# Daily Activity and Multimodal Travel Planner: Phase II Final Report

**Ryuichi Kitamura**
**Cynthia Chen**
**Jiayu Chen**

**California PATH Research Report**
**UCB-ITS-PRR-99-1**

This work was performed as part of the California PATH Program of the University of California, in cooperation with the State of California Business, Transportation, and Housing Agency, Department of Transportation; and the United States Department of Transportation, Federal Highway Administration.

Report for MOU 339

January 1999

ISSN 1055-1425

# Daily Activity and Multimodal Travel Planner

Phase II Final Report

Submitted by

Ryuichi Kitamura
Institute for Transportation Studies
Department of Civil and Environmental Engineering
University of California
Davis, CA 95616

Cynthia Chen
Institute for Transportation Studies
Department of Civil and Environmental Engineering
University of California
Davis, CA 95616

Jiayu Chen
Institute for Transportation Studies
Department of Civil and Environmental Engineering
University of California
Davis, CA 95616

January, 1999

## Abstract

It is important that our travel be organized in an efficient way. One way to achieve this is to provide travelers with a trip planner that produces efficient travel itineraries for them. It is desired that a trip planner possess the following features in order to be useful: be able to handle multiple destinations, multiple constraints, and multiple modes; and be able to adjust travelers' preferences under different circumstances. These features cannot be found in existing trip planners.

The goal is then to develop an "Itinerary Planner" that possesses all of these features. The Itinerary Planner attempts to identify the most desirable itinerary from among all feasible alternatives. The desirability of an itinerary is measured by an objective function, which is defined a weighted sum of seven criteria (i.e., attributes of the itinerary), The weights represent travelers' preferences to the attributes (e.g., total travel time and monetary cost). The Itinerary Planner first uses initial values of the preference weights established in a previous study in the Bay Area. After a series of operations, the Planner selects the two "best" itineraries which have the best and the second best objective function values. Then, the Planner presents the selected itineraries to the traveler. If the traveler is not satisfied with either itinerary, the Planner asks the traveler to indicate the itinerary that they prefer to the other. Based on the selection made by the user, the Planner updates the preference weights and re-selects another two itineraries. This process is repeated until the traveler is satisfied with one of the selected itineraries.

A prototype is developed for downtown San Francisco. It is demonstrated that the Planner is capable of effectively generating alternative itineraries for a tour that involves multiple trips and multiple modes, with complex constraints, and that the Planner prototype serves as a practical tool for travelers in itinerary planning.

## Executive Summary

### 1. Introduction

Since our movement is restricted by the amount of available time and the travel speed, it is important that our travel be efficiently organized such that the time resource can be best utilized to engage in activities in an efficient manner. One way to achieve this goal is by developing efficient travel itineraries. The objective of this project is to develop such a tool that can assist the traveler in developing an efficient itinerary in which multiple locations can be visited with minimal waste.

Also in the scope of this project is the development of an information system that will aid the traveler in using public transit in a complex tour in which multiple locations are visited. Underlying this are the beliefs that the availability of information affects the decision to use public transit, and that people will make complex tours by public transit if they are shown that it is possible and convenient to do so.

These considerations have motivated the "Itinerary Planner," computer software that assists the traveler by proposing to him/her efficient itineraries for visiting multiple locations using alternative travel modes. Given the set of locations the traveler wishes to visit and the constraints associated with the visits, the Itinerary Planner develops alternative itineraries for the visits interactively with the traveler, or, the user. The planner presents alternative itineraries to the user, and the user indicates the Planner which itinerary is more preferable. The planner in turn takes the feedback from the user and updates its objective function to better reflect the user's preferences, then generates another set of itineraries. This process is iterated until a satisfactory itinerary is found.

### 2. Outline of the Travel Itinerary Planner

The Itinerary Planner is designed to construct an itinerary that is suitable for engaging in a series of activities at different locations. It is a multi-modal planning package with both highway and transit data bases, and accounts for users' preferences and constraints as it interactively builds an itinerary. An *itinerary* here comprises a set of sequenced *activity locations* to visit, *trips* which connect these locations.

The Itinerary Planner considers a set of trip attributes (e.g., travel time and cost, waiting time etc.) while identifying the best itinerary for the user. Quite often an optimal itinerary that is superior to others in all attributes does not exist. For instance, an itinerary with a minimal travel time may have a large monetary cost. The user is naturally interested in an itinerary that has the best combination of trip attributes. An Interactive Programming Method (IPM) is used in the Planner in locating such an itinerary. The IPM brings the user into the decision making process, and updates the objective function based on inputs from the user.

The Planner consists of three components: User Inputs, Planner Executor, and System Outputs. User Inputs include information from the user on locations to visit and constraints associated with the visits. The input information is fed to the Planner Executor, which analyzes the input information, evaluates all possible itineraries, and computes preference scores using initial preference weights for the respective trip attributes. Two alternative itineraries with highest and second-highest preference scores are then selected by the Planner and presented to the user. This constitutes the outputs to the user. The user is asked to indicate which of the two itineraries he/she prefers more, and whether the preferred itinerary is satisfactory or not. If the user is not satisfied with either of the two, the Planner then updates the preference weights based on the user's choice between the two itineraries, and evaluates all possible itineraries again using updated weights. The process is repeated until the Planner finds an itinerary that is satisfactory to the user.

## 3. Planner Algorithms

At a very first glance the Planner problem may look like a typical Traveling Salesman Problem (TSP), where an optimal route to visit a given set of destinations is sought. Yet the Planner is not exactly a typical TSP problem. Many difficulties would arise if we applied algorithms used to solve TSP problems to the Planner problem because of the various types of constraints associated with the Planner problem. In addition, it is desired that the Planner can adapt to users with different preferences to trip attributes.

The IPM method adopted by the Planner to accommodate users of different preferences towards attributes of travel itineraries (e.g., total travel time, monetary cost, and walk distance). The method comprises a procedure to update the user’s preference weights and an algorithm to locate an optimum itinerary using the weights. Different users possess different weights, and the same user may have varying preferences under different circumstances. The initial weights are updated to represent a particular user's preferences under a particular circumstance in the Planner.

The user's preference function is formulated as linear in parameters; the total utility a user derives from an itinerary is a weighted sum of the set of its attributes. The attributes incorporated in the current version of the Planner include: travel time by taxi, travel time by transit, travel time by cable car, travel cost, walk time, number of transfers and waiting time.

A simple and yet efficient scheme to update users’ preferences is devised and adopted in this study. Using initial weights that were established from an earlier mode choice study in the San Francisco Bay Area, the Planner first selects two alternative itineraries with the highest and second-highest values. Then, the Planner asks the user to select the one they would prefer more. The Planner then compares the value of each attribute between these two alternatives. The Planner adjusts the preference weights by doubling the coefficient if the attribute value of the preferred itinerary is less than that for the other route, and halving it if the attribute value for the preferred itinerary is greater than that for the other itinerary.

## 4. Planner Prototype Development

The Planner Prototype consists of three components: user inputs, system outputs and planner executor.

### *Assumptions*

The prototype is developed as a public-access terminal. The source (the beginning location) of the itinerary, therefore, is one of such terminals. In the prototype, the user can choose as the source of the itinerary one of the eight fixed public kiosks located throughout the study area. Downtown San Francisco is selected as the study area. Ten transit lines run in the study area.

The first assumption is that only a single travel mode is used to travel between two consecutive activity locations. The second major assumption is that all visits take place at places whose exact geographical locations are specified by the user ("hard locations"). The user is free to choose any point on the map as activity locations to visit.

### *User Inputs*

At the onset of the Planner execution, the Planner presents the user with a map of the downtown San Francisco area on which eight kiosks are shown. The Planner prompts the user to input a set of variables. These variables are essential for the search of desirable itineraries. The variables are inputted in the following order:

- Source (starting point of an itinerary)
- Attributes of a visit (repeated as necessary)
    - Location
    - Timing constraints
    - Duration constraints
- Sink (ending point of an itinerary)
- Sequencing constraints
- Mode preference

***System Outputs***

Outputs to the user consist of two highlighted itineraries selected by the Planner. As noted earlier, an itinerary can be made in combination of more than one mode; different travel modes are designated on the map by different colors. Written directions are also provided, guiding the user in following the itinerary. For taxi, automobile or walking, written directions indicate what mode to take on which street for how long. For public transit, written directions indicate which transit line to take, the names of stops to board/alight, and to transfer between transit lines

***Planner Executor***

Figure 1, the flow chart of the Planner Executor, illustrates how the Planner Prototype functions. Step 1 is the user inputs where the user supplies information on locations to visit, constraints associated with each visit, mode preferences, etc. In Step 2, the Planner locates the nearest nodes to the activity locations selected by the user and finds a minimum path between every possible pair of activity locations and stores the results in a temporary database. In Step 3 through Step 5, the Planner enumerates all possible sequences, computes the arrival time and departure time at each visit, and checks the feasibility of every sequence against the timing, duration and sequence constraints specified by the user. In Step 6, the Planner evaluates the generalized cost of every feasible sequence and selects two itineraries with lowest and second-lowest values. If the user is not satisfied with either itinerary, he/she is then asked to indicate which is more preferred. The Planner then takes the user's input, updates the preference weights and repeats operations from Step 2. If the user is satisfied with one of the itineraries, the Planner ceases operation and goes to “END”.

## 5. Conclusions

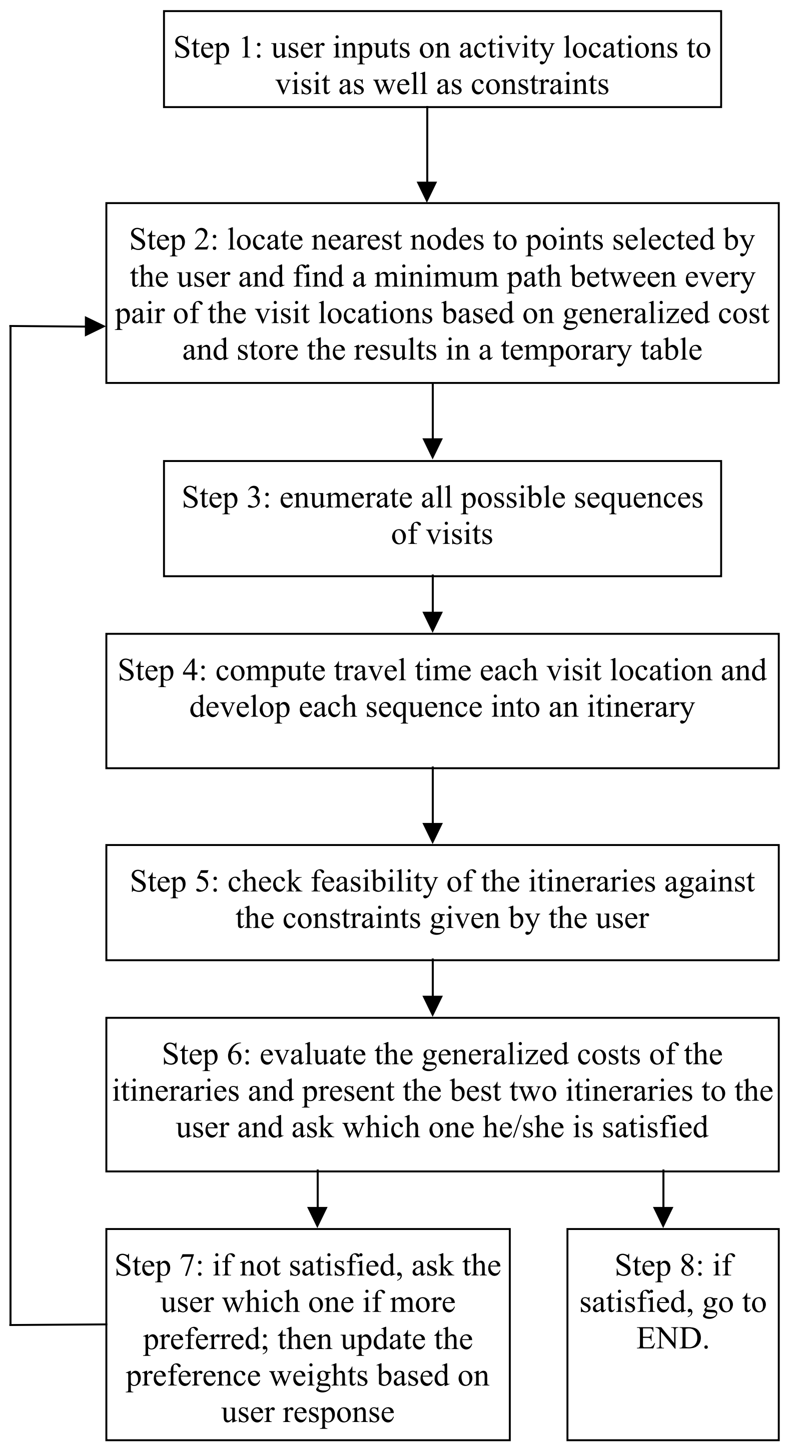
The Itinerary Planner developed so far is one of the first itinerary planners that incorporate multiple destinations, multiple criteria and constraints, and multiple modes of travel. It is also one of the first itinerary planner that brings the user into the decision making process and employs a preference function that updates users’ preferences interactively.

The Planner prototype is developed in a small test area: downtown San Francisco. Being one of the first multi-mode, multi-trip travel planning aids, the prototype is subject several limitations. The effectiveness of the algorithm adopted to update preference weights is yet to be examined. Nevertheless, the Planner presents a new way to save

travel time, relieve traffic congestion and encourage transit use by helping travelers to organize their trips in an efficient manner.

**Keywords**
Trip planning, itinerary, multi-mode, traveling salesman problem (TSP), interactive programming method (IMP), preference function, preference weights, multi-criteria

**Figure 1: Flow chart of Planner Executor**


Step 1: user inputs on activity locations to visit as well as constraints
Step 2: locate nearest nodes to points selected by the user and find a minimum path between every pair of the visit locations based on generalized cost and store the results in a temporary table
Step 3: enumerate all possible sequences of visits
Step 4: compute travel time each visit location and develop each sequence into an itinerary
Step 5: check feasibility of the itineraries against the constraints given by the user
Step 6: evaluate the generalized costs of the itineraries and present the best two itineraries to the user and ask which one he/she is satisfied
Step 7: if not satisfied, ask the user which one if more preferred; then update the preference weights based on user response
Step 8: if satisfied, go to END.

## 1. Introduction

Travel constitutes an integral part of our daily life. Only by traveling are we able to engage in a variety of activities at different locations. Since the extension of our movement is restricted by the amount of time that is available and the speed with which we can travel, it is important that our travel be efficiently organized such that the time resource can be best utilized to engage in activities in an efficient manner. One approach to achieving this is to choose less congested and faster routes. The use of in-vehicle advanced traveler information systems (ATIS) for this purpose has been discussed extensively in the intelligent transportation systems (ITS) literature.

Little attention has been directed, on the other hand, to achieving the same goal by developing an efficient travel itinerary. This becomes important when a traveler visits a number of places in a tour. Examples include a delivery truck driver who delivers goods to multiple locations, or a tourist who wishes to visit a number of attractions. In these cases the traveler is interested in not only the best route connecting successive locations for visit, but also the best sequence of visiting the locations. The problem, however, is an extremely complex one to solve, whose solution may often be not obvious to the traveler. The objective of this project is to develop a tool that can assist the traveler in developing an efficient itinerary in which multiple locations can be visited with minimal waste.

Also in the scope of this project is the development of an information system that will aid the traveler in using public transit in a complex tour in which multiple locations are visited. Underlying this is the belief that the availability of information affects the decision to use public transit, and that people will make complex tours by public transit if they are shown that it is possible and convenient to do so (Abdel-Aty et al., 1996).

These considerations have led to the conception of the "Travel Itinerary Planner", a computer software that assists the traveler by proposing to him/her efficient itineraries for visiting multiple locations using alternative travel modes. Given the set of locations the traveler wishes to visit and the constraints associated with the visits, the Planner develops alternative itineraries for the visits interactively with the traveler, or, the user. The Planner presents alternative itineraries, and the user indicates to the Planner which itinerary is more preferable. The Planner in turn takes the feedback from the user and updates its objective function to better reflect the user's preferences. This process is iterated until a satisfactory itinerary is found.

The development of the Travel Itinerary Planner may consist of two stages: first it is developed as a public kiosk, as is done in this study, and then as a personal unit in the future. The inputs to and outputs from the Planner are identical in both stages. The critical difference is whether the Planner has memory or not. As a public kiosk, the Planner serves as a transportation guide for many anonymous users and it does not store information about each user's preferences. It simply refreshes itself every time a new user uses it. As a personal unit, on the other hand, the Planner can serve as a transportation guide for a specific user because it can be designed to store information about the user's preferences and past history of travel. The Planner would learn and tend to become increasingly intelligent over time. In other words, the more the user uses it, the less time the Planner should take to identify an itinerary that satisfies the user.

Using a combination of public transit, private auto and walking may be more attractive and efficient for the traveler for visiting a series of locations than using just one mode alone. For example, the traveler might drive to a park-and-ride facility to park the car and take the subway into the city to avoid traffic congestion, then visit several locations on foot. Or the traveler might take taxi to accomplish a time-critical task (e.g., to catch an express delivery service) and then walk back to the office. It is thus essential that the Travel Itinerary Planner should take multiple travel modes into consideration while developing a desirable itinerary for the user.

This is the final report of the Itinerary Planner project which reports on the development of a prototype Itinerary Planer. The prototype represents a first step of the development which is based on a set of assumptions. The principal assumptions adopted in the prototype development include: the Planner is a public kiosk such that the starting point (the "source") of the itinerary is one of the pre-selected locations in the study area; visit locations are exactly known (“hard locations"; see Section 2); and  single mode is used to travel between two consecutive activity locations (excluding access and egress walk to and from public transit). The report provides an overview of the algorithms that are adopted in the Planner, documents in detail how the prototype has been developed and how it functions, and points to the future research directions.

The report is organized as follows. Section 2 of the report provides an outline of the Itinerary Planner. Section 3 offers a brief review of existing trip planners developed by organizations in both public and private domains. Section 4 offers a literature review on the Traveling Salesman Problem (TSP) algorithms and proposes an alternative algorithm to be used in the Planer. Section 5 details the actual development of the Planner prototype, identifies existing problems and points out research directions in the next-stage developments. The conclusions of this study are presented in Section 6. The Planner prototype code is enclosed as Appendix.

# 2. Outline of the Travel Itinerary Planner

The Itinerary Planner is designed to construct a suitable itinerary for engaging in a series of activities at different locations. It is a multi-modal planning package with both highway and transit data bases, and accounts for the user's preferences and constraints as it interactively builds an itinerary. Presented in this section is the outline of the Planner. Section 2.1 formally and mathematically presents the problem with which the Planner deals. Section 2.2 offers an overall picture of how the Planner works as a whole. Following this in Sections 2.3 and 2.4 is a description of each of the major components of the Planner: inputs, outputs and the Planner Executor. The Planner Executor collects information on the user's preferences, updates the coefficients that represent user preferences, and assists the user in discovering the best compromise itinerary. In Section 2.4, issues involved in itinerary building are discussed; the concept of "interactive programming", which is the principal mechanism for itinerary development, is presented.

## 2.1. Problem Formulation

The Itinerary Planner is designed to construct an itinerary that is suitable for engaging in a series of activities at different locations. It is a multi-modal planning package with both highway and transit data bases, and accounts for users' preferences and constraints as it interactively builds an itinerary.

An *itinerary* here comprises a set of sequenced *activity locations* to visit, and *trips* which connect these locations. The duration of a visit at a location is called *sojourn* duration. There are two classes of locations for visit. The first class includes locations that are specified as an exact geographical location by the address, cross-roads, or the name associated with the location (e.g., a landmark). Such a location is referred to as a *hard location*. The second class includes locations that are specified just as a class of opportunities for activities, such as a grocery store. A location in this class is referred to as a *soft location*. Given the set of locations to visit, an anticipated duration and a set of constraints associated with each visit, the Planner determines the sequence of the visits and finds best ways of traveling among the locations for visit.

Attributes of a trip that can be considered by the Planner include[1]:

| | |
|---|---|
| Auto Trip: | route |
| | expected travel time |
| | travel distance |
| | departure time and expected arrival time |
| | specific features of the route (e.g., scenic route) |
| Transit Trip: | route (a transit line or lines) |

[1] Not all of these attributes have been included in the Planner prototype. Those features that are included in the Planner Prototype are discussed in Section 5 of this report.

| | |
|---|---|
| | expected travel time |
| | fare |
| | transit stops for initial boarding, transfer, and alighting |
| | number of transfers |
| | scheduled departure time or arrival time at each pertinent stop |
| | expected waiting time |
| | access and egress walking distance and time |
| | specific features of the trip |
| Taxi Trip: | expected travel time |
| | approximate fare |
| Walk Trip: | travel distance |
| | expected travel time |
| Bicycle Trip: | travel distance |
| | expected travel time |

Although it is desirable that an optimal itinerary be identified, difficulties may arise in doing so. Difficulties exist most probably in the process of identifying user preferences, and also in the computational requirements associated with itinerary optimization. The Planner therefore aims at building a suitable, rather than optimal, itinerary that is well acceptable to the user, while interacting with the user to identify his/her preferences in developing an itinerary. The user interface of the Planner includes components that facilitate the interaction with the user.

The Planner problem is not a classical optimization problem where one can set the first derivative equal to zero and calculate the values of decision variables at which the objective function is maximized. Instead, it is a combinatorial problem with unknown coefficients of the objective function, _ . Firstly, it is a combinatorial problem because it includes discrete elements such as visit locations and modes of travel, and the desirability of an itinerary depends on the sequence in which the locations are visited and what travel mode is used to reach each visit location. When the number of visit locations is large, there can be quite a large number of combinations of location sequences and travel modes which need be evaluated to locate a solution.

Secondly, the coefficients of the objective function of the problem are unknown. Since the optimization problem aims at identifying the most desirable itinerary, its objective function, which is a measure of the desirability of an itinerary, must reflect each user's preferences. These preferences are represented by coefficient vector _ , which is referred to in this report as "*preference weights*". The problem is that the preferences, and therefore the values of _ , are unknown simply because preferences vary from user to user and there is no way of determining what preferences a user has when the Planner commences its task of building itineraries for the user.

In sum, the difficulty is thus two-fold: (1) there is no procedure by which this combinatorial problem can be analytically solved, and (2) the coefficients of the objective function need to be identified as part of problem solving. The Planner conducts an exhaustive search of all feasible itineraries in dealing with the first point; as for the second point, the Planner extracts information from user inputs to continuously update estimates of _ , in a trial-and-error process.

## 2.2. System Components

The Planner consists of three components: User Inputs, Planner Executor, and System Outputs. In this section, these three components are outlined.

### 2.2.1. User Inputs

The input from the user defines a specific itinerary development problem. In this section, the input variables are defined.

The user inputs to the Planner comprise:

- the initial and final locations,
- the set of locations to be visited,
- anticipated duration of stay (sojourn duration) at each location, and constraints associated with the duration,
- constraints on the sequence of the visits,
- constraints associated with the timing of each visit, and
- available travel modes and preferences toward alternative modes.

Constraints on the sequence of the visits refer to such constraints as "location A must be visited before location B, or location A must be the last location to visit". There may be a lower or upper bound to the sojourn duration at a location, e.g., "must spend at least 30 minutes" or "cannot stay more than 1 hour at a location".

- *Activity Location*

As defined above, the location of a visit is considered as *hard* if it is a specific place where the activity is to take place; the location of a visit is considered *soft* if it is one of a set of alternative locations where the activity can be pursued. In case of hard locations, their exact geographical locations are to be provided by the user. Soft locations can be thought of as a set of opportunities such as grocery stores; the set of opportunities is generated by the Planner.

The Planner prototype described later in this report is set up to deal only with hard locations. The user can specify hard locations by either giving the exact address or clicking on the map. Except for the source of the tour, the user is free to select any points within the study area as activity locations to visit or as their final destinations ("sinks"). The tour is expected to start at one of the kiosks, i.e., the source is one of the kiosks.

- *Timing of the Visit*

Timing constraints refer to the time windows associated with a visit. When the user wishes to arrive at the *i-th* location no earlier than $a_i$ and no later than $b_i$, the time window of the visit is expressed as $(a_i, b_i)$. Complexity of timing constraints increases when the user wishes to arrive at and leave the *i-th* location respectively within certain intervals. In this case, the time windows can be expressed as $((c_i, d_i); (e_i, f_i))$; the arrival at, and the departure from, the *i-th* location can take place only during intervals of $(a_i ,b_i)$ and $(e_i ,f_i)$, respectively. Only time constraints with a single time window are dealt with in the Planner prototype. In other words, the earliest arrival time at, and the latest departure time from, each visit location can be specified.

- *Duration of the Visit*

The duration of a visit at the *i-th* location may be subject to upper or/and lower bounds: $(d_{li}, d_{ui})$. For instance, the user may want to shop for no less than 10 minutes $(d_{li})$ as the minimum duration at i-th location, but no more than 2 hours $(d_{ui})$ as the maximum duration at i-th location. Note that $(d_{li}, d_{ui})$ is different from $(a_i, b_i)$ or $((c_i, d_i); (e_i, f_i))$ described above; $(d_{li}, d_{ui})$ refers to time length, while the latter refers to the time of day.

- *Sequence of the Visits*

Constraints on the sequence of visits arise when one of the following happens: (i) visit *A* must be made before or after visit *B*, (ii) as a special case of (i), visit *A* must be the first or the last to be made, and (iii) there must be a certain number of visits to be made between visits *A* and *B.* Only the first type of sequence constraints is considered in the Planner prototype.

