path_no = %d\n", i);*/
   path_no = i;
   **pfinal = shortp(pfinal, from, to, path_no);
   **pfinal = alloc_time(pfinal, path_no);
   k = 0;
   while(pfinal[path_no][k] && k < SEGM_NO)
        k++;
   --k; /* resets to last k */
   dest_time[path_no][0] = pfinal[path_no][k]->dtime;/* time before
conflict removal */
   if (path_no > 1 && dest_time[path_no][0] > dest_time[path_no-1][1])
      {  /* 2nd time around if time before conflict removal in this
round */
         /* is more than time after conflict removal in last round.
*/
      time = dest_time[path_no-1][1];
      if (storepath) /* if path occup_time's are meant to be stored */
         {
         store_time(path_no-1);
         for(j = 0; pfinal[path_no-1][j]; j++)
            {
/*printf("2: Pfinal[%d][%d]->atime = %f, Pfinal[%d][%d]->dtime = %f\n",
path_no, j, pfinal[path_no][j]->atime, path_no, j, pfinal[path_no][j]-
>dtime);*/
            if (pfinal[path_no-1][j]->atime != pfinal[path_no-1][j]-
>dtime)
```

```
                {
                delay = (pfinal[path_no-1][j]->dtime - pfinal[path_no-
1][j]->atime);
                stat.veh_blocked[veh_i_d] += delay;
                stat.pt_blocked[pfinal[path_no-1][j]->point] += delay;
                }
            }
         }
         while(path_no)
              {
              flag = 1;
              for(j = 0; j <= k; j++)
                 {
                 if (pfinal[path_no][j] && flag)
                    {
                 /*printf("sshortp1: pfinal[%d][%d]->point = %d\n",
path_no, j, pfinal[path_no][j]->point);*/
                 free(pfinal[path_no][j]);
                    }
                 else
                    flag = NULL;
                 }
                 path_no--;
              }
/*printf("0: Returning from sshortp, tnow = %f, time = %f\n", tnow,
time);*/
              return time;
      }
/*printf("sshortp: veh_i_d = %d\n", veh_i_d);*/
   **pfinal = conflict(pfinal, path_no, veh_i_d);
/*printf("returned from conflct()\n");*/
   dest_time[path_no][1] = pfinal[path_no][k]->dtime; /* time after
conflict */
/*printf("dest_time[%d][0] = %f, dest_time[%d][1] = %f\n", path_no,
dest_time[path_no][0], path_no, dest_time[path_no][1]);*/
   if (dest_time[path_no][1] == dest_time[path_no][0]) /* if no conflict
*/
      {
      if (storepath)
         {
         store_time(path_no);
         }
/*printf("sshortp: pfinal[%d][%d]->dtime = %f\n",path_no, k,
pfinal[path_no][k]->dtime);*/
      time = pfinal[path_no][k]->dtime;
      while(path_no)
           {
           flag = 1;
           for(j = 0; j <= k; j++)
              {
              if (pfinal[path_no][j] && flag)
                 {
                 /*printf("sshortp1: pfinal[%d][%d]->point = %d\n",
path_no, j, pfinal[path_no][j]->point);*/
                 free(pfinal[path_no][j]);
                 }
              else
                 flag = NULL;
              }
              path_no--;
```

```
            }
/*printf("1: Returning from sshortp, tnow = %f, time = %f\n", tnow,
time);*/
      return time;
      }
   if (!((int) t_time[0]) || (dest_time[path_no][1] < t_time[1])) /*
stores earliest*/
      {           /* arrival at dest. time for all the paths being
calculated */
      t_time[0] = (float) path_no;
      t_time[1] = dest_time[path_no][1];
/*printf("path_no = %d, t_time[0] = %f, t_time[1] = %f\n",
path_no,t_time[0],t_time[1]);*/
      }
   }
   path_no = (int) t_time[0];
   if (storepath)
      {
      store_time(path_no);
      for(j = 0; pfinal[path_no][j]; j++)
         {
/*printf("2: Pfinal[%d][%d]->atime = %f, Pfinal[%d][%d]->dtime = %f\n",
path_no, j, pfinal[path_no][j]->atime, path_no, j, pfinal[path_no][j]-
>dtime);*/
         if (pfinal[path_no][j]->atime != pfinal[path_no][j]->dtime)
            {
            delay = (pfinal[path_no][j]->dtime - pfinal[path_no][j]-
>atime);
            stat.veh_blocked[veh_i_d] += delay;
            stat.pt_blocked[pfinal[path_no][j]->point] += delay;
            }
         }
      }
   time = t_time[1];
/*printf("path_no = %d, time = %f, t_time[0] = %f\n", path_no, time,
t_time[0]);*/
   while(path_no)
        {
        flag = 1;
        for(j = 0; j <= k; j++)
           {
           if (pfinal[path_no][j] && flag)
              {
              /*printf("sshortp1: pfinal[%d][%d]->point = %d\n",
path_no, j, pfinal[path_no][j]->point);*/
              free(pfinal[path_no][j]);
              }
           else
              flag = NULL;
           }
           path_no--;
        }
/*printf("2: Returning from sshortp, tnow = %f, time = %f\n", tnow,
time);*/
   return time;
}


store_time(path_no)
int path_no;
```

```
{
int x, pt_a, flag, atime, dtime;
int occup_begin, occup_end;
struct time *p;
/*printf("Reached store_time\n");*/
for(x=1; x+1 < SEGM_NO; x++) /* the rest of the code stores */
   {                         /* occup times in master list  */
   if (pfinal[path_no][x+1])
      {
      flag = 1;
      pt_a = pfinal[path_no][x]->point; /* for each x it goes thru the
foll. */
      while(flag)
           {
           flag = 0;
           atime = pfinal[path_no][x-1]->dtime; /* dept. time from prev
pt */
           dtime = pfinal[path_no][x+1]->atime; /* arrival time at next
pt */
           if (!number[pt_a]->occup_time) /* if there is no occup_time
*/
              {                    /* then store at top of occup_time
list */
/*printf("Reached !number[pt_a]->occup_time\n");*/
              p = (struct time *) malloc(sizeof(struct time));
              if (!p)
                 {
                 printf("Memory allocation failure for struct time:
sshortp\n");
                 exit(1);
                 } /* occupation of pt. x is from time veh. leaves last
pt to */
              p->occup_begin = pfinal[path_no][x-1]->dtime;/* time it
reaches */
              p->occup_end = pfinal[path_no][x+1]->atime;  /* next pt.
*/
              p->prior = NULL;
              p->next = NULL;
              number[pt_a]->occup_time = p;
              }
           else /* if (number[pt_a]->occup_time) */
              {
              occup_begin = number[pt_a]->occup_time->occup_begin;
              occup_end = number[pt_a]->occup_time->occup_end;
              if (dtime < occup_begin) /* if new times before 1st
occup_time */
                 {                     /* insert pointer at top */
                 p = (struct time *) malloc(sizeof(struct time));
                 if (!p)
                    {
                    printf("Memory alloc failure for struct time:
sshortp\n");
                    exit(1);
                    }
                 p->occup_begin = pfinal[path_no][x-1]->dtime;
                 p->occup_end = pfinal[path_no][x+1]->atime;
                 p->prior = NULL;
                 p->next = number[pt_a]->occup_time;
                 number[pt_a]->occup_time->prior = p;
                 }
```

```
                else if (dtime == occup_begin)
                   {
                   number[pt_a]->occup_time->occup_begin = atime;
                   }
                else /* dtime > occup_begin */
                   {
                   if (atime < occup_begin) /* new time overlaps 1st
occup_time*/
                      {
                      number[pt_a]->occup_time->occup_begin =
pfinal[path_no][x-1]->dtime;
                      if (dtime > occup_end)
                         {
                         number[pt_a]->occup_time->occup_end =
pfinal[path_no][x+1]->dtime;
                         reset(pt_a, number[pt_a]->occup_time->occup_end);
                         }                    /* need to pass variables */
                      else /* if (dtime <= occup_end) */
                         ; /* do nothing                */
                      }
                   else /* atime >= occup_begin */
                      {
                      if (dtime <= occup_end) /* if newtime within
occup_time */
                         ;                    /* do nothing
*/
                      else if(atime <= occup_end)
                         {
                         number[pt_a]->occup_time->occup_end =
pfinal[path_no][x+1]->dtime;
                         reset(pt_a, number[pt_a]->occup_time->occup_end);
                         }
                      else /* atime > occup_end */
                         {
                         if (number[pt_a]->occup_time->next) /* if next do
again*/
                            {/* since new time is not around present
occup_time */
                            flag = 1;
                            number[pt_a]->occup_time = number[pt_a]-
>occup_time->next;
                            }
                         else /* if no occup_time->next */
                            {
                            p = (struct time *) malloc(sizeof(struct
time));
                            if (!p)
                               {
                               printf("sshortp:Mem alloc failure, struct
time\n");
                               exit(1);
                               }
                            p->occup_begin = pfinal[path_no][x-1]->dtime;
                            p->occup_end = pfinal[path_no][x+1]->atime;
                            p->prior = number[pt_a]->occup_time;
                            number[pt_a]->occup_time->next = p;
                            p->next = NULL;
                            }
                         }
                      }
```

```
                    }
                }
            }
        }
    else
        break; /* to break out of for loop, careful unorthodox break */
    }
}


reset(a, dtime) /* removes all in bet. occup_time's before new occup_end
*/
int a;
float dtime;
{
struct time *y, *z;
/*printf("Reached reset: \n");*/
y = number[a]->occup_time; /* to remember top of list */

if (number[a]->occup_time->next && dtime >= number[a]->occup_time->next-
>occup_begin)
    {
    number[a]->occup_time = number[a]->occup_time->next;
    while(number[a]->occup_time->next && dtime >= number[a]->occup_time-
>next->occup_begin)
        {                   /* while occup_end is more than next
occup_begin */
        z = number[a]->occup_time;
        number[a]->occup_time = number[a]->occup_time->next;
        number[a]->occup_time->prior = z->prior;
        z->prior->next = number[a]->occup_time;
        free(z);
        }

    if (number[a]->occup_time) /* if not more than occup_end of next
occup_time*/
        {
        z = number[a]->occup_time;
        if (number[a]->occup_time->occup_end > dtime)
            y->occup_end = number[a]->occup_time->occup_end;
        if (number[a]->occup_time->next)
            {
            y->next = number[a]->occup_time->next;
            number[a]->occup_time->next->prior = y;
            }
        else /* if all next fall before dtime or don't exist */
            y->next = NULL;
        number[a]->occup_time = y; /* status unchanged */
        free(z);
        }
    else /* if !number[a]->occup_time ie. no overlap before new occup_end
*/
        ;
    }
/*printf("Returning from reset: \n");*/
}
```

Shortp3.c

```
/* shortp4.c takes a 'from' and 'to' point as input and calculates the
*/
/* shortest dist. between them.


