Title
Daily Activity and Multimodal Travel Planner: Phase 1 Report

Authors
Kitamura, Ryuichi
Chen, Cynthia

Publication Date
1998-09-01

CALIFORNIA PATH PROGRAM
INSTITUTE OF TRANSPORTATION STUDIES
UNIVERSITY OF CALIFORNIA, BERKELEY

Daily Activity and Multimodal Travel Planner: Phase 1 Report

Ryuichi Kitamura, Cynthia Chen
University of California, Davis

California PATH Working Paper
UCB-ITS-PWP-98-23

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 Transportation, Federal Highway Administration.

Report for MOU **228**

September 1998

ISSN 1055-1417

Daily Activity and Multimodal Travel Planner

Phase I Draft Report

Submitted by

Ryuichi Kitamura
Department of Civil Engineering Systems
Kyoto University
Sakyo-ku, Kyoto 606-01
Japan

ITS, Academic Surge 2028
Department of Civil and Environmental Engineering
University of California
Davis, CA 95616

and

Cynthia Chen
ITS, Academic Surge 2028
Department of Civil and Environmental Engineering
University of California
Davis, CA 95616

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 move, 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 extensively discussed in the ITS literature.

Little attention has been directed, on the other hand, to achieving the same goal by developing efficient travel itineraries. This becomes important when a traveler visits a number of places in a tour. Examples include a delivery truck driver who is supposed to deliver goods to multiple locations, or a tourist who wishes to visit a number of attraction spots. 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 a 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 beliefs that the availability of information affects the decision to use public transit in important manners, 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 led to the conception of the "Travel Planner," a computer software which 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 Travel Planner develops alternative itineraries for the visits interactively with the traveler, or, the user. The planner presents alternative itineraries, 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. This process is iterated until a satisfactory itinerary is found.

The development of the Travel Planner consists of two stages: first it is developed as a public kiosk, and then as a personal unit. The inputs to and outputs from the Planner are identical in both stages. The critical difference is whether the Planner has a 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 user uses it. **As a** personal unit, on the other hand, the Planner serves as a transportation guide for a specific user and it therefore is able to store information about the user's preferences or past history of travel. The Planner would learn and tend to

become smarter 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 modes of travel and walking might be more attractive and efficient for the traveler for visiting a series locations than using any 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 Planner should take multi-modal itineraries into consideration while developing a desirable itinerary for the user.

This is a first-phase report of the Planner project which reports on the development of a prototype Planner that has been developed. The prototype represents a first step of development which is based on a set of assumptions. The primary assumptions are: the Planner is a public kiosk; the private auto is not used; and all destination locations are uniquely specified by the user. This report provides an overview of the algorithms that are in the Planner, documents in detail how the prototype was developed and how it functions, reports on the design and results of initial user test, and points to future research directions.

This report is organized as follows. Section **2** of the report provides an outline of the Travel Planner. Section 3 offers a literature review on the traveling salesman problem and proposes an alternative algorithm to be used in the Planner. Section 4 details the actual development of the Planner prototype, identifies existing problems and points out research directions in the next-stage of Planner development. Section 5 presents the experimental design and its results'. Code developed for the Planner is enclosed in Appendix.

[1] Note that Section 5 is not included in this draft report because the experiment is expected to be completed in December, 1997. This section will be included in the phase-1 final report which will be delivered at the end of December, 1997.

2. Travel Itinerary Planner: Outline of the Planner

The Travel Planner is designed to construct a suitable itinerary to engage 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. Presented in this section is the outline of the Planner. Section 2.1 formally presents the Planner problem. Section 2.2 offers an overall picture of how the Planner works as a whole. Following this is a description of each of the main components: inputs, outputs and the Planner Executor, in Sections 2.3 and 2.4. The Planner Executor collects information on the user's preferences, updates the preference weights, and assists the user in discovering the best compromise route. 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. Detailed descriptions of the algorithms used in Planner Executor are provided in Section 4 of this report.

2.1. Problem Formulation

The Travel Planner is designed to construct an itinerary which 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 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.

The following attributes of a trip are considered by the Planner.

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)
	expected travel time

fare
names of 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. Difficulties exist most probably in the process of identifying user preferences, and in computational requirements associated with itinerary optimization. The Planner therefore aims at building a suitable, rather than optimal, itinerary that is well acceptable by the user, while working interactively with the user in identifying his/her preferences and developing an itinerary. The user interface of the Planner includes components that facilitate the interaction with the user.

Mathematically, the problem can be expressed as follows:

$$\text{Maximize} \quad f(\beta, x) \tag{1}$$

$$\text{Subject to} \quad g_i(x) = 0, \qquad i = 1, 2, \ldots, n \tag{2}$$

where

β = an Mxl vector, $(\beta_1, \beta_2, \ldots, \beta_M)$, where β_i (i =1, …, M) is the coefficient of the i-th attribute of an itinerary,
n = number of constraints, and
x = an Mxl vector, $(x_1, x_2, \ldots, x_M)$, where x_j (j = 1, …, M) is the j-th attribute of an itinerary.

Attributes of our interest include travel time, travel cost, number of transfers, waiting time, and walking time. No specific functional form is indicated by Eq. (1).

The constraints, $g_1(x), \ldots, g_n(x)$, represent timing constraints, duration constraints and sequencing constraints associated with the problem. For example, one of the most basic constraints is: the latest possible arrival time at current location is greater than or equal to

the earliest possible arrival time at the previous location, plus the minimum duration there, plus the travel time from there to the current location.

Values of the elements of x are known to the Planner, but not those of β, which represent the user's preferences. The Planner problem is not a simple 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 β. The difficulty is 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 therefore extracts information from user inputs to continuously update estimates of β, in a trial-and-error process.

2.2. System Components

The Planner consists of three stages: User Inputs, Planner Executor, and Outputs to the User. User Inputs include information from the user on locations to visit and constraints associated with the visits. The input information is then fed to the Planner Executor, which analyzes the input information, evaluates all possible itineraries, and computes preference scores using β. 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 weights, β, based on user's choice between the two itineraries, and evaluates all possible itineraries again using updated β. The process is repeated until the Planner finds an itinerary which is satisfactory to the user. Sections 2.3 to 2.4 describe each component in detail.