- *Mode Constraints*

Mode constraints may be static or dynamic. Static constraints refer to those that do not change along the course of an itinerary. For example, the user may have no access to a particular travel mode; or the user does not consider taking a certain mode, even though it is available. These constraints can be specified at the outset of an itinerary development session with the Planner. Dynamic constraints come into effect as results of various conditions that arise or vanish as an itinerary evolves. For example, if a trip is made outside the operating hours of public transit, then public transit is not available for the trip. Modal constraints may also lead to constraints on visits. For example, if the traveler parks a private automobile at a visit location, and travels to the next visit location on another mode (e.g., walk), the traveler must return to the former location to pick up the automobile. Stationary constraints are identified by the user, while dynamic constraints are identified and incorporated into the itinerary development process automatically by the Planner.

- *Beginning and Ending Points (Source and Sink)*

The user indicates when and where he/she wishes to start and end the entire tour. The only constraints associated with the beginning and ending points of a tour are concerned with the timing of the visit. The user is asked to enter earliest and latest departure times from the source, and earliest and latest arrival times at the sink.

### 2.2.2. System Outputs

The outputs to the user consist of two itineraries selected by the Planner. In addition to the visual display of these two itineraries, their attributes are also provided to the user upon request. Desired attributes include:

1. the sequence of the visits,
2. travel mode for each trip in the itinerary,
3. total travel cost associated with the itinerary, and
4. total walking time in the itinerary.

I*n addition*, the following is desired *if transit is taken,*

5. bus stop names and transfer locations,
6. number of transfers, and
7. waiting time.

### 2.2.3. Planner Executor

The Planner Executor synthesizes user inputs; enumerates all possible itineraries and examines their feasibility against the user-provided constraints; presents alternative itineraries to the user and asks the user to rank them; and updates preference weights, _
, based on the user response. During the interactive decision making process, the Planner searches and updates the preference weights to best represent the preference structure the user has. This process continues until the Planner discovers an itinerary that is acceptable to the user.

## 2.3. Conclusion

We note that an individual as a decision maker may not even consciously enumerate all the criteria he/she uses in the decision making process. It would also be reasonable to assume that the individual would in general not quantify his/her preferences when making a choice that involves trade-offs among multiple factors. Furthermore, the preferences of a given individual could vary from situation to situation. Yet he/she is often faced with a set of alternatives none of which is superior to the others in every aspect, and is forced to exercise complex trading off among the criteria in order to find a best compromise solution. The Planner is developed as an aid for the individual making this decision.

The decision making problem at hand can be characterized by:

(a) the multitude of attributes that affect the desirability of an alternative solution,
(b) our limited ability in identifying how these attributes affect an individual's decision making,
(c) heterogeneity across individuals in their preferences and decision making processes, and
(d) variability from situation to situation in preference and decision making process even for a given individual.

In light of these, it is suggested that finding a satisfactory itinerary should involve an iterative process in which the user is brought into the solution process and his/her preferences are quantified.

This multi-criteria, iterative and interactive approach adopts Traveling Salesman Problem (TSP) algorithms (discussed in Section 4) as part of the interactive solution procedure. The solution procedure can be outlined as:

1. Given a set of locations to visit, an exhaustive search for an optimum solution is performed by generating, based on the preference weights, the best feasible itinerary for each of all possible sequences of the visit locations.
2. A set of solutions is selected from among those generated in the previous step, including the optimum solution under the current preference weights, and presented to the user.
3. The user is asked to choose the most desirable itinerary from the set presented to him/her.
4. If the chosen itinerary is acceptable to the user, the procedure is terminated. Otherwise the preference weights are updated based on the choice made by the user, according to the procedure described in Section 4.3 of this report, and Steps 1 through 4 are repeated.

The Planner presents the user with an optimum itinerary according to the current preference weights along with alternative solutions (e.g., the fastest itinerary or least-costly itinerary).[2] The user's preferences are measured interactively based on his/her choice of an alternative from among the set presented. Further discussions can be found in Section 5 of this report.

[2] In the current version of the prototype, a second-best solution is presented.

## 3. Review of Existing Trip Planners

The idea of trip planner, i.e., a decision support tool for trip planning, is not new. In fact, many trip planners had existed prior to the development of the Itinerary Planner of this study. The question then is how unique the Itinerary Planner is in comparison to existing trip planners. This section offers a brief review of selected trip planners that exist and compares them with the Itinerary Planner developed in this study. The review evolves around the following set of questions that may reveal differences between the existing trip planners and the Itinerary Planner:

1. Is the planner designed for a single trip or for a series of trips?;
2. Is the planner designed for a single mode or for multiple modes?;
3. Is the planner based on a single criterion or multiple criteria?;
4. What types of constraints can be incorporated into the planner?;
5. How friendly is the user-interface of the planner?; and
6. Can the user modify travel plans produced by the trip planner?

Both public-sector organizations and private firms have developed trip planners that are now publicly available. Public-sector organizations include transit operators and transportation planning organizations. The ones developed by public-sector organizations are usually available free of charge, while those developed by private firms are often for sale. It is worthy to note that commercial trip planners are designed for vacation planning where the duration is more than one day while the Itinerary Planner is a daily activity travel planner.

The trip planners developed by public-sector organizations that are reviewed here include those from the Tri-county Metropolitan Transportation District of Oregon, the Santa Barbara Metropolitan Transportation District, Ventura County Transportation Commission (VCTC), Metro system in the Puget Sound region, and the Bay Area Transit Information Project. Private-sector trip planners are reviewed including AAA MAP 'N' GO 3.0 from DeLorme, TRIP PLANNER 98 from Microsoft Expedia, and TRIPMAKER DELUXE 1998 from Rand McNally. The trip planner developed by the Tri-county Metropolitan Transportation District of Oregon and those developed by AAA, Microsoft and Rand McNally are PC-based and can be installed in kiosks. Table 1 compares the Itinerary Planner prototype of this study to existing trip planners.

**Table 1. Features of Itinerary Planner Prototype as Compared to Existing Trip Planners**

| | TI Planner* | Planner 1 | Planner 2 | Planner 3 | Planner 4 | Planner 5 | Planner 6 | Planner 7 | Planner 8 |
|---|---|---|---|---|---|---|---|---|---|
| No. of criteria | Simultaneous multiple criteria (travel time, travel cost, waiting time, number of transfers, etc.) | N/A | N/A | One of the pre-specified criteria (fastest, fewest transfers, minimum walking) | N/A | N/A | One of the pre-specified criteria (fastest, shortest, most scenic) | One of the pre-specified criteria (fastest, shortest, most scenic, major roads) | One of the pre-specified criteria (fastest, shortest, most scenic) |
| Travel modes | Transit, auto, walk, taxi | Transit | Transit | Transit | Transit | Transit | Auto | Auto | Auto |
| Types of constraints | Sojourn duration, time of day, sequence | Time of day | N/A | Time of day | N/A | Time of day | Speed limit, time of day, no. of stops | Speed limit, time of day, no. of stops | Speed limit, time of day, no. of stops |
| User-friendliness | Average | Poor | Poor | Poor | Poor | Poor | Good | Good | Good |
| No. of days | Daily | Daily | Daily | Daily | Daily | Daily | Multiple | Multiple | Multiple |
| No. of trips | Up to 7 | Single | Single | Single | Single | Single | Multiple | Multiple | Multiple |
| Plan modification | Yes | No | No | No | No | No | Yes | Yes | Yes |

* The Itinerary Planner prototype of this study. Planner 1: By Tri-county Metropolitan Transportation District of Oregon. Planner 2: By Santa Barbara Metropolitan Transportation District. Planner 3: By Ventura County Transportation Commission. Planner 4 : By Metro system in the Puget Sound Region. Planner 5: By Bay Area Transit Information Project. Planner 6: AAA MAP 'N' GO 3.0 from DeLorme. Planner 7: TRIP PLANNER 98 from Microsoft Expedia. Planner 8: TRIPMAKER DELUXE 1998 from Rand McNally.

*Single Mode vs. Multiple Modes*

Interestingly enough, the public-sector trip planners and private-sector planners can be distinguished on the basis of the travel modes they incorporate. Public-sector trip planners deal exclusively with public transit, while private-sector planners deal exclusively with the automobile. None of the trip planners reviewed here offers the capability of developing a travel plan in which both public transit and the automobile are used. Among the public-sector trip planners reviewed, only the ones from VCTC and the Tri-county Metropolitan Transportation District of Oregon combine information from different transit lines (e.g., Line 20 and Line 30) and provide the user a trip itinerary, given origin and destination, that involves a transfer(s) between transit lines. All the other public transit planners offer schedules for different transit lines separately without integrating them. The Itinerary Planner of this study incorporates transfers between public transit lines and offers transfer instructions to the user.

*Single Criterion vs. Multiple Criteria*

Some existing trip planners (e.g., Planners 3, 6, 7, 8 in Table 1) provide the option of selecting a criterion to use from among a set of criteria. These planners, however, are capable of incorporating only one criterion at a time. Available criteria include: travel time, distance, scenic route or not, number of transfers, major roads, and walking distance.

*Incorporating Constraints*

Constraints are incorporated into the trip planners to varying degrees. Among the transit trip planners developed by public-sector organizations, only the planners from Tri-county Metropolitan Transportation District of Oregon, VCTC and the one by BART in the Bay Area Transit Information Project deal with temporal constraints. The VCTC planner asks the date when the user is planning to take the trip and desired departure and arrival times. With the BART system, the user indicates on what day the trip is taken (weekdays, Saturdays, or holidays and Sundays). The user may also ask the planner to present schedules of trains whose departure or arrival times fall within an interval.

Among the planners for automobile trips, TRIPMAKER DELUXE offers the most extensive options. TRIPMAKER DELUXE let the user specify how fast he/she wants to drive, in what hours to drive (e.g., between 10 A.M. to 3 P.M.) and how often and how long to stop and stretch the legs (e.g., for 20 minutes every 2 hours). The user can also set up a daily budget of travel expenses. TRIPMAKER DELUXE also deals with duration constraints. For example, the user can specify how much time he/she plans to spend at each of the places he/she wishes to visit on the way.

## *User-interface*

User-interface consists of inputs from the user and outputs to the user. It can be either text-based or map-based, or both. Overall, the commercial software packages sold in the market offer significantly better user-interface than those from public-sector organizations. The former also offer more extensive options and information for trip planning. For example, the three trip planners commercially available (AAA MAP 'N GO, TRIP PLANNER 98 and TRIPMAKER DELUXE 1998) all offer listings for lodging, restaurants and attractions at places the user plans to visit. They also make use of multimedia features and present information in ways that cannot be done by conventional media such as a guidebook or a map. For example, Microsoft's TRIP PLANNER 98 has several 360-degree moving images of tourist attractions including a breathtaking view of the 18-th century houses preserved on Philadephia's Society Hill. TRIPMAKER DELUXE 98 has video tours of popular destinations like the Everglades in South Florida, replete with pelicans and alligators.

The public-sector trip planners reviewed here are quite unexciting and sometimes not at all user-friendly. For example, the planner from VCTC accepts only text-based user input. To input the origin and destination of a trip, the user must know either the street address, or a landmark recognized by the system. The planners from Santa Barbara Metropolitan Transportation District and from Puget Sound Metro provide a map in which the entire region is divided into several sub-regions. Once the user chooses one region on the map, the selected sub-region is displayed and the transit routes passing through that sub-region are indicated by text.

The planner from the Bay Area Transit Information Project seems to be most user-unfriendly; its database contains information on all transit systems in the nine counties in the Metropolitan Transportation Commission (MTC) region. To get from an origin to a destination, the user must know which transit line he/she needs to take; the planner does not provide this information to the user. In this sense, what is provided by the Bay Area Transit Information Project is not exactly a trip planner. Then, the user can click on a transit line and find out the schedule. As for the BART system, it is worthy to note that the user can view the entire region that BART system covers and select origin stations and destination stations on a map.

## *Modifying Travel Plans*

For a variety of reasons, it may so happen that a travel plan generated by a trip planner may not be exactly what the user was anticipating. In such cases, it is important that the user can give instructions to the planner such that the travel plan can be revised. Only those commercially available trip planners allow the user to make adjustments on the travel plan. Adjustments can be made on the duration of stay, departure time, etc. After a few adjustments, it is likely that those planners will provide a satisfactory itinerary to the user.

*Single Trip vs. Multiple Trips*

Among all the trip planners reviewed, only those commercially available trip planners (Planners 6, 7, and 8 in Table 1) can produce travel itineraries linking multiple destinations. The public sector trip planners can only deal with a single trip connecting an origin and a destination.

*Summary*

Every trip planner has advantages and disadvantages and none of the planners reviewed here possesses all the desirable features to make an ideal trip planner.

There are a few features possessed by the Itinerary Planner of this study that are not found in the existing trip planners reviewed here. The first is being multi-modal; the commercial trip planners exclude public transit and the public-sector trip planners exclude automobile. The second is being multi-criteria. None of the existing trip planners is capable of dealing with more than one criteria at the same time; at best, the user is allowed to choose one criterion from a list of pre-selected criteria. The third is being adjusting preference weights to meet different users' needs at different times.

As Section 5 of this report will reveal, the Itinerary Planner prototype developed in this study is significantly better than the public sector trip planners in several aspects. The commercial trip planners are better than this Itinerary Planner prototype in their user-interface. Additionally, the commercial trip planners also provide lists of opportunities including lodging, restaurants, and tourist attractions. These features should be incorporated in the next stage of the Itinerary Planner development.

## 4. Planner Algorithms

In this section, the traveling salesman problem (TSP) is formally presented (Section 4.1), and algorithms for this class of problem are reviewed (Section 4.2). The review includes both exact and approximate algorithms. These algorithms, however, do not incorporate types of constraints that are critically important in the Itinerary Planner problem. The purpose of the section is to describe the differences between the Planner problem and the TSP, and to elaborate on reasons why TSP algorithms are not adequate for the Planner problem. In Section 4.3, the review turns to the literature pertinent to the constraining dimensions of the Itinerary Planner. The conclusion of the review on TSP algorithms follows in Section 4.4; the conclusion addresses the applicability of TSP algorithms to the Planner problem. In Section 4.5, the multiple-criteria nature of the Planner problem is discussed. The approach to multiple-criteria problems which is adopted in this study, the interactive programming method, is presented in Section 4.6.

### 4.1. Traveling Salesman Problem

If none of the constraints described in the previous section is present, and if all visits are at hard locations, the Itinerary Planner problem can be thought of as a typical Traveling Salesman Problem (TSP). The TSP problem is defined as follows:

> Consider a graph ($V, A$), where $V = \{v_1, v_2, ..., v_n\}$ is a vertex set, containing $n$ vertices, two of which are special nodes of origin and destination and the others represent locations to visit, and $A = \{(v_i, v_j)$: $i \neq j, v_i, v_j$ _ $V\}$ is an arc set. Associated with each arc, $(v_i, v_j)$, is a nonnegative cost, denoted by $c_{ij}$. The TSP problem can then be stated as follows: Given ($V, A$) and $C = \{c_{ij}\}$, find an optimal route from the origin to destination, covering every vertex in the network, with the least total cost.

When $c_{ij} = c_{ji}$, the problem is symmetrical; when $c_{ij}$ _ $c_{ji}$, the problem is asymmetrical. The review contained in this report only considers the asymmetrical problem because symmetrical problems can be treated as a special case of asymmetrical problems. In the rest of this report as in the previous secitons, the origin and destination of a tour to visit several locations shall be called "source" and "sink", respectively. The terms "origin" and "destination" shall always be used in association with a trip to avoid confusion.

The discussion that follows provides a brief review of recent algorithmic developments for the TSP. This review, however, is not intended to be a survey which performs an exhaustive review of all the TSP algorithms developed so far; interested readers are referred to Laporte (1992, 1993), Lawler (1985) and Reinelt (1991) for comprehensive reviews. Instead, this review focuses on those TSP algorithms that are relevant to the development of the Planner algorithm.

## 4.2. TSP Algorithms

TSP algorithms can be classified into two categories: exact algorithms and approximate algorithms. Exact algorithms locate the optimal solution by exhaustive search of all possibilities, while the approximate algorithms avoid exhaustive search and attempt to find a solution which is within a certain proximity of the optimal solution.

### 4.2.1. Exact Algorithms

For the exact algorithm, the problem is formulated as a mathematical programming problem.

(TSP1)

Minimize $\quad \underset{i?j}{} c_{ij} x_{ij}$

Subject to $\quad \overset{n}{\underset{j=1}{}} x_{ij} = 1 \quad (i = 1,2,\ldots,n) \qquad (1)$

$\overset{n}{\underset{i=1}{}} x_{ij} = 1 \quad (j = 1,2,\ldots,n) \qquad (2)$

$\underset{i\in S\ j\in S'}{} x_{ij} \ ?\ 1 \qquad (3)$

$x_{ij} \in \{0,1\} \quad (i, j = 1, 2,\ldots, n; i \ ?\ j) \qquad (4)$

where $x_{ij}$ is a binary variable, taking on the value of 1 if arc $(v_i, v_j)$ is used in the optimal solution, and 0 otherwise; and *S* and *S*' are partitions of *V*.

The solution to TSP1 should be a single tour covering all vertices in the network. Constraints (1) and (2) guarantee that every vertex is entered once and left once. Constraints (1) and (2) alone do not guarantee a single tour to be formed, however; in fact, more than one sub-tour, each containing less than *n* vertices, may be formed under constraints (1) and (2) alone.

Formation of more than one sub-tours is prevented with constraint (3). Suppose *S* and *S*' are mutually exclusive subsets of *V*, each of which contains less than n vertices, and suppose *S*' and *S* together form *V* (*S* _ *S*' = *V*). Without constraint (3), the solution to TSP1 could be two separate paths connecting vertices within *S* and *S*', respectively, but with no links connecting *S* and *S*'.

The TSP1 problem can not be solved analytically; it is a combinatorial optimization problem for which it is known that no analytical solutions exist. Hoffman and Wolfe (1985), described the nature of the problem as:

"...we are trying to minimize total distance, so the problem is one of the optimization; but we can not immediately employ the methods of differential calculus by setting derivatives to zero, because we are in a

> combinatorial situation: our choice is not over a continuum but over the set of all tours."

Hoffman and Wolfe (1985) noted that algorithms such as cutting planes, branch and bound, and dynamic programming have been used to find an exact optimal solution to TSP1.

A notable development is by Miller and Pekny (1991). The argument that Miller and Pekny made was that arcs with high $c_{ij}$ are far less likely to be considered in an optimal tour compared to those with low $c_{ij}$. Thus, without contaminating the optimal solution, the size of the problem can be reduced by removing those arcs with high $c_{ij}$. Based on this logic, arcs with $c_{ij}$ greater than a threshold value _ were initially removed from consideration. Miller and Pekny constructed a dual problem to the primal problem of TSP1 above; they demonstrated that under certain conditions involving _ , the optimal solution to the dual problem are equivalent to that to the primal problem where no _ is involved. Thus, whenever these conditions are met, the optimal solution is found with a reduced-sized dual problem. If the conditions are not met, the value of _ is increased; a new network with updated vertices and arcs are constructed. The conditions for obtaining the optimal solution to the dual problem are checked again. This process is applied repeatedly until the conditions for the equivalence between the dual and primal solutions are met.

Exact algorithms which involve exhaustive search of all possibilities are limited to problems with a relatively small number of vertices and, usually, with few constraints. Even when they are applicable, exact algorithms tend to require more computing time compared to approximate algorithms, which are described below.

### 4.2.2. Approximate Algorithms

Laporte (1993) classified heuristic algorithms into two categories: tour construction procedures which incorporate vertices step by step into a solution, and tour improvement procedures which first generate a feasible but not optimal solution and then improve the solution by repeatedly removing and adding vertices into the solution. Laporte (1993) noted that the best approach would be composite procedures which basically combine the tour construction and improvement procedures.

Potvin (1993) noted that the most successful composite procedures that have been developed included CCAO heuristics, GENIUS, and Lin-Kernighan procedures. These procedures have achieved heuristic solutions within about 2% of the optimal solution. Interested readers are referred to Golden et al. (1985), Gendreau et al. (1992), and Johnson (1990) for these procedures.

## 4.3. TSP with Constraints

The formulation defined in TSP1 above stands as a theoretical problem in the sense that there is no real-world problem as simple as TSP1. Real-world applications are formulated as variations of TSP1, with constraints unique to the situation. For instance, in a delivery problem, customers may impose some temporal constraints that delivery can be done only during certain periods of the day. This instance involves timing constraints. Another example may involve order constraints. When more than one emergency call comes in, the dispatcher will prioritize them and send an emergency response team in a descending order of the degree of emergency.

Some constraints are considered in both TSP and the Planner problem. Discussed in this section are those constraints pertinent to the Planner problem. Applying exact algorithms to problems with constraints is straightforward; therefore, the discussion of this section is more pertinent to approximate algorithms than to exact algorithms.

### 4.3.1. Temporal Constraints

Temporal constraints here refer to those associated with the timing of a visit. As noted earlier, the arrival time at, and the departure time from, a location may be constrained by an earliest (or latest) possible arrival time and a latest (or earliest) possible departure time; the arrival and departure times may be constrained by two intervals (i.e., an arrival and departure can be made only during specific time intervals, respectively); or combinations of these.

The first type of constraint, the visit to a location is constrained by a single time window, has been studied in the TSP field and is directly applicable to the Planner problem. The definition of the problem is essentially the same as TSP1, except that each vertex, $v_i$, is associated with a time window $(a_i, b_i)$, where $a_i$ is the earliest starting time for the visit and $b_i$ is latest ending time for it. In the second type of constraints, the visit to a location is constrained by two time windows, where both arrival and departure can be made only within a certain time interval. In most of the problems formulated, an arrival before an earliest arrival time is allowed, but one has to wait. In this case the objective of the problem is the minimization of not only the total travel time, but also the waiting time.

Both exact and approximate algorithms have been developed for the TSP with time windows. As Gendreau et al. (1995) noted, exact algorithms suffer from the relatively small size of the problems they can handle (e.g., up to 100 vertices) and requirements for tight time windows for good performance. As an alternative to exact algorithms, heuristic searches have been developed to solve TSPs with time windows. For a survey on algorithms for TSPs with time windows, the reader is referred to Solomon & Desrosiers (1988).

One of the successful methods developed is called "generalized insertion heuristics" procedure (Gendreau, 1995). This procedure consists of two steps: (1) construction of a feasible itinerary from the source to the sink, covering all vertices in the network, and (2) application of a post-optimization procedure. The post-optimization procedure starts with removing the vertex immediately after the source and re-inserting it at every possible place in the itinerary (see Gendreau, et. al, 1992, for detailed description of this algorithm). Whenever insertion is possible, the objective function is evaluated. If the objective function is improved with the insertion, the new itinerary is accepted; otherwise, it is discarded. Post-optimization procedure is repeated until it passes all vertices except the source and sink and no further improvements can be made.

The research in TSP algorithms with time windows has so far considered only one type of temporal constraints: a single time window representing the earliest arrival time and the latest departure time from a location of visit. This type of constraint is considered in the Planner. As noted earlier, the Planner in addition considers situations where the arrival at, and the departure from, a location are constrained by two time windows. To the authors' knowledge, no TSP algorithm has been developed to directly address this type of constraint.

### 4.3.2. Duration Constraints

The user may place constraints on the duration of the visit at a location. The constraint on the duration of the visit can be expressed by an interval: $(d_{li}, d_{ui})$, where $d_{li}$ is the minimum duration and $d_{ui}$ is the maximum duration of the visit at location $i$. When $d_{li} = d_{ui}$, the duration of the visit to location $i$ is fixed; when $d_{li} < d_{ui}$, it is flexible. No approximate algorithms, to the authors' knowledge, have been developed to solve the TSP problem with constraints on the duration of the visit. It is not difficult, though, to introduce duration constraints into exact algorithms as long as the required computing time is not excessive.

### 4.3.3. Sequence Constraints

In some of the network and distribution problems for which TSP algorithms have been developed, the sequence of visits is considered as one of the constraints. For instance, in repair job problems, there may be certain types of repair jobs requiring higher priority than other types; similar issues arise in emergency vehicle dispatching problem. These problems are termed as "clustered traveling salesman problems," where vertices (or, say, customers) are assigned to clusters $(V_0, V_1, \ldots, V_m)$, where $V_0$ contains a single vertex for the source. Visits to clusters must be made in the order of $0, 1, \ldots, m$.

Though both the Planner problem and the clustered TSP consider sequence constraints, differences exist. Firstly, in the Planner problem, not every vertex (location to visit) is assigned a prior order of visit. In other words, the sequence constraints in the Planner problem are not as restrictive as those in the clustered TSP. Consequently, the set of alternative itineraries is larger for the Planner problem than for a clustered TSP in which the same number of visits is considered. Secondly, clustered TSP algorithms require an order of visits to be specified before their execution. This may create difficulties for the Planner problem because the user may not always be aware of sequence requirements inherent in the set of visits. Clustered TSP algorithms would therefore require some revision to address sequence requirements of the Planner problem.

## 4.4. Applicability of Existing TSP Algorithms

When only one criterion, say travel time, is used in selecting a desired itinerary, the problem is the often encountered traveling salesman problem. In the course of designing the Planner, it was realized that more than one criterion may be adopted by the user when comparing and evaluating alternative itineraries. For example, the total travel time and monetary cost of travel associated with an itinerary may serve as evaluation criteria simultaneously. In the case of travel mode choice, a traveler would often choose a travel mode while considering the trade-off between travel time and monetary cost; a traveler may normally commute to work by bus because it takes longer but it is cheaper, but when he/she is late for an urgent meeting, he/she may choose to take a cab, which is faster but more expensive. Likewise, there are many criteria that a driver may adopt when choosing from among alternative routes, including: travel time, travel distance, fractions of roadways by class, number of left turns, number of traffic lights, esthetic aspects of the roadside, etc.