/* Algorithm for shortest path between two vertices. */
/* for(i = 1; i <=; i++)                             */
/*    Aij = 0                                        */
/* for(k =1; k <= n; i++)                            */
/*    for(i = 1; i <= n; i++)                        */
/*       for(j = 1; j <= n; j++)                     */
/*          Aij = min(Aij, Aik + Akj)                */


#include <stdio.h>


/*struct mapname
       {
       int map; /* What is map, do I need it. *//*
       char *name; /* Name of decision point. *//*
       int visited;/* Indicates whether point has been visited (=0) or
not. *//*
       int sdist[PATH_NO]; /* For shortest dist between from_no & this
pt., *//*
       } *agvs[MAX];       /* for a particular shortp path == PATH_NO
*//*

struct matrix                       /* Material Flow Matrix. *//*
            {
            char name[3];           /* Name of decision pt. eg, A1, B34
*//*
            float coord[2];           /* Stores x,y coordinates of point
*//*
            char type[9];           /* Type of decision pt. *//*
            int dest[MAX_DESTT]; /* points it will travel to from this
pt. *//*
            struct random *rand; /* Stores random no. linked list & data
*//*
            float c_dist[CONNECT]; /* array showing dist of connecting
pts *//*
            int c_index[CONNECT];  /* array showing index of connecting
pt.*//*
            float c_coord[CONNECT][2];/* array showing coord of connec
pt. *//*
            char c_name[CONNECT][3];/* array showing name of connecting
pt. *//*
            int visited;
            } *number[MAX];

struct path    /* To store data on shortest path calculations. This *//*
          {    /* may not necessarily be the final shortest path.   *//*
          int point;    /* Stores index number of point. *//*
          int vel;      /* Velocity at that decision point. *//*
          float atime;  /* Arrival time at that decision point. *//*
          float dtime;  /* Departure time from that decision point. *//*
          } *pfinal[PATH_NO][SEGM_NO]; /* From time.c */
```

```
/*struct nodeparent /* Used for storing ancestors & descendants on a
path. *//*
        {
        int parent[PATH_NO];
        } spath[MAX];*/


int top_of_stack;
int laststop;
int pathlist[PATH_NO][SEGM_NO];
void push();
extern struct ev_lst *top;



struct path *shortp(pfinal, from_no, to_no, path_no)
struct path *pfinal[][SEGM_NO];
int from_no, to_no;
int path_no;
{
struct path *p;
int x = 0, y;
int temp_path_no;
int i, j, k, l, m, n, q;
int nnode, dist1, flag;
int counter = 0, do_rest;
float dist;
void pop();
top_of_stack = 0; /* Initialized to 0 every time generate() fn. is
called. */
/*printf("reached shortp4.c\n");*/
for (i = 1; i < MAX; i++)
if (number[i])
    {
    agvs[i] = (struct mapname *) malloc(sizeof(struct mapname));
    if (!agvs[i])
       {
       printf("Out of memory for agvs in shortp4.c\n");
       exit(1);
       }

    y = 0;
    flag = 1;
    dist = INFINITY;
    while(y < 4 && flag)
       {
       if (number[from_no]->c_index[y] == i)
          {
          dist = number[from_no]->c_dist[y];
          flag = 0;
          }
       y++;
       }

    agvs[i]->visited = 0;          /* Initializes the parameters */
    for(y = 1; y <= path_no; y++)
       {
       agvs[i]->sdist[y] = dist;      /* Initializes distance of pt     */
```

```
        spath[i].parent[y] = from_no; /* as parent node for each node. */
        }
    }
agvs[from_no]->visited = 1; /* Indicates when it has visited a point. */
for (i = 1; i < MAX; i++) /* FIRST time around (i = 1)   */
if (agvs[i])
    {                                   /* it finds the closest direct
point  */
    flag = 0;                           /* from 'from_no'. Repeats it for
the */
    do_rest = 1;
    for (j = 1; j < MAX; j++)           /* points which haven't been
visited. */
    if (agvs[j] && !agvs[j]->visited) /* If not visited. */
        {
        if (!flag)
           {
           dist1 = agvs[j]->sdist[1]; /* dist of 'j' from from_no */
           nnode = j; /* j indicates the index no. of point. */
           flag = 1;
           }
        else
           {
           if (dist1 >= agvs[j]->sdist[1])
              {
              dist1 = agvs[j]->sdist[1]; /* dist of 'j' from from_no */
              nnode = j;
              }
           }
        }
    agvs[nnode]->visited = 1;
/*printf("nnode = %d, to_no = %d, from_no = %d\n", nnode, to_no,
from_no);*/
    if (nnode == to_no)
       counter +=1;
/*printf("counter = %d, path_no = %d\n", counter, path_no);*/
    if (nnode == to_no || counter == path_no)
       {                                  /* index no.'s of parent nodes
resp. */
       top_of_stack = 0;
       push(path_no, to_no);              /*  into a stack till it
*/
       temp_path_no = path_no;            /* reaches from_no           */
       while(!spath[nnode].parent[temp_path_no])
            temp_path_no--;
       laststop = spath[nnode].parent[temp_path_no];
/*printf("laststop = %d, spath[%d].parent[%d] = %d\n", laststop, nnode,
path_no, spath[nnode].parent[path_no]);*/
       while (laststop != from_no)
            {
          push(path_no, laststop);
/*printf("spath[%d].parent[%d] = %d\n", laststop, path_no,
spath[laststop].parent[path_no]);*/
            temp_path_no = path_no;
            while(!spath[laststop].parent[temp_path_no])
                 temp_path_no--;
          laststop = spath[laststop].parent[temp_path_no];
          }
            push(path_no, laststop);
/*printf("top_of_stack = %d\n", top_of_stack);*/
```

```
            if (top_of_stack == 2)
             {
             /*printf("DIRECT PATH\n");*/
                 p = (struct path *) malloc(sizeof (struct path));
                 if(!p)
                   {
                   printf("Allocation failure for struct path in
shortp()\n");
                   exit(1);
                   }
                 p->point = pathlist[path_no][0];
                 pfinal[path_no][1] = p;
/*printf("pfinal[%d][1]->point = %d\n", path_no, pfinal[path_no][1]-
>point);*/
                 p = (struct path *) malloc(sizeof (struct path));
                 if(!p)
                   {
                   printf("Allocation failure for struct path in
shortp()\n");
                   exit(1);
                   }
                 p->point = pathlist[path_no][1];
                 pfinal[path_no][0] = p;
/*printf("pfinal[%d][0]->point = %d\n", path_no, pfinal[path_no][0]-
>point);*/
                 pfinal[path_no][2] = NULL;

                 for(i = 1; i < MAX; i++)
                 if (number[i])
                    {
                    free(agvs[i]);
                    free(spath[i]);
                    }

                 return **pfinal;
             }
            else if (top_of_stack > 2) /* Lists path of AGV  */
             {                    /* starting with from_no. */
                 x = 0;
             while (top_of_stack)
                   {
                       pop();
                       p = (struct path *) malloc(sizeof (struct path));
                       if(!p)
                         {
                         printf("Alloc failure: struct path in
shortp\n");
                         exit(1);
                         }
                       p->point = pathlist[path_no][top_of_stack];
                       pfinal[path_no][x] = p;
/*printf("1: pfinal[%d][%d]->point = %d\n", path_no,x,
pfinal[path_no][x]->point);*/
                       pfinal[path_no][x+1] = NULL;
                       x++;
                       }
                         if (counter == path_no)
                            {

                               for(i = 1; i < MAX; i++)
```

```
                        if (number[i])
                           {
                           free(agvs[i]);
                           free(spath[i]);
                           }

                     return **pfinal;
                         }
                      else
                         do_rest = 0;
                }
                else
                   {
                   printf("Error in top_of_stack\n");
                   exit(1);
                   }
        }
        if (do_rest)
        for (k = 0; k < MAX; k++)
        if (agvs[k] && (agvs[k]->visited == 0 || k == to_no))
          {                                  /* For that 'j' which is
closest */
                                             /* it checks all other points
*/
                                             /* to see if the shortest
*/
                                             /* path or via 'j' is closer.
*/
/*printf("k = %d\n", k);*/
           flag = 1;
           for(l = 1; l <= path_no; l++) /* l indicates which sdist of
nnode*/
           if (flag)
              {                        /* is being considered to see if the
path */
              for(m = 1; m <= path_no; m++) /* via 'nnode' or existing
sdist*/
              if (flag)
                 {                                /* is shorter for node
'k'. m */
                 q = 0;          /* indicates the sdist of 'k' being
considered */
                 while(number[nnode]->c_index[q] != k && q < 4)
                      {
                      q++;
                      }
                 if (q<4 && agvs[k]->sdist[m] > agvs[nnode]->sdist[l] +
number[nnode]->c_dist[q]) /* if path to 'k' is greater than via nnode */
                    {
                                /* finds which sdist via 'nnode' is
shorter */
                    for(n = path_no; n > m + 1; n--)/* than which sdist
of 'k'.*/
                       {          /* Pushes the sdist back one till
insertion. */
                       agvs[k]->sdist[n] = agvs[k]->sdist[n-1];
                       }
                    agvs[k]->sdist[m] = agvs[nnode]->sdist[l] +
number[nnode]->c_dist[q];
                    spath[k].parent[m] = nnode;
```