2.3. User Inputs and Outputs

As outlined above, the Planner, given the user inputs, evaluates all feasible itineraries and presents the user with two alternative itineraries to choose from. The user in turn tells the Planner which itinerary he/she prefers more, and the Planner updates the preference weights based on the response. Itinerary building evolves in this manner through the interaction between the user and the Planner.

The final output of the Planner consists of the final and accepted itinerary presented in enough detail and specificity such that the user will be able to execute the itinerary without prior knowledge about the area. The contents and presentation format of the output will be presented in Section 4 of this report.

2.3.1. User Inputs

The input from the user defines a specific itinerary development problem. In this section, the input variables are first defined. An outline of user interface to obtain these input variables is presented in Section 4.

The user inputs to the Planner comprise:

- the initial and final location,
- 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.

As noted earlier, there are two classes of locations for visit: *hard locution* and *soft location.*

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.

Constraints associated with the timing of a visit refer to the time window or time windows associated with the visit, and are primarily determined by the nature of the activity pursued during the visit. For example, the activity may have to start by a certain time of day, or may have to be completed by a certain time of day. In general, the starting time and/or the ending time of an activity may be constrained to take place either before a time point, after a time point, or within a period.

- *Activity Locution*

As defined above, the location of a visit is considered as *hard* if it is the only place where the activity can 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.

The location of a visit, either hard or soft, is specified by the user by giving the exact address or cross-roads, or, in case of prominent locations such as landmarks, by the name commonly referred to by the local people. 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 that has been developed considers only 25 geographically-fixed hard locations. This is primarily to reduce coding complexities and computational requirements in the early stages of Planner development. The user selects locations for his/her visits from among the 25 potential locations. Selection is completed by manipulating a scroll bar presented to the user on the screen. The selection indicated by the user will be shown on a map on the screen after user inputs are completed.

- *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 leave the location no later than b_i, the time window of the visit is expressed as (a_i, b_i). Complexity of the timing constraints increases when the user wishes to arrive and leave the *i-th* location during 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 a single time window is considered in the Planner prototype. The user is asked if there is earliest arrival time or latest arrival time associated with each location he/she would like to visit. If the user responds with an "YES", the Planner then solicits earliest and/or latest arrival times from the user.

- *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}), but no more than 2 hours (d_{ui}). 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.

The Planner prototype only considers a single window as duration constraints. In other words, the user is asked to enter a minimum duration and a maximum duration at each location he/she wishes to visit.

- *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) 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 over time. For example, the user may have no access to a 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 itinerary development. Dynamic constraints come into effect depending on 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 internally by the Planner.

The modes incorporated into the Planner prototype include transit lines, cable cars, taxi, and walk. The automobile is not included in the prototype at this point primarily due to two reasons. Firstly, one of the principal objectives of developing the Planner is to provide better information on public transportation and encourage more transit use. A higher priority is therefore placed on public transit than on the private automobile in initial stages of Planner development. Secondly, it is not common in reality for the traveler to use both an automobile and public transportation for a series of trips. A prototype which does not incorporate the automobile is thus considered to be able to cater to transit users sufficiently.

- *Beginning and Ending Points*

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

2.3.2. Outputs to Users

When the Planner generates an itinerary, it is shown on a map. There is also an "attribute button" shown on the screen. When the attribute button is clicked, an attribute window appears and contains information about the itinerary, such as travel time, travel cost, walking time, number of transfers, etc.

Visualization of more than one alternative itinerary on a single map is not possible when itineraries share many routes (e.g., transit lines) in common. This happens when relatively simple street and transit networks are used for the prototype as simple networks offer a rather limited number of alternative routes between any two points. Consequently, although the attributes of two itineraries may not be identical, they may not produce any

discernible difference when presented visually on screen. In this case, the user will have to compare the two itineraries by comparing their attributes in the attribute window rather than visually on the map.

2.4. 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, β , based on the user response. During the interactive decision making process, the Planner "learns" and updates the preference function to best represent the preference structure the user has. This process continues until the Planner discovers an itinerary that is acceptable to the user. The section presents the rationale that underlies two features of the Planner design that have already been discussed: (i) formulation of the problem as a multi-criteria optimization problem, and (ii) adoption of an interactive programming approach. Further discussions on interactive programming can be found in Section 3.6 of this report.

Formulation of Planner Problem as a Multi-Criteria Optimization Problem

There can be many criteria that may be adopted by the user when comparing and evaluating alternative itineraries. For example, the total travel time or monetary cost of travel associated with an itinerary may serve as such a criterion. When only one criterion is used in selecting a desired itinerary, the problem is the often encountered as a traveling salesman problem (TSP) which is formally defined in Section 3. In the course of designing the Planner, however, it was realized that more than one criterion is often used by the traveler. In the case of travel mode, a traveler often chooses travel mode while considering the trade-off between travel time and monetary cost. For example, the 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.

One approach to incorporating multiple criteria into the Planner problem is to develop a generalized cost function in which all relevant criteria are weighted and combined to form a uni-dimensional indicator of the desirability of an itinerary. The problem, however, is that there has been no such cost function developed for itineraries that involve multiple stops. More importantly, it is unrealistic to assume that one generalized cost function is applicable to every traveler, or for every situation for a given traveler. For example, it is well established that the value of time varies from person to person depending on income and other factors, many of which are probably unobserved. The above example of the choice between bus and taxi suggests that the value of time of a given person varies from

occasion to occasion, depending on situational factors, which are again often unobserved. In case of our problem, the nature of the problem is much more complex because preferences toward activity sequencing, timing, and sojourn duration at different locations are introduced into the problem.'

Furthermore, we note that an individual as a decision maker may not even be aware of the criteria he/she uses in the decision making process. In general, the individual is not capable of expressing his preferences in quantitative terms. Yet he/she is often faced with a set of non-dominant alternatives, none of which is superior to the others in every aspect, and is forced to exercise complex trading off among the criteria to find a best compromise solution. Given

(a) the multitude of factors that affect the desirability an alternative solution,
(b) our limited ability in identifying and measuring all factors that affect an individual's decision making,
(c) heterogeneity across individuals in their preferences and decision making procedures, and
(d) variability in preferences and the decision making procedure even for a given individual,

the single criterion approach or the generalized cost approach is not necessarily a realistic option for our problem.