Even if there were one single objective to be achieved in the Planner problem, existing TSP algorithms do not seem to be suitable for immediate application to the Planner problem for several reasons. First is the multi-dimensionality of the constraints in the Planner problem. Most of the TSP algorithms reviewed here consider one-dimensional constraints; for instance, either time windows or *a priori* orders of visits, but not both. Secondly, there are constraints (such as one on the duration of a visit) that are considered by the Planner, but that have not yet been extensively studied for TSP. To tackle the types of problems addressed by the Planner, conventional TSP algorithms must be appropriately expanded.

## 4.5. Multiple-Criteria Programming Problem

The problem at hand cannot be expressed as an optimization problem based on a single criterion. Multiple criteria that a traveler might consider when selecting from among alternative itineraries must be properly represented by the Planner Executor. The approach adopted here is to combine a set of criteria (i.e., trip attributes) using preference weights. More specifically, a linear combination of selected trip attributes weighted by preference weights is evaluated as the generalized cost of travel on each network link by travel mode, and minimum paths are sought between visit locations using these generalized costs. Alternative itineraries are built by the Planner using these minimum paths between visit locations. The approach thus assumes linear compensation between any pair of trip attributes (e.g., travel time versus travel cost).

Given that the problem is formulated as a multiple-criteria programming problem, the task of articulating the user's preferences still remains. For this purpose, an interactive multiple objective mathematical programming (MOMP) procedure is adopted in this study. MOMP procedures

> "attempt to generate a best compromise solution by progressive articulation of preferences of a decision maker facing multiple criteria

> with complex tradeoffs. … The motivation has come from the increasing recognition of the multi-objective nature of decision problems and has been enhanced by the increasing power and accessibility of computers. … the goal is to create a computationally efficient decision aid which puts the information provided to and solicited from the decision maker (DM) in a form that is easy to understand and provide" (Aksoy et al., 1996).

For problems involving multiple objectives (or criteria) which are often conflicting with each other, a dominant solution[1] in general does not exist. Instead, a set of non-dominant solutions exists. The objective for the Planner, then, is to aid its user in finding the most desirable compromise solution. Without input from the user, however, the Planner is not able to distinguish the value (to the user) of one solution from that of another in the solution set. The Planner must solicit preference information from the user, identify the preference structure the user has, and find that solution which maximizes the user's preference function. One approach which facilitates this is termed "interactive programming method." The interactive programming method proposed for the Planner is described in the rest of this section.

## 4.6. Interactive Programming Method

The discussions of this section is divided into three parts. Firstly, the interactive programming approach adopted for the Planner is presented. Secondly, the method of evaluating alternative itineraries while reflecting the user preferences is described. Lastly, the method adopted by the Planner to update the weights that represent the user's preferences to respective attributes of the itinerary is described. These three questions facilitate the implementation of the Interactive Programming Method in the Itinerary Planner Prototype directly.

### 4.6.1. Interactive Programming Approach

The interactive programming approach adopted by the Planner takes on the following form: the Planner presents alternative itineraries (which are intermediate solutions) to the user, who in turn indicates which itinerary is most desirable; the Planner updates the user's preference weights based on the indicated choice, and generates another set of alternative itineraries. The preference weights define the utility of a solution itinerary and are elements of the objective function. Thus the objective function is interactively adjusted through the adjustment of the preference weights in the process of searching for an optimal itinerary.

Two interactive approaches can be adopted for itinerary development: "itinerary construction" and "itinerary improvement." In the itinerary construction approach, an

[1] A "dominant solution" is a solution (in this case an itinerary) which is superior to other solutions with respect to every one of the criteria under consideration.

itinerary is constructed incrementally by specifying a trip connecting two locations to visit, one by one with user input. In the itinerary improvement approach, an initial feasible itinerary is first generated, then improved through interaction with the user. These two approaches have both advantages and disadvantages.

The itinerary construction approach assumes that if the user is satisfied with each segment of the itinerary connecting two consecutive locations to visit, given the portion of the itinerary developed so far, then he/she is also satisfied with the entire itinerary. With this approach, the user takes an active role (e.g., deciding which mode to take and which location to visit next) and is burdened with more tasks while instructing the Planner to find a suitable itinerary. Furthermore, this approach may not offer a solution that is at all close to the optimal solution if the decisions made in early phases of itinerary construction are globally far from optimal.

The itinerary improvement approach is able to select globally optimal itineraries if the algorithm performs exhaustive search of all relevant itineraries. Because the size of the problems anticipated for the Planner is relatively small in terms of the number of locations to be visited, performing an exhaustive search is not impractical. Yet, whether such a solution can be reached through interaction with the user is not certain because, for one thing, the user may not be consistent in revealing their preferences. If this in fact is the case, then the notion of "optimal" solution is meaningless: at the end of the programming process, the user will have an acceptable solution; whether this is optimal or not cannot be determined unless user preferences are unambiguously identified.

Another dimension which needs to be introduced into the analytical scope here is the cost of searching for better solutions. It is inconceivable that a user would go through hundreds of iterations to locate the "optimal" solution as the cost of this search process would not be justifiable to the user. Furthermore, the "optimal" solution may not be discernibly different from nearby sub-optimal solutions. The interactive programming procedure must be efficient in the sense that it can locate an acceptable, hopefully near-optimal, solution within a number of iterations that a typical user is willing to perform. Clearly the emphasis must be on presenting an acceptable solution, or a set of alternative solutions, to the user, but not locating the optimal solution.

The Itinerary Planner Prototype adopts the itinerary improvement approach where itineraries are improved and selected via a number of interactions with the user. This choice is based on mainly two reasons. Firstly, the number of activity locations to visit (excluding source and sink) is expected to be relatively small (in the prototype it is limited to six), thus, performing an exhaustive search of all possible itineraries is not impractical. Secondly, the tour improvement approach reduces the user's burden in selecting the best compromise itinerary.

### 4.6.2. Selecting the Best Itinerary

In making a choice from among a set of alternative solutions, different users may adopt different strategies in comparing alternatives, eliminating inferior ones, and selecting the most satisfactory solution. Alternative approaches to selecting a solution are first discussed in this section, followed by a description of the approach adopted by the Planner.

There are many approaches in selecting the most desirable solution from among a set of possible solutions. If the evaluation criteria are clearly defined and organized into one index, then appropriate mathematical optimization techniques can be adopted to locate the solution. This is not always the case. A sample of few approaches that may be adopted in such cases are presented below.

*Concordance Analysis*

Concordance analysis is one approach to rank-ordering alternative solutions. According to Jankowski (1995)

> "the concordance analysis determines the ranking of alternatives by means of pairwise comparison of alternatives. The comparison is based on calculating the concordance measure which represents the degree of dominance of alternative *i* over alternative *i'* for all the criteria for which *i* is equal or better than *i'*, and the discordance measure which represents the degree of dominance of alternative *i'* over alternative i for all the criteria for which *i'* is better than *i*."

Calculation of concordance and discordance measures are carried out on each pair of alternatives and a final score is then calculated for every alternative. Selection of a best alternative is based on the final score. Procedure of calculating concordance and discordance measures can be found in Nijkamp *et al.* (1990).

*Ideal Point*

With the ideal point approach, the user is asked to locate the ideal solution in a *p*-dimensional criteria space, i.e., the user specifies the ideal value for each single criteria. Then the distance between the ideal solution and each alternative is evaluated. A best alternative is identified as the one with the shortest distance from the ideal solution.

In the Planner problem, the ideal travel time, travel cost, etc., may be reasonably assumed to be 0. The user may have specific desirable values for other variables, e.g., the starting time of an activity, or the sequence of two activities. The ideal point approach appears to be very effective for the latter variables. Whether the Planner can identify weights associated with the *p* criteria in a practical manner remains as a problem.

*Non-compensatory Approach*

The non-compensatory approach assumes that an inferior value of a certain criterion cannot be compensated for by superior values of other criteria. Thus, if a threshold value exists for an criterion and if an alternative does not satisfy that threshold, then the alternative is inferior and should be eliminated from the solution set. With this approach, the user is asked to specify a trade-off range and rank the importance of each criterion. This information is then used to select the best alternative for the user.

*Approach Adopted in This Study*

It is assumed in this study that in evaluating alternative itineraries, the user uses implicit preference weights that are associated with attributes of the itinerary, including travel time, travel cost, walking time, and number of transfers. The itinerary that the user is most satisfied with has the best combination of attribute values. Suppose the utility (or attractiveness) to the user of the itinerary is expressed by the objective function, and that this objective function is defined in terms of the preference weights and the attributes of the itinerary. Namely, the objective function is expressed as

*f(x: _ ) = x__*

where

$x = \{x_1, x_2, \ldots, x_n\}$ is a vector of the arguments of the objective function, and
$\beta = \{\beta_1, \beta_2, \ldots, \beta_n\}$ is a vector of the preference weights.

The preference weights approach is adopted in this study for two reasons. Firstly, the preference weights approach assumes linear compensatory relations among the itinerary attributes, or a linear trade-off behavior in decision making, which has been adopted as an assumption in many behavioral studies. Secondly, the preference weights approach is relatively easy to implement. The tasks to be performed by the Planner with respect to the objective function are thus two-fold: it must first estimate values of _ efficiently, then find an itinerary, or a set of itineraries, which offer acceptable values of *f* using the updated values of _ .

### 4.6.3. Updating the Preference Weights

Many alternative methods exist that can be used by the Planner to solicit preference information from the user. In this section, alternative methods that may be used to solicit users' preference information are first discussed. This is followed by a discussion on what is adopted by the Planner.

One method is to ask the user for trade-off ratios directly. For example, we may ask the user how much more he/she is willing to pay to decrease the travel time by 10 minutes (this approach is called “contingent valuation method”). This willingness-to-pay information can be used by the Planner to establish the rate of substitution between travel time and monetary cost, which can be used in the search for a better itinerary with an optimal balance between travel time and cost.

Alternatively, one may adopt a formal measurement methodology, such as conjoint measurement. In this method, combinations of values of selected attributes (e.g., travel time and travel cost) are presented to the user based on an experimental design. The responses obtained from the user are used to construct a preference function that offers the combined utility of, in this case, travel time and travel cost.

The third way to obtain preference information is the family of methodologies used in the analysis of discrete choice data, in particular, stated-preference data. For example, the Planner may generate a set of alternative itineraries and ask the user to rank-order them. Or it may present a set of itineraries and ask the user which itinerary he/she prefers the most. This exercise is repeated as needed to establish a preference function for each user. There are well established statistical methods to analyze data thus obtained and to establish preference functions.

Preference weights associated with the respective attributes of the itinerary are not observable; users themselves probably do not recognize their preferences as weights. Thus, it is a daunting task for the user if he/she were asked to specify trade-off ratios between attributes. The second and third methods described above, though doable, are not practical in that they both require a substantial amount of repeated experimental exercises from the user to obtain a set of updated weights. With these considerations in mind, the Itinerary Planner adopts a simple strategy to update the weights: increase the value of the weight when the attribute value of the preferred itinerary (as indicated by the user) is less than that of the other itinerary, and decrease the weight when the attribute value of the preferred itinerary is greater than that of the other.

## 4.7. Conclusions

This section has offered a review of Traveling Salesman Problem (TSP) algorithms pertinent to the Itinerary Planner prototype development. It is concluded that direct implementation of TSP algorithms is not applicable to the Itinerary Planner problem. Instead, an Interactive Programming Method (IMP) is adopted in this study where the user is brought into the decision making process to locate the best compromise itinerary. To implement the IMP procedure, the Planner adopts a tour improvement procedure where itineraries are improved and selected via a number of interactions with the user. The user’s preference function is assumed to be a linear compensatory function where preference weights for attributes of the itinerary are updated repeatedly. It is not guaranteed that the IMP procedure will locate the best compromise itinerary for the user. It, however, will produce suitable itineraries while accommodating that user preferences which vary from user to user or from occasion to occasion.

## 5. Planner Prototype Development

This section is devoted to the documentation of the Planner prototype development. The assumptions used in the development are laid out in Section 5.1. As described in Section 2 of this report, the Planner prototype consists of three major parts: user inputs, outputs to the user, and the Planner Executor. User inputs and outputs are described in Section 5.2 while Planner Executor is presented in Section 5.3. An example is shown in Section 5.4. In Section 5.5, limitations of the current prototype are pointed out and future research directions are identified.

### 5.1. Study Area and Assumptions

The prototype is developed as a public-access terminal. The source of the itinerary, therefore, is one of such terminals. In the prototype, the user can choose as the source of the itinerary one of the eight fixed public kiosks located throughout the study area.

Downtown San Francisco is selected as the study area. The study area extends to the intersection of Marina Blvd. and Casa Way in the northwest direction, Market street and 17th Street in the southwest direction, Illinois Street and 18th Street in the southeast direction and the Embarcadero in the northeast direction. The area includes many of the tourist attractions of San Francisco including Union Square and Fisherman's Wharf.

There are ten transit lines in the study area. Horizontally in the east-west direction, bus line No. 45 runs on Union Street; bus line No. 1 runs on Sacramento street; cable car runs on California Street; bus line No. 38 runs on Geary street; bus line No. 31 runs on Eddy street; bus line No. 5 runs on McAllister Street. Vertically in the north-south direction, bus line No. 42 runs on Van Ness Street and bus lines No. 30 and No. 39 run on Stockton Street.

In this effort, which represents the initial development of the Planner prototype, two major assumptions are introduced to create a working prototype of the Itinerary Planner.

The first assumption is that only a single travel mode is used to travel between two consecutive activity locations. In other words, although the itinerary is multi-modal in which any combination of automobile, taxi, walk and public transit may be used, each trip in the itinerary can be made only by a single mode. (Walking to and from public transit is not counted as a separate mode, and the Planner prototype can include the combination, walk - public transit - walk.) Public transit includes bus and cable car.

The second major assumption is that all visit locations are hard locations. This assumption has been introduced to simplify coding and to reduce data requirements. The user is free to choose any point on the map as activity locations to visit.

## 5.2. User-Interface

This section consists of two parts: input variables collected from the user by the Planner at the outset of a session, and output variables presented to the user by the Planner.

### 5.2.1. User Inputs

At the onset of the Planner execution, the Planner presents the user with a map of the downtown San Francisco area on which eight kiosks are shown. The Planner prompts the user to input the set of variables that are described in Section 2 of this report. These variables are essential for the search of desirable itineraries. The variables are inputted in the following order:

- Source (starting point of an itinerary)
- Attributes of a visit (repeated as necessary)
  - Location
  - Timing constraints
  - Duration constraints
- Sink (ending point of an itinerary)
- Sequencing constraints
- Mode preference

In the rest of this section, each item in the above list is described.

*Source (Starting Point of an Itinerary)*

Eight kiosks are available for the user to select as source of their itineraries and these eight kiosks include: Ferry Building, The Cannery, Ghirardelli Square, Civic Center, Jackson/Hyde Kiosk, Union Square, Moscone Center, and Post/Pierce Kiosk. After the user selects one, he/she is prompted to input time of day constraints (e.g., earliest and latest departure time) associated with the source.

*Attributes of a Visit*

Attributes of a visit are obtained from the user. These attributes include location, time of day and duration constraints.

A location can be specified by the user by:

- giving an exact street address, or
- clicking on the map directly.

The exact street address is most precise, but the user may not always know it. Specifying a location by cross-streets or landmarks is not available in the current

Planner prototype. The other option is to specify an activity location by clicking directly on the map. This option is usable only when the user can tell the location on the map. But if the user knows where it is, then clicking on the map or touching the screen to pinpoint the location may be the easiest and the fastest.

Time of day constraints concern the earliest possible arrival time and the latest possible departure time; for each activity location to visit, the Planner asks the user to input, if any, the earliest possible arrival time at the location and the latest possible departure time from the location. Duration constraints concern the minimum possible duration and the maximum possible duration. The Planner prompts the user to input minimum and maximum possible duration if they exist.

*Sink (Ending Point of an Itinerary)*

Information obtained from the user about the sink is similar to that obtained about a visit, except for that the user is not required to input duration constraints. Additionally, time of day constraints associated with the sink refer to the earliest possible arrival time and the latest possible arrival time.

*Sequencing Constraints*

Sequencing constraints refer to a particular order of visits that must be maintained in the itinerary. In the prototype Planner, the user is able to specify that a particular location must be visited before or after another location.

*Mode Preference*

The user can specify his/her mode preferences in the Planner prototype. As noted earlier, the Planner prototype incorporates four modes: private automobile, taxi, public transit, and walk. The user can eliminate any of the four modes or any combination of modes and ask Planner to develop an itinerary with one or more preferred modes.

### 5.2.2. Outputs to the User

Outputs to the user consist of two highlighted itineraries selected by the Planner. As noted earlier, an itinerary can be made in combination of more than one mode; different travel modes are designated on the map by different colors. Written directions are also provided, guiding the user in following the itinerary. For taxi, automobile or walking, written directions indicate what mode to take on which street for how long. For public transit, written directions indicate which transit line to take, the names of stops to board/alight, and to transfer between transit lines

## 5.3. Planner Executor

Figure 1, the flow chart of the Planner Executor, illustrates how the Itinerary Planner functions. Step 1 is the user inputs where the user supplies information on locations to visit, constraints associated with each visit, mode preferences, etc. In Step 2, the Planner locates the nearest nodes to the activity locations selected by the user and finds a minimum path between every possible pair of activity locations and stores the results in a temporary database. In Step 3 through  Step 5, the Planner enumerates all possible sequences, computes the arrival time and departure time at each visit, and checks the feasibility of every sequence against the timing, duration and sequence constraints specified by the user. In Step 6, the Planner evaluates the generalized cost of every feasible sequence and selects two itineraries with lowest and second-lowest values. If the user is not satisfied with either itinerary, he/she is then asked to indicate which is more preferred. The Planner then takes user's inputs, updates the preference weights and repeats operations from Step 2. If the user is satisfied with one of the itineraries, the Planner ceases operation and goes to "END". Further discussion on each of these steps is provided below.

### *Street and Transit Networks*

The Planner operates on two networks of the study area: one is the street network on which the user can take automobile, walk, or taxi and the other is the transit network on which the user can take transit or walk.

The street map of downtown San Francisco, which comes with the ArcView software package[1], is used in the prototype. This map contains every street in the downtown and its adjoining areas. Each street segment (i.e., between two sccuessive intersections) is coded with attribute data including its length, the name of the street, and the direction(s) of traffic on it. For the current prototype, the data base for the streets in downtown and its peripheral areas includes over 3,000 street links.

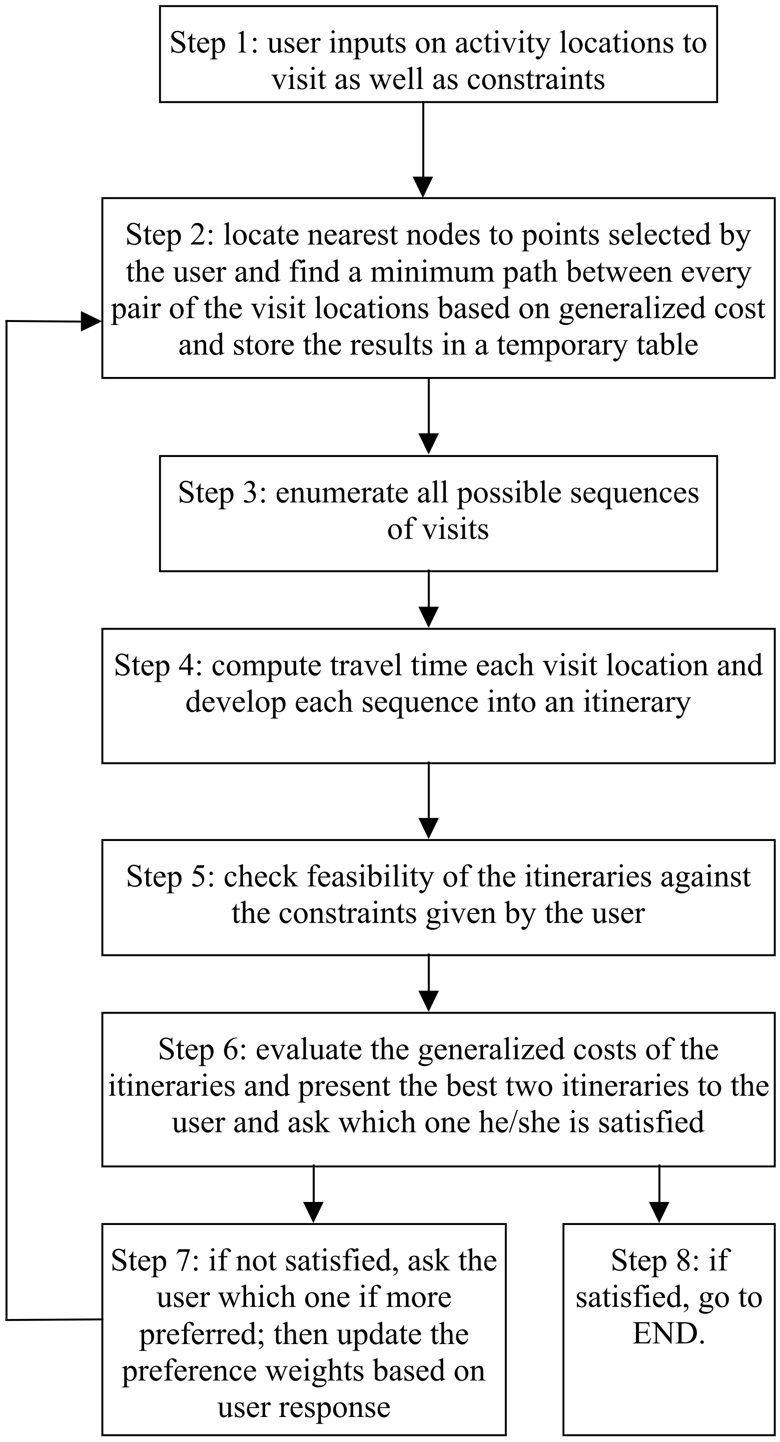
The transit map of the study area is coded on the top of the street map. At the intersection where two transit lines cross, dummy transfer nodes and link are added such that waiting time, resulted from transfers, from one transit line to the other can be properly represented. In areas that are not covered by transit lines, the user is assumed to walk. Each record in this transit data base is coded with attribute data including the name of the street, the name of the transit line, cross street, and whether it is a dummy transfer link or not.

### *Dijkstra's Algorithm*

There are in general more than one path between any two activity locations selected by the user; Dijsktra's algorithm is applied to locate the best path between two activity

[1] ArcView is a GIS program developed by ESRI.

**Figure 1: Flow chart of Planner Executor**


Step 1: user inputs on activity locations to visit as well as constraints
Step 2: locate nearest nodes to points selected by the user and find a minimum path between every pair of the visit locations based on generalized cost and store the results in a temporary table
Step 3: enumerate all possible sequences of visits
Step 4: compute travel time each visit location and develop each sequence into an itinerary
Step 5: check feasibility of the itineraries against the constraints given by the user
Step 6: evaluate the generalized costs of the itineraries and present the best two itineraries to the user and ask which one he/she is satisfied
Step 7: if not satisfied, ask the user which one if more preferred; then update the preference weights based on user response
Step 8: if satisfied, go to END.

locations. The term "best " here implies having the highest value of the preference function that is a weighted sum of seven trip attributes. These seven attributes are: travel cost, bus travel time, waiting time, walk time, number of transfers, cable car travel time, and taxi travel time. Thus the generalized cost of each network link, expressed as the negative of the preference function value associated with the link, is used in the minimum path search in the Planner prototype.

Trip attributes above are determined by mode based on a set of assumptions on travel speed and initial costs etc. For transit, nine transit lines are assumed to have a speed of 400 meters per minute and a fare of \$1; the other one (line 4 representing cable car line) is assumed to have a speed of 267 meters per minute and a fare of \$2. For automobile and taxi, the Planner assumes a travel speed of 667 meters per minute, or 40 kilometers per hour. Travel cost associated with automobile includes two parts: parking cost and operating cost. For all visit locations except the sink, the user must pay an initial cost of \$5 and the operating cost for automobile is assumed to be \$0.0003 per meter. Travel cost associated with taxi includes two parts as well: initial fare and distance-based fare. The initial fare is \$5 and the distance-based fare is assumed to be \$0.0001 per meter.

Every time the user transfers to a different transit line, there will be waiting time involved. The waiting time resulted from transferring to different transit lines are assumed to be 13, 11, 12, 11, 15, 13, 11, 12, 9, and 25 minutes respectively for ten transit lines.
The Planner assumes a walking speed of 50 meters per minute and there is no travel cost in the dollar sense associated with walking.

*Generation of All Possible Sequences of Mixed Modes*

Sequences of visits are generated with and without a private auto used to travel. With an automobile involved, the itinerary can be made by a combination of auto, walk and public transit, but not taxi; it was assumed that the user will not use taxi when a private automobile is available. Without an automobile involved, the itinerary can be made by a combination of taxi, walk and public transit.

In case an auto is used, Table 1 shows the number of mode sequences by N (the number of locations including source and sink) that can be made for a particular sequence of visits when transit, private auto and walk are used. When private auto is used, special caution must be taken on the parking constraint. The user can park the auto anywhere except at the source, and take transit or walk to the next activity location. However, he or she must go back to where the auto is parked to fetch the car and then go to the subsequent activity location. The number of sequences shown in the table reflects this constraint. As noted earlier, the maximum number of visit locations that can be accommodated in the current prototype of the Itinerary Planner is six (or eight if the source and sink are included).