```
                    if (k == to_no && counter)
                       counter += 1;
                    temp_path_no = m;
                    while(path_no >= temp_path_no)
                         {
                         spath[k].parent[temp_path_no+1] = NULL;
                         temp_path_no++;
                         }
                    temp_path_no = m;
                    while(spath[nnode].parent[temp_path_no+1] && path_no
>= temp_path_no && spath[nnode].parent[temp_path_no+1] != from_no)
                         {
                         spath[k].parent[temp_path_no+1] = nnode;
                         agvs[k]->sdist[temp_path_no+1] = agvs[nnode]-
>sdist[temp_path_no+1] + number[nnode]->c_dist[q];
                         temp_path_no++;
                         }
                    flag = NULL;

/* Now get out of m & l for loops */
/*printf("1: spath[%d].parent[%d] = %d, spath[%d].parent[%d] = %d\n", k,
m, nnode, k, m+1, spath[k].parent[m+1]);*/
                    }
                else /* Repeat till you find connecting pt to
nnode/reach MAX */
                    ;
                }
             }
          }
   }
   /*printf("Error in calculating shortest path\n");*/

   for(i = 1; i < MAX; i++)
   if (number[i])
      {
      free(agvs[i]);
      free(spath[i]);
      }

   return **pfinal;
   /*exit(1);*/
}


/*index(via, i)
char *via[MAX];
int i;
{
int loc;
char *p;

loc = *via[i] - 'A'; /* *via[i] == via[i][0] *//*
p = &via[i][1]; /* Trying to access numerical string after first letter
*//*
loc += (atoi(p) - 1) * 26 + 1;
if (loc > 2600)
   {
   printf("Shortp4: Cell out of bounds\n");
   exit(1);
   }
```

```
return(loc);
}*/


void push(path_no, node)
int path_no, node;
{
pathlist[path_no][top_of_stack] = node; /* Check definition of pathlist.
*/
top_of_stack++;
/*printf("node = %d\n", node);*/
}

void pop()
{
if (top_of_stack > 0)
   top_of_stack--;
}
```

Time3.c

```
/* time.c will allocate time intervals to selected path and store in
list. */
/* Important, most/all of the points may be at constant distance from
*/
/* adjacent points.
*/

/* #include "struct.h"*/

/*time()
{
get path
take each section of path
calculate travel time for each section
v2 - u2 = 2as
v = u + at
s = ut + 1\2 at2
Case I Constant acceleration
Case II Constant velocity
Case III Constant retardation
ttime1 = (dist - sqr(v)/2*a) / v;
ttime2 = v/a;
ttime = ttime1 + ttime2;
dist = sqrt(sqr(x2 - x1) + sqr(y2 - y1))
}*/


/*allocate travel time for each section
store in master list*/



/*extern struct matrix
              {                       /* Material Flow Matrix. *//*
              char *name;             /* Name of decision pt. eg, A1, B34
*//*
              int coord[2];           /* Stores x,y coordinates of point
*//*
              char *type;             /* Type of decision pt. *//*
              int c_dist[CONNECT];    /* array showing dist of connecting
pts *//*
              int c_index[CONNECT]; /* array showing index of connecting
pt.*//*
              char c_name[CONNECT]; /* array showing name *//*
              int visited;            /* & dist. of connecting  points.
*//*
              } *number[MAX];


struct path    /* To store data on shortest path calculations. This */
/*
         {    /* may not necessarily be the final shortest path.   *//*
         int point;    /* Stores index number of point. */ /*
         int vel;      /* Velocity at that decision point. *//*
         float atime;  /* Arrival time at that decision point. *//*
         float dtime;  /* Departure time from that decision point. *//*
         } *pfinal[SEGM_NO][PATH_NO];*/
```

```
/* Need to consider the case of only one segment on the path of the
vehicle */
/* with veh. crossing that segment without reaching maximum velocity
*/


#include <stdio.h>

float ttime();
double dist();
extern float tnow, tend;

struct path *alloc_time(pfinal, path_no)
struct path *pfinal[PATH_NO][SEGM_NO];
int path_no;
{
int i, pt_a, pt_b;
float u, v, d;
float time1, time2, time3, time;
float dist3;
int k = 0;
int last_pt;
float travel_dist = 0.0, s = 0.0, stop_dist;
float stop_dist1, stop_dist2, stop_dist3;
/*printf("Reached time2.c\n");*/
while(pfinal[path_no][k + 1]) /* Finds total dist. on path. */
     {
     travel_dist += dist(pfinal[path_no][k]->point, pfinal[path_no][k +
1]->point);
     k++;
     }
     pfinal[path_no][0]->atime = tnow;
     pfinal[path_no][0]->dtime = pfinal[path_no][0]->atime;
     for (i = 0; i < SEGM_NO; i++) /* 'i' indicates segment for which
travel */
         {              /* time is being calculated & stored in master
list. */
         if (pfinal[path_no][i + 1])
            {
             pt_a = pfinal[path_no][i]->point;
             pt_b = pfinal[path_no][i + 1]->point;
             travel_dist -= dist(pt_a, pt_b);
/*printf("travel_dist = %f\n", travel_dist);*/
             stop_dist = pow((double) MAX_VEL, 2.0)/(2.0*((double)
ACC));
             if (travel_dist >= stop_dist) /* While there's time to
stop.*/
                {
                pfinal[path_no][i+1]->atime = pfinal[path_no][i]->dtime
+ ttime(pt_a, pt_b, i);
                pfinal[path_no][i+1]->dtime = pfinal[path_no][i+1]-
>atime;
/*printf("1: time2.c: pfinal[%d][%d]->dtime = %f pfinal[%d][%d]->atime =
%f\n",path_no, i, pfinal[path_no][i]->dtime, path_no, i+1,
pfinal[path_no][i+1]->atime);*/

                }

             else if(i==0 && dist(pt_a,pt_b)>(pow((double)
MAX_VEL,2.0))/(2*((double) ACC)))
```

```
                {
                dist3 = dist(pt_a, pt_b);
/*printf("1: pt_a = %d, pt_b = %d, dist3 = %f\n", pt_a, pt_b, dist3);*/
                time1 = (double) MAX_VEL/((double) ACC); /* v = at */
                s  = .5*ACC*pow(time1, 2.0); /* s = .5at2 */
                time3 = (double) MAX_VEL/((double) ACC); /* v = at */
                time2 = (dist3 - 2*s)/((double) MAX_VEL); /* s = ut */
                time = time1 + time2 + time3;
                pfinal[path_no][1]->atime = pfinal[path_no][0]-
>dtime+time;
/*printf("time1 = %f, time2 = %f, time3 = %f\n", time1,time2,time3);*/
                pfinal[path_no][1]->dtime = pfinal[path_no][1]->atime;
                }

             else /* If required to decelerate to a stop */
                {
                travel_dist += dist(pt_a, pt_b); /* is this right */
/*printf("travel_dist = %f\n", travel_dist);*/
                if (s == 0.0) /* If required to start decelerating. */
                   { /* 's' = dist. veh. can go without starting
retardation. */
/*printf("dist(pt_a, pt_b) = %f\n", dist(pt_a, pt_b));*/
                   s = travel_dist - (pow((double) MAX_VEL,
2.0))/(2*((double) ACC));
                   time1 = s/((double) MAX_VEL);/*it has to start
slowing*/
                   time2 = sqrt(2*(dist(pt_a, pt_b) - s)/((double)
ACC));/* in this */
                   time = time1 + time2;    /* node hence time1 & time2.
*/
/*printf("time1 = %f, time2 = %f\n", time1, time2);*/
                   pfinal[path_no][i+1]->atime = pfinal[path_no][i]-
>dtime+time;
/*printf("2: time2.c: pfinal[%d][%d]->dtime = %f pfinal[%d][%d]->atime =
%f\n",path_no, i, pfinal[path_no][i]->dtime, path_no, i+1,
pfinal[path_no][i+1]->atime);*/
                   pfinal[path_no][i+1]->dtime = pfinal[path_no][i+1]-
>atime;
                   u = (double) MAX_VEL - ((double) ACC)*time2;
                   travel_dist -= dist(pt_a, pt_b); /* is this right */
                   }
                else /* (s != 0) *//* If already deccelerating. */
                   {
                   d = dist(pt_a, pt_b);
                   v = u;
                   u = sqrt(pow(v, 2.0) - 2*((double) ACC)*d);
                   time = (v - u)/((double) ACC);
/*printf("time = %f, v = %f, u = %f, d = %f\n", time, v, u, d);*/
                   pfinal[path_no][i+1]->atime = pfinal[path_no][i]-
>dtime+time;
/*printf("3: time2.c: pfinal[%d][%d]->atime = %f\n",path_no,i+1,
pfinal[path_no][i+1]->atime);*/
                   pfinal[path_no][i+1]->dtime = pfinal[path_no][i+1]-
>atime;
                   }
                }
           }
         else
            break; /* breaks out of for loop, careful unorthodox break
*/
```