Interactive Programming Approach

We therefore adopt the interactive Multiple Objective Mathematical Programming (MOMP) procedures. 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)

Because these features should be possessed by the Planner, it has been determined that the Planner should be developed as a MOMP procedure. Adopting the MOMP approach, however, does not imply an abandonment of TSP algorithms. In fact, the TSP algorithms are used as part of the Planner's interactive solution procedure, which is outlined below:

[1] Preferences to these attributes are not incorporated in the Planner prototype.

1. Given a set of location to visit, an exhaustive search for an optimum solution is performed by generating the best feasible itinerary for each of all possible sequences based on the preference function.
2. **A** set of solutions is selected from among those generated in the previous step, including the optimum solution, 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 function is 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 function along with alternative solutions (e.g., the fastest itinerary and cheapest 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 4 of this report.

[2] In the prototype so far developed, a second-best solution is presented.

3. Planner Algorithms

In this section, the traveling salesman problem (TSP) is formally presented (Section 3.1), and algorithms for this class of problem are reviewed (Section 3.2). 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 3.3, the review turns to the literature pertinent to the constraining dimensions of the Travel Planner. The conclusion of the review on TSP algorithms follows in Section 3.4; the conclusion addresses the applicability of TSP algorithms to the Planner problem. In Section 3.5, the multiple-objective nature of the Planner problem is discussed. An alternative approach to the TSP algorithms: interactive programming method, is presented in Section 3.6.

3.1. Traveling Salesman Problem

If none of the constraints described in the previous section is present, and if all visits have hard locations, the Travel 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 others represent locations to visit, and $A = \{(v_i, v_j): i \neq j, v_i, v_j \in V\}$ is an arc set. Each arc, (v_i, v_j), is associated with 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 $c_{ij} \neq c_{ji}$, the problem is asymmetrical. The review contained in this report only considers the asymmetrical problem; there is simply no symmetrical transportation network in real world.

The discussion that follows provides a brief review of recent algorithmic developments for the TSP. This section, 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.

3.2. 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.

3.2.1. Exact Algorithms

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

(TSP1)

$$\text{Minimize} \quad \sum_{i \neq j} c_{ij} x_{ij}$$

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

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

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

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

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.

Constraints (1) and (2) guarantee that every vertex is entered once and left once. The solution to TSP1 should be a single tour covering all vertices in the network. 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 ($S \cup S' = V$). 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).

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. In addition, a great number of algorithms have been developed. One of the notable developments 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 the 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 λ 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 h, the optimal solution to the dual problem are equivalent to that to the primal problem where no λ 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 λ 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. Exact algorithms also require longer computing time compared to approximate algorithms which are described below.

3.2.2. 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 optimality. Interested readers are referred to Golden et al. (1985), Gendreau et al. (1992), and Johnson (1990) for these procedures.

3.3. TSP with Constraints

The formulation defined in TSPl above stands as a theoretical problem in the sense that there is no real-world problem as simple as TSPl. 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 most relevant to approximate algorithms.

3.3.1. Temporal Constraints

Temporal constraints here refer to the timing of a visit. As noted earlier, there are two types of constraints regarding the timing of a visit: 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 area and are 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 only be made 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. Thus, the objective of the problem is the minimization of not only the total travel time, but also the waiting time.

Both exact algorithms 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 the 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). In plain words, the procedure consists of two steps: (1) construction of a feasible itinerary from the origin to the destination, covering all vertices in the network, and (2) apply the post-optimization procedure. The post-optimization procedure starts with removing the vertex immediately after the origin 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 origin and destination 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. Although no TSP studies have, to the authors' knowledge, directly researched this type of constraint, the algorithms developed for TSP with single time window can probably be modified and applied to this problem.

3.3.2. Duration Constraints

As discussed in Section 2.3.1, the user may place constraints on the duration of the visit to 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 to 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 check against duration constraints in exact algorithms as long as the required computing time is not excessive.

3.3.3. Order 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. 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_o contains a single vertex for the origin. 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 order constraints in the Planner problem are not as restrictive as those in the clustered TSP. Consequently, the set for alternative itineraries is larger for the Planner problem than for the 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 order requirements inherent in the set of visits. Clustered TSP algorithms would therefore require some revision to address order requirements for the Planner problem.

3.4. Applicability of Existing TSP Algorithms

All TSP algorithms, exact or approximate, are designed to achieve a sole objective: given a set of locations to visit in a network and a travel time/cost matrix associated with the network, find the optimal route with a minimum travel time or cost. **A** unique optimal solution usually exists when only one objective is considered. However, this is not exactly the case in the Travel Planner problem, where there exist more than one criterion which are often conflicting with each other.

Even if there were one single objective to be achieved in the Planner, 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 the duration of a visit) that are considered by the Planner, but that have not yet been extensively studied in TSP. To tackle the type of problem addressed by the Planner, alternative methodologies other than conventional TSP algorithms must be developed that can provide the user with a solution that is feasible and as close to the optimal solution. The methods adopted by the Planner is summarized in Section 3.6 of this report.

3.5. Multiple-Objective Programming Problem

Multiple-objective programming problems are the ones in which there are more than one criterion used to evaluate the desirability of each alternative solution. If a firm pursues just to maximize its profit, then it is a single-objective problem; if a firm pursues to maximize its profit and to provide the best possible working environment for its employees, then it is a multiple-objective problem. As noted earlier, the traveler may consider travel time, cost, number of transfers, etc. when choosing from among a set of itineraries. Therefore, the problem of this project, i.e., selecting the most desirable itinerary for visiting a set of activity opportunities under a certain set of constraints, is a multiple-objective problem.

For problems involving multiple objectives which are often conflicting with each other, a dominant solution' often 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 non-dominant 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.