Without a private auto involved, there could be $3^{N-1}$ mode sequences for each sequence of visits. Since there are altogether (N-2)! Sequences of visit locations, the total of number of sequences is (N-2)! _ $3^{N-1}$ when transit, taxi and walk are considered.

**Table 1. Number of Mode Sequences with Auto Involved***

| No. of activity locations | N = 4 | N = 5 | N = 6 | N = 7 | N = 8 |
|---|---|---|---|---|---|
| No. of mode sequences | $2^2$+1= 5 | 2_ $2^2$+$2^3$ +1=17 | 3_$2^2$+2_ $2^3$+2_ $2^4$+1=61 | 4_$2^2$+3_ $2^3$+5_$2^4$+ 3 ×$2^5$+1=217 | 5_$2^2$+4_$2^3$+9_ $2^4$+8_$2^5$+5_ $2^6$+1=773 |

* three modes: private auto, walk and public transit, are considered here.

*Feasibility Check*

Feasibility check is concerned with two types of constraints: timing and duration constraints, and sequence constraints. Timing and duration constraints imply that for any two consecutive locations in a sequence, the latest possible departure time at a location must be greater than or equal to the arrival time at the previous location, plus the travel time from the previous location to the current location, plus the duration at the previous location. A similar condition can be applied to the earliest possible arrival time. In the prototype, however, it is assumed that the user can wait at the visit location if he/she arrives there before the earliest possible arrival time. Sequence constraints imply that a feasible sequence must satisfy the particular order of visits specified by the user. These constraints are checked for each sequence of visits and for all possible mode combinations.

*Calculation of Preference Function Values*

Following the feasibility check, the value of the preference function is calculated for each feasible sequence, and two itineraries with highest and second-highest preference function values are selected. These two itineraries are then presented to the user. The user is then asked whether he/she is satisfied with either itinerary. If not, the Planner will update the preference weights and go through a second iteration. If yes, the Planner will conclude its search.

The preference function is specified in the prototype as:

$$f = \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_4 x_4 + \beta_5 x_5 + \beta_6 x_6 + \beta_7 x_7$$

where, $x_1$, $x_2$, $x_3$, $x_4$, $x_5$, $x_6$, *and* $x_7$ are bus travel time, cable car travel time, auto (taxi) travel time, travel cost, waiting time, number of transfers, and walk time respectively.

*Updating Preference Weights*

Preference weights used in the first iteration are called initial weights. They are based on the coefficients of a travel mode choice model estimated in a previous study in the San Francisco Bay Area (Train, 1976). The initial weights are updated to better reflect the user's preferences in a iterative manner using the method described in Section 4.6.3. This approach is adopted because (i) it does not require extensive inputs from the user to update the preference weights, and (ii) the preference function is refined interactively when the user continues to search for better itineraries; thus more refined preference valuations will be adopted for those users who are willing to search extensively for a better itinerary.

## 5.4. An Example

Suppose that the user indicated to the Planner the set of locations to visit and constraints associated with each visit as shown in Table 2. In addition, suppose the user also indicated that the work place must be visited before the grocery store, that in turn must be visited before the movie theater. The user did not impose any mode constraints.

**Table 2. User Inputs (example)**

| Location | Earliest arrival | Latest departure | Min. duration | Max. duration |
|---|---|---|---|---|
| Union Square | 6am | 8am | None | None |
| Work Place | 7am | 9am | None | None |
| Lunch Place | Noon | 1pm | None | None |
| Grocery Store | 5pm | None | None | None |
| Movie Theater | 7pm | None | None | None |
| Home | None | None | None | None |

The Planner produced two itineraries as output. The sequence of visits is the same for both itineraries, which is: Union Square _ work place _ lunch place _ grocery store → movie theater _ home. Other output attributes for both itineraries are shown in Table 3 and Table 4, respectively.

Given the initial two itineraries, the user indicated that he preferred the second itinerary because it had lower in-vehicle travel time, waiting time, walk time, as well as a smaller number of transfers, though it had a higher cost compared to the first itinerary. The Planner then took the input, updated the weights and re-produced another set of two itineraries. The two revised itineraries again have an identical sequence of visits. Their output attributes are shown in Table 5 and Table 6.

**Table 3. Itinerary Attributes of Itinerary 1 (1st iteration)**

| From | To | Mode | In-veh. (min) | Cost($) | Wait(min) | transfers | Walk (min) |
|---|---|---|---|---|---|---|---|
| Union Sq. | Work | Transit | 6 | 2 | 20.9 | 2 | 2 |
| Work | Lunch | Transit | 3.4 | 1 | 14.9 | 0 | 6.7 |
| Lunch | Grocer | Transit | 4.7 | 1 | 10 | 0 | 7.6 |
| Grocer | Movie | Transit | 25.6 | 1 | 15.9 | 1 | 5 |
| Movie | Home | Transit | 6.8 | 1 | 16.9 | 1 | 4.8 |
| For the entire itinerary | | | | | | | |
| Union Sq. | Home | Transit | 46.5 | 6 | 78.6 | 4 | 26.1 |

**Table 4. Itinerary Attributes of Itinerary 2 (1st iteration)**

| From | To | Mode | In-veh. (min) | Cost($) | Wait (min) | transfers | Walk (min) |
|---|---|---|---|---|---|---|---|
| Union Sq. | Work | Transit | 6 | 2 | 20.9 | 2 | 2 |
| Work | Lunch | Transit | 3.4 | 1 | 14.9 | 0 | 6.7 |
| Lunch | Grocer | Transit | 4.7 | 1 | 10 | 0 | 7.6 |
| Grocer | Movie | Taxi | 1.9 | 5.39 | 0 | 0 | 0 |
| Movie | Home | Transit | 6.8 | 1 | 16.9 | 1 | 4.8 |
| For the entire itinerary | | | | | | | |
| Union Sq. | home | | 22.8 | 11.39 | 62.7 | 3 | 21.1 |

**Table 5. Itinerary Attributes of Itineraries 1 (2nd iteration)**

| From | To | Mode | In-veh. (min) | Cost($)[1] | Wait (min) | transfers | Walk (min) |
|---|---|---|---|---|---|---|---|
| Union Sq. | Work | Auto | 2.1 | 5.14 | 0 | 0 | 0 |
| Work | Lunch | Auto | 2.1 | 5.14 | 0 | 0 | 0 |
| Lunch | Grocer | Auto | 2.3 | 5.15 | 0 | 0 | 0 |
| Grocer | Movie | Auto | 1.9 | 5.13 | 0 | 0 | 0 |
| Movie | Home | Auto | 3.2 | 0.21 | 0 | 0 | 0 |
| For the entire itinerary | | | | | | | |
| Union Sq. | home | Auto | 11.6 | 20.77 | 0 | 0 | 0 |

[1] Parking cost is included in the "cost" category.

**Table 6. Itinerary Attributes of Itineraries 2 (2nd iteration)**

| From | To | Mode | In-veh. (min) | Cost($)[1] | Wait (min) | transfers | Walk (min) |
|---|---|---|---|---|---|---|---|
| Union Sq. | Work | Transit | 6 | 2 | 20.9 | 2 | 2 |
| Work | Lunch | Taxi | 2.1 | 5.42 | 0 | 0 | 0 |
| Lunch | Grocer | Taxi | 2.3 | 5.47 | 0 | 0 | 0 |
| Grocer | Movie | Taxi | 1.9 | 5.39 | 0 | 0 | 0 |
| Movie | Home | Taxi | 3.2 | 5.65 | 0 | 0 | 0 |
| For the entire itinerary | | | | | | | |
| Union Sq. | home | | 15.5 | 23.93 | 20.9 | 2 | 2 |

[1] Parking cost is included in the "cost" category.

The Planner updated the preference weights in a manner that was consistent with our expectation. The feedback from the user after the first iteration (the user preferred the second itinerary with a higher cost but less travel time) indicated that the initial weight on travel time was too low and the initial weight on travel cost was too high. For the second iteration, the Planner increased the weight on travel time and lowered the weight on travel cost. The itineraries produced at the second iteration reflected this adjustment.

The Planner prototype is computationally efficient. For the above example, on a Pentium II PC, it took about 1 second for the Planner Executor to locate the two itineraries and took about 5 seconds for the user-interface to display the two itineraries. With 6 activity locations (the maximum number of activity locations allowed in the current prototype), it took about 7 seconds for the Planner Executor to locate the two best itineraries.

## 5.5. Limitations

The current prototype of the Planner is limited in several aspects. These limitations are mostly related to the algorithm development. Regarding the simple algorithm for preference weight updating adopted by the Itinerary Planner, one may ask how we can ensure that an interactive multi-objective mathematical programming (MOMP) procedure will converge to true values in the end. Firstly, an investigation is yet to be made into the convergence characteristics of the simple algorithm used in the prototype. In addition, convergence to true values requires that users be consistent throughout the problem solving process (Shin & Ravindran, 1991). This can be considered a serious drawback of interactive MOMP procedures (French, 1984). Inconsistencies that may be exhibited by the user during decision making can be reduced by presenting him easy and simple questions (Shin & Ravindran, 1991). The question about convergence, however, is not fully investigated in this study and certainly deserves future research.

Another limitation is that the current Planner prototype can not incorporate soft locations. This limitation is primarily due to the lack of a data base to accommodate soft locations.

## 6. Conclusion

The Itinerary Planner developed in this project is one of the first travel planning tools that incorporate multiple destinations, multiple objectives and constraints, and multiple modes of travel. It is also one of the first travel planners that bring the user into the itinerary development process and continuously adjust the objective function used to evaluate alternative itineraries such that the user's preferences can be best reflected in selecting an itinerary.

The development of the Itinerary Planner thus represents an ambitious effort to incorporate the many novel features into one software package for travel planning assistance. The Planner prototype developed in this project, however, is still in its prototype stage and unavoidably exhibits certain limitations. In particular, it has not been extensively tested. Some of the concepts and algorithms that have been introduced into the prototype need to be tested and refined. The user-interface has room for improvement; data bases for different types of opportunities (e.g., hotels, restaurants, banks) are desired such that the user can find where to go by keying in the desired type of opportunity; and the method of updating the objective function should be rigorously tested and improved. Despite these limitations, it is believed that the project has shown the Planner is an effective tool that can help travelers organize their itineraries in an efficient manner while observing many constraints that may exist. It is expected that the Planner will provide not only personal benefits to the user but also social benefits through increased transit use and reduced traffic congestion.

## References:

1. Abdel-Aty, M., Kitamura, R. and Jovanis, P. Investigating Effect of Advanced Traveler Information on Commuter Tendency to Use Transit. *Transportation Research Record*, 1550, National Research Council, Washington, D.C., 1996, pp. 65-72.

2. Aksoy, Y., Butler, T., and Minor, E. Comparative Studies in Interactive Multiple Objective Mathematical Programming. *European Journal of Operational Research*, Vol. 89, No.2, 1996, pp. 408-422.

3. French, S. Interactive Multiple Objective Programming: its Aims, Applications and Demands. *Journal of the Operational Research Society*, Vol.35, No.9, 1984, pp. 827-834.

4. Gendreau, M., Hertz, A., and Laporte, G. New Insertion and Post-Optimization Procedures for the Traveling Salesman Problem. *Operations Research*, Vol.40, 1992, pp. 1086-1094.

5. Gendreau, M., Hertz, A., Laporte, G., and Stan, M. *A Generalized Insertion Heuristics for the Traveling Salesman Problem with Time Windows*. Universite de Montreal, Centre de recherche sur les transports, 1995.

6. Golden and Stewart "Empirical Analysis of Heuristics". Traveling Salesman Problem. A Guided Tour of Combinatorial Optimization, John Wiley & Sons, 1985.

7. Hoffman, A. J., and Wolfe, P. "History". The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization. Edited by Lawler, E. L. et. al., Wiley, 1985.

8. Jankowski, P., 1995, "Integrating Geographical Information Systems and Multiple Criteria Decision-making Methods". International Journal of Geographical Information Systems, vol. 9, no. 3, pp. 251-273.

9. Johnson, D.S. "Local Optimization and the Traveling Salesman Problem". Automata, Languages and Programming, Lecture Notes in Computer Science 443, Springer-Verlag, 1990.

10. Laporte, Gilbert "THE TRAVELING SALESMAN PROBLEM - An Overview of Exact and Approximate Algorithms". European Journal of Operational Research 59(2), 1992, pp. 231-247.

11. Laporte, Gilbert "Recent Algorithmic Developments for the Traveling Salesman Problem and the Vehicle Routing Problem". Universite de Montreal, Centre de recherche sur les transports, 1993.

12. Miller and Pekny "Exact Solution of Large Asymmetric Traveling Salesman Problems". Science, Vol. 251, No. 4995, 1991, pp. 754-761.

**References (continued):**

13. Nijkamp, P., and Rietveld, P., and Voogd, H. Multicriteria Evaluation in Physical Planning. Amsterdam, Netherlands; New York : North-Holland ; New York, N.Y., U.S.A. : Distributors for the U.S. and Canada, Elsevier Science Pub., 1990.

14. Potvin, Jean-Yves "The Traveling Salesman Problem: A Neural Network Perspective". Universite de Montreal, Centre de recherche sur les transports, 1993.

15. Reinelt, G. The Traveling Salesman: Computational Solutions for TSP Applications. Springer-Verlag, 1991.

16. Shin, W. and Ravindran, A. Interactive Multiple Objective Optimization: Survey 1 - Continuous Case. *Computers & Operations Research*, Vol.18, No.1, 1991, pp.97-114.

17. Solomon, M. and Desrosiers, J. Time Window Constrained Routing and Scheduling Problems. *Transportation Science*, Vol.22, No.1, 1988, pp.1-13.

# APPENDIX

## TP: TRAVELING PLANNER C++ CODE

INTRODUCTION:

*What* is TP and *when* is it useful?

TP is a C++ software specifically developed for an exhaustive search of the optimal path which minimizes a weighted objective function with seven transportation attributes involved: traveling cost, walking time, waiting time, transit transfer number, transit traveling time, taxi traveling time, and cable traveling time. TP consists of two parts: a kernel part executing a minimum spanning tree search of the transportationally optimal path between two given location nodes, and an out-layer part realizing an exhaustive search of the optimal itinerary for user specified visiting location sequence. TP1.CPP executes the first run with the initial weights of the transportation attributes, then, TP2.CPP realizes the repeated runs with updated weights until user satisfied. All the static and dynamic input files are from Arcview user interface and the output file which reports the searched optimal itinerary are return to Arcview for user interface.

*What* are included in TP?

- Input file list:

| | | |
|---|---|---|
| Static: | G_sanf1.dat | Street network file (adjacent lists of all nodes) |
| | G_tran.dat | Transit network file {adjacent lists of all nodes) |

| | | |
|---|---|---|
| | businfo.dat | Transit line speed and cost file |
| | buswaitT.dat | Transit line waiting time file |
| Dynamic: | input.dat | User specified visiting location information |
| | seq.dat | Location sequence constraints |
| | total.dat | Total location number and travel mode selection |
| | userpref.dat | User preferred path number |

The 'static' file denotes an unchangeable file and the 'dynamic' file is dynamically changed in the program running.

- Output file list:

| | | |
|---|---|---|
| Tmp: | fpathattr.dat | Searched path attribute values |
| Dynamic: | fpathone.dat | Searched first path node and link sequences |
| | fpathtwo.dat | Searched first path node and link sequences |
| | fattrone.dat | Searched first path attributes |
| | fattrtwo.dat | Searched first path attributes |
| | fnodecoorone.dat | Visiting location's coordinates for Path one |
| | fnodecoortwo.dat | Visiting location's coordinates for Path two |

The 'Tmp' file is the intermediate file used by the updated program runs.

- Program list:

| | | |
|---|---|---|
| Main: | Tp1.cpp | First run of Traveling Planner |
| | Tp2.cpp | Updated runs of Traveling Planner |
| Subroutine: | Tsp_grap.cpp | Minimum spanning tree algorithm |
| | Tppath.cpp | Constrainted exhaustive search of optimal path |
| | Strings.cpp | Assign number to text strings |
| | Seqlist.cpp | Sorting stack of minimum spanning tree |

| | | |
|---|---|---|
| Header: | Strings.h | head file of Strings.cpp |
| | Seqlist.h | head file of Seqlist.cpp |
| | Graph.h | head file of Tsp_grap.cpp |
| | Tppath.h | head file of Tppath.cpp |

The main programs Tp1.cpp and Tp2.cpp execute the first and undated runs for searching the optimal path which minimizes the objective function involving seven transportation attributes. In the kernel part, a minimum spanning tree search is conducted to find the optimal path between two given location nodes. Then, an out-layer part executes a constrainted exhaustive search of the optimal itinerary for an user specified location sequence.

The minimum spanning tree search algorithm, represented by the program tsp_grap.cpp, includes four functional parts: inputting the static files of the street /transit network and transit information data, building the graph and data structure of the algorithm, initializing and expanding the minimum spanning tree, and outputting the optimal path between all possible location pairs of a given location sequence to a database which is used by next step for the exhaustive optimal search. The program strings.cpp improves the efficiency of network-node search greatly by applying the binary tree search technique, especially when the total number of the network nodes is big. For a N-nodes network, a linear search is in a time cost of $O$(N). However, the standard binary tree search, by setting and updating left, right, and middle (the average of the left and the right) values, can reduce the time cost to $\log_2$(N). The program seqlist.cpp improves the efficiency of the minimum spanning tree search greatly by applying a linear-stack sorting technique which significantly reduces the minimum distance searching time at each step of expanding the minimum spanning tree.

The constrainted exhaustive search of the optimal itinerary, represented by the program Tppath.cpp, includes seven functional parts: inputting the dynamic files of the user specified visiting location information and constraints (both travel time and visiting sequence constraints), generating all possible node (location) pairs which is of an amount of (N-1)+$(N-2)^2$ with N denoting the total number of the user specified location number, generating all possible node (location) sequences which is of an amount of N! with N being the total location number, generating all possible travel-mode (walk, taxi, and bus) sequences which is of an amount of $3^{N-1}$ with N the total location number, generating all possible auto-mode node (location) sequences which is lack of a general formula for the total amount but the estimated number is bigger than N!, constraintedly and exhaustively searching the optimal itinerary in which all possible node and mode combinations are checked for the time and sequence feasibility and then the first and second optimal itinerary are selected in the feasible ones, and outputting the resulted first and second optimal itinerary.

- Variable Description list:

Although the effort has been made to make the variables self-explainable, a list of the variable description still be summed here for the purpose of clearance.

(1) Variables for tsp_grap.cpp:

| | |
|---|---|
| MAX_NODES: | Network node-number limit, specified as 4000 in current situation. |
| MAX_EDGENUM: | Network edge-number limit, specified as 1000 in current situation. |
| MAX_BUSLINES: | Network edge-number limit, specified as 101 in current situation. |
| N: | Total visiting locations. |
| start, startIndex: | start searching node and its number index. |
| finish, finishIndex: | finish searching node and its number index. |
| busStartDone[MAX_BUSLINES]: | Bus start done index for a path search. |
| NInS: | Program variable of active minimum spanning tree node. |
| inS_seq[MAX_NODES]: | Index array of active minimum spanning tree node |
| Num_edge: | Link number of a searched path between a node pair. |
| edgeseq[MAX_EDGENUM]: | array of link sequence of a searched path between a node pair. |

| | |
|---|---|
| travel_mode: | Travel mode index: 1 - walk, 2 - taxi, 3 - bus, 4 - auto. |
| walk_speed: | Walk speed specified as 50 (meter/min) in current situation. |
| taxi_speed: | Taxi speed specified as 667 (meter/min) in current situation. |
| TaxiStartCost: | Taxi start cost specified as 5000 (1.0E+3$) in current situation. |
| AutoParkCost: | Auto parking cost specified as 5000 (1.0E+3$) in current situation. |
| weight[7]: | Array of objective function weights. |
| taxi_cost_rate: | Taxi cost rate specified as 0.0003 ($/meter) in current situation. |
| auto_cost_rate: | Auto cost rate specified as 0.0001 ($/meter) in current situation. |
| CostZoominFactor: | Cost unit-transfer factor specified as 1000.0. |
| CostZoomoutFactor: | Cost unit-transfer factor specified as 0.1. |
| TimeZoominFactor: | Time unit-transfer factor specified as 1000.0. |
| TimeZoomoutFactor: | Time unit-transfer factor specified as 0.01. |
| LenZoomoutFactor: | Length unit-transfer factor specified as 0.001. |
| TransfernumZoominFactor: | Transfer number transfer factor specified as 10.0. |
| minChangetoE_1minFactor: | Minute changed to 0.1 minute factor specified as 10.0. |
| waitT[MAX_BUSLINES][MAX_BUSLINES][1]: | Array of transfer (from bus i to bus j) or initial (bus i) wait time. |
| businfo[MAX_BUSLINES][2]: | Array of bus line speed and cost information. |
| current_length: | Objective function length of current link. |
| current_len,: | Length of current link. |
| current_traveltime: | Total travel time of current link. |
| current_travelcost: | Total travel cost of current link. |
| current_walkT: | Walk time of current link. |
| current_waitT: | Wait time of current link. |
| current_transfernum: | Bus transfer number of current link. |
| current_transitTravelT: | Transit travel time of current link. |
| current_cableTraverT: | Cable travel time of current link. |
| current_taxiTravelT: | Taxi travel time of current link. |

## (2) Variables for Tppath.cpp:

| | |
|---|---|
| NODEROWS: | Array index limit1 of nodeseq array, specified as 800 in current situation. |
| NODECOLS: | Array index limit2 of nodeseq array, specified as 10 in current situation. |
| MODEROWS: | Array index limit1 of modeseq array, specified as 2500 in current situation. |
| MODECOLS: | Array index limit2 of modeseq array, specified as 10 in current situation. |
| MAX_NODE_NUMBER: | Maximum network node-number, specified as 4000 in current situation. |

| | |
|---|---|
| NODE_NUM_LIMIT: | Network node-number limit, specified as 4001 in current situation. |
| NO_FEASIBILITY_LIMIT: | No feasibility constant, specified as -9999. |
| nodeseq[NODEROWS][NODECOLS]: | Array of all possible visiting-node sequence for walk, taxi, bus mode. |
| modeseq[MODEROWS][MODECOLS]: | Array of all possible visiting-mode sequence for walk, taxi, bus mode. |
| auto_nodeseq[800][15]: | Array of all possible visiting-node sequence for auto mode. |
| auto_modeseq[800][15]: | Array of all possible visiting-node sequence for auto mode. |
| addmodes[800][20]: | Array for restoring intermediate calculation result. |
| CurAutoNodeseq[15]: | Array for restoring intermediate calculation result. |
| s[NODECOLS+2]: | Array for restoring intermediate calculation result. |
| seq[NODECOLS+1][2]: | Array for restoring intermediate calculation result. |
| loc_info[10][10]: | Array of location information of user specified visiting locations |
| pair[50][4][200]: | Array of searched path data for all possible pair combination of visiting locations. |
| includeMode[4]: | Including index of travel mode (walk, taxi, bus, or auto): 1 - included, 0 - not included. |

## LIST OF TP(Traveling Planner) CODE

**Tp1.cpp:**
```
// tp1.cpp:        Code One of Traveling Planner Project studied at TIS of UC Davis
// Author:         Jiayu Chen, TIS, UCD
// Created date:  Feb, 1998
// Last Modified: November, 1998

#include <iostream.h>
#include "graph.h"
#include "strings.h"
#include "tppath.h"

#include "strings.cpp"
#include "tsp_grap.cpp"
#include "tppath.cpp"
#include "seqlist.cpp"

int main()
{        graph gstreet, gtransit;
         TPpath p;
         strings Tnode_street, Tnode_transit, Tbus, Tdum;
         int isTransit;
         char ch;

         ifstream fstreet("g_sanf1.dat", ios::in | ios::nocreate);
         if (fstreet.fail()) error("Cannot open fstreet file.");
         ifstream ftransit("g_tran.dat", ios::in | ios::nocreate);
         if (ftransit.fail()) error("Cannot open ftransit file.");

         isTransit=0;
         gstreet.build(isTransit, Tnode_street, Tdum, Tdum, fstreet);       // Build adjacency lists of
street-network
         gstreet.ini_weight();
         gstreet.SetParameters();
         gstreet.InputBusInfo();
//       int n = Tnode_street.size();
//       cout <<"total street-Node number: "<<n<< endl;
//       cout <<"  Node_street         Node#"<<endl;
//       for (i=0; i<n; i++) cout<<setw(6)<<Tnode_street[i]<<setw(6)<<i<<endl;

         isTransit=1;
         gtransit.build(isTransit, Tnode_transit, Tbus, Tdum, ftransit);       // Build adjacency lists of
transit-network
         gtransit.ini_weight();
         gtransit.SetParameters();
         gtransit.InputBusInfo();
//       n = Tnode_transit.size();
//       cout <<"total transit-Node number: "<<n<< endl;
//       cout <<"  Node_transit         Node#"<<endl;
//       for (i=0; i<n; i++) cout<<setw(6)<<Tnode_transit[i]<<setw(6)<<i<<endl;
//       n = Tbus.size();
//       cout <<"total bus number: "<<n<< endl;
//       cout <<"  busname         busindex"<<endl;
//       for (int i=0; i<n; i++) cout<<setw(6)<<Tbus[i]<<setw(6)<<i<<endl;

         p.get_locations_info(gstreet, Tnode_street, Tnode_transit);
//       cout <<endl<<"see:"<<endl;
//       for (int k=0; k<5; k++) cout<<setw(10)<<p.loc_info[k][0];
//       cin>>ch;
         p.location_pairs();

         p.bestpath_allpairs(Tnode_street, Tnode_transit, gstreet, gtransit, Tbus);

         p.modeseqs();
//       int N=p.get_N();  cout << "N=" << N << endl;
//       int m=(int)pow(3,N-1);
//       for (int i=0; i<m; i++) {
//                for (int j=0; j<N-1; j++) cout<<setw(6)<<p.modeseq[i][j];
//                cout<<endl;}

         p.nodeseqs();
//       n=p.get_locnum();
//       m = 1;
//       for (i=1; i<(n+1); i++)  m *=  i;
//       for (i=0; i<m; i++) {
//                for (int j=0; j<n; j++) cout<<setw(6)<<p.nodeseq[i][j];
//                cout<<endl<<endl;}

         if (p.includeMode[3] == 1) {
                  p.auto_seqs();
/*                int N=p.get_N();
                  int m=p.get_AutoRowIndex();  cout << "AutoRowIndex=" << m << endl;
                  for (int i=0; i<m; i++)
                  {        for (int j=0; j<N+4; j++) cout<<setw(3)<<p.auto_nodeseq[i][j];
                           cout <<"     ";
                           for (j=0; j<N+3; j++) cout<<setw(3)<<p.auto_modeseq[i][j];
                           cout<<endl;
                  }
*/
                  p.ModifyAuto_seqs();
/*                N=p.get_N();
                  m=p.get_AutoRowIndex();  cout << "AutoRowIndex=" << m << endl;
                  for (i=0; i<m; i++)
                  {        for (int j=0; j<N+4; j++) cout<<setw(3)<<p.auto_nodeseq[i][j];
                           cout <<"     ";
                           for (j=0; j<N+3; j++) cout<<setw(3)<<p.auto_modeseq[i][j];
                           cout<<endl;
                  }
*/       }

         p.constraited_bestpath(Tnode_street, Tnode_transit);

         cout << "Enter any letter-key to continue." <<endl;
         cin >> ch;

         return 0;
}
```