```
        }

        last_pt = pfinal[path_no][i]->point;
        if (strcmp(number[last_pt]->type, "parking"))
           {
            pfinal[path_no][i]->dtime += transfer_time;
           }

        return **pfinal;
}


float ttime(pt_a, pt_b, i) /* 'i' indicates segment on path for which
travel */
int pt_a, pt_b;      /* time is being calculated and pt_a, pt_b are
*/
int i;               /* the index numbers of points on segment.
*/
{
float s, w;
float time, time1, time2;
static float u = 0.0; /* u indicates initial vel. at each section. */
float ddist[SEGM_NO]; /* Indicates length of that segment. */
ddist[i] = dist(pt_a, pt_b);
/*printf("ddist[i] = %f\n", ddist[i]);*/
if (i == 0) /* if (first section) */
   {
   if (MAX_VEL <= sqrt(2.0*ACC*ddist[0])) /* if reaches MAX_VEL. */
      {
      time1 = (double) MAX_VEL/((double) ACC); /* v = at */
      s  = .5*ACC*pow(time1, 2.0); /* s = .5at2 */
      time2 = (ddist[0] - s)/((double) MAX_VEL); /* s = ut */
      time = time1 + time2;
      u = (double) MAX_VEL;
/*printf("0:  s = %f, time 1 = %f, time2 = %f\n", s, time1, time2);*/
      }
   else
      {
      time = sqrt(2.0*ddist[0]/((double) ACC)); /* s = .5at2 */
/*printf("1: time = %f\n", time);*/
      u = (double) ACC*time; /* v = at */
      }
   }
else
   {
   if (u == (double) MAX_VEL)
      {
      time = ddist[i]/((double) MAX_VEL); /* s = ut */
/*printf("2: time = %f\n", time);*/
      }
   else if((double) MAX_VEL <= sqrt(pow(u, 2.0) +2.0*((double)
ACC)*ddist[i]))/*if reaches MAX_VEL*/
      {
      time1 = ((double) MAX_VEL - u)/(double) ACC; /* v = u + at */
      s = (pow((double) MAX_VEL, 2.0) - pow(u, 2.0))/(2.0*((double)
ACC));
      time2 = (ddist[i] - s)/(double) MAX_VEL; /* s = ut */
      time = time1 + time2;
/*printf("3: time = %f\n", time);*/
      }
```

```
   else /* If still won't MAX_VEL in this segment. */
      {
      w = sqrt(pow(u, 2.0) + 2.0*((double) ACC)*ddist[i]);
      time = (w - u)/(double) ACC;
/*printf("4: time = %f\n", time);*/
      u = w;
      }
   }
   return(time);
}


double dist(a, b)
int a, b;
{
double d;
double x1, x2, y1, y2, z;
x1 = number[a]->coord[0];
y1 = number[a]->coord[1];
x2 = number[b]->coord[0];
y2 = number[b]->coord[1];
if (x1 < x2)
   {
   z = x1;
   x1 = x2;
   x2 = z;
   }
if (y1 < y2)
   {
   z = y1;
   y1 = y2;
   y2 = z;
   }

d = sqrt(pow((x1 - x2), 2.0) + pow((y1 - y2), 2.0));
return(d);
}
```

Conflct3.c

```
/* Conflict.c will take as input a shortest path with timing data & */
/* a master list containing occupation times of nodes/segments.  It */
/* will identify regions of conflict and stop vehicle at preceeding */
/* point for time required to stop conflict and so on to finally    */
/* a conflict free path with occupation times of points on path.    */

#include <stdio.h>

extern float tnow;

struct path *conflict(pfinal, path_no)
struct path *pfinal[][SEGM_NO];
int path_no;
{
int x;
int a, i;
float occup_begin, occup_end, atime, dtime, delay;
struct time *temp_top;
/*printf("Reached conflct.c\n");*/

x=0;
while(pfinal[path_no][x])
{
/*printf("Confl1: pfinal[%d][%d]->atime = %f, pfinal[%d][%d]->dtime =
%f\n", path_no,x,pfinal[path_no][x]->atime, path_no,x,
pfinal[path_no][x]->dtime);*/
x++;
}

for(i = 1; i < SEGM_NO; i++) /* i=1 since don't need to check for 1st pt
*/
   {
   if (pfinal[path_no][i])
      {
/*printf("i = %d\n", i);*/
      a = pfinal[path_no][i]->point; /* a represents index no of pt on
path */
      atime = pfinal[path_no][i]->atime; /* Arrival time at pt */
      dtime = pfinal[path_no][i]->dtime; /* departure time from pt on
path. */
      if (number[a]->occup_time)
         update_time(a);
      if (number[a]->occup_time) /* if there exists some occup_time */
         {
         occup_begin = number[a]->occup_time->occup_begin; /* Time occup
of   */
         occup_end = number[a]->occup_time->occup_end; /* segm begins &
ends. */
         temp_top = number[a]->occup_time;
         while((int) (occup_begin * INFINITY))
               {
               if ((atime >= occup_begin && atime <= occup_end) || (dtime
>= occup_begin && dtime <= occup_end)) /* If conflict. */
                  {
                  delay = occup_end - atime;
/*printf("Conflict present: i = %d\n", i);*/
/*printf("atime = %f, dtime = %f, occup_begin = %f, occup_end = %f\n",
atime, dtime, occup_begin, occup_end);*/
```

```
/*printf("delay = %f\n", delay);*/
                  if (delay != 0.000)
                     i = remove(i, delay, path_no);

x=0;
while(pfinal[path_no][x])
{
/*printf("Confl2: pfinal[%d][%d]->atime = %f, pfinal[%d][%d]->dtime =
%f\n", path_no,x,pfinal[path_no][x]->atime, path_no,x,
pfinal[path_no][x]->dtime);*/
x++;
}

                  occup_begin = NULL;
                  }
               else if (atime > occup_end) /* If no confl & atime greater
than */
                  {/*present occup_time check if conflict with next
occup_time.*/
                  if (number[a]->occup_time->next) /* if next occup_time.
*/
                     {
/*printf("Reached problem 3: a = %d\n", a);*/
                     number[a]->occup_time = number[a]->occup_time->next;
                     occup_begin = number[a]->occup_time->occup_begin;
                     occup_end = number[a]->occup_time->occup_end;
                     }
                  else
                     occup_begin = NULL;
                  }
               else
                  occup_begin = NULL;
               }
               number[a]->occup_time = temp_top;
            }
         }
      else
         break;
      }
x = 0;
while(pfinal[path_no][x])
{
/*printf("Confl3: pfinal[%d][%d]->atime = %f, pfinal[%d][%d]->dtime =
%f\n", path_no,x,pfinal[path_no][x]->atime, path_no,x,
pfinal[path_no][x]->dtime);*/
x++;
}
   return **pfinal;
}


update_time(a) /* removes old occup_time ie those before tnow for
segments */
int a;
{
float time;
struct time *y = NULL;
/*printf("Reached update_time a = %d\n", a);*/
 /* if there is a occup_time, remove if the  */
 /* occup_time is before tnow, since you can't live in the past. */
```

```
   time = number[a]->occup_time->occup_end;
/*printf("time = %f, tnow = %f\n", time, tnow);*/
   while (tnow > time && (int) time) /* time has to have some value */
         {
         if (number[a]->occup_time->next)
            {
/*printf("Reached problem 1:\n");*/
            y = number[a]->occup_time;
            number[a]->occup_time = number[a]->occup_time->next;
            if (y->prior) /* if middle link */
               {
               number[a]->occup_time->prior = y->prior;
               number[a]->occup_time->prior->next = number[a]-
>occup_time;
               }
            else /* if 1st link */
               number[a]->occup_time->prior = NULL;
            time = number[a]->occup_time->occup_end;
            free(y);
            }
                           /* goto next occup_time for same segment if
any. */
         else /* If no occup_time->next and this is before tnow */
            {
/*printf("Reached problem 2:\n");*/
            y = number[a]->occup_time;
            number[a]->occup_time = NULL;
            free(y);
            time = 0.0;
            }
         /* -keep doing till you reach occup-time which is after ptime
or no
            more occup-time (for a particular segment)
         -If no occup-time is > ptime put 1st occup_time == NULL.  If
some
          occup_time > ptime, then put it at head of linked list. */
         }
}


remove(i, delay, path_no) /* doesn't check for more than 1  */
                          /* occup_time for prev_pt: DEFECT */
int i, path_no; /* 'i' indicates point no */
float delay;
{
int prev_pt, pres_pt, j, flag;
float patime, pdtime, poccup_begin, poccup_end; /* contains data on prev
pt */
struct time *temp_top;
/*printf("Reached remove()\n");*/

if(i)
  {
  prev_pt = pfinal[path_no][i-1]->point; /* data on last pt on short
path */
  patime = pfinal[path_no][i-1]->atime;
  pdtime = pfinal[path_no][i-1]->dtime + delay;
  if (number[prev_pt]->occup_time)
     update_time(prev_pt);
```

```
/*printf("prev_pt = %d\n", prev_pt);*/
  if (!number[prev_pt]->occup_time || i == 1)
     {                              /* if prev_pt doesn't have any
occup_times */
/*printf("Reached !number[prev_pt]->occup_time || i == 1\n");*/
     pfinal[path_no][i-1]->dtime += delay;
     pfinal[path_no][i]->atime += delay;
     pfinal[path_no][i]->dtime += delay;
     /* below need to be careful that += delay doesn't go beyond tend */
     j = i;
     while(pfinal[path_no][i+1])
          {
          pfinal[path_no][i+1]->atime += delay;
          pfinal[path_no][i+1]->dtime += delay;
          i++;
          }
          return j;
     }

  temp_top = number[prev_pt]->occup_time;
  flag = 1;
  while(number[prev_pt]->occup_time && flag)
       {
       poccup_begin = number[prev_pt]->occup_time->occup_begin;/*occup
data on*/
       poccup_end = number[prev_pt]->occup_time->occup_end;    /*
previous pt */
       if ((patime < poccup_begin && pdtime > poccup_begin) && (patime >
poccup_begin && pdtime < poccup_end)) /* while causing conflict in
previous segment */
          {                         /* ie head on collision
*/
          i--;
          if (!i)
             {
             /*printf("Conflict at begining in remove()\n"); */
             pfinal[path_no][i]->dtime = INFINITY;
             while(pfinal[path_no][i+1])
                  {
                  pfinal[path_no][i+1]->atime = INFINITY;
                  pfinal[path_no][i+1]->dtime = INFINITY;
                  i++;
                  }
                  return i;
             }
          delay = poccup_end - patime;  /* wait time in prev pt to prev
pt to */
          prev_pt = pfinal[path_no][i-1]->point; /* clear conflict in
prev pt */
          patime = pfinal[path_no][i-1]->atime;
          pdtime = pfinal[path_no][i-1]->dtime + delay;
          poccup_begin = number[prev_pt]->occup_time->occup_begin;
          poccup_end = number[prev_pt]->occup_time->occup_end;
          }
          if (number[prev_pt]->occup_time->next)
              number[prev_pt]->occup_time = number[prev_pt]->occup_time-
>next;
          else
             flag = NULL;
       }
```

```
        number[prev_pt]->occup_time = temp_top;


        /* Now there's no conflict in prev pt caused by vehicle having to
wait */
/*printf("Reached adding delay to prev_pt\n");*/
        pres_pt = pfinal[path_no][i]->point;
        pfinal[path_no][i-1]->dtime += delay;
        pfinal[path_no][i]->atime = number[pres_pt]->occup_time-
>occup_end;
        pfinal[path_no][i]->dtime += delay;
        j = i;
        while(pfinal[path_no][i+1])
             {
             pfinal[path_no][i+1]->atime += delay;
             pfinal[path_no][i+1]->dtime += delay;
             i++;
             }
             return j;
  }
}
```

Veh_all3.c

```
/* veh_all1.c */
/*Vehicle Allocation
Assumption: If a vehicle waits at the drop off pt. instead of going
            to parking then the pt. must be off the beaten path

Consider each vehicle and find where it's located, or will be located.