3.6. Interactive Programming Method

Three issues are discussed in this section. Firstly, what programming approach should the Planner adopt?; in other words, in what way should the Planner build the itinerary for the user? Secondly, what method is to be used by the Planner to select an optimal one from a set of solutions? Lastly, what method is to be used by the Planner to update the weights that represent the user's preferences to respective attributes of the itinerary?

3.6.1. Interactive Programming Approach

The interactive programming approach adopted by the Planner takes on the following form: the Planner presents alternative itineraries to the user, who in turn indicates which itinerary is most desirable; the Planner updates the user's preference function based on the indicated choice, and generates another set of alternative itineraries. Thus the preference function, or say the objective function, is interactively adjusted 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, the Planner constructs an itinerary incrementally by specifying a trip connecting two locations to visit, one by one with user input. In the itinerary improvement approach, the Planner generates an initial feasible itinerary and improves it 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. Probably more importantly, the final itinerary connecting all

[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 objectives under consideration.

locations to visit could very well be optimal in its parts, but not at all optimal as a whole. In particular, the optimal itinerary would be precluded in this approach in the event where segments selected in earlier stages of itinerary development are not identical with those in the optimal itinerary.

The itinerary improvement approach, on the other hand, prevents those locally but not globally optimal itineraries to be selected, 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 can be 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.

With the consideration above, the approach adopted by the Planner is itinerary improvement approach. In other words, the Planner selects the entire itinerary and improves it as a whole.

3.6.2. Selecting the Best Itinerary

In making a choice from a set of alternatives, different users may adopt different strategies to eliminate inferior ones, compare promising ones, and select the most satisfactory itinerary. Evaluation methods directly relate to how the Planner updates user's preference function. Alternative approaches to selecting a solution are first discussed in this section, followed by a description of the approach adopted by the Planner.

Preference function

It is assumed that in evaluating alternative itineraries, the user uses an implicit preference function whose variables are various attributes of the itinerary, including travel time,

travel cost, walking time, number of transfers, etc. The itinerary the user is most satisfied with has the best combination of attribute values and the largest value of the preference function. The preference function may be expressed as:

$$y = x\beta,$$

where

$x = \{x_1, x_2, \ldots, x_n\}$, representing arguments of the preference function, and $\beta' = \{\beta_1, \beta_2, \ldots, \beta_n\}'$, representing the weights of the respective arguments.

The tasks to be performed by the Planner with respect to the preference function are two-fold: it must first estimate values of β efficiently, then find an itinerary, or a set of itineraries, which offer acceptable values of y using the updated values of β .

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 upon the final score. Procedure of calculating concordance and discordance measures can be found in Nijkamp and van Delft (1977) and 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 when the value of a certain criterion falls below a trade-off range, it cannot be compensated for by superior values of other variables. Thus, 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.

*The Approach Adopted in **the** Study*

The preference function approach is adopted in this effort to develop a Planner prototype. The primary reason is its practicality, not so much with the evaluation of the desirability of each alternative itinerary, but more with the measurement of the user's preferences.

3.6.3. Updating the Weights

The use of a preference function requires the Planner to first estimate the weights and then select the one with the highest value. 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 "contingency 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 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 exercise for the user to obtain a set of updated weights.

A simple and yet efficient scheme to update users' preferences is devised and adopted in this study. Using initial weights established from previous mode choice studies in the Bay Area, the Planner first selects two alternatives with the highest and second-highest values. Then, the Planner asks the user to select the one they would prefer. The Planner compares value of each attribute between these two alternatives. The Planner doubles the coefficient if its attribute value of the preferred route is less than that for the other route and halves the coefficient if its attribute value of the preferred route is greater than that for the other route.

This procedure is repeated with a new set of itineraries selected with a new set of weights. This approach is adopted because (i) it does not require extensive inputs from the user to establish a preference function, and (ii) the preference function is refined interactively when the user continue 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. Further discussions on preference updating can be found in Section 4.3.

4. Planner Prototype Development

This section is devoted to the documentation of the development of the Planner prototype. Assumptions adopted in the development are laid out in Section 4.1. **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 which updates user's preference function. User inputs and outputs are described in Section 4.2 while Planner Executor is presented in Section 4.3. In Section 4.4, problems are identified in the prototype development and research directions are pointed out in the next stage of the development. Documentation of the coding for both the transit network and the Planner Executor is presented in Sections 4.5 and 4.6.

4.1. Assumptions

As the effort of this project represents the initial development of the Planner prototype, a number of assumptions are introduced to create a working prototype for future development. These assumptions concern mainly with the complexity of the transit network in the prototype and the flexibility of use provided to the user.

The transit network used in the prototype is a simplified version of the network in downtown San Francisco. The network consists of five MUNI lines, one cable-car line, plus streets for taxi and walking which provide access to various attraction points. **A** simple transit network has allowed us to produce a prototype in a timely manner. The prototype yet provides a means for us to probe into some general research questions. For example, suppose there exists such an itinerary planner in the market. How much will people be willing to pay to use it? What is the nature of the market for such a device? What features of the Planner do users value? Will it in fact encourage transit use? The prototype developed here will be refined and used to address some of these questions.

User's flexibility is rather restricted in the Planner prototype developed thus far. Potential locations for visit are restricted to a set of 25 specific locations. The restriction was imposed mainly due to concerns with computational time and programming requirements. Allowing the user to choose visit locations without restriction on a map would mean that the Planner will first have to identify attributes of trips (e.g., travel time and travel cost) made to and from each location to other visit locations. This would require extensive multi-modal minimum path search before conducting any optimization. With the imposed restriction, network data can be pre-possessed and computational requirements can be thus reduced. It is anticipated that that this restriction can be removed in the next stage of development with efficient algorithms and computer code.

Another assumption is that all visit locations are hard locations. This assumption has been introduced to simplify coding and to reduce data requirements. Travel modes are restricted in the prototype to exclude the automobile. The modes included are: bus (MUNI), cable car, taxi and walk. Including the private automobile in multi-modal

contexts implies an additional constraint that the automobile parked must be fetched some time later. The choice of parking location is another dimension that needs to be incorporated while considering the trade-off between parking cost and walking distance. Due to added complexity in algorithm development and coding, it was decided not to include the automobile in this initial prototype.