**tp2.cpp:**

```
// tp2.cpp:      Code Two of Traveling Planner Project studied at ITS of UC Davis
// Author:       Jiayu Chen, ITS, UCD
// Created date: Feb., 1998
// Last Modified: November, 1998

// #include "tp_code.h"
#include <iostream.h>
#include "graph.h"
#include "strings.h"
#include "tppath.h"

#include "strings.cpp"
#include "tsp_grap.cpp"
#include "tppath.cpp"
#include "seqlist.cpp"

int main()
{        graph gstreet, gtransit;
         strings Tnode_street, Tnode_transit, Tbus, Tdum;
         TPpath p;
         int isTransit;
         char ch;

         ifstream fstreet("g_sanf1.dat", ios::in | ios::nocreate);
         if (fstreet.fail()) error("Cannot open fstreet file.");
         ifstream ftransit("g_tran.dat", ios::in | ios::nocreate);
         if (ftransit.fail()) error("Cannot open ftransit file.");

         isTransit=0;
         gstreet.build(isTransit, Tnode_street, Tdum, Tdum, fstreet);      // Build adjacency lists of
street-network
         gstreet.update_weight();

         gstreet.SetParameters();
         gstreet.InputBusInfo();
//       int n = Tnode_street.size();
//       cout <<"total street-Node number: "<<n<< endl;
//       cout <<"  Node_street        Node#"<<endl;
//       for (i=0; i<n; i++) cout<<setw(6)<<Tnode_street[i]<<setw(6)<<i<<endl;

         isTransit=1;
         gtransit.build(isTransit, Tnode_transit, Tbus, Tdum, ftransit);      // Build adjacency lists of
transit-network
         gtransit.update_weight();
         gtransit.SetParameters();
         gtransit.InputBusInfo();
//       n = Tnode_transit.size();
//       cout <<"total transit-Node number: "<<n<< endl;
//       cout <<"  Node_transit        Node#"<<endl;
//       for (i=0; i<n; i++) cout<<setw(6)<<Tnode_transit[i]<<setw(6)<<i<<endl;
//       n = Tbus.size();
//       cout <<"total bus number: "<<n<< endl;
//       cout <<"   busname         busindex"<<endl;
//       for (i=0; i<n; i++) cout<<setw(6)<<Tbus[i]<<setw(6)<<i<<endl;
         p.get_locations_info(gstreet, Tnode_street, Tnode_transit);
//       cout <<endl<<"see:"<<endl;
//       for (int k=0; k<5; k++) cout<<setw(10)<<p.loc_info[k][0];
//       cin>>ch;
         p.location_pairs();

         p.bestpath_allpairs(Tnode_street, Tnode_transit, gstreet, gtransit, Tbus);

         p.modeseqs();
//       int N=p.get_N();  cout << "N=" << N << endl;
//       int m=(int)pow(3,N-1);
//       for (int i=0; i<m; i++) {
//               for (int j=0; j<N-1; j++) cout<<setw(6)<<p.modeseq[i][j];
//               cout<<endl;}

         p.nodeseqs();
//       n=p.get_locnum();
//       m = 1;
//       for (i=1; i<(n+1); i++)  m *=  i;
//       for (i=0; i<m; i++) {
//               for (int j=0; j<n; j++) cout<<setw(6)<<p.nodeseq[i][j];
//               cout<<endl<<endl;}

         if (p.includeMode[3] == 1) {
                 p.auto_seqs();
//               int N=p.get_N();
//               int m=p.get_AutoRowIndex();  cout << "AutoRowIndex=" << m << endl;
//               for (int i=0; i<m; i++)
//               {       for (int j=0; j<N+4; j++) cout<<setw(3)<<p.auto_nodeseq[i][j];
//                       cout <<"    ";
//                       for (j=0; j<N+3; j++) cout<<setw(3)<<p.auto_modeseq[i][j];
//                       cout<<endl;
//               }

                 p.ModifyAuto_seqs();
//               int N=p.get_N();
//               int m=p.get_AutoRowIndex();  cout << "AutoRowIndex=" << m << endl;
//               for (int i=0; i<m; i++)
//               {       for (int j=0; j<N+4; j++) cout<<setw(3)<<p.auto_nodeseq[i][j];
//                       cout <<"    ";
//                       for (j=0; j<N+3; j++) cout<<setw(3)<<p.auto_modeseq[i][j];
//                       cout<<endl;
//               }
         }

         p.constraited_bestpath(Tnode_street, Tnode_transit);

         cout << "Enter any letter-key to continue." <<endl;
         cin >> ch;

         return 0;
}
```

**Tppath.h:**

```
#ifndef __TPPATH_H__
#define __TPPATH_H__

#include <iostream.h>
#include <iomanip.h>
#include <stdio.h>
#include <math.h>

#include "strings.h"
#include "graph.h"

void error(const char *s);

class TPpath {
public:
         enum {  NODEROWS=800,                     //=MAX_locnom !   //
                 NODECOLS=10,                      //=MAX_locnum+1   //
                 MODEROWS=2500,                    //=3^(MAX_N-1)    //
                 MODECOLS=10,                      //=MAX_N          //
                 MAX_NODE_NUMBER=4000,
                 NODE_NUM_LIMIT=4001,
                 NO_FEASIBILITY_LIMIT=-9999};
         int nodeseq[NODEROWS][NODECOLS], modeseq[MODEROWS][MODECOLS];
         int auto_nodeseq[800][15], auto_modeseq[800][15], addmodes[800][20], CurAutoNodeseq[15];
         int s[NODECOLS+2], seq[NODECOLS+1][2], loc_info[10][10], pair[50][4][200];
         int includeMode[4];
         int get_locnum() {return locnum;}
         int get_N() {return N;}
         int get_AutoRowIndex() {return AutoRowIndex;}
         int get_addmodes(int ChangeLinkNum);
         int SeqFeasibility();
         inline int WalkTaxiBusTimeFeasibility(int modeseqth);
         inline int AutoTimeFeasibility(int modeseqth);
         void auto_seqs();
         void ModifyAuto_seqs();
         void get_WholeNodeseq(int nodeseqth);
         void get_CurAutoNodeseq(int auto_row);
         void zero_parkSeqs();
         void one_parkSeqs();
         void two_parkSeqs();
         void three_parkSeqs();
         void insertNode(int *nodebuf, int park_loc, int pick_loc);
         void insertMode(int *modebuf, int insertPos, int insertLen, int r);
         void insertTwoNode(int *nodebuf, int park_loc1, int pick_loc1, int park_loc2, int pick_loc2);
         void insertThreeNode(int *nodebuf, int park_loc1, int pick_loc1, int park_loc2, int pick_loc2, int
park_loc3, int pick_loc3);
         void insertTwoMode(int *modebuf, int insertPos1, int insertLen1, int insertPos2, int insertLen2,
int r);
```

```
        void insertThreeMode(int *modebuf, int insertPos1, int insertLen1, int insertPos2, int insertLen2,
int insertPos3, int insertLen3, int r);
        void get_locations_info(graph &g, strings &Tnode_street, strings &Tnode_transit);
        void get_activeNodeindex(int x, int y);
//      void get_activeNodeCoor(int nodeindex, int travelmode);
        void location_pairs();
        void bestpath_allpairs(strings &Tnode_street, strings &Tnode_transit, graph &gstreet, graph
&gtransit, strings &Tbus);
        void constraited_bestpath(strings &Tnode_street, strings &Tnode_transit);
        void getTaxiBestpathforAllpairs(graph &g);
        void getAutoBestpathforAllpairs(graph &g);
        void ReportBestpathforAllpairs();
        void ReportOneFeasiblePath(int isAuto, int nodeseqth, int modeseqth);
        void ReportPath(int pathIndex, int isAuto, int report_value, int report_nodeseq, int
report_modeseq);
        void OutputPath(strings &Tnode_street, strings &Tnode_transit, ofstream &f1, ofstream &f2, ofstream
&f3, ofstream &f4, int pathIndex, int isAuto, int output_value, int output_nodeseq, int output_modeseq);
        void nodeseqs();
        void modeseqs();
private:
        int locnum, N, Num_modeseqs, Num_pairs, endrow, AutoRowIndex, nodecoor[MAX_NODE_NUMBER][2];
        int Nodeindex_street, Nodeindex_transit, total_nodecoor;
        int NodeindexSeq_street[MAX_NODE_NUMBER], NodeindexSeq_transit[MAX_NODE_NUMBER];
        int travel_mode, path_value, path_traveltime;
        enum {nil=-1, inf=(INT_MAX/2), ZERO=0, AUTO_TRAVEL_MODE=4};
};

#endif
```