Find straight line dist. of vehicle from demand point.

Find approximate time it will take to get to demand point at constant
velocity in a straight line.

If vehicle is busy, add time required for it to get free to estimated
(approximate) travel time.

Take five (three) earliest vehicles which will reach demand point.

Obtain shortest path for each vehicle one by one & pass through conflict
resolver and time.d to obtain precise time of arrival at demand point.

Allocate that vehicle to that demand & put on event list.

Put status of vehicle on vehicle list.*//*

struct path    /* To store data on shortest path calculations. This *//*
          {    /* may not necessarily be the final shortest path.   *//*
          int point;    /* Stores index number of point. *//*
          int vel;      /* Velocity at that decision point. *//*
          float atime;  /* Arrival time at that decision point. *//*
          float dtime;  /* Departure time from that decision point. *//*
          } *pfinal[PATH_NO][SEGM_NO]; /* From time.c */



#include <stdio.h>

double dddist();


/*struct vehicle
              {
              int status;/* Busy: unloaded/loaded, parking, charging,
idle. *//*
              int dest_pt;  /* dest_pt & dest_time, ie which pt. it's
headed *//*
              int dest_time;/* for & what time it will reach there.
*//*
              int ded_area; /* which if any dedicated areas does it
service. *//*
              } vehicle_no[VEHICLE_NO];*/


extern float tnow, tend;
extern int no_veh;
extern struct ev_lst *top, *last, *top_veh, *last_veh;
extern int all_veh_busy;
```

```
veh_alloc(request, dest) /* 'request' is the point requesting a vehicle
*/
int request, dest;
{
extern float loading[VEHICLE_NO];
int alpha, beta, busy;
struct ev_lst *i, *temp;
int storepath;
int j, k, m, flag;
int x, y, doit;
float dest_time, ddest_time/* actual time vehicle reaches dest */;
int veh_i_d = 0;
float dist1, time1, ttemp;
float ttime[MAX_TIMENO][2];     /* MAX.. indicates which earliest time,
while
                                  [2] indicates earliest time & vehicle
no.  */
float waite = 0.0, arrival_time = 0.0;
/*printf("Reached veh_alloc()\n");*/
j = 0;
while(j < MAX_TIMENO)
     {
     ttime[j][0] = 0.0; /* earliest time */
     ttime[j][1] = (float) INFINITY; /* vehicle no */
     j++;
     }
i = (struct ev_lst *) malloc(sizeof(struct ev_lst));
if (!i)
   {
   printf("Memory allocation failure for ev_lst in veh_all1.c\n");
   exit(1);
   }
/*for(k = 0; k < no_veh; k++)
printf("vehicle_no[%d].status = %d\n", k, vehicle_no[k].status);*/
busy = 1;
for(k = 0; k < no_veh; k++) /* for all the vehicles */
if (vehicle_no[k].status != 1) /* if vehicle is not busy & UNLOADED */
if (!(top_veh && vehicle_no[k].status == 2)) /* ie no pt. in queue for
veh. */
   {                            /* ie is not answering a request     */
   doit = 1;
   if (vehicle_no[k].ded[0]) /* if this vehicle is a dedicated veh. */
      {
      doit = 0;
      for(x = 1; x < DED; x++)
         if (vehicle_no[k].ded[x] == request)
            {
            doit = 1;
            busy = 0; /* at least one vehicle can be allocated */
            }
      }
   else
      busy = 0;



   /* if (number[request]->ded)
      {
```

```
printf("YES ded_veh\n");
      x = 0;
      if (!strcmp(number[request]->type, "request"))
         {
         while(k != request_veh[request][x] && x < VEHICLE_NO - 1)
              x++;
              if (x == (VEHICLE_NO - 1))
                 doit = NULL;
              else
                 busy = 0; /* At least one vehicle can be allocated *//*
         }
      else
         {
         while(k != machining_veh[request][x] && x < VEHICLE_NO - 1)
              x++;
         if (x == (VEHICLE_NO - 1))
            doit = NULL;
         else
            busy = 0; /* At least one vehicle can be allocated *//*
         }
      }
   else
      busy = 0; /* At least one vehicle can be allocated */




   if (doit) /* if this veh can be alloc for this pt */
      {
      dist1 = dddist(vehicle_no[k].dest_pt, request); /* gets app dist
bet pts*/
      time1 = dist1/ (float) MAX_VEL; /* calculates app. travel time bet
pt's */
      if (vehicle_no[k].status == 2 || vehicle_no[k].status == 3) /* if
busy*/
         time1 += (vehicle_no[k].dest_time - tnow);/*add time to be
free*/
      flag = 1;
      j = 0;
      while(j < MAX_TIMENO && flag) /* Finds MAX_TIMENO earliest times.
*/
           {
           if (!ttime[j][0]) /* if very first round thru loop */
              {
              ttime[j][0] = time1;
              ttime[j][1] = (float) k; /* identifies which vehicle */
   /*printf("ttime[%d][1] = %d\n", j, (int) ttime[j][1]);*/
              flag = NULL;
              }
           else /* if (ttime[j][0]) */
              {
              if (time1 < ttime[j][0])
                 {
                 m = MAX_TIMENO - 1;
                 while(m > j && m > 0) /* inserts time1 in right place
*/
                      {
                      ttime[m][0] = ttime[m - 1][0];
                      ttime[m][1] = ttime[m - 1][1];
```

```
                        m--;
                        }
                    ttime[j][0] = time1;
                    ttime[j][1] = (float) k;
   /*printf("ttime[%d][1] = %d\n", j, (int) ttime[j][1]);*/
                    flag = NULL;
                    }
                }
                j++;
            }
      }
   else
      ; /* don't process for this veh. since it's not ded. to this pt.
*/
   }
if (busy == 1) /* All vehicles are busy */
   {
   all_veh_busy++;
   temp = (struct ev_lst *) malloc(sizeof(struct ev_lst));
   if (!temp)
      {
      printf("Memory allocation failure for ev_lst in veh_all1.c\n");
      exit(1);
      }
   temp->which = request;
   temp->dest = INFINITY;
   temp->time = tnow;
   top_veh = load_veh(temp, top_veh);
   if (!strcmp(number[request]->type, "request"))
      {
      x = 0;
      while(machining_pt[x][0] != dest)
           x++;
      machining_pt[x][1] = 0; /* m/c is freed since no veh avail for
alloc */
      }
/*printf("All vehicles busy\n");*/
   return;
   }

/*for(j = 0; j<MAX_TIMENO; j++)
   printf("ttime[%d][1] = %d\n", j, (int) ttime[j][1]);*/

   dest_time = 0;
   beta = vehicle_no[(int) ttime[0][1]].dest_pt;
   for(j = 0; j < MAX_TIMENO; j++)   /* Calculates time of arrival at
pt. */
   if ((int) ttime[j][1] != INFINITY)
      { /* generating request for selected vehicles & keeps earliest
one. */
      alpha = vehicle_no[(int) ttime[j][1]].dest_pt;
/*printf("alpha = %d, ttime[j][1] = %d, request = %d\n", alpha, (int)
ttime[j][1], request);*/
      if (alpha == request)
         { /* arrival_time same as i->time since veh cant answer request
*/
         veh_i_d = (int) ttime[j][1]; /* until it has finished unloading
*/
         if (vehicle_no[veh_i_d].status == 2) /*if busy add time to get
free*/
```

```
            {
            i->time = vehicle_no[veh_i_d].dest_time + transfer_time; /*
+time */
            waite = i->time - tnow - transfer_time;                  /* to
load */
            if (i->time - transfer_time > tend)
               waite -= (i->time - transfer_time - tend);
            }
         else
            {
            i->time = tnow + transfer_time;
            waite = 0.0;
            }

            if (i->time - transfer_time < tend)
               {
               loading[veh_i_d] += transfer_time;
               if (i->time >= tend)
                  loading[veh_i_d] -= (i->time - tend);
               }
            y = 0;
            if(!strcmp(number[request]->type, "request"))
              {
              while(request_pt[y] != request)
                   y++;
              stat.raw_part_waiting[y][0] += waite;
              stat.raw_part_waiting[y][1] += 1.0;
/*printf("1: stat.raw_part_waiting = %f, For pt %d\n",
stat.raw_part_waiting[y][0], request);*/
              }
            else
              {
              while(machining_pt[y][0] != request)
                   y++;
              stat.fin_part_waiting[y][0] += waite;
              stat.fin_part_waiting[y][1] += 1.0;
              }

         i->from = request;
         i->which = request; /* where the event occurs, ie where it's
going */
         i->dest = dest;
         strcpy(i->type, "pickup");
         i->veh = veh_i_d; /* which vehicle */
/*printf("Veh_all1: i->veh = %d\n", i->veh);*/
         top = load(i, top);
         if (vehicle_no[veh_i_d].status == 5 ||
vehicle_no[veh_i_d].status == 6)
            stat.veh_idle[veh_i_d][1] += (tnow -
stat.veh_idle[veh_i_d][0]);
         vehicle_no[veh_i_d].status = 1;
         vehicle_no[veh_i_d].dest_pt = request;
         vehicle_no[veh_i_d].dest_time = i->time;
         return;
         }
      else if (j==0/* || beta != alpha*/) /* if two veh's are not at
same pt. */
         {
         storepath = 0;
```