4.2. User-Interface

This section consists of two parts: input variables collected from the user by the Planner, and output variables presented to the user by the Planner. User interface for preference updating will be reported separately.

4.2.1. User Inputs

The Planner prompts the user to input the set of variables described in Section 4 of this report. These variables are essential for the search of desirable itineraries. It is proposed that the variables are inputted with the following sequence:

Beginning location
Attributes of a visit (repeated as necessary)
    Location
    Timing constraints
    Duration constraints
Sequencing constraints
Ending location

In the rest of this section, the input format for geographic location is first discussed. Following this is the discussion of each item in the above list.

- *Geographical Location*

As noted earlier, a location can be specified by the user by giving:

exact street address,
cross-roads, or
landmark name.

As noted earlier, only hard locations are included in this phase of the study. The exact street address is most precise, but the user may not always know it. Specifying a location in terms of cross-roads (i.e., nearest intersection) is less precise, but in many cases would guarantee adequate levels of geographical proximity. It is anticipated that users can tell

the nearest intersection in terms of the cross-roads even when they cannot tell the precise street address.

For certain locations such as landmarks, their names will uniquely specify their exact locations (e.g., "Coit Tower" in San Francisco). In the gray area are such expressions as "Safeway on Covell Street." Whether the Planner can determine the location based on such expressions depends on the quality of its database. In addition, there is a potential problem that such an expression does not uniquely identify a location because there may be more than one "Safeway on Covell Street." The Planner should be soliciting more information from the user when the user input does not uniquely identify a location.

A location may be input by typing its address, cross-roads or name through the keyboard, or pointing the location on the map displayed on screen. The latter option is feasible 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 easiest and fastest. In case a street name or landmark name is input through the keyboard, the user interface is adopted in which possible alternative names are displayed in full as the user types first letters of the location name. This scheme has been adopted in some in-vehicle navigation systems or spell checkers in word processing software packages.

In the prototype of this study, only a limited number of potential visit locations are considered. In the prototype, therefore, the user is asked to select a list of locations to visit using a scroll bar.

In the rest of this section, an outline of the user interface for the input variables is presented.

Phase 1 Prototype Implementation:

- *Beginning Location*

The Planner presents the user a window which lists all 25 potential locations to visit. These 25 locations scatter around downtown San Francisco and its adjoining areas. These 25 locations are shown on the map as well. The user is instructed to select a location with a scroll bar.

PLANNER> "Please select one as your origin."

Beginning location also concerns timing constraint. More specifically the Planner also asks the user to identify the earliest and latest possible departure times if any. Timing constraints are described as follows.

- *Timing Constraints*

PLANNER> "Do you need to leave this location before a certain time?"

PLANNER> (if "Yes") "Select the hour before which you must leave this location"

PLANNER> "Select AM or PM"

PLANNER> "Select the minutes before which you must leave this location"

PLANNER> "Do you need leave this location after a certain time?"

PLANNER> (if "Yes") "Select the hour after which you must leave this location"

PLANNER> "Select AM or PM"

PLANNER> Select the minutes after which you must leave this location

- *Attribute of a Visit: Location*

PLANNER> "Please select one of the following: (1) select another location to visit, (2) select the terminal point."

The Planner presents the user with the window containing location names. Based on the answer to the above question, the user is asked to select another location to visit or a destination as the terminal point of the itinerary.

- *Duration Constraints*

PLANNER> "At least how many minutes do you plan to spend at this location?"

PLANNER> " At most how many minutes do you plan to spend at this location?"

- *Sequencing Constraints*

PLANNER> "Do you have to visit this location before visiting another location?"

PLANNER> (If "Yes") Must visit *the location the user just selected* before visiting:

The window with destination locations is then presented to the user; the user shall select one location from the window as the one he/she must visit after visiting the current location. The list of the other locations for potential visit must be appropriately defined

based on the past user input. For example, if the user has indicated that Location B must be visited *before* Location **A,** then, Location A must be eliminated from the list of other locations, when locations that must be visited *after* Location B are asked.

• *Ending location*

This segment is reached when the user indicate to "select the terminal point" earlier in response to the first prompt for location selection.

PLANNER> "Please select one as your destination point."

The Planner presents the user with window in which all *25* locations are identified. The user is asked to select one from the list as his/her destination point.

Timing constraints may be associated with the ending location as well. The Planner therefore asks the user to indicate the earliest and latest arrival times at the ending location, using the format for timing constraints described above.

4.2.2. Outputs to the User

Outputs to the user are essentially two maps. As noted earlier, the Planner identifies two routes and displays them on the two maps. In each map, a route is identified, linking all locations to visit. **A** switch button is provided on screen such that the user can switch between the two maps and compare the two routes selected by the Planner.

The two routes are highlighted in red and yellow and are labeled. If the route is made by transit, it is labeled with route names such as "MUNI 42"; if the route is made by taxi alone or walk alone, the route is highlighted on a map on which major streets are labeled. The Planner prototype does not supply detailed written directions for the user as the output; it, however, lists the sequence of visits from origin to destination on the side of the map.

In the current prototype the stop name where the user can transfer from one transit line to another is not displayed on screen. This capability, however, is being developed and will surely be available in the next stage of the development.

Figures 4.1 and Figure 4.2 show sample routes produced by the Planner. In this case, the user chose six locations to visit, including the origin and destination. The Planner's first choice is shown in Figure **4.1;** the tour is made by walk and the total walk time is 203 minutes. The second choice is shown in Figure 4.2; the tour is made by a combination of transit and walk. The attributes of this itinerary can be accessed by clicking the button with a label of "**A**". The attributes of this itinerary are as follows: travel cost of *$5,* transit

travel time of 11 minutes, total wait time of 5 minutes, a total **walk** time of 119 minutes and two transfers. The order of the visits is shown at the side in legend.