**Tppath.cpp:**

```
// TPpath:        Finding the best path of Traveling Planner Project studied at TIS of UC Davis
// Author:        Jiayu Chen, TIS, UCD
// Created date:  Feb., 1998
// Last Modified: November, 1998

// TPpath: Finding the best path of Traveling Planner Project studied at TIS of UC Davis
// Author: Jiayu Chen, TIS, UCD
// Created date:  Feb., 1998
// Last Modified: July, 1998
#include <iostream.h>
#include <iomanip.h>
#include <stdio.h>
#include <math.h>

#include "strings.h"
#include "graph.h"

void error(const char *s);

void TPpath::get_locations_info(graph &g, strings &Tnode_street, strings &Tnode_transit)
{       char s1[40], s2[40], buf[255];
        int j1, j2, j3, j4, j5, j6;

        ifstream ftotal("total.dat", ios::in | ios::nocreate);
        if (ftotal.fail()) error("Cannot open total.dat file.");
        while ( !ftotal.eof() ) {
                ftotal.getline(buf,255);
                if (buf[0] != '#') sscanf(buf, "%d %d %d %d %d\n", &locnum,
       &includeMode[0], &includeMode[1], &includeMode[2], &includeMode[3]);
        }
        ftotal.close();
        N=locnum+2;

        ifstream fnodecoor("table.txt", ios::in | ios::nocreate);
        if (fnodecoor.fail()) error("Cannot open aggr.dat file.");
        int row=0;
        while ( !fnodecoor.eof() ) {
                fnodecoor.getline(buf,255);
                if (buf[0] != '#') {
                        sscanf(buf, "%s %s %d %d", &s1, &s2, &j1, &j2);
                        NodeindexSeq_street[row] = Tnode_street.add(s1);
                        NodeindexSeq_transit[row] = Tnode_transit.add(s2);
                        nodecoor[row][0]=j1;
                        nodecoor[row][1]=j2;
                        row++;
                }
        }
        total_nodecoor = row;
        fnodecoor.close();

    ifstream flocinfo("input.dat", ios::in | ios::nocreate);
        if (flocinfo.fail()) error("Cannot open input.dat file.");
        row=0;
        while ( !flocinfo.eof() ) {
                flocinfo.getline(buf,255); // = flocinfo.get(buf,255,'\n');
                if (buf[0] != '#') {
                        sscanf(buf, "%d %d %d %d %d %d", &j1, &j2, &j3, &j4, &j5, &j6);
                        get_activeNodeindex(j1, j2);
                        loc_info[row][0] = Nodeindex_street;
                        loc_info[row][1] = Nodeindex_transit;

                        if ( j3 == NO_FEASIBILITY_LIMIT) loc_info[row][2] = j3;
                        else loc_info[row][2] = (int) (g.minChangetoE_1minFactor * j3);
//earliest arrival time

                        if ( j4 == NO_FEASIBILITY_LIMIT) loc_info[row][3] = j4;
                        else loc_info[row][3] = (int) (g.minChangetoE_1minFactor * j4);
//latest departure time

                        if ( j5 == NO_FEASIBILITY_LIMIT) loc_info[row][4] = j5;
                        else loc_info[row][4] = (int) (g.minChangetoE_1minFactor * j5);          //min
duration

                        if ( j6 == NO_FEASIBILITY_LIMIT) loc_info[row][5] = j6;
```

```
                              else loc_info[row][5] = (int) (g.minChangetoE_1minFactor * j6);           //max
duration

                              loc_info[row][6] = j1;
                              loc_info[row][7] = j2;
                              row++;
                    }
          }
          flocinfo.close();

          ifstream fseq("seq.dat", ios::in | ios::nocreate);
          if (fseq.fail()) error("Cannot open seq.dat file.");
          row=0;
          while ( !fseq.eof() ) {
                    fseq.getline(buf,255);
                    if (buf[0] != '#') {
                              sscanf(buf, "%d %d", &j1, &j2);
                              seq[row][0] = j1; seq[row][1] = j2;
                              row++;
                    }
          }
          fseq.close();
}

void TPpath::get_activeNodeindex(int x, int y)
{ int row, min_row;
   double min=(double)inf, D;
   for (row=0; row<total_nodecoor; row++)
   { D = abs(nodecoor[row][0]-x) + abs(nodecoor[row][1]-y);
             if (D < min) {min = D; min_row = row;}
  }
   Nodeindex_street = NodeindexSeq_street[min_row];
   Nodeindex_transit = NodeindexSeq_transit[min_row];
}

/*void TPpath::get_activeNodeCoor(int nodeindex, int travelmode)
{ int row, cur_row;

   for (row=0; row<total_nodecoor; row++)
  {       if (travelmode == 2)
          {         if (NodeindexSeq_transit[row] == nodeindex) {
                              cur_row = row;
                              break;}
          }
          else
          {         if (NodeindexSeq_street[row] == nodeindex) {
                              cur_row = row;
                              break;}
          }
  }
   NodeXcoor = nodecoor[cur_row][0];
   NodeYcoor = nodecoor[cur_row][1];
}*/

void TPpath::location_pairs()
{
          Num_pairs=(N-1)+(N-2)*(N-2);
          endrow=Num_pairs;
          for (int row=0; row<N-1; row++){
                    for (int i=0; i<3; i++) {
                              pair[row][i][0]=1;
                              pair[row][i][1]=row+2;
                    }
          }

          int k=2;
          while ( row<endrow )
          {         for(int j=1; j<N+1; j++)
                    {         if(j!=1 && j!=k)
                              {         for (int i=0; i<3; i++){
                                                  pair[row][i][0]= k;
                                                  pair[row][i][1]=j;}
                                        row++;
                              }
                    }
                    k++;
          }
}

void TPpath::bestpath_allpairs(strings &Tnode_street, strings &Tnode_transit, graph &gstreet,
                                                                              graph &gtransit, strings &Tbus)
{         int i, street_loc_seq[10], transit_loc_seq[10];

          for (int k=0; k<N; k++) street_loc_seq[k]=loc_info[k][0];
          for (k=0; k<N; k++) transit_loc_seq[k]=loc_info[k][1];

          for (int j=0; j<N-1; j++)
          {         gstreet.start = street_loc_seq[j];
                    gstreet.startIndex = j + 1;
                    i = 0;
                    if ((includeMode[0] == 1) || (includeMode[1] == 1) || (includeMode[3] == 1))
                    {         gstreet.travel_mode=i+1;
                              gstreet.prepare(Tbus);
                              while (gstreet.nInS < gstreet.n) {
                                        int ret = gstreet.ExpandSets(Tnode_street, Tbus, &street_loc_seq[0], N,
i);
                                        if (ret > 0)
                                                  break;
                              }
                              gstreet.output(Tnode_street, N, Num_pairs, pair, &street_loc_seq[0]);
                    }

                    i=2;
                    if ((includeMode[2] == 1) || (includeMode[3] == 1))
                    {         gtransit.start=transit_loc_seq[j];
```

```
                              gtransit.startIndex=j+1;
                              int n = Tbus.size();
                              for (int ibus=0; ibus<n; ibus++) gtransit.busStartDone[ibus]=0;
                              gtransit.travel_mode=i+1;
                              gtransit.prepare(Tbus);
                              while (gtransit.nInS < gtransit.n) {
                                        int ret = gtransit.ExpandSets(Tnode_transit, Tbus, &transit_loc_seq[0],
N, i);
                                        if (ret > 0)
                                        break;
                              }
                              gtransit.output(Tnode_transit, N, Num_pairs, pair, &transit_loc_seq[0]);
                    }
          }
          getTaxiBestpathforAllpairs(gstreet);
          getAutoBestpathforAllpairs(gstreet);
          ReportBestpathforAllpairs();
}

void TPpath::getTaxiBestpathforAllpairs(graph &g)
{         int row, kk, taxi_value, TaxiTravelT;
          float Length, TaxiCostValue;

          int k=1;
          int appliedMode=0;
          for (row=0; row<Num_pairs; row++)
          {         if (pair[row][appliedMode][2] == inf) taxi_value = pair[row][appliedMode][2];
                    else
                    {         Length = (1/g.lenZoomoutFactor) * pair[row][appliedMode][4];
                              TaxiTravelT = (int) (Length/g.taxi_speed);
                              TaxiCostValue = g.taxi_cost_rate * Length + g.taxiStartCost;
                              taxi_value = (int)(g.weight[0]*TaxiCostValue + g.weight[6]*TaxiTravelT);
                    }
                    pair[row][k][0]=pair[row][appliedMode][0];
                    pair[row][k][1]=pair[row][appliedMode][1];
                    pair[row][k][2]=taxi_value;
                    pair[row][k][3]=(int) (g.timeZoomoutFactor * TaxiTravelT);
                    pair[row][k][4]=pair[row][appliedMode][4];
                    pair[row][k][5]=(int) (g.costZoomoutFactor * TaxiCostValue);
                    pair[row][k][11]=pair[row][k][3];
                    pair[row][k][12]=pair[row][appliedMode][12];
                    for (kk=13; kk<13+pair[row][k][12]; kk++)
                              pair[row][k][kk] = pair[row][appliedMode][kk];
          }

}

void TPpath::getAutoBestpathforAllpairs(graph &g)
{         int row, kk, auto_value, AutoTravelT;
          float Length, AutoCostValue;

          int k=3;
          int appliedMode=0;
          for (row=0; row<Num_pairs; row++)
          {         if (pair[row][appliedMode][2] == inf) auto_value = pair[row][appliedMode][2];
                    else
                    {         Length = (1/g.lenZoomoutFactor) * pair[row][appliedMode][4];
                              AutoTravelT = (int) (Length/g.taxi_speed);
                              AutoCostValue = g.auto_cost_rate * Length;
                              if (pair[row][appliedMode][1] != N) AutoCostValue = AutoCostValue +
g.AutoParkCost;
                              auto_value = (int)(g.weight[0]*AutoCostValue + g.weight[6]*AutoTravelT);
                    }
                    pair[row][k][0]=pair[row][appliedMode][0];
                    pair[row][k][1]=pair[row][appliedMode][1];
                    pair[row][k][2]=auto_value;
                    pair[row][k][3]=(int) (g.timeZoomoutFactor * AutoTravelT);
                    pair[row][k][4]=pair[row][appliedMode][4];
                    pair[row][k][5]=(int) (g.costZoomoutFactor * AutoCostValue);
                    pair[row][k][11]=pair[row][k][3];
                    pair[row][k][12]=pair[row][appliedMode][12];
                    for (kk=13; kk<13+pair[row][k][12]; kk++)
                              pair[row][k][kk] = pair[row][appliedMode][kk];
          }

}

void TPpath::ReportBestpathforAllpairs()
{         int k, row, kk;

          cout << "in TPpath::bestpath_allpairs:" <<endl;
          cout << "row, travel_mode, startV, endV, TotalD, Total_time(1.0E-1min), total_len(meter),
total_cost(c), "
                    << "total_walkT(1.0E-1min), total_waitT(1.0E-1min), total_transfernum(#),
total_transitTravelT(1.0E-1min), "
                    << "total_cableTraverT(1.0E-1min), total_taxi/autoTravelT(1.0E-1min), Num_edge" << endl;
          for (row=0; row<Num_pairs; row++)
          {         for (k=0; k<4; k++)
                    {
          cout<<row<<setw(3)<<k+1<<setw(3)<<pair[row][k][0]<<setw(3)<<pair[row][k][1]<<setw(11);
                              for (kk=2; kk<13; kk++) cout << pair[row][k][kk] << setw(10);
                              cout << endl;
                    }
          }

}

void TPpath::constraited_bestpath(strings &Tnode_street, strings &Tnode_transit)
{         int flag1, flag2, flag3, min_value=inf, sec_value=inf, isAuto, autoMin, autoSec;
          int min_modeseq=ZERO, min_nodeseq=ZERO, sec_modeseq=ZERO, sec_nodeseq=ZERO, pathIndex;
          Num_modeseqs=(int)pow(3,N-1);

          int m = 1;
          for (int i=1; i<N-1; i++)  m *=  i;
          for (int nodeseqth=0; nodeseqth<m; nodeseqth++)
```

```
         {        get_WholeNodeseq(nodeseqth);

                  // check sequence constraint(see if seqence s[i] is satisfied for seq.dat)
                  flag1 = SeqFeasibility();

                  if (flag1 == 1)
                  {
                           //for walk and taxi modes:
                           if ((includeMode[0] == 1) || (includeMode[1] == 1) || (includeMode[2] == 1))
                           {        for (int modeseqth=0; modeseqth<Num_modeseqs; modeseqth++)
                                    {
                                             //  check the time constraint of nodeseqs[nodeseqth] for all
modeseqs
                                             flag2 = WalkTaxiBusTimeFeasibility(modeseqth);
                                             if (flag2 == 1)
                                             {        isAuto=0; //ReportOneFeasiblePath(isAuto, nodeseqth,
modeseqth);

                                                      if (path_value<min_value) {
                                                               sec_value=min_value; sec_modeseq=min_modeseq;
sec_nodeseq=min_nodeseq; autoSec=autoMin;
                                                               min_value=path_value; min_modeseq=modeseqth;
min_nodeseq=nodeseqth; autoMin=0;}

                                                      else if (path_value<sec_value) {
                                                               sec_value=path_value; sec_modeseq=modeseqth;
sec_nodeseq=nodeseqth; autoSec=0;}
                                             }
                                    }
                           }

                           //for auto mode:
                           if (includeMode[3] == 1)
                           {        for (int auto_row=0; auto_row<AutoRowIndex; auto_row++)
                                    {
                                             //  check the time constraint of nodeseqs[nodeseqth] for all
auto_modeseqs
                                             get_CurAutoNodeseq(auto_row);

                                             flag3 = AutoTimeFeasibility(auto_row);
                                             if (flag3 == 1)
                                             {        isAuto=1; //ReportOneFeasiblePath(isAuto, nodeseqth,
auto_row);

                                                      if (path_value<min_value) {
                                                               sec_value=min_value; sec_modeseq=min_modeseq;
sec_nodeseq=min_nodeseq; autoSec=autoMin;
                                                               min_value=path_value; min_modeseq=auto_row;
min_nodeseq=nodeseqth; autoMin=1;}

                                                      else if (path_value<sec_value) {
                                                               sec_value=path_value; sec_modeseq=auto_row;
sec_nodeseq=nodeseqth; autoSec=1;}
                                             }
                                    }
                           }

                  }
         }

         if ((min_value == inf) && (sec_value == inf))
         {        cout << "No feasible path available." << endl << endl;
                  exit(1);
         }

         ofstream fpathone("fpathone.dat", ios::out);
         ofstream fpathtwo("fpathtwo.dat", ios::out);
         ofstream fattrone("fattrone.dat", ios::out);
         ofstream fattrtwo("fattrtwo.dat", ios::out);
         ofstream fpathattr("fpathattr.dat", ios::out);
         ofstream fnodecoorone("fnodecoorone.dat", ios::out);
         ofstream fnodecoortwo("fnodecoortwo.dat", ios::out);
//       fpathattr-rewind function
         fpathattr << "#total_cost, total_walkT, total_waitT, total_transfernum,"
                           << "total_transitTravelT, total_cableTraverT, total_taxi/autoTravelT"
                           << endl;

         if (autoMin == 0) {        pathIndex=1; isAuto=0;
                                                      ReportPath(pathIndex, isAuto, min_value, min_nodeseq,
min_modeseq);
                                                      OutputPath(Tnode_street, Tnode_transit, fpathone,
fattrone, fpathattr, fnodecoorone, pathIndex, isAuto, min_value, min_nodeseq, min_modeseq);}
         if (autoMin == 1) {        pathIndex=1; isAuto=1;
                                                      ReportPath(pathIndex, isAuto, min_value, min_nodeseq,
min_modeseq);
                                                      OutputPath(Tnode_street, Tnode_transit, fpathone,
fattrone, fpathattr, fnodecoorone, pathIndex, isAuto, min_value, min_nodeseq, min_modeseq);}
         if (autoSec == 0) {        pathIndex=2; isAuto=0;
                                                      ReportPath(pathIndex, isAuto, sec_value, sec_nodeseq,
sec_modeseq);
                                                      OutputPath(Tnode_street, Tnode_transit, fpathtwo,
fattrtwo, fpathattr, fnodecoortwo, pathIndex, isAuto, sec_value, sec_nodeseq, sec_modeseq);}
         if (autoSec == 1) {        pathIndex=2; isAuto=1;
                                                      ReportPath(pathIndex, isAuto, sec_value, sec_nodeseq,
sec_modeseq);
                                                      OutputPath(Tnode_street, Tnode_transit, fpathtwo,
fattrtwo, fpathattr, fnodecoortwo, pathIndex, isAuto, sec_value, sec_nodeseq, sec_modeseq);}
         fpathattr.close();
}

void TPpath::get_WholeNodeseq(int nodeseqth)
{        int j;
         s[0]=1;
         for (j=1; j<N-1; j++) s[j]=nodeseq[nodeseqth][j-1]+1;
         s[N-1]=N;
```

```
}

void  TPpath::get_CurAutoNodeseq(int  auto_row)
{        int  i;

         CurAutoNodeseq[0]  =  auto_nodeseq[auto_row][0];
         for(i=0;  i<auto_nodeseq[auto_row][0];  i++)
                  CurAutoNodeseq[i+1]=s[auto_nodeseq[auto_row][i+1]-1];
}

int  TPpath::SeqFeasibility()
{        int  flag,  j1,  j2,  j5,  j6;

         flag = 1;
         for (j1 = 0; j1 < (N-1); j1++) {
                  if  (seq[j1][1]  !=  NO_FEASIBILITY_LIMIT)
                  {        j5 = 0;
                           j6 = 0;
                           for (j2 = 0; j2 < (N-1); j2++) {
                                    if (s[j2] == seq[j1][0]) j5 = j2;
                                    if (s[j2] == seq[j1][1]) j6 = j2;}
                           if (j5 > j6) {
                                    flag = 0;
                                    break;}
                   }
         }

         return  flag;
}

inline  int  TPpath::WalkTaxiBusTimeFeasibility(int  modeseqth)
{        int  flag,  startNode,  endNode;
         int  earlest_arrival_time,  startNode_arrival_time,  min_duration;
         int  startNode_departure_time,  startNode_latest_departure_time;
         int  endNode_arrival_time,  endNode_latest_departure_time;

         flag = 1;
         path_traveltime  =  0;
         path_value  =  0;

         for  (int  sth=0;  sth<N-1;  sth++)
    {    travel_mode  =  modeseq[modeseqth][sth]-1;
                  startNode=s[sth];
                  endNode=s[sth+1];
                  if (((travel_mode  ==  0)  &&  (includeMode[0]  ==  0))  ||
                           ((travel_mode  ==  1)  &&  (includeMode[1]  ==  0))  ||
                           ((travel_mode  ==  2)  &&  (includeMode[2]  ==  0)))
                  {        flag = 0;
                           return  flag;
                  }

                  for  (int  pairth=0;  pairth<Num_pairs;  pairth++)
                  {        if(  (pair[pairth][travel_mode][0]  ==  startNode)  &&  (pair[pairth][travel_mode][1]
==  endNode)  )
                           {        if  (pair[pairth][travel_mode][2]  ==  inf)  {flag  =  0;  return  flag;}
                                    path_value  =  path_value  +  pair[pairth][travel_mode][2];
                                    path_traveltime  =  path_traveltime  +  pair[pairth][travel_mode][3];
                                    break;
                           }
                  }

                  earlest_arrival_time  =  loc_info[startNode-1][2];
                  if  (sth  ==  0)  startNode_arrival_time  =  earlest_arrival_time;
                  else
                  {        if  (endNode_arrival_time  !=  -1)  startNode_arrival_time  =  endNode_arrival_time;

                           if  (endNode_arrival_time  ==  -1)  startNode_arrival_time  =  earlest_arrival_time;
                  }

                  min_duration  =  loc_info[startNode-1][4];
                  if  (min_duration  ==  NO_FEASIBILITY_LIMIT)  min_duration  =  0;

//......check  if  start  node  departure  time  satisfies  the  feasibility  condition
                  startNode_latest_departure_time  =  loc_info[startNode-1][3];
                  if  ((startNode_latest_departure_time  !=  NO_FEASIBILITY_LIMIT)  &&  (startNode_arrival_time
!=  NO_FEASIBILITY_LIMIT))
                  {        startNode_departure_time  =  startNode_arrival_time  +  min_duration;
                           if  (startNode_departure_time  >  startNode_latest_departure_time)  {
                                    flag = 0;
                                    return  flag;  }
                  }

//......check  if  the  end  node  arrival  time  satisfies  the  feasibility  condition
                  endNode_latest_departure_time  =  loc_info[endNode-1][3];
                  if  ((endNode_latest_departure_time  !=  NO_FEASIBILITY_LIMIT)  &&  (startNode_arrival_time  !=
NO_FEASIBILITY_LIMIT))
                  {        endNode_arrival_time  =  startNode_arrival_time  +  min_duration  +  path_traveltime;
                           if  (endNode_arrival_time  >  endNode_latest_departure_time)  {
                                    flag = 0;
                                    return  flag;  }
                  }  else
                  {        if  (startNode_arrival_time  !=  NO_FEASIBILITY_LIMIT)  endNode_arrival_time  =
startNode_arrival_time  +  min_duration  +  path_traveltime;
                           if  (startNode_arrival_time  ==  NO_FEASIBILITY_LIMIT)  endNode_arrival_time  =  -1;
                  }

         }

         return  flag;
}

inline  int  TPpath::AutoTimeFeasibility(int  modeseqth)
{        int  flag,  endPoint,  startNode,  endNode;
         int  earlest_arrival_time,  startNode_arrival_time,  min_duration;
         int  startNode_departure_time,  startNode_latest_departure_time;
         int  endNode_arrival_time,  endNode_latest_departure_time;

         flag = 1;
         path_traveltime  =  0;
         path_value  =  0;
         endPoint=CurAutoNodeseq[0];

         for  (int  sth=0;  sth<endPoint-1;  sth++)
         {        travel_mode  =  auto_modeseq[modeseqth][sth+1]-1;
          startNode=CurAutoNodeseq[sth+1];
                  endNode=CurAutoNodeseq[sth+2];

                  if (((travel_mode  ==  0)  &&  (includeMode[0]  ==  0))  ||
                          ((travel_mode  ==  1)  &&  (includeMode[1]  ==  0))  ||
                          ((travel_mode  ==  2)  &&  (includeMode[2]  ==  0)))
                  {        flag = 0;
                          return  flag;
                  }

                  for  (int  pairth=0;  pairth<Num_pairs;  pairth++)
                  {        if(  (pair[pairth][travel_mode][0]  ==  startNode)  &&  (pair[pairth][travel_mode][1]
==  endNode)  )
                          {        if  (pair[pairth][travel_mode][2]  ==  inf)  {flag  =  0;  return  flag;}
                                  path_value  =  path_value  +  pair[pairth][travel_mode][2];
                                  path_traveltime  =  path_traveltime  +  pair[pairth][travel_mode][3];
                                  break;
                          }
                  }

                  earlest_arrival_time  =  loc_info[startNode-1][2];
                  if  (sth  ==  0)  startNode_arrival_time  =  earlest_arrival_time;
                  else  startNode_arrival_time  =  endNode_arrival_time;


                  min_duration  =  loc_info[startNode-1][4];
                  if  (min_duration  ==  NO_FEASIBILITY_LIMIT)  min_duration  =  0;

                  startNode_latest_departure_time  =  loc_info[startNode-1][3];
                  if  ((startNode_latest_departure_time  !=  NO_FEASIBILITY_LIMIT)  &&  (startNode_arrival_time
!=  NO_FEASIBILITY_LIMIT))
                  {        startNode_departure_time  =  startNode_arrival_time  +  min_duration;
                          if  (startNode_departure_time  >  startNode_latest_departure_time)  {
                                  flag = 0;
                                  return  flag;  }
                  }

                  endNode_latest_departure_time  =  loc_info[endNode-1][3];
                  if  ((endNode_latest_departure_time  !=  NO_FEASIBILITY_LIMIT)  &&  (startNode_arrival_time  !=
NO_FEASIBILITY_LIMIT))
                  {        endNode_arrival_time  =  startNode_arrival_time  +  min_duration  +  path_traveltime;
                          if  (endNode_arrival_time  >  endNode_latest_departure_time)  {
                                  flag = 0;
                                  return  flag;  }
                  }  else
                  {        if  (startNode_arrival_time  !=  NO_FEASIBILITY_LIMIT)  endNode_arrival_time  =
startNode_arrival_time  +  min_duration  +  path_traveltime;
                          if  (startNode_arrival_time  ==  NO_FEASIBILITY_LIMIT)  endNode_arrival_time  =
startNode_arrival_time;
                  }

         }

         return  flag;
}

void  TPpath::ReportOneFeasiblePath(int  isAuto,  int  nodeseqth,  int  modeseqth)
{        if  (isAuto  ==  0)
         {        for  (int  ssth=0;  ssth<N;  ssth++)  cout  <<  setw(3)  <<  s[ssth];
                  cout  <<  ",";
                  for  (ssth=0;  ssth<N-1;  ssth++)
                          cout  <<  setw(3)  <<  resetiosflags(ios::left)  <<  modeseq[modeseqth][ssth];
         }
         if  (isAuto  ==  1)
         {        for  (int  ssth=0;  ssth<CurAutoNodeseq[0];  ssth++)
                          cout  <<  setw(3)  <<  CurAutoNodeseq[ssth+1];
                  cout  <<  ",";
                  for  (ssth=0;  ssth<CurAutoNodeseq[0]-1;  ssth++)
                          cout  <<  setw(3)  <<  resetiosflags(ios::left)  <<  auto_modeseq[modeseqth][ssth+1];
         }
         cout  <<  ",";
         cout  <<  resetiosflags(ios::left)  <<  "   path_value="
                   <<  setw(7)  <<  resetiosflags(ios::left)  <<  path_value
                   <<  resetiosflags(ios::left)  <<  "   nodeseqth="
                   <<  setw(1)  <<  resetiosflags(ios::left)  <<  nodeseqth
                   <<  resetiosflags(ios::left)  <<  "   modeseqth="
                   <<  setw(2)  <<  resetiosflags(ios::left)  <<  modeseqth<<endl;
}

void  TPpath::ReportPath(int  pathIndex,  int  isAuto,  int  report_value,  int  report_nodeseq,  int  report_modeseq)
{        int  j;
//       path:
         cout<<"path"<<pathIndex<<":  isAuto="<<isAuto<<endl;
         if  (report_value  <  inf)
         {        get_WholeNodeseq(report_nodeseq);

                  if  (isAuto  ==  0)
                  {        for  (j=0;  j<N;  j++)  cout  <<  setw(3)  <<  s[j];
                          cout  <<  ",";
                          for  (int  sth=0;  sth<N-1;  sth++)
                                  cout  <<  setw(3)  <<  modeseq[report_modeseq][sth];
                  }
                  if  (isAuto  ==  1)
                  {        get_CurAutoNodeseq(report_modeseq);
                          for  (j=0;  j<CurAutoNodeseq[0];  j++)
                                  cout  <<  setw(3)  <<  CurAutoNodeseq[j+1];
```

```
                                cout  <<  ",";
                                for  (int  sth=0;  sth<auto_modeseq[report_modeseq][0];  sth++)
                                           cout  <<  setw(3)  <<  auto_modeseq[report_modeseq][sth+1];
                      }
                      cout  <<  ",  ";
                      cout  <<  "   path_value="              <<report_value
                                 <<  "   path_nodeseq="       <<report_nodeseq
                                 <<  "   path_modeseq="       <<report_modeseq<<endl;
           }
           else  cout  <<  "No  path"  <<  pathIndex  <<  "  available."   <<endl;
}

void  TPpath::OutputPath(strings  &Tnode_street,  strings  &Tnode_transit,
                                                                 ofstream  &f1,  ofstream  &f2,  ofstream  &f3,  ofstream
&f4,
                                                                 int  pathIndex,  int  isAuto,  int  output_value,
                                                                 int  output_nodeseq,  int  output_modeseq)
{          int  j,  endPoint,  startNode,  endNode,  true_travel_mode,  path_attr[20];
           char  startV[30],  endV[30];
           for  (int  k=0;  k<20;  k++)  path_attr[k]=0;

//         Path:
           if  (output_value  <  inf)
           {          get_WholeNodeseq(output_nodeseq);

                      if  (isAuto  ==  0)  endPoint  =  N;
                      if  (isAuto  ==  1){
                                get_CurAutoNodeseq(output_modeseq);
                                endPoint  =  CurAutoNodeseq[0];
                      }

                      f2  <<  "#startV,  endV,  travel_mode,  Total_time(1.0E-1min),  total_len(meter),  total_cost(c),
"
                                <<"total_walkT(1.0E-1min),  total_waitT(1.0E-1min),  total_transfernum(#),  "
                                <<"total_transitTravelT(1.0E-1min),  total_cableTraverT(1.0E-1min),
total_taxi/autoTravelT(1.0E-1min)"
                                <<  endl;

                      for  (int  sth=0;  sth<endPoint-1;  sth++)
                      {          if  (isAuto  ==  0)
                                {          travel_mode=modeseq[output_modeseq][sth]-1;
                                           true_travel_mode  =  travel_mode+1;
                                           startNode=s[sth];  endNode=s[sth+1];
                                }
                                if  (isAuto  ==  1)
                                {          travel_mode=auto_modeseq[output_modeseq][sth+1]-1;
                                           true_travel_mode  =  travel_mode+1;
                                           startNode=CurAutoNodeseq[sth+1];  endNode=CurAutoNodeseq[sth+2];
                                }

                                if  (true_travel_mode  ==  3  )  {
                                           strcpy(startV,  Tnode_transit[loc_info[startNode-1][1]]);
                                           strcpy(endV,  Tnode_transit[loc_info[endNode-1][1]]);
                                }  else  {
                                           strcpy(startV,  Tnode_street[loc_info[startNode-1][0]]);
                                           strcpy(endV,  Tnode_street[loc_info[endNode-1][0]]);
                                }

                                int  NodeXcoor  =  loc_info[startNode-1][6];
                                int  NodeYcoor  =  loc_info[startNode-1][7];
                                f4  <<  setw(10)  <<  startV  <<  setw(12)  <<  NodeXcoor  <<  setw(12)  <<  NodeYcoor  <<  "
";
                                NodeXcoor  =  loc_info[endNode-1][6];
                                NodeYcoor  =  loc_info[endNode-1][7];
                                f4  <<  setw(10)  <<  endV  <<  setw(12)  <<  NodeXcoor  <<  setw(12)  <<  NodeYcoor  <<  endl;

                                for  (int  pairth=0;  pairth<Num_pairs;  pairth++)
                                {          if(  (pair[pairth][travel_mode][0]==startNode)  &&
                                                     (pair[pairth][travel_mode][1]==endNode)  )
                                           {
                                                     for  (k=2;  k<12;  k++)
                                                                path_attr[k]  =  path_attr[k]  +
pair[pairth][travel_mode][k];
                                                     path_attr[2]  =  output_value;

                                                     f1  <<  startV  <<  "  "  <<  endV  <<  endl;
                                                     int  N_edge=pair[pairth][travel_mode][12];
                                                     for  (j  =  13;  j  <  13  +  N_edge;  j++)
                                                                f1  <<  pair[pairth][travel_mode][j]  <<  "  "
                                                                    <<  true_travel_mode  <<  endl;

                                                     int  k  =  travel_mode;  int  col  =  pairth;
                                                     f2  <<  setw(10)  <<  startV  <<  setw(10)  <<  endV  <<  setw(4)  <<
true_travel_mode;
                                                     for  (int  kk=3;  kk<12;  kk++)  f2  <<  setw(10)  <<  pair[col][k][kk];
                                                     f2  <<  endl;

                                                     break;
                                           }
                                }
                      }

                      f2  <<  "#Path  atrributes:"  <<endl;
                      f2  <<  "#TotalD,  Total_time(1.0E-1min),  total_len(meter),  total_cost(c),  total_walkT(1.0E-
1min),  total_waitT(1.0E-1min),  "
                                <<"total_transfernum(#),  total_transitTravelT(1.0E-1min),
total_cableTraverT(1.0E-1min),  "
                                <<"total_taxi/autoTravelT(1.0E-1min)"  <<  endl;
                      for  (int  k=2;  k<12;  k++)  f2  <<  setw(10)  <<  path_attr[k];
                      f2  <<  endl;
                      for  (k=5;  k<12;  k++)  f3  <<  setw(10)  <<  path_attr[k];
                      f3  <<  endl;

                      f1.close();
                      f2.close();
```

```
                f4.close();
        }
        else cout << "No Path" << pathIndex << " available." << endl;
}

void  TPpath::auto_seqs()
{
  switch (N) {
  case 1:
          break;
  case 2:
    AutoRowIndex=0;
    zero_parkSeqs();
    break;
  case 3:
    AutoRowIndex=0;
    zero_parkSeqs();
    break;
  case 4:
    AutoRowIndex=0;
    zero_parkSeqs();
        AutoRowIndex++;
    one_parkSeqs();
    break;
  case 5:
    AutoRowIndex=0;
    zero_parkSeqs();
        AutoRowIndex++;
    one_parkSeqs();
    break;
  case 6:
    AutoRowIndex=0;
    zero_parkSeqs();
        AutoRowIndex++;
    one_parkSeqs();
    two_parkSeqs();
    break;
  case 7:
    AutoRowIndex=0;
    zero_parkSeqs();
        AutoRowIndex++;
    one_parkSeqs();
    two_parkSeqs();
    break;
  case 8:
    AutoRowIndex=0;
    zero_parkSeqs();
        AutoRowIndex++;
    one_parkSeqs();
    two_parkSeqs();
    three_parkSeqs();
    break;
  }

}

void  TPpath::ModifyAuto_seqs()
{       int i, j;
        for (i=0; i<AutoRowIndex; i++)
        {       for (j=0; j<N+3; j++) if (auto_modeseq[i][j] == 2) auto_modeseq[i][j] = 3;}
}

void  TPpath::zero_parkSeqs()
{       int i;
        auto_nodeseq[AutoRowIndex][0]=N;
        for (i=1; i<N+1; i++) auto_nodeseq[AutoRowIndex][i]=i;
        auto_modeseq[AutoRowIndex][0]=N-1;
        for (i=1; i<N; i++) auto_modeseq[AutoRowIndex][i]=AUTO_TRAVEL_MODE;
}

void  TPpath::insertNode(int *nodebuf, int park_loc, int pick_loc)
{       int i, newbuf[10];

        for (i=0; i<pick_loc; i++) newbuf[i]=nodebuf[i];
        newbuf[pick_loc]=park_loc;
        for (i=pick_loc+1; i<N+1; i++) newbuf[i]=nodebuf[i-1];
        for (i=0; i<N+1; i++) nodebuf[i]=newbuf[i];
}

void  TPpath::insertMode(int *modebuf, int insertPos, int insertLen, int r)
{       int i, newbuf[30];
        for (i=0; i<insertPos; i++) newbuf[i]=AUTO_TRAVEL_MODE;
        for (i=insertPos; i<insertPos+insertLen; i++) newbuf[i]=addmodes[r][i-insertPos];
        for (i=insertPos+insertLen; i<(N-1)+1; i++) newbuf[i]=AUTO_TRAVEL_MODE;

        for (i=0; i<(N-1)+1; i++) modebuf[i]=newbuf[i];
}

void  TPpath::one_parkSeqs()
{       int i, row, park_loc, pick_loc, nodebuf[10], modebuf[100];

        for (park_loc=2; park_loc<N-1; park_loc++)
        {       for (pick_loc=park_loc+1; pick_loc<N; pick_loc++)
                {

                        for (i=0; i<N; i++) nodebuf[i] = auto_nodeseq[0][i+1];
                        insertNode(&nodebuf[0], park_loc, pick_loc);

                        int ChangeLinkNum = pick_loc - park_loc + 1;
                        int total_row = get_addmodes(ChangeLinkNum);

                        for (row=AutoRowIndex; row<AutoRowIndex+total_row; row++)
                        {       auto_nodeseq[row][0] = auto_nodeseq[0][0] + 1;
                                for (i=0; i<N+1; i++) auto_nodeseq[row][i+1] = nodebuf[i];
                                for (i=0; i<N-1; i++) modebuf[i] = auto_modeseq[0][i+1];
```

```
                                        int insertPos = park_loc - 1;
                                        int insertLen = pick_loc - park_loc + 1;
                                        insertMode(&modebuf[0], insertPos, insertLen, row);
                                        auto_modeseq[row][0] = auto_modeseq[0][0] + 1;
                                        for (i=0; i<(N-1)+1; i++) auto_modeseq[row][i+1]=modebuf[i];
                                }
                                AutoRowIndex=row;

                        }
                }

}

void TPpath::insertTwoNode(int *nodebuf, int park_loc1, int pick_loc1, int park_loc2, int pick_loc2)
{       int i, newbuf[15];

        for (i=0; i<pick_loc1; i++) newbuf[i]=nodebuf[i];
        newbuf[pick_loc1]=park_loc1;
        for (i=pick_loc1+1; i<pick_loc2+1; i++) newbuf[i]=nodebuf[i-1];
        newbuf[pick_loc2+1]=park_loc2;
        for (i=pick_loc2+2; i<N+2; i++) newbuf[i]=nodebuf[i-2];
        for (i=0; i<N+2; i++) nodebuf[i]=newbuf[i];
}

void TPpath::insertTwoMode(int *modebuf, int insertPos1, int insertLen1, int insertPos2, int insertLen2, int
r)
{       int i, newbuf[30];
        for (i=0; i<insertPos1; i++) newbuf[i]=AUTO_TRAVEL_MODE;
        for (i=insertPos1; i<insertPos1+insertLen1; i++) newbuf[i]=addmodes[r][i-insertPos1];
        for (i=insertPos1+insertLen1; i<insertPos2+1; i++) newbuf[i]=AUTO_TRAVEL_MODE;
        for (i=insertPos2+1; i<insertPos2+insertLen2+1; i++) newbuf[i]=addmodes[r][i-insertPos2+insertLen1-
1];
        for (i=insertPos2+insertLen2+1; i<(N-1)+(insertLen1+insertLen2); i++) newbuf[i]=AUTO_TRAVEL_MODE;


        for (i=0; i<(N-1)+2; i++) modebuf[i]=newbuf[i];
}

void  TPpath::two_parkSeqs()
{       int i, row, park_loc1, pick_loc1, park_loc2, pick_loc2, nodebuf[15], modebuf[100];

        for (park_loc1=2; park_loc1<N-3; park_loc1++)
        {       for (pick_loc1=park_loc1+1; pick_loc1<N-2; pick_loc1++)
                {
                        for (park_loc2=park_loc1+2; park_loc2<N-1; park_loc2++)
                        {       for (pick_loc2=park_loc2+1; pick_loc2<N; pick_loc2++)
                                {
                                        if (pick_loc1 < park_loc2)
                                        {       for (i=0; i<N; i++) nodebuf[i] = auto_nodeseq[0][i+1];
                                                insertTwoNode(&nodebuf[0], park_loc1, pick_loc1,
park_loc2, pick_loc2);

                                                int ChangeLinkNum = (pick_loc1 - park_loc1 + 1) +
(pick_loc2 - park_loc2 + 1);
                                                int total_row = get_addmodes(ChangeLinkNum);

                                                for (row=AutoRowIndex; row<AutoRowIndex+total_row;
row++)
                                                {       auto_nodeseq[row][0] = auto_nodeseq[0][0] +
2;
                                                        for (i=0; i<N+2; i++) auto_nodeseq[row][i+1]
= nodebuf[i];
                                                        for (i=0; i<N-1; i++) modebuf[i] =
auto_modeseq[0][i+1];
                                                        int insertPos1 = park_loc1 - 1;
                                                        int insertLen1 = pick_loc1 - park_loc1 + 1;
                                                        int insertPos2 = park_loc2 - 1;
                                                        int insertLen2 = pick_loc2 - park_loc2 + 1;
                                                        insertTwoMode(&modebuf[0], insertPos1,
insertLen1, insertPos2, insertLen2, row);
                                                        auto_modeseq[row][0] = auto_modeseq[0][0] +
2;
                                                        for (i=0; i<(N-1)+2; i++)
auto_modeseq[row][i+1]=modebuf[i];
                                                }
                                                AutoRowIndex=row;
                                        }
                                }
                        }
                }
        }

}

void TPpath::insertThreeNode(int *nodebuf, int park_loc1, int pick_loc1, int park_loc2, int pick_loc2, int
park_loc3, int pick_loc3)
{       int i, newbuf[15];

        for (i=0; i<pick_loc1; i++) newbuf[i]=nodebuf[i];
        newbuf[pick_loc1]=park_loc1;
        for (i=pick_loc1+1; i<pick_loc2+1; i++) newbuf[i]=nodebuf[i-1];
        newbuf[pick_loc2+1]=park_loc2;
        for (i=pick_loc2+2; i<pick_loc3+2; i++) newbuf[i]=nodebuf[i-2];
        newbuf[pick_loc3+2]=park_loc3;
        for (i=pick_loc3+3; i<N+3; i++) newbuf[i]=nodebuf[i-3];
        for (i=0; i<N+3; i++) nodebuf[i]=newbuf[i];
}

void TPpath::insertThreeMode(int *modebuf, int insertPos1, int insertLen1, int insertPos2, int insertLen2,
int insertPos3, int insertLen3, int r)
{       int i, newbuf[30];
        for (i=0; i<insertPos1; i++) newbuf[i]=AUTO_TRAVEL_MODE;
        for (i=insertPos1; i<insertPos1+insertLen1; i++) newbuf[i]=addmodes[r][i-insertPos1];
        for (i=insertPos1+insertLen1; i<insertPos2+1; i++) newbuf[i]=AUTO_TRAVEL_MODE;
        for (i=insertPos2+1; i<insertPos2+insertLen2+1; i++) newbuf[i]=addmodes[r][i-insertPos2+insertLen1-
1];
        for (i=insertPos2+insertLen2+1; i<insertPos3+2; i++) newbuf[i]=AUTO_TRAVEL_MODE;
        for (i=insertPos3+2; i<insertPos3+insertLen3+2; i++) newbuf[i]=addmodes[r][i-
insertPos3+insertLen1+insertLen2-2];
        for (i=insertPos3+insertLen3+2; i<(N-1)+(insertLen1+insertLen2); i++) newbuf[i]=AUTO_TRAVEL_MODE;


        for (i=0; i<(N-1)+3; i++) modebuf[i]=newbuf[i];
}

void TPpath::three_parkSeqs()
{       int i, row, park_loc1, pick_loc1, park_loc2, pick_loc2, park_loc3, pick_loc3, nodebuf[15],
modebuf[100];

        for (park_loc1=2; park_loc1<N-5; park_loc1++)
        {       for (pick_loc1=park_loc1+1; pick_loc1<N-4; pick_loc1++)
                {
                        for (park_loc2=park_loc1+2; park_loc2<N-3; park_loc2++)
                        {       for (pick_loc2=park_loc2+1; pick_loc2<N-2; pick_loc2++)
                                {
                                        for (park_loc3=park_loc2+2; park_loc3<N-1; park_loc3++)
                                        {       for (pick_loc3=park_loc3+1; pick_loc3<N; pick_loc3++)
                                                {
                                                        if ((pick_loc1 < park_loc2) && (pick_loc2 <
park_loc3))
                                                        {
                                                                for (i=0; i<N; i++) nodebuf[i] =
auto_nodeseq[0][i+1];
                                                                insertThreeNode(&nodebuf[0],
park_loc1, pick_loc1, park_loc2, pick_loc2, park_loc3, pick_loc3);

                                                                int ChangeLinkNum = (pick_loc1 -
park_loc1 + 1) + (pick_loc2 - park_loc2 + 1) + (pick_loc3 - park_loc3 + 1);
                                                                int total_row =
get_addmodes(ChangeLinkNum);

                                                                for (row=AutoRowIndex;
row<AutoRowIndex+total_row; row++)
                                                                {       auto_nodeseq[row][0] =
auto_nodeseq[0][0] + 3;
                                                                        for (i=0; i<N+3; i++)
auto_nodeseq[row][i+1] = nodebuf[i];
                                                                        for (i=0; i<N-1; i++)
modebuf[i] = auto_modeseq[0][i+1];
                                                                        int insertPos1 = park_loc1
- 1;
                                                                        int insertLen1 = pick_loc1
- park_loc1 + 1;
                                                                        int insertPos2 = park_loc2
- 1;
                                                                        int insertLen2 = pick_loc2
- park_loc2 + 1;
                                                                        int insertPos3 = park_loc3
- 1;
                                                                        int insertLen3 = pick_loc3
- park_loc3 + 1;

        insertThreeMode(&modebuf[0], insertPos1, insertLen1, insertPos2, insertLen2, insertPos3,
insertLen3, row);
                                                                        auto_modeseq[row][0] =
auto_modeseq[0][0] + 3;
                                                                        for (i=0; i<(N-1)+3; i++)
auto_modeseq[row][i+1]=modebuf[i];
                                                                }
                                                                AutoRowIndex=row;
                                                        }
                                                }
                                        }
                                }
                        }
                }
        }

}

int TPpath::get_addmodes(int ChangeLinkNum)
{       int j1, j2, j3, j4, j5, j6, total_row;
        int row = AutoRowIndex-1;

        switch(ChangeLinkNum){
        case 2:
                for (j1=0; j1<2; j1++)
                {       for (j2=0; j2<2; j2++)
                        {       row++;
                                addmodes[row][0]=j1+1;
                                addmodes[row][1]=j2+1;
                        }
                }
                total_row = row-(AutoRowIndex-1);
                break;
        case 3:
                for (j1=0; j1<2; j1++)
                {       for (j2=0; j2<2; j2++)
                        {       for (j3=0; j3<2; j3++)
                                {       row++;
                                        addmodes[row][0]=j1+1;
                                        addmodes[row][1]=j2+1;
                                        addmodes[row][2]=j3+1;
                                }
                        }
                }
                total_row = row-(AutoRowIndex-1);
                break;
        case 4:
```

```
                    for (j1=0; j1<2; j1++)
                    {        for (j2=0; j2<2; j2++)
                             {        for (j3=0; j3<2; j3++)
                                      {        for (j4=0; j4<2; j4++)
                                               {        row++;
                                                        addmodes[row][0]=j1+1;
                                                        addmodes[row][1]=j2+1;
                                                        addmodes[row][2]=j3+1;
                                                        addmodes[row][3]=j4+1;
                                               }
                                      }
                             }
                    }
                    total_row = row-(AutoRowIndex-1);
                    break;
          case 5:
                    for (j1=0; j1<2; j1++)
                    {        for (j2=0; j2<2; j2++)
                             {        for (j3=0; j3<2; j3++)
                                      {        for (j4=0; j4<2; j4++)
                                               {        for (j5=0; j5<2; j5++)
                                                        {        row++;
                                                                 addmodes[row][0]=j1+1;
                                                                 addmodes[row][1]=j2+1;
                                                                 addmodes[row][2]=j3+1;
                                                                 addmodes[row][3]=j4+1;
                                                                 addmodes[row][4]=j5+1;
                                                        }
                                               }
                                      }
                             }
                    }
                    total_row = row-(AutoRowIndex-1);
                    break;
          case 6:
                    for (j1=0; j1<2; j1++)
                    {        for (j2=0; j2<2; j2++)
                             {        for (j3=0; j3<2; j3++)
                                      {        for (j4=0; j4<2; j4++)
                                               {        for (j5=0; j5<2; j5++)
                                                        {        for (j6=0; j6<2; j6++)
                                                                 {        row++;
                                                                          addmodes[row][0]=j1+1;
                                                                          addmodes[row][1]=j2+1;
                                                                          addmodes[row][2]=j3+1;
                                                                          addmodes[row][3]=j4+1;
                                                                          addmodes[row][4]=j5+1;
                                                                          addmodes[row][5]=j6+1;
                                                                 }
                                                        }
                                               }
                                      }
                             }
                    }
                    total_row = row-(AutoRowIndex-1);
                    break;
          }

   return total_row;
}

void  TPpath::nodeseqs()
{         int row;
          int j1, j2, j3, j4, j5, j6, j7, j8;
          int n=locnum;

switch (n) {
case 1:
          nodeseq[0][0]=1;
          break;
case 2:
          row=-1;
          for (j1=0; j1<n; j1++) {
           for (j2=0; j2<n; j2++) {
            if  (j2!=j1) {
                       row++;
                        nodeseq[row][0] = j1+1;
                        nodeseq[row][1] = j2+1;
            }
           }
  }
          break;
case 3:
          row=-1;
          for (j1=0; j1<n; j1++) {
           for (j2=0; j2<n; j2++) {
            if  (j2!=j1) {
             for (j3=0; j3< n; j3++) {
                    if  ((j3!=j2) && (j3!= j1)) {
                           row++;
                      nodeseq[row][0] = j1+1;
                      nodeseq[row][1] = j2+1;
                       nodeseq[row][2] = j3+1;
                    }
             }
            }
           }
          }
          break;
case 4:
          row=-1;
          for (j1=0; j1<n; j1++) {
           for (j2=0; j2<n; j2++) {
            if  (j2!=j1) {
             for (j3=0; j3<n; j3++) {
                     if  ((j3!=j2) && (j3!=j1)) {
                      for (j4=0; j4<n; j4++) {
                       if  ((j4!=j3) && (j4!=j2) && (j4!=j1)) {
                           row++;
                            nodeseq[row][0] = j1+1;
                            nodeseq[row][1] = j2+1;
                            nodeseq[row][2] = j3+1;
                           nodeseq[row][3] = j4+1;
                       }
                      }
                     }
            }
           }
          }
         }
         break;
case  5:
         row=-1;
         for (j1=0; j1<n; j1++) {
          for (j2=0; j2<n; j2++) {
           if  (j2!=j1) {
             for (j3=0; j3<n; j3++) {
              if  ((j3!=j2) && (j3!=j1)) {
               for (j4=0; j4<n; j4++) {
                if  ((j4!=j3) && (j4!=j2) && (j4!=j1)) {
                        for (j5=0; j5< n; j5++) {
                                 if  ((j5!=j4) && (j5!=j3) && (j5!=j2) && (j5!=j1)) {
                                          row++;
                                          nodeseq[row][0] = j1+1;
                                          nodeseq[row][1] = j2+1;
                                          nodeseq[row][2] = j3+1;
                                          nodeseq[row][3] = j4+1;
                                          nodeseq[row][4] = j5+1;
                         }
                 }
                }
          }
              }
             }
    }
          }
         }
         break;
case  6:
         row=-1;
         for (j1=0; j1<n; j1++) {
          for (j2=0; j2<n; j2++) {
           if  (j2!=j1) {
             for (j3=0; j3<n; j3++) {
                     if  ((j3!=j2) && (j3!=j1)) {
                      for (j4=0; j4<n; j4++) {
                       if  ((j4!=j3) && (j4!=j2) && (j4!=j1)) {
                        for (j5=0; j5<n; j5++) {
                                 if  ((j5!=j4) && (j5!=j3) && (j5!=j2) && (j5!= j1)) {
                                  for (j6=0; j6<n; j6++) {
                                   if  ((j6!=j5) && (j6!=j4) && (j6!=j3) && (j6!=j2) && (j6!=j1)) {
                                       row++;
                                        nodeseq[row][0] = j1+1;
                                        nodeseq[row][1] = j2+1;
                                        nodeseq[row][2] = j3+1;
                                        nodeseq[row][3] = j4+1;
                                        nodeseq[row][4] = j5+1;
                                        nodeseq[row][5] = j6+1;
                                  }
                                 }
                                }
                        }
                       }
                      }
                     }
            }
           }
          }
         }
         break;
case  7:
         row=-1;
         for (j1=0; j1<n; j1++) {
          for (j2=0; j2<n; j2++) {
           if  (j2!=j1) {
             for (j3=0; j3<n; j3++) {
                     if  ((j3!=j2) && (j3!=j1)) {
                      for (j4=0; j4<n; j4++) {
                       if  ((j4!=j3) && (j4!=j2) && (j4!=j1)) {
                        for (j5=0; j5<n; j5++) {
                                 if  ((j5!=j4) && (j5!=j3) && (j5!=j2) && (j5!=j1)) {
                                  for (j6=0; j6<n; j6++) {
                                   if  ((j6!=j5) && (j6!=j4) && (j6!=j3) && (j6!=j2) && (j6!=j1)) {
                                    for (j7=0; j7<n; j7++) {
                                            if  ((j7!=j6) && (j7!=j5) && (j7!=j4) && (j7!=j3) && (j7!=j2) &&
(j7!=j1)) {
                                                row++;
                                                 nodeseq[row][0] = j1+1;
                                                 nodeseq[row][1] = j2+1;
                                                 nodeseq[row][2] = j3+1;
                                                 nodeseq[row][3] = j4+1;
                                                 nodeseq[row][4] = j5+1;
                                                 nodeseq[row][5] = j6+1;
                                                 nodeseq[row][6] = j7+1;
                                           }
                                   }
                       }
                                 }
                                }
                        }
```

```
                            }
                          }
                        }
              }
            }
          }
        }
        break;
case 8:
        row=-1;
        for (j1=0; j1<n; j1++) {
         for (j2=0; j2<n; j2++) {
          if  (j2!=j1) {
           for (j3=0; j3<n; j3++) {
            if  ((j3!=j2) && (j3!=j1)) {
             for (j4=0; j4<n; j4++) {
              if  ((j4!=j3) && (j4!=j2) && (j4!=j1)) {
                     for (j5=0; j5<n; j5++) {
                      if  ((j5!=j4) && (j5!=j3) && (j5!=j2) && (j5!=j1)) {
                       for (j6=0; j6<n; j6++) {
                        if  ((j6!=j5) && (j6!=j4) && (j6!=j3) && (j6!=j2) && (j6!=j1)) {
                         for (j7=0; j7<n; j7++) {
                          if  ((j7!=j6) && (j7!=j5) && (j7!=j4) && (j7!=j3) && (j7!=j2) && (j7!=j1)) {
                           for (j8=0; j8<n; j8++) {
                            if  ((j8!=j7) && (j8!=j6) && (j8!=j5) && (j8!=j4) && (j8!=j3) && (j8!=j2) &&
(j8!=j1)) {
                                                      row++;
                                                       nodeseq[row][0] = j1+1;
                                                       nodeseq[row][1] = j2+1;
                                                       nodeseq[row][2] = j3+1;
                                                       nodeseq[row][3] = j4+1;
                                                       nodeseq[row][4] = j5+1;
                                                       nodeseq[row][5] = j6+1;
                                                       nodeseq[row][6] = j7+1;
                                                       nodeseq[row][7] = j8+1;
                                            }
                          }
                         }
                        }
                       }
                      }
                     }
                    }
                   }
              }
             }
            }
           }
          }
        }
        break;

}

}


void  TPpath::modeseqs()
{       int j1, j2, j3, j4, j5, j6, j7, row;
//      int  p32=(int)pow(3,2);p33=(int)pow(3,3);p34=(int)pow(3,4);p35=(int)pow(3,5);p36=(int)pow(3,6);
        int  endnum=3;

switch (N) {
case 2:
        modeseq[0][0] = 1;
        modeseq[1][0] = 2;
        modeseq[2][0] = 3;
        break;
case 3:
        row=-1;
        for (j1=0; j1<endnum; j1++) {
         for (j2=0; j2<endnum; j2++) {
//                row=3*j1+j2;
                  row++;
                  modeseq[row][0] = j1+1;
                  modeseq[row][1] = j2+1;
         }
  }
        break;
case 4:
        row=-1;
        for (j1=0; j1<endnum; j1++) {
         for (j2=0; j2<endnum; j2++) {
          for (j3=0; j3<endnum; j3++) {
//              row=p32*j1+3*j2+j3;
                   row++;
                   modeseq[row][0] = j1+1;
                   modeseq[row][1] = j2+1;
                   modeseq[row][2] = j3+1;
          }
         }
        }
        break;
case 5:
        row=-1;
        for (j1=0; j1<endnum; j1++) {
         for (j2=0; j2<endnum; j2++) {
          for (j3=0; j3<endnum; j3++) {
           for (j4=0; j4<endnum; j4++) {
//                   row=p33*j1+p32*j2+3*j3+j4;
                    row++;
                     modeseq[row][0] = j1+1;
                     modeseq[row][1] = j2+1;
                     modeseq[row][2] = j3+1;
                     modeseq[row][3] = j4+1;
```

```
                }
               }
              }
             }
             break;
case 6:
             row=-1;
             for (j1=0; j1<endnum; j1++) {
              for (j2=0; j2<endnum; j2++) {
               for (j3=0; j3<endnum; j3++) {
                for (j4=0; j4<endnum; j4++) {
                 for (j5=0; j5<endnum; j5++) {
//                        row=p34*j1+p33*j2+p32*j3+3*j4+j5;
                          row++;
                           modeseq[row][0] = j1+1;
                           modeseq[row][1] = j2+1;
                           modeseq[row][2] = j3+1;
                           modeseq[row][3] = j4+1;
                           modeseq[row][4] = j5+1;
                       }
                }
               }
              }
             }
             break;
case 7:
             row=-1;
             for (j1=0; j1<endnum; j1++) {
              for (j2=0; j2<endnum; j2++) {
               for (j3=0; j3<endnum; j3++) {
                for (j4=0; j4<endnum; j4++) {
                 for (j5=0; j5<endnum; j5++) {
                  for (j6=0; j6<endnum; j6++) {
//                                row=p35*j1+p34*j2+p33*j3+p32*j4+3*j5+j6;
                                  row++;
                                   modeseq[row][0] = j1+1;
                                   modeseq[row][1] = j2+1;
                                   modeseq[row][2] = j3+1;
                                   modeseq[row][3] = j4+1;
                                   modeseq[row][4] = j5+1;
                                   modeseq[row][5] = j6+1;
                        }
                       }
                }
               }
              }
             }
             break;
case 8:
             row=-1;
             for (j1=0; j1<endnum; j1++) {
              for (j2=0; j2<endnum; j2++) {
               for (j3=0; j3<endnum; j3++) {
                for (j4=0; j4<endnum; j4++) {
                 for (j5=0; j5<endnum; j5++) {
                  for (j6=0; j6<endnum; j6++) {
                   for (j7=0; j7<endnum; j7++) {
//                                row=p36*j1+p35*j2+p34*j3+p33*j4+p32*j5+3*j6+j7;
                                  row++;
                                   modeseq[row][0] = j1+1;
                                   modeseq[row][1] = j2+1;
                                   modeseq[row][2] = j3+1;
                                   modeseq[row][3] = j4+1;
                                   modeseq[row][4] = j5+1;
                                   modeseq[row][5] = j6+1;
                                   modeseq[row][6] = j7+1;
                         }
                        }
                       }
                }
               }
              }
             }
             break;

}

}
```