```
         if (vehicle_no[(int) ttime[j][1]].status == 2 ||
vehicle_no[(int) ttime[j][1]].status == 3)
            {
            ttemp = tnow;
            tnow = vehicle_no[(int) ttime[j][1]].dest_time;
            ddest_time = sshortp(alpha, request, storepath, (int)
ttime[j][1]);
            tnow = ttemp;
            }
         else
            {
            ddest_time = sshortp(alpha, request, storepath, (int)
ttime[j][1]);
            }

/*printf("dest_time = %f,ddest_time = %f,j = %d\n", dest_time,
ddest_time, j);*/
         /* In prior line need to make sure that only the
            shortest path used is stored in master list.  */
         if (dest_time == 0.0 || ddest_time < dest_time)
            {
            dest_time = ddest_time;
            veh_i_d = (int) ttime[j][1];
            }
         }
      }
      strcpy(i->type, "pickup");
      i->veh = veh_i_d;
      i->which = request;
      i->dest = dest;
      i->from = vehicle_no[veh_i_d].dest_pt; /* pt where vehicle is at
now */

      storepath = 1;
      if ((vehicle_no[veh_i_d].status == 2 || vehicle_no[veh_i_d].status
== 3) && vehicle_no[veh_i_d].dest_time>tnow)
         {
         ttemp = tnow;
         tnow = vehicle_no[veh_i_d].dest_time;
/*printf("veh_i_d = %d\n", veh_i_d);*/
         i->time = sshortp(alpha, request, storepath, veh_i_d);
         tnow = ttemp;
         }
      else
         {
/*printf("veh_i_d = %d\n", veh_i_d);*/
         i->time = sshortp(alpha, request, storepath, veh_i_d);
         }

      arrival_time = i->time - transfer_time;
      waite = arrival_time - tnow;

      if (arrival_time > tend)
         {
         waite -= (arrival_time - tend);
         }
      else
         {
         loading[veh_i_d] += transfer_time;
         if (i->time > tend)
```

```
            loading[veh_i_d] -= i->time - tend;
/*printf("loading[%d] = %f\n", veh_i_d, loading[veh_i_d]);*/
         }

/*printf("i->time = %f, tnow = %f, tend = %f\n", i->time, tnow, tend);*/
/*printf("vehicle_no[veh_i_d].dest_time = %f, waite = %f\n",
vehicle_no[veh_i_d].dest_time,waite);*/

       y = 0;
       if (!strcmp(number[request]->type, "request"))
          {
          while(request_pt[y] != request)
               y++;
          stat.raw_part_waiting[y][0] += waite;
          stat.raw_part_waiting[y][1] += 1.0;
/*printf("1: stat.raw_part_waiting = %f, For pt %d\n",
stat.raw_part_waiting[y][0], request);*/
          }
        else
          {
          while(machining_pt[y][0] != request)
               y++;
          stat.fin_part_waiting[y][0] += waite;
          stat.fin_part_waiting[y][1] += 1.0;
          }

      if (vehicle_no[veh_i_d].status == 2 || vehicle_no[veh_i_d].status
== 3)
         {                                          /* this is for
subtracting */
           /* the diff. in tnow & when the veh will be free in the
future */
         if (vehicle_no[veh_i_d].dest_time > tend)  /* from veh_busy
times*/
            waite = 0.0;
         else
            if (vehicle_no[veh_i_d].dest_time &&
vehicle_no[veh_i_d].dest_time != (double) INFINITY)
               waite -= (vehicle_no[veh_i_d].dest_time - tnow);
         }
         stat.veh_busy_empty[veh_i_d] += waite;

/*printf("stat.veh_busy_empty[%d] =
%f\n",veh_i_d,stat.veh_busy_empty[veh_i_d]);
printf("veh_alloc: i->time = %f, tnow = %f\n", i->time, tnow);
printf("Loading event pickup in veh_alloc\n");*/
      top = load(i, top);
      if (vehicle_no[veh_i_d].status == 5 || vehicle_no[veh_i_d].status
== 6)            stat.veh_idle[veh_i_d][1] += (tnow -
stat.veh_idle[veh_i_d][0]);
      vehicle_no[veh_i_d].status = 1;
      vehicle_no[veh_i_d].dest_pt = request;
      vehicle_no[veh_i_d].dest_time = i->time;
}


double dddist(a, b)
int a, b;
{
double dist1, dist;
```

```
double x1, x2, y1, y2, z;
x1 = number[b]->coord[0];
y1 = number[b]->coord[1];
x2 = number[a]->coord[0];
y2 = number[a]->coord[1];
if (x1 < x2)
   {
   z = x1;
   x1 = x2;
   x2 = z;
   }
if (y1 < y2)
   {
   z = y1;
   y1 = y2;
   y2 = z;
   }
/*printf("x1 = %f, y1 = %f, x2 = %f, y2 = %f\n", x1,y1,x2,y2);*/
dist1 = pow((x1 - x2), 2.0) + pow((y1 - y2), 2.0);
dist = sqrt(dist1);
return dist;
}
```

Machall3.c

```
#include <stdio.h>

mach_alloc(c)
int c;
{
extern int all_mach_busy;
extern float mach_busy[MACHINING][2];
extern struct ev_lst *top_mach;
int dest, x, storepath, e;
float time, ttime;
struct ev_lst *temp2;
dest = INFINITY;
time = NULL;
for(x = 0; x < MACHINING; x++) /* finds closest idle m/c */
   {
   if (machining_pt[x][0] && !machining_pt[x][1]) /* if m/c exists & is
free */
      {
/*printf("machining_pt[%d][1] = %d\n", x, machining_pt[x][1]);*/
/*printf("c = %d, machining_pt[%d][0] = %d\n", c, x,
machining_pt[x][0]);*/
       storepath = NULL;
       e = INFINITY;
       ttime = sshortp(c, machining_pt[x][0], storepath,e);
       if (!time || ttime < time)
          {
          time = ttime;
          dest = x;
          }
       }
   }

   if (dest != INFINITY) /* if some m/c was idle */
      {
      machining_pt[dest][1] = 1; /* set m/c to busy */
      mach_busy[dest][1] = tnow;
/*printf("mach_busy[%d][0] = %f, mach_busy[%d][1] = %f\n", dest,
mach_busy[dest][0], dest, mach_busy[dest][1]);*/
      dest = machining_pt[dest][0];
      }
   else /* load in load_mach queue for m/c to get free. */
      {
/*printf("All m/c's busy\n");*/
all_mach_busy += 1;
      temp2=(struct ev_lst *)malloc(sizeof(struct ev_lst));
      if (!temp2)
         {
         printf("Mem alloc fail for ev_lst in sim_run\n");
         exit(1);
         }
      temp2->which = c;
      temp2->veh = INFINITY;
      temp2->dest = INFINITY;
      temp2->time = tnow;
      top_mach = load_mach(temp2, top_mach);
      }
      return dest;
}
```

Ev_lst3.c

```
/* event_lst() holds a doubly linked list for storing event notices. */

#include <stdio.h>

/*On occurance of event notice the event list returns two variables.
  Type of event notice
      demand for vehicle at particular point
      demand for charging of particular vehicle
      arrival of vehicle at destination: includes download/transfer time
      arrival of vehicle to pickup part: model logic to get shortp
  Which point/vehicle
      name/index of point requiring vehicle
      name of vehicle needing to be charged
Manipulations of Event list
  Load event notice in correct place
  Delete a particular event notice
  Remove first event notice from event list and execute logic
  Advance clock to next event notice on list*/


/*struct ev_lst
            { int time; char type[9]; int which; int dest;
            struct ev_lst *prior;
            struct ev_lst *next;
            };*/


/* Load an event notice on to the event list. */
/* Create a doubly linked list in sorted order.
   A pointer to the first element is returned because
   it is possible that a new element will be inserted
   at the top of the list.                              */

struct ev_lst *load(i, top) /* store in sorted order */
struct ev_lst *i;  /* new element */
struct ev_lst *top;  /*first element in list */
{
struct ev_lst *old, *p;
extern struct ev_lst *last;

/*printf("Ev_lst: i->time = %f, i->which = %d, i->type = %s, i->from =
%d, i->veh = %d\n", i->time, i->which, i->type, i->from, i->veh);*/

if (last == NULL)  /* first element in list */
   {
   i->next = NULL;
   i->prior = NULL;
   last = i;
   /*printf("1st event\n");*/
qq(1, top);
   return i;
   }

p = top;  /* start at top of list */
old = NULL;
/*printf("p->time = %f, i->time = %f\n", p->time, i->time);*/
while(p)
     {
```

```
/*printf("p->time = %f, i->time = %f\n", p->time, i->time);*/
     if (p->time <= i->time) /* while time of new event is more    */
        {                    /* than time of event being looked at */
        old = p;
        p = p->next;
        }
     else
        {
        if (p->prior)
           {
           p->prior->next = i;
           i->next = p;
           i->prior = p->prior;
           p->prior = i;
qq(2, top);
           return top;
           }
        i->next = p;  /* new first element */
        i->prior = NULL;
        p->prior = i;
qq(3, top);
        return i;
        }
     }
     old->next = i;  /* put on end */
     i->next = NULL;
     i->prior = old;
     last = i;
qq(4, top);
     return top;
}