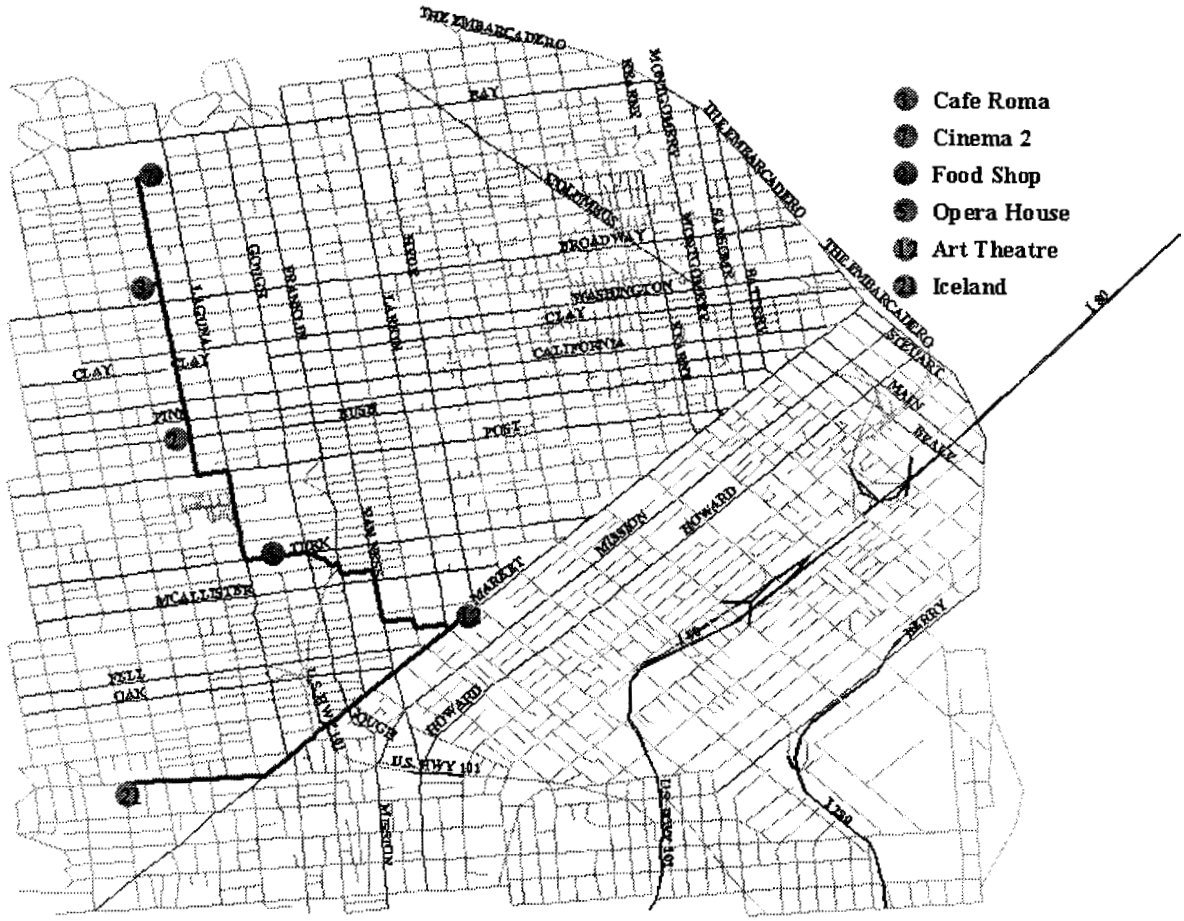

Figure 4.1: **A** Sample Itinerary Selected by the Planner

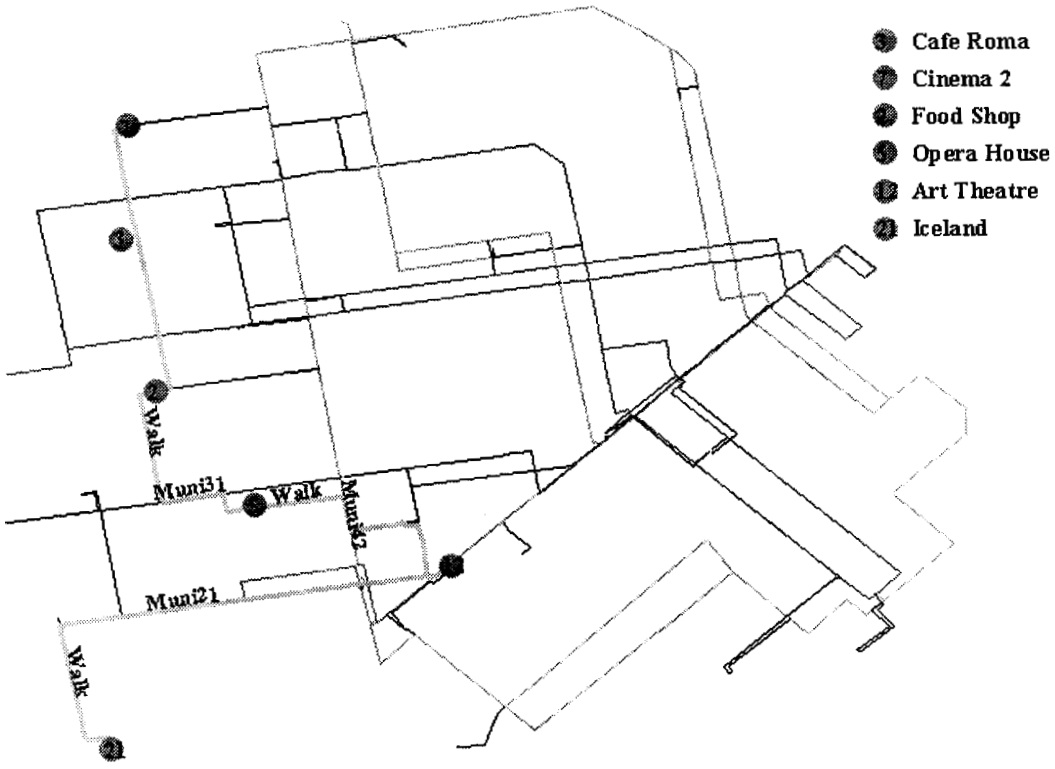

Figure **4.2:** An Alternative Sample Itinerary Selected by the Planner

4.3. Planner Executor

Once the user inputs are completed, the Planner evaluates each possible sequence and identifies feasible ones. Based on the preference function, the Planner then evaluates each feasible sequence and selects the two with highest and second-highest values. Feasibility check consists of two tasks: timing and duration constraints as well as sequence constraints. Timing and duration constraints imply that for any two consecutive locations in a sequence, the latest arrival time at the current location must be greater than or equal to the sum of the earliest departure time at previous location and the travel time from previous location to the current location and the minimum duration at the previous location. Sequence constraints imply that a feasible sequence must satisfy the order specified by the user.