**Graph.h**

```
#ifndef __GRAPH_H__
#define __GRAPH_H__

#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <limits.h>

#include "strings.h"
#include "seqlist.h"

struct edge{int num, len, edgeindex, isbus, busindex, isDumLink, PrevBus, NextBus; edge *next;};

class graph {
         seqlist * graphSeqList;
public:
         enum {MAX_NODES=4000, MAX_EDGENUM=1000, MAX_BUSLINES=101};
         int n, N, start, finish, startIndex, finishIndex, taxiStartDone, busStartDone[MAX_BUSLINES];
         int inS_seq[MAX_NODES], edgeseq[MAX_EDGENUM];
         int nInS, travel_mode, Num_edge, walk_speed, taxi_speed, taxiStartCost, AutoParkCost;
         float weight[7], taxi_cost_rate, auto_cost_rate;
         float costZoominFactor, lenZoomoutFactor, timeZoominFactor, costZoomoutFactor, timeZoomoutFactor;
         float transfernumZoominFactor, minChangetoE_1minFactor;
         void InputBusInfo();
```

```
        void ini_weight();
        void update_weight();
        void SetParameters();
        void prepare(const strings &Tbus);
        void build(int isTransit, strings &Tnode, strings &Tbus, strings &Tdum, ifstream &file);
        void get_OFLength(int islink);
        void output(const strings &Tnode, int total_locN, int Num_pairs, int pair[50][4][200], int
*loc_seq);
        void ReportPath(const strings &Tnode, int iNode, int current_edgeindex);
        int ExpandSets(const strings &Tnode, const strings &Tbus, int *loc_seq, int total_locN, int
travelmode);
        int get_linkAttributes(const strings &Tbus, int i, int j);
        int get_edgeindex(int i, int j);
        int GetPrevIsbus(int PrevNode, int i);
        int GetTransitSpeed(int busnum);
        int GetTransitCost(int busnum);
        int GetTransferWaitT(int bus1, int bus2);
        int GetWaitT(int bus);
private:
        struct NodeType{int count, connect, inS, D, len, travel_time, travelcost, walkT, waitT,
                transfernum, transitTravelT, cableTraverT, taxiTravelT; edge *link;} *list;
        strings Tbus;
        int waitT[MAX_BUSLINES][MAX_BUSLINES][1], businfo[MAX_BUSLINES][2];
        int current_length, current_len, current_traveltime, current_travelcost, current_walkT,
current_waitT;
        int current_transfernum, current_transitTravelT, current_cableTraverT, current_taxiTravelT;
        int ReadName(strings &Tnode, istream &file, char &PrevChar);
        enum {nil=-1, inf=(INT_MAX/2), ZERO=0};
        void recalculate(const strings& Tbus, int w);
};

#endif
```