/* Delete the top element from a doubly linked list */
struct ev_lst *dldelete(top)
struct ev_lst *top;  /*first item in list */
{
struct ev_lst *i;
i = top;
top = i->next;
if (top)
   top->prior = NULL;
free(i);
return top;
}


qq(x, top)
struct ev_lst *top;
int x;
{
struct ev_lst *q;
/*printf("Loop%d\n", x);*/
q = top;
while(q)
     {
     /*printf("Ev_lst: q->time = %f, q->which = %d, q->dest = %d, q-
>type = %s, q->from = %d, q->veh = %d\n", q->time, q->which, q->dest, q-
>type, q->from, q->veh);*/
     q = q->next;
```

```
     }
}


struct ev_lst *load_veh(temp, top_veh) /* store in sorted order */
struct ev_lst *temp;  /* new element */
struct ev_lst *top_veh;  /*first element in list */
{
struct ev_lst *old, *p;
extern struct ev_lst *last_veh;

/*printf("Reached load_veh\n");*/
qqq(top_veh);
if (last_veh == NULL)  /* first element in list */
   {
   temp->next = NULL;
   temp->prior = NULL;
   last_veh = temp;
   return temp;
   }

p = top_veh;  /* start at top of list */
old = NULL;
/*printf("temp->time = %f, p->time = %f\n", temp->time, p->time);*/
while(p)
     {
     if (p->time < temp->time) /* while time of new event is more    */
        {                      /* than time of event being looked at */
        old = p;
        p = p->next;
        }
     else
        {
        if (p->prior)
           {
           p->prior->next = temp;
           temp->next = p;
           temp->prior = p->prior;
           p->prior = temp;
           return top_veh;
           }
        temp->next = p;  /* new first element */
        temp->prior = NULL;
        p->prior = temp;
        return temp;
        }
     }
     old->next = temp;  /* put on end */
     temp->next = NULL;
     temp->prior = old;
     last = temp;
     return top_veh;
}


struct ev_lst *delete_veh(top_veh)
struct ev_lst *top_veh;  /*first item in list */
{
struct ev_lst *temp;
temp = top_veh;
```

```
top_veh = temp->next;
if (top_veh)
   top_veh->prior = NULL;
else
   last_veh = NULL;
free(temp);
return top_veh;
}


qqq(top_veh)
struct ev_lst *top_veh;
{
struct ev_lst *tempveh;
tempveh = top_veh;
while(tempveh)
     {
     /*printf("Ev_lst: tempveh->time = %f, tempveh->which = %d, tempveh-
>dest = %d\n", tempveh->time, tempveh->which, tempveh->dest);*/
     tempveh = tempveh->next;
     }
}

struct ev_lst *load_mach(temp, top_mach) /* store in sorted order */
struct ev_lst *temp;  /* new element */
struct ev_lst *top_mach;  /*first element in list */
{
struct ev_lst *old, *p;
extern struct ev_lst *last_mach;

/*printf("Reached load_mach\n");*/
qqqq(top_mach);
if (last_mach == NULL)  /* first element in list */
   {
   temp->next = NULL;
   temp->prior = NULL;
   last_mach = temp;
   return temp;
   }

p = top_mach;  /* start at top of list */
old = NULL;
/*printf("temp->time = %f, p->time = %f\n", temp->time, p->time);*/
while(p)
     {
     if (p->time < temp->time) /* while time of new event is more    */
        {                      /* than time of event being looked at */
        old = p;
        p = p->next;
        }
     else
        {
        if (p->prior)
           {
           p->prior->next = temp;
           temp->next = p;
           temp->prior = p->prior;
           p->prior = temp;
           return top_mach;
           }
```

```
        temp->next = p;  /* new first element */
        temp->prior = NULL;
        p->prior = temp;
        return temp;
        }
     }
     old->next = temp;  /* put on end */
     temp->next = NULL;
     temp->prior = old;
     last = temp;
     return top_mach;
}


struct ev_lst *delete_mach(top_mach)
struct ev_lst *top_mach;  /*first item in list */
{
struct ev_lst *temp;
/*printf("Reached delete_mach\n");*/
temp = top_mach;
top_mach = temp->next;
if (top_mach)
   top_mach->prior = NULL;
else
   last_mach = NULL;
free(temp);
return top_mach;
}


qqqq(top_mach)
struct ev_lst *top_mach;
{
struct ev_lst *tempmach;
tempmach = top_mach;
while(tempmach)
     {
     /*printf("Ev_lst: tempmach->time = %f, tempmach->which = %d,
tempmach->dest = %d\n", tempmach->time, tempmach->which, tempmach-
>dest);*/
     tempmach = tempmach->next;
     }
}
```

Rand3.c

```
/* rand.c contains various random number generators. */

double ran3(), normal(), erlang(), weibull();
double gaussian(), gam1(), gam2(), gamma();
double expon(), beta(), beta1();
double triang(), triang1(), lognormal();
int poisson();
int bernoulli(), binomial();


/*  random deviate generation  */


#define MBIG 1000000000
#define MSEED 161803398
#define MZ 0
#define FAC (1.0/MBIG)
#define PI 3.141592654
#define E 2.718281828

#define NSEEDS 3
int seed[NSEEDS] = {-31, -663, -345};
int riff[NSEEDS] = {0,0,0};



float ran1()
{
static long int a = 100001;

a = (a*125) % 2796203;
return (float) a / 2796203;
}


float ran2()
{
static long int a = 1;

a = (a*100001 + 3) % 1717;
return (float) a / 1717;
}


float ran4()
{
static long int a = 203;

a = (a*10001 + 3) % 1717;
return (float) a / 1717;
}


float random()
{
float f;

f = ran4();
```

```
if (f > .5) return ran1();
else return ran2();
}


int poisson(n, mean)
int n;          /*  r.n. stream          */
int mean;   /*  mean (num expected) */
{
   double a, b;
   int i;
   double u_ip1;

   a = exp( - (double) mean );
   b = 1.0;
   i = 0;

   while(1)
       {
       u_ip1 = ran3(n);
       b *= u_ip1;
       if ( b < a )
          return i;
       else
          i++;
       }
}



uniform(n, lo, hi)
int n, lo, hi;
{
return (lo + (hi - lo)*(ran3(n)));
}



double expon(n, beta) /* error returns -Inf */
int n;
int beta;  /*  mean  */
{
   return ( -((float) beta)*log(ran3(n)) );
}



double gaussian(n) /* error returns both negative and positive no's
around 1 */
int n;
{
   static int iset = 0;
   static double gset;
   double fac, r, v1, v2;

   if (iset == 0)
      {
      do {
         v1 = 2.0 * ran3(n) - 1.0;
```

```
            v2 = 2.0 * ran3(n) - 1.0;
            r = v1*v1 + v2*v2;
            } while (r >= 1.0);
         fac = sqrt(-2.0*log(r)/r );
         gset = fac * v1;
         iset = 1;
         return (fac * v2);
         }
      else
         {
         iset = 0;
         return gset;
         }
}


double normal(n, mean, stdev)
int n;
int mean;
int stdev;
{
    return ( ((float) mean) + gaussian(n) * ((float) stdev));
}


double lognormal(n, mean, stdev)
int n;
double mean;
double stdev;
{
    return (exp( normal(n, mean, stdev)));
}


double erlang(n, beta, m)
int n;
double beta;     /*  mean    */
int m;          /*  number of exponential samples  */
{
   /*  implementation of m-Erlang random deviate  */
   /*  generator as described in Law and Kelton   */
   /*  on page 254                                */

   double u2 = 1.0;
   int i;

   for (i=1; i<=m; i++)
      u2 *= ran3(n);

   return (-beta/m)*log(u2);
}



double ran3(n)
int n;
{
/* Returns a uniform random deviate between 0.0 and 1.0. Set idum to */
/* any negative value to initialize or reinitialize the sequence.    */
   static int inext[NSEEDS], inextp[NSEEDS];
```

```
    static long ma[NSEEDS][56];
    long mj, mk;
    int i, ii, k;
    int *idum;

    idum = &seed[n];
    if (*idum < 0 || riff[n] == 0)
       {
       riff[n] = 1;
       mj = MSEED - (*idum < 0 ? -*idum : *idum);
       mj %= MBIG;
       ma[n][55] = mj;
       mk = 1;
       for (i=1; i<=54; i++)
          {
          ii = (21 * i) % 55;
          ma[n][ii] = mk;
          mk = mj - mk;
          if (mk < MZ)
             mk += MBIG;
          mj = ma[n][ii];
          }
       for (k=1; k<=4; k++)
          for (i=1; i<=55; i++)
             {
             ma[n][i] -= ma[n][1+(i+30) % 55];
             if (ma[n][i] < MZ)
                ma[n][i] += MBIG;
             }
       inext[n] = 0;
       inextp[n] = 31;
       *idum = 1;
       }
    if (++inext[n] == 56)
       inext[n] = 1;

    if (++inextp[n] == 56)
       inextp[n] = 1;
    mj = ma[n][inext[n]] - ma[n][inextp[n]];
    if (mj < MZ)
       mj += MBIG;
    ma[n][inext[n]] = mj;
    return mj*FAC;
}

 double triang1(n, c) /* 'c' varies between 0-1 */
int n;
double c;  /*  mode in a [0,1] triangular dist.  */
{
    /*  This function generates a triang(0, 1, c) random deviate   */
    /*  according to the procedure in Law and Kelton on page 261.  */

    double u;

    u = ran3(n);
    if ( u <= c )
       return sqrt(c*u);
    else
       return (1.0 - sqrt((1.- c)*(1.- u)) );
}
```

```
double triang(n, lo, hi, mid)
int n;
int lo;
int hi;
int mid;
{
   double d;

   d = (mid-lo)/(hi-lo);
   return ( ((float) lo) + (((float) hi) - ((float) lo))*triang1(n, d)
);
}

/*double gam1(n, alpha)
int n;
double alpha;   /*  shape parameter  *//*
{
   /*  This function generates a gamma(alpha, 1) random   */
   /*  deviate for the case where alpha < 1 as described  */
   /*  in Law and Kelton (pp. 255-258).  Note that alpha  */
   /*  is assumed to be greater then 0                    */