Feasibility check results in a set in which every sequence is feasible. Evaluation of each feasible sequence against the preference function is the next step. Since three modes (transit combined with walk, taxi alone, and walk alone) are considered in the Planner

prototype, the Planner evaluates $3 \times n!$ sequences, where n is the number of locations to visit excluding the origin and destination. Seven variables are considered in the value function including travel cost, transit travel time, waiting time, walk time, number of transfers, cable-car travel time, and taxi travel time. For the mode of transit combined with walk, all seven variables may have positive values except taxi travel time; for the mode of taxi alone, only travel cost and taxi travel time have positive values; for the mode of walk alone, only walk time has a positive value.

The Planner starts its evaluation of alternative itineraries with initial weights for the attributes in the preference function, and selects those two itineraries with the highest and second-highest values. The initial weighs are established in previous travel mode choice studies in the San Francisco Bay Area. If the user is not satisfied with either itinerary, the Planner updates the weights to better represent the user's preferences. Obviously some algorithm must be devised for updating preference weights. It is considered efficient if the devised algorithms can converge to the true values quickly; it is considered reliable if the devised algorithms can converge at the same vector starting from different initial values.

As briefly mentioned in Section 3 of this report, the basic idea of the preference updating procedure is as follows: 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. How much the weight should be increased or decreased, however, remains to be determined through a trial-and-error process.

Suppose that the user is not satisfied with the itineraries presented, say, the ones shown in Figures 4.1 and 4.2. The user is then asked to choose a more preferable one between the two. Suppose the user has chosen the one in Figure 4.2. The Planner then updates the weights and presents another two itineraries to the user. These two updated itineraries are shown in Figures 4.3 and **4.4.** In this example, weights are either doubled or halved depending on the comparison of the two itineraries. Both updated itineraries involve taxi as the only travel mode. Itinerary 1 (in Figure 4.3) has a cost of $26.01 and a travel time of 13 minutes while itinerary **2** has a cost of *$3* 1 and a travel time of 14 minutes.