**tsp_grap.cpp:**

```
// Copyright(c) 1996 Leendert Ammeraal. All rights reserved.
// This program text occurs in Chapter 9 of
//
//     Ammeraal, L. (1996) Algorithms and Data Structures in C++,
//        Chichester: John Wiley.

// shpath: Finding the shortest path: Dijkstra's algorithm
//    (to be linked with strings.cpp, see Section 9.5).

// Modified by Jiayu Chen (Fab. 1998) at TIS of UCD
//        to fit the purpose of Traveling Planner Project
#include <stdio.h>
#include <iomanip.h>
#include <ctype.h>
#include <string.h>

#include "graph.h"

void graph::SetParameters()
{       taxi_cost_rate=(float)0.0003;       //($/meter)
        auto_cost_rate=(float)0.0001;       //($/meter)
        taxiStartCost=5000;                                   //(1.0E+3$)
        AutoParkCost=5000;                                    //(1.0E+3$)
        walk_speed=50;                                        //50(meter/min)
        taxi_speed=667;                                       //667(meter/min)
        costZoominFactor=(float)1000.;
        costZoomoutFactor=(float)0.1;
        timeZoominFactor=(float)1000.;
        timeZoomoutFactor=(float)0.01;
        lenZoomoutFactor=(float)0.001;
        minChangetoE_1minFactor=(float)10.;
        transfernumZoominFactor=(float)10.;
}

void error(const char *s)
{  cout << s << endl; exit(1);}

void graph::InputBusInfo()
{       char buf[255];
        int j1, j2, j3;

        int i=1;
        ifstream fbuswaiT("buswaitT.dat", ios::in | ios::nocreate);
        if (fbuswaiT.fail()) error("Cannot open buswaitT.dat file.");
        while ( !fbuswaiT.eof() ) {
                fbuswaiT.getline(buf,255); // = fbuswaiT.get(buf,255,'\n');
                if (buf[0] != '#') {
                        sscanf(buf, "%d %d %d", &j1, &j2, &j3);
                        waitT[j1][j2][0] = (int) (j3 * timeZoominFactor);
                }
        }
        fbuswaiT.close();

        ifstream fbusinfo("businfo.dat", ios::in | ios::nocreate);
        if (fbusinfo.fail()) error("Cannot open businfo.dat file.");
        int row=0;
        while ( !fbusinfo.eof() ) {
                fbusinfo.getline(buf,255);
                if (buf[0] != '#') {
                        sscanf(buf, "%d %d", &j1, &j2);
                        businfo[row][0] = j1;
        //transit_speed
                        businfo[row][1] = (int) (j2 * costZoominFactor);     //transit_cost
                        row++;
                }
        }
        fbusinfo.close();
}

int graph::ReadName(strings &Tnode, istream &file, char &PrevChar)
{  char s[100];
```

```
    s[0] = '\n';
    do
    {   PrevChar = s[0];
        file.get(s[0]);
    }   while (!file.fail() && !isalpha(s[0]));
    if (file.fail()) return -1;
    file >> setw(99) >> &s[1];
    return Tnode.add(s);
}

void graph::build(int isTransit, strings &Tnode, strings &Tbus, strings &Tdum, ifstream &file)
{   int i, j, len, edgeindex, isbus, busindex, isDumLink, PrevBus, NextBus, dumv;
    edge *p;
    char PrevChar, buf[100];
    const int ChunkSize = 64;
    N = n = 0;
    list = NULL;
    for (;;)
    {   j = ReadName(Tnode, file, PrevChar);
        if (j < 0) break;
        if (PrevChar == '\n')
        {  i = j;
           j = ReadName(Tnode, file, PrevChar);
        }
        file >> len;
        file >> edgeindex;
        if (isTransit == 1)
            {       file >> isbus;
                    if (isbus == 0) {file >> dumv; file >> dumv; file >> dumv; file >> dumv;}
                    if (isbus == 1) {
                            file >> isDumLink;
                            if (isDumLink == 0) {
                                    file >> setw(100) >> buf; busindex = Tbus.add(buf);
                                    file >> dumv; file >> dumv;}
                            if (isDumLink == 1) {
                                    file >> setw(100) >> buf; int dumindex = Tdum.add(buf);
                                    file >> setw(100) >> buf; PrevBus = Tbus.add(buf);
                                    file >> setw(100) >> buf; NextBus = Tbus.add(buf);}
                    }
            }
        if (file.eof()) break;
//       cout << "Nodei:" << Tnode[i] << "   Nodej" <<Tnode[j] << endl;
        if (file.fail()) error("Incorrect input file.");
        int max = i > j ? i : j;
        if (max >= N)
        {   NodeType *listOld = list;
            list = new NodeType[N += ChunkSize];
            for (int v=0; v<N; v++)
                if (v < n) list[v] = listOld[v]; else
                {  list[v].count = 0;
                   list[v].link = NULL;
                }
            delete[]listOld;
        }
        if (max >= n) n = max + 1;
        p = new edge;
        p->num = j; p->len = len; p->edgeindex = edgeindex;
            p->isbus = isbus;        p->busindex = busindex;
            p->isDumLink = isDumLink;         p->PrevBus = PrevBus; p->NextBus = NextBus;
        p->next = list[i].link;
        list[i].link = p;
        list[j].count++;
    }
    file.close();
}

void graph::ini_weight()
{       ofstream fweight("fweight.dat", ios::out);
        if (fweight.fail()) error("Cannot open fweight file.");

        weight[0] = (float) (0.1);          /* cost weight                                   */
        weight[1] = (float) (0.001);        /* waiting time weight                      */
        weight[2] = (float) (0.07);                 /* walking time weight                   */
        weight[3] = (float) (0.53);                 /* transfer number weight           */
        weight[4] = (float) (0.001);        /* transit-travel time weight       */
        weight[5] = (float) (0.001);        /* cable-travel time weigh          */
        weight[6] = (float) (0.3);          /* taxi/auto-travel time weight     */
        fweight << "#Weights of the objective function:" << endl;
        for (int k = 0; k < 7; k++) fweight << setw(10) <<weight[k];
        fweight.close();
}

void graph::update_weight()
{       char buf[255];
        int prefer_route, prefer_index;
        int j1, j2, j3, j4, j5, j6, j7, route_value[2][7];

//      getting previous weights
        ifstream finweight("fweight.dat", ios::in);
        if (finweight.fail()) error("Cannot open fweight.dat file.");
        while ( !finweight.eof() ) {
                finweight.getline(buf,255);
                if (buf[0] != '#') sscanf(buf, "%f %f %f %f %f %f %f",
                                                &weight[0],&weight[1],&weight[2],&weight[3],
                                                &weight[4],&weight[5],&weight[6]);
        }
        finweight.close();

//      getting the route attributes of 1st and 2nd route
        ifstream fpathattr("fpathattr.dat", ios::in);
        if (fpathattr.fail()) error("Cannot open pathattr.dat file.");
        int row=0;
        while ( !fpathattr.eof() ) {
                fpathattr.getline(buf,255);
                if (buf[0] != '#') {
                        sscanf(buf, "%d %d %d %d %d %d %d", &j1, &j2, &j3, &j4, &j5, &j6, &j7);
                        route_value[row][0] = j1;
                        route_value[row][1] = j2;
                        route_value[row][2] = j3;
                        route_value[row][3] = j4;
                        route_value[row][4] = j5;
                        route_value[row][5] = j6;
                        route_value[row][6] = j7;
                        row++;
                }
        }
        fpathattr.close();

//      getting user prefer_index
        ifstream fuserpref("userpref.dat", ios::in);
        if (fuserpref.fail()) error("Cannot open userpref.dat file.");
        fuserpref.getline(buf,255);
        sscanf(buf, "%d", &prefer_route);
        if (prefer_route == 1) prefer_index = 1;
        if (prefer_route == 2) prefer_index = -1;
        fuserpref.close();

//      updating weights
        if(route_value[0][0] < route_value[1][0]) weight[0]=weight[0]*(float)(pow(2,prefer_index));
        if(route_value[0][0] > route_value[1][0]) weight[0]=weight[0]/(float)(pow(2,prefer_index));
        if(route_value[0][1] < route_value[1][1]) weight[1]=weight[1]*(float)(pow(2,prefer_index));
        if(route_value[0][1] > route_value[1][1]) weight[1]=weight[1]/(float)(pow(2,prefer_index));
        if(route_value[0][2] < route_value[1][2]) weight[2]=weight[2]*(float)(pow(2,prefer_index));
        if(route_value[0][2] > route_value[1][2]) weight[2]=weight[2]/(float)(pow(2,prefer_index));
        if(route_value[0][3] < route_value[1][3]) weight[3]=weight[3]*(float)(pow(2,prefer_index));
        if(route_value[0][3] > route_value[1][3]) weight[3]=weight[3]/(float)(pow(2,prefer_index));
        if(route_value[0][4] < route_value[1][4]) weight[4]=weight[4]*(float)(pow(2,prefer_index));
        if(route_value[0][4] > route_value[1][4]) weight[4]=weight[4]/(float)(pow(2,prefer_index));
        if(route_value[0][5] < route_value[1][5]) weight[5]=weight[5]*(float)(pow(2,prefer_index));
        if(route_value[0][5] > route_value[1][5]) weight[5]=weight[5]/(float)(pow(2,prefer_index));
        if(route_value[0][6] < route_value[1][6]) weight[6]=weight[6]*(float)(pow(2,prefer_index));
        if(route_value[0][6] > route_value[1][6]) weight[6]=weight[6]/(float)(pow(2,prefer_index));
        float sum = weight[0] + weight[1] + weight[2] + weight[3]
                                + weight[4] + weight[5]+ weight[6];
        weight[0] = weight[0]/sum;
        weight[1] = weight[1]/sum;
        weight[2] = weight[2]/sum;
        weight[3] = weight[3]/sum;
        weight[4] = weight[4]/sum;
        weight[5] = weight[5]/sum;
        weight[6] = weight[6]/sum;

        ofstream foutweight("fweight.dat", ios::out);
        if (foutweight.fail()) error("Cannot open fweight.dat file.");
        foutweight << "#Weights of the objective function:" << endl;
        for (int k=0; k<7; k++) foutweight << setw(10) <<weight[k];
        foutweight.close();
}

int graph::get_edgeindex(int i, int j)
{       edge *p = list[i].link;
        while (p)
        {       if (p->num == j)   return (p->edgeindex);
                p = p->next;
        }
        return inf;
}

void graph::get_OFLength(int islink)
{       if (islink == 1)
        {       current_length =(int)(weight[0] * current_travelcost +
                                                weight[1] * current_waitT +
                                                weight[2] * current_walkT +
                                                weight[3] * transfernumZoominFactor *
current_transfernum +
                                                weight[4] * current_transitTravelT +
                                                weight[5] * current_cableTraverT +
                                                weight[6] * current_taxiTravelT);
                current_traveltime = current_waitT +
                                                        current_walkT +
                                                        current_transitTravelT +
                                                        current_cableTraverT +
                                                        current_taxiTravelT;
        }
        if (islink ==0)
        {       current_length=inf;
                current_traveltime=inf;
        }
}

int graph::get_linkAttributes(const strings &Tbus, int i, int j)
{       int transit_speed, PrevNode;
        edge *p = list[i].link;

        while (p)
        {       if (p->num == j)   // return (p->len);
                {       current_len=p->len;
                        current_travelcost=0;
                        current_waitT=0;
                        current_walkT=0;
                        current_transfernum=0;
                        current_transitTravelT=0;
                        current_cableTraverT=0;
                        current_taxiTravelT=0;
                        if (travel_mode == 1) current_walkT=(p->len)/walk_speed;
                        if (travel_mode == 2) {
                                current_travelcost=current_travelcost+(int)((p->len)*taxi_cost_rate);
                                current_taxiTravelT=(p->len)/taxi_speed;
                        }
                        if (travel_mode == 3) {
```

```
                                    if (p->isbus == 0) current_walkT=(p->len)/walk_speed;
                                    if (p->isbus == 1) {
                                             if (p->isDumLink == 0) {
                                                      if (i == start) {
                                                               current_waitT=GetWaitT(p->busindex);
                                                               current_travelcost=GetTransitCost(p-
>busindex);
                                                               busStartDone[p->busindex]=1;
                                                      }

                                                      if (i != start) {
                                                               PrevNode=list[i].connect;
                                                               int ret=GetPrevIsbus(PrevNode,i);
                                                               if (ret == 0) {
                                                                        current_transfernum=1;
                                                                        current_waitT=GetWaitT(p->busindex);
                                                                        current_travelcost=GetTransitCost(p-
>busindex);
                                                               }
                                                               if (busStartDone[p->busindex] == 0)
                                                                        busStartDone[p->busindex]=1;
                                                      }

                                                      transit_speed=GetTransitSpeed(p->busindex);
                                                      if (Tbus[p->busindex] != "cable-car")
current_transitTravelT=(p->len)/transit_speed;
                                                      if (Tbus[p->busindex] == "cable-car")
current_cableTraverT=(p->len)/transit_speed;
                                             }

                                             if (p->isDumLink == 1) {
                                                      current_travelcost=GetTransitCost(p->NextBus);
                                                      current_waitT=GetTransferWaitT(p->PrevBus, p-
>NextBus);
                                                      current_transfernum=1;
                                             }
                                    }
                           }
                           return 1;
                  }
                  p = p->next;
         }

         current_len=inf;
         current_travelcost=inf;
         current_waitT=inf;
         current_walkT=inf;
         current_transfernum=inf;
         current_transitTravelT=inf;
         current_cableTraverT=inf;
         current_taxiTravelT=inf;
         return 0;
}

int graph::GetPrevIsbus(int PrevNode, int i)
{        edge *p = list[PrevNode].link;
         while (p)
         {        if (p->num == i) return p->isbus;
                  p = p->next;
         }
         return inf;
}

int graph::GetTransitSpeed(int busnum)
{        return businfo[busnum][0]; }

int graph::GetTransitCost(int busnum)
{        return businfo[busnum][1]; }

int graph::GetWaitT(int bus)
{        return (int) (waitT[bus+1][bus+1][0]); }

int graph::GetTransferWaitT(int bus1, int bus2)
{        return (int) (waitT[bus1+1][bus2+1][0]); }


void graph::prepare(const strings &Tbus)
{        int islink;
         for (int v=0; v<n; v++)
         {        if (v==start) {list[v].D=ZERO; list[v].travel_time=ZERO;}
                  if (v!=start) {
                           islink = get_linkAttributes(Tbus,start,v);
                           get_OFLength(islink);
                           list[v].D = current_length;
                           list[v].len = current_len;
                           list[v].travel_time = current_traveltime;
                           list[v].travelcost = current_travelcost;
                           list[v].waitT = current_waitT;
                           list[v].walkT = current_walkT;
                           list[v].transfernum = current_transfernum;
                           list[v].transitTravelT = current_transitTravelT;
                           list[v].cableTraverT = current_cableTraverT;
                           list[v].taxiTravelT = current_taxiTravelT;
                  }
//                list[v].D = (v == start ? 0 : length(start, v));
                  list[v].inS = 0;
                  list[v].connect = start;
         }
         list[start].inS = 1;
         nInS = 1;
         inS_seq[0]=start;
         graphSeqList = new seqlist();
         edge *pp = list[start].link;
         while(pp) {
                  graphSeqList->addNewNode(pp->num, list[pp->num].D);
```

```
                    pp = pp->next;
          }
// cout << "In prepare graphSeqList = " << graphSeqList << "\n";
}


int graph::ExpandSets(const strings &Tnode, const strings &Tbus, int *loc_seq, int total_locN, int
travelmode)
{         int w = inf, min = INT_MAX, deadNodeIndex;
          char deadNode[30];

// Choose a vertex w in inS's adjacent vertices such that list[w].D
// is a minimum:

#if 0
          for (Ins_v=0; Ins_v<nInS; Ins_v++) {
                    edge *p = list[inS_seq[Ins_v]].link;
                    while (p) {
                              v = p->num;
                              if (list[v].inS == 0)
                              {         if (list[v].D < min)
                                        {         min = list[v].D;
                                                  w = v;
                                        }
                              }
                              p = p->next;
                    }
          }
#endif

          // cout << "Expand graphSeqList = " << graphSeqList << "\n";
          seqNode * ref  = graphSeqList->popFirstNode();
          if(! ref ) {
                    strcpy(deadNode, Tnode[deadNodeIndex]);
                    if( travelmode == 2) cout << "Transit-network broken at "<<deadNode<<"."<<endl;
                    else cout << "The path broken at Street-network "<<deadNode<<"."<<endl;
                    return 1;
          }

          w = ref->nodeNumber; delete ref;
          deadNodeIndex = w;
//        Move w from V to S:
          list[w].inS = 1;
          inS_seq[nInS++]=w;

//        Check if proper activity locations are included in inS:
          int IsFinish = 1;
          for (int i=1; i<total_locN; i++) IsFinish=IsFinish*list[loc_seq[i]].inS;
          if (IsFinish == 1) return 1;


          recalculate(Tbus, w);

          return 0;
}

void      graph::recalculate(const strings & Tbus, int w)
{
//        Update distances list[v].D and predecessors list[v].connect
//        for all adjacencies v of w in V:
          edge *p = list[w].link;
          while (p) {
                    int v = p->num;
                    if (list[v].inS == 0)
                    {         int islink=get_linkAttributes(Tbus,w,v);
                              get_OFLength(islink);
                              int sumD = list[w].D + current_length;
                              int sum_len = list[w].len + current_len;
                              int sum_traveltime = list[w].travel_time + current_traveltime;
                              int sum_travelcost = list[w].travelcost + current_travelcost;
                              int sum_waitT = list[w].waitT + current_waitT;
                              int sum_walkT = list[w].walkT + current_walkT;
                              int sum_transfernum = list[w].transfernum + current_transfernum;
                              int sum_transitTravelT = list[w].transitTravelT + current_transitTravelT;
                              int sum_cableTraverT = list[w].cableTraverT + current_cableTraverT;
                              int sum_taxiTravelT = list[w].taxiTravelT + current_taxiTravelT;;

                              if (sumD < list[v].D)
                              {
                    // cout << "recalculate: graphSeqList = " << graphSeqList << "\n";
                                        graphSeqList->addNode(v, sumD, list[v].D);
                                        list[v].D = sumD;
                                        list[v].len = sum_len;
                                        list[v].travel_time = sum_traveltime;
                                        list[v].travelcost = sum_travelcost;
                                        list[v].waitT = sum_waitT;
                                        list[v].walkT = sum_walkT;
                                        list[v].transfernum = sum_transfernum;
                                        list[v].transitTravelT = sum_transitTravelT;
                                        list[v].cableTraverT = sum_cableTraverT;
                                        list[v].taxiTravelT = sum_taxiTravelT;
                                        list[v].connect = w;
                              }
                    }
                    p = p->next;
          }
}


void graph::output(const strings &Tnode, int total_locN, int Num_pairs, int pair[50][4][200], int *loc_seq)
{         int start_vertex, end_vertex, current_edgeindex, cur_row, Vseq[1000];

          for (int k=1; k<total_locN; k++)
          {         if (start != loc_seq[k])
```

```
                    {        finish = loc_seq[k];
                             finishIndex=k+1;
                             Num_edge = 0;
//                           cout << "start       finish"<<endl;
//                           cout  <<setw(10)<<start<<setw(10)<<finish<<endl;

                             for(int row=0; row<Num_pairs; row++)
                             {        if ((startIndex == pair[row][travel_mode-1][0]) && (finishIndex ==
pair[row][travel_mode-1][1]))
                                      {        cur_row=row;
                                               break;
                                      }
                             }

                             if (list[finish].D == inf) {
                                      pair[cur_row][travel_mode-1][2]=list[finish].D;
                                      ReportPath(Tnode, finish, 0);
                             }
                             else {
                                      // Replace predecessors with successors:
                                      int j = finish, i, jnext = 0;
                                      int Vcount=0; Vseq[Vcount]=j;
                                      for (;;)
                                      {        i = list[j].connect;
                                               Vcount++;
                                               Vseq[Vcount]=i;
                                               if (j == start) break;
                                               j = i;
                                      }

                                      i = Vseq[Vcount-1];
                                      int end_vertexIndex=Vcount-2;
                                      for (;;)
                                      {        if(i != finish)
                                               {        start_vertex = i;
                                                        end_vertex = Vseq[end_vertexIndex];
                                                        current_edgeindex = get_edgeindex(start_vertex,
end_vertex);
                                                        edgeseq[Num_edge++]=current_edgeindex;
                                               }

                                               //ReportPath(Tnode, i, current_edgeindex);

                                               if (i == finish) break;
                                               i = end_vertex;
                                               end_vertexIndex=end_vertexIndex-1;
                                      }

                                      pair[cur_row][travel_mode-1][2]=list[finish].D;
                                      pair[cur_row][travel_mode-1][3]=(int)(timeZoomoutFactor  *
list[finish].travel_time);
                                      pair[cur_row][travel_mode-1][4]=(int)(lenZoomoutFactor  *
list[finish].len);
                                      pair[cur_row][travel_mode-1][5]=(int)(costZoomoutFactor  *
list[finish].travelcost);
                                      pair[cur_row][travel_mode-1][6]=(int)(timeZoomoutFactor  *
list[finish].walkT);
                                      pair[cur_row][travel_mode-1][7]=(int)(timeZoomoutFactor  *
list[finish].waitT);
                                      pair[cur_row][travel_mode-1][8]=list[finish].transfernum;
                                      pair[cur_row][travel_mode-1][9]=(int)(timeZoomoutFactor  *
list[finish].transitTravelT);
                                      pair[cur_row][travel_mode-1][10]=(int)(timeZoomoutFactor  *
list[finish].cableTraverT);
                                      pair[cur_row][travel_mode-1][11]=(int)(timeZoomoutFactor  *
list[finish].taxiTravelT);
                                      pair[cur_row][travel_mode-1][12]=Num_edge;
                                      for (int col = 13; col < 13 + Num_edge; col++)
pair[cur_row][travel_mode-1][col]  =  edgeseq[col-13];
                             }
                    }
          }
}

void graph::ReportPath(const strings &Tnode, int iNode, int current_edgeindex)
{         int i = iNode;
          if ((i == finish) && (current_edgeindex == 0)) {
                    cout<<"No path from Location "<<startIndex<<" to Location "<<finishIndex<<endl<<endl;
                    //cout << "graphSeqList is empty !!! \n";
                    return;
          }
          if (i == start)
          {         cout << "Shortest path: travel_mode=" << travel_mode << endl << endl;
                    cout << "          Vertex               Distance            Edgeindex\n";
          }

          cout << (i == finish ? "finish = " :
                             i == start  ? "start  = " :
             "         ")
                    << setw(20) << setiosflags(ios::left) << Tnode[i] << "   "
                    << setw(6) << resetiosflags(ios::left) << list[i].D;
          if(i != finish) cout << setw(15) << resetiosflags(ios::left) << current_edgeindex;

          cout << endl<< endl;
}
```

**Seqlist.h:**

```
#ifndef __SEQLIST_H__
#define __SEQLIST_H__

struct seqNode {
          int nodeNumber;
          int distance;
          seqNode * next;
};
```

```
class seqlist {
        seqNode * head;
        int     nNode;
public:
        seqlist();
        ~seqlist();
        void addNode(int nodeNumber, int distance, int oldDistance);
        void addNewNode(int nodeNumber, int distance);
        void seqlist::deleteNode(int nodeNumber, int D);
        inline total() { return nNode; } ;
        seqNode * popFirstNode();
};

#endif /* __SEQLIST_H__ */
```

**Seqlist.cpp:**

```
#if 0
struct seqNode {
        int nodeNumber;
        seqNode * next;
};

class seqlist {
        seqNode * head;
        int     nNode;
        seqlist(int number);
        ~seqlist();
        addNewNode(int nodeNumber, int distance);
        inline total() { return nNode; } ;
        int popFirstNode();
};
#endif

#include "seqlist.h"

seqNode* seqlist::popFirstNode()
{
        seqNode * p = head;

        if(p) {
                head = p->next;
                int n = p->nodeNumber;
                nNode --;
                // cout << "Pop node: " << n << "\n";
// cout.flush();

                return p;
        }
        return 0;
}
void seqlist::addNewNode(int nodeNumber, int distance)
{
// cout << "add " << nodeNumber << " " << distance << "\n";
// cout.flush();
        seqNode *p = new seqNode;
        p->nodeNumber = nodeNumber;
        p->next = 0;

        if(head == 0) {
                head = p;
                return;
        }

        seqNode * thisOne = head;
        seqNode * nextOne = head;
        while(nextOne)  {
// cout << "nextOne = " << nextOne << " node = " << nextOne->nodeNumber << "\n";

                if(distance > nextOne->distance) {
                        thisOne = nextOne;
                        nextOne = nextOne->next;
                } else if(distance == nextOne->distance) {
                        if(nodeNumber == nextOne->nodeNumber) {
                                // return if we are already in there
                                delete p;
                                return;
                        }
                        thisOne = nextOne;
                        nextOne = nextOne->next;
                } else {
                        // add here
                        p->next = thisOne->next;
                        thisOne->next = p;
                        return;
                }
        }
        p->next = thisOne->next;
        thisOne->next = p;
}

void seqlist::addNode(int nodeNumber, int distance, int oldDistance)
{
// cout << "add " << nodeNumber << " " << distance << " old=" << oldDistance << "\n";
// cout.flush();
        seqNode *p = new seqNode;
        p->nodeNumber = nodeNumber;
        p->distance   = distance;
        p->next = 0;

        if(head == 0) {
                head = p;
                return;
        }
        seqNode * thisOne = head;
```

```
         seqNode * nextOne = head;
         while(nextOne)   {
                  if(distance > nextOne->distance) {
                           thisOne = nextOne;
                           nextOne = nextOne->next;
                  } else if(distance == nextOne->distance) {
                           if(nodeNumber == nextOne->nodeNumber) {
                                    // return if we are already in there
                                    delete p;
                                    return;
                           }
                           thisOne = nextOne;
                           nextOne = nextOne->next;
                  } else {
                           // add here
                           break;
                  }
         }
         p->next = thisOne->next;
         thisOne->next = p;
         deleteNode(nodeNumber, oldDistance);
}

void seqlist::deleteNode(int nodeNumber, int D) {
         seqNode * p = head;

         if(p->nodeNumber == nodeNumber &&
             p->distance    == D)
         {
                  head = p->next;
                  return;
         }

         while(p->next) {
// cout << p->nodeNumber << " " << p->distance << "\n";
                  if(p->next->nodeNumber == nodeNumber &&
                     p->distance           == D) {
                           p->next = p->next->next;
                           return;
                  }
                  p = p->next;
         }
}


seqlist::~seqlist()
{
         seqNode *p = head;
         while(p) {
                  seqNode * q = p;
                  p = p->next;
                  delete q;
         }
}

seqlist::seqlist()
{
         head = 0;
         nNode = 1;
}
```

**Strings.h:**

```
// Copyright(c) 1996 Leendert Ammeraal. All rights reserved.
// This program text occurs in Chapter 9 of
//
//     Ammeraal, L. (1996) Algorithms and Data Structures in C++,
//        Chichester: John Wiley.

// strings.h: Strings stored and represented
//             by ints 0, 1, 2, ...
#ifndef __STRINGS_H__
#define __STRINGS_H__

#include <iostream.h>

class stringindex {
         int num;
         const char * key;
         stringindex * left;
         stringindex * right;

public:
         stringindex(int n, const char * s) { left = right = 0; key = s; num = n; }
         ~stringindex() { if(left) delete left; if(right) delete right; }
         void add(int n, const char * s);
         stringindex * search(const char * p);
         int index() { return num; }
         const char * value() { return key; }
};

class strings {
public:
    strings(){p = NULL; N = n = 0; root = 0; }
    ~strings();
    int add(const char *s);
     char *operator[](int i)const{return p[i];}
    // int read(istream &f);
     int size()const{return n;}
private:
    stringindex * root;
    enum {ChunkSize = 64};
    int search(const char *s);
    char **p;
    int n, N;
};
```

```
#endif
```

**Strings.cpp:**

```
// Copyright(c) 1996 Leendert Ammeraal. All rights reserved.
// This program text occurs in Chapter 9 of
//
//     Ammeraal, L. (1996) Algorithms and Data Structures in C++,
//        Chichester: John Wiley.

// strings.cpp: See header file strings.h
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <string.h>


#include "strings.h"

typedef char *charptr;

void stringindex::add(int n, const char * s) {
         int ret = strcmp(s, key);
         if(ret == 0) return;
         if(ret < 0)  { // left
              if(left)
                   left->add(n, s);
              else
                   left = new stringindex(n, s);
         }
         if(ret > 0) { // right
              if(right)
                   right->add(n, s);
              else
                   right = new stringindex(n, s);
         }
    }
stringindex * stringindex::search(const char * s) {
         int ret = strcmp(s, key);
         if(ret == 0)
              return this;
         if(ret < 0) {
              return left ? left->search(s) : 0;
         } else {
              return right ? right->search(s) : 0;
         }
}

strings::~strings()
{   for (int i=0; i<n; i++) delete[](p[i]);
    delete[]p;
}
int strings::add(const char *s)
{
    int i = search(s);
    if(i >= 0)
         return i;     // we already found it
    if (n == N)
    {  // Re-allocate:
       char **pOld = p;
       p = new charptr[N += ChunkSize];
       for (i=0; i<n; i++) p[i] = pOld[i];
       delete[]pOld;
    }
    p[n] = new char[strlen(s) + 1];
    strcpy(p[n], s);
    if(root == 0) { // first time
         root = new stringindex(n, p[n]);
    } else {
         root->add(n, p[n]);
    }
    return n ++;
}

int strings::search(const char * s)
{
    // for (int i=0; i<n; i++)
    // if (strcmp(p[i], s) == 0) return i;
    if(root) {
         stringindex * ret = root->search(s);
         if(ret > 0)
              return ret->index();
    }
    return -1;
}


#if 0
int strings::read(istream &f)
{   char buf[100];
    f >> setw(100) >> buf;
    return f.fail() ? -1 : add(buf);
}
#endif
```