   /* double u1, u2, b, p, y;

   b = (E + alpha)/E;
   while(1)
      {
      u1 = ran3(n);
      p = b * u1;

      if ( p > 1.0 )
        {
        y = -log((b-p)/alpha);
        u2 = ran3(n);
        if (u2 <= pow(y, (alpha - 1.0)) )
           return y;
        }
      else  /*  p is le 1.0  *//*
        {
        y = pow(p, (1.0/alpha));
        u2 = ran3(n);
        if ( u2 <= exp(-y) )
           return y;
        }
      }


double gam2(n, alpha)
int n;
double alpha;  /*  shape parameter  */
/*{
   /*  This function generates a gamma(alpha, 1) random   */
   /*  deviate for the case where alpha > 1 as described  */
   /*  in Law and Kelton (pp. 255-258)                    */

/*   double a, b, q, th, d, u1, u2, v, y, z, w;
```

```
    a = pow( (2.0*alpha - 1.0), -0.5 );
    b = alpha - log(4.0);
    q = alpha + 1.0/alpha;
    th = 4.5;
    d = 1.0 + log(th);

    while(1)
       {
       u1 = ran3(n);
       u2 = ran3(n);

       v = a * log( u1/(1-u1) );
       y = alpha * exp(v);
       z = u1*u1*u2;
       w = b + q*v - y;

       if ( (w + d - th*z) >= 0.0 )
          return y;
       if ( w >= log(z) )
          return y;
       }
}


*double gamma(n, alpha, beta)
int n;
double alpha;    /*  shape parameter  *//*
double beta;     /*  scale parameter  */
/*{
   /*  This function generates a gamma(alpha, beta)      */
   /*  random deviate for alpha > 0 and beta > 0.        */
   /*  This function calls either gam1, gam2, or expon   */
   /*  depending upon the value of alpha.                */

/*    if ( alpha < 1.0 )
      return (beta * gam1(n, alpha));
    else if (alpha > 1.0 )
      return (beta * gam2(n, alpha));
    else  /*  alpha == 1.0  *//*
      return (beta * expon(n, 1.));

}*/


/* double beta1(n, alpha1, alpha2)
intn;
double alpha1;  /* shape parameter 1 *//*
double alpha2;  /* shape parameter 2 *//*
{
   /*  This function generates a beta(alpha1, alpha2)  */
   /*  random deviate on the interval [0,1] according  */
   /*  to the procedure on page 260 of Law and Kelton. */

/*    double y1, y2;

   y1 = gamma(n, alpha1, 1.0);
   y2 = gamm(n, alpha2, 1.0);

   return (y1/y1 + y2));
```

```
}


double beta(n, a, b, alpha1, alpha2)
int n;
double a;     /*  beginning of interval  *//*
double b;     /*  end of interval        *//*
double alpha1;    /*  shape parameter 1  *//*
double alpha2;    /*  shape parameter 2  *//*
{
   /*  This function generates a beta(alpha1, alpha2)    */
   /*  random deviate on the interval [a,b] according    */
   /*  to the procedure on page 260 of Law and Kelton.   *//*

   double x;

   x = beta1(n, alpha1, alpha2);
   return (a + (b-a)*x);
}*/


/*double weibull(n, alpha, beta)
double alpha;     /*  shape parameter  *//*
double beta;      /*  scale parameter  *//*
{
   return pow( (-log(ran3(n))), (1./alpha) ) * beta;
} */
```

```
/* struct.c h/


#define MAX 2600 /* Indicates no. of points which may be entered. */
#define PATH_NO 3 /* Indicates no. of paths which may be calculated. */
#define SIZE 3  /* Size indicates length of point name. */
#define CONNECT 4 /* Indicates no. of connecting points possible. */
#define INFINITY 9999
#define LIST 99 /* Indicates max. no. of points on path. */
#define SEGM_NO 99 /* Indicates max no. of segments on path. */
#define VEHICLE_NO 17 /* Indicates max no. of vehicles. */
#define MAX_TIMENO 3 /* Max. no. of times which will be calculated. */
#define MAX_VEL 5 /* Max. velocity of vehicle */
#define ACC 2.5 /* acceleration of vehicle */
#define DEST 9 /* max. no. of points which can be dest. of that decision
pt. */
#define REQUEST 9 /* max. no of request/workstation points */
#define MACHINING 9 /* max. no. of machining workcells */
#define CHARGING 3 /* max. no of charging points */
#define PARKING 3 /* max. no of parking points */
#define MAX_DEST 9 /* max. no of destinations veh. can go to from any
pt. */
#define MAX_DESTT MAX_DEST + 1
#define DED 9 /* max. no. of points which can be serviced by one veh. */

int request_pt[REQUEST]; /* stores index no of request/workstation pt's
*/
int machining_pt[MACHINING][2]; /* [0]=index_no, [1]=status (busy=1,
free=0) */
int charging[CHARGING]; /* stores index no of charging pt's */
int parking[PARKING]; /* stores index no of parking pt's */
int request_veh[REQUEST][VEHICLE_NO];     /* indicates which veh's can
*/
int machining_veh[MACHINING][VEHICLE_NO]; /* answer this pt's requests
*/
int all_veh_busy;

struct vehicle /* From vehicle.c */
             {
             int status; /* Busy: unloaded=1/loaded=2, parking=3 */
                         /* charging=4, idle=5 , waiting=6 at dropoff
pt. */
             int dest_pt;   /* dest_pt & dest_time, ie which pt. it's
headed */
             float dest_time;        /* for & what time it will reach
there. */
             int ded[DED]; /* which if any dedicated points does it
service. */
             } vehicle_no[VEHICLE_NO];


struct random
            {
            int ran_no[4];
            int multiple; /* 0: single dest.; 1: alternative multiple
dest. */
            struct random *prior; /* 2: multiple dest., with own random
no  */
            struct random *next;
```

```
            };


struct time
          {
          float occup_begin; /* Time veh. gets to point at start of
segment. */
          float occup_end;   /* Time veh. leaves point at end of
segment. */
          struct time *next;
          struct time *prior;
          };


struct matrix                          /* Material Flow Matrix. */
            {
            char name[3];              /* Name of decision pt. eg, A1, B34
*/
            float coord[2];            /* Stores x,y coordinates of point */
            char type[9];              /* Type of decision pt. */
            int dest[MAX_DESTT]; /* points it will travel to from this
point. */
            struct random *rand; /* Stores random no. linked list & data
*/
            float c_dist[CONNECT];   /* array showing dist of connecting
pts */
            int c_index[CONNECT];  /* array showing index of connecting
pt.*/
            float c_coord[CONNECT][2];/* array showing coord of connec
pt. */
            char c_name[CONNECT][3];/* array showing name of connecting
pt. */
            struct time *occup_time;
            int ded;
            } *number[MAX];


struct mapname /* From shortp.c */
       {
       /*int map; /* What is map, do I need it. */
       char *name; /* Name of decision point. */
       int visited; /* Indicates whether point has been visited (=0) or
not. */
       float sdist[PATH_NO]; /* For shortest dist between two points,
for a */
       } *agvs[MAX];       /* particular shortp path == PATH_NO
*/


struct nodeparent /* Used for storing ancestors on a path. */
       { /* From shortp.c */
       int parent[PATH_NO];
       } spath[MAX];
```

```
struct ev_lst /* From ev_lst.c */
            {
            float time; /* time at which event will occur */
            char type[9]; /* type of event */
            int which; /* point at/for which event notice occurs. */
            int dest;
            int veh; /* which vehicle */
            int from; /* where it's coming from */
            struct ev_lst *prior;
            struct ev_lst *next;
            };


struct path    /* To store data on shortest path calculations. This */
          {    /* may not necessarily be the final shortest path.   */
          int point;    /* Stores index number of point. */
          int vel;      /* Velocity at that decision point. */
          float atime;  /* Arrival time at that decision point. */
          float dtime;  /* Departure time from that decision point. */
          } *pfinal[PATH_NO][SEGM_NO]; /* From time.c */



struct statistics
                {                                   /* [][0] = last time idle
*/
                float veh_idle[VEHICLE_NO][2]; /* [][1] = total time
idle */
                float veh_blocked[VEHICLE_NO];
                float veh_busy_empty[VEHICLE_NO]; /* waite */
                float veh_busy_loaded[VEHICLE_NO];
                float veh_parking[VEHICLE_NO];
                float raw_part_waiting[REQUEST][2];/*av time pt. waits
for veh*/
                float fin_part_waiting[MACHINING][2];/* [][0]=av. time
*/
                float pt_blocked[MAX];                /* [][1]=no. of
times */
                } stat;


void init_number()           /* Initializes each element in pointer
array  */
{                            /* number[MAX] to NULL, indicating that
*/
register int t;              /* there is no entry in that location.
*/
for(t = 0; t < VEHICLE_NO; ++t)
   {
   number[t] = NULL;
   }
}

void init_vehicle()
{
register int t;
for(t = 0; t < VEHICLE_NO; ++t)
```

```
   vehicle_no[t].dest_pt = NULL;
}


void init_stat()
{
register int t;
for(t = 0; t < VEHICLE_NO; ++t)
   {
   stat.veh_idle[t][0] = NULL;
   stat.veh_idle[t][1] = NULL;
   stat.veh_blocked[t] = NULL;
   stat.veh_busy_empty[t] = NULL;
   stat.veh_busy_loaded[t] = NULL;
   }
for(t = 0; t < REQUEST; ++t)
   {
   stat.raw_part_waiting[t][0] = NULL;
   stat.raw_part_waiting[t][1] = NULL;
   }
for(t = 0; t < MACHINING; ++t)
   {
   stat.fin_part_waiting[t][0] = NULL;
   stat.fin_part_waiting[t][1] = NULL;
   }
for(t = 0; t<MAX; ++t)
   stat.pt_blocked[t] = NULL;
}
```