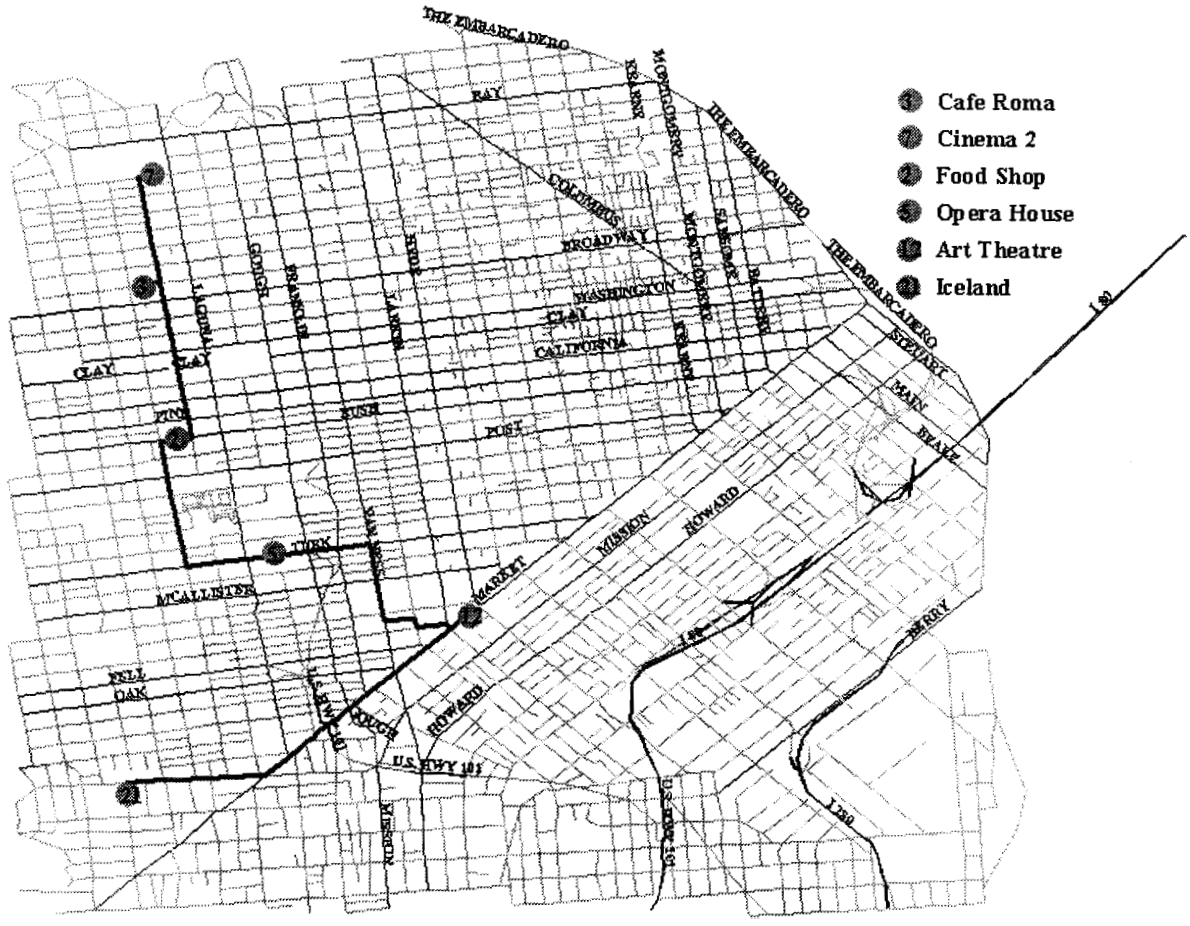

Figure **4.3:** An Updated Itinerary Selected by the User

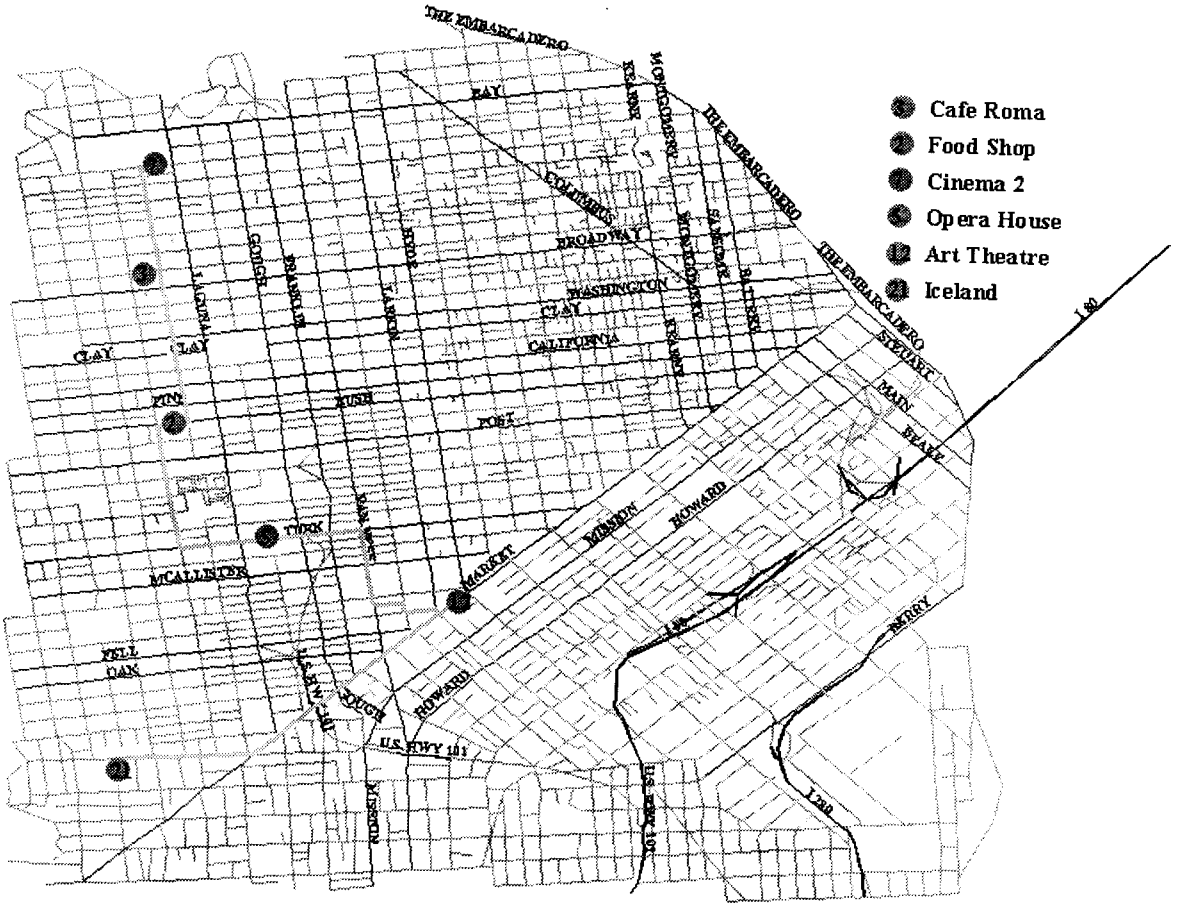

Figure 4.4: An Alternative Updated Itinerary Selected by the User

4.4. Next-Stage Development

Several problems appear to prevail throughout the updating process. These problems mostly relate to the issue of sensitivity. For example, in some instances especially when the number of visits is relatively small (consequently the set of alternative itineraries is small), one or two itineraries in particular sequences appear to dominate. In addition, when a set of initial weights is used, one mode appears to dominate the others throughout the updating process. The dominance of either a particular sequence or a particular mode over the rest leads to the result that optimal itineraries selected by the Planner is sometimes not sensitive to user inputs.

There are a few conceivable reasons for the apparent dominance of certain sequences or modes despite user inputs. Firstly, there may be indeed some sequences or modes that are dominant. When dominant sequences or modes do exist, the iterative nature of the updating process adopted by the Planner does not function. Secondly, initial weights may be a source of the problem. During the experiment that tested different initial weights, we

found that different weights result in different solutions obtained by the Planner algorithm for updating the preference weights.

Several approaches can be taken to remedy the problem in the next stage of Planner development. In terms of dominating sequences or modes, the Planner should identify a dominant choice if it exists, inform the user of the choice, and terminate the search procedure. Additionally, sensitivity analysis should also be conducted to identify lower and upper bounds of each preference coefficient within which the utility ranking will not change, given that other preference coefficients are fixed. The results can be conveyed to the user by the Planner to suggest that some preference coefficients should be updated dramatically. In terms of initial weights, some conjoint measurement may be performed such that an appropriate set of initial weights can be identified and used for a particular user.

In addition to the remedies mentioned above, we will also propose a more complex network than the current one to be used in the next stage development. Users will not be restricted to choose locations to visit from a fixed set of 25 hard locations. Instead, users are free to choose any locations by either entering the address or landmarks or clicking on the map.

4.5. Documentation - Map Display

The Travel Planner considers three modes in its trip planning: taxi alone, walk alone, and transit combined with walk, To calculate the best compromise route for the user, the Planner must utilize a network on which the calculation can be based. The modes of taxi alone and walk alone use a street map as its network while transit uses a transit network. It is therefore necessary that at least three themes must be prepared to facilitate the operation in the Planner. These three themes include: 1) a theme of attraction points containing 25 locations for users to visit, 2) a theme of street map, and 3) a theme of transit network. The documentation as follows describes each theme in turn.

A Theme of Attraction Points

The very first task of coding a theme of attraction points was to select 25 locations in the map of downtown San Francisco. The number of locations was limited to 25 in order to simplify the preliminary development. In the process of selection, it is desired that these twenty-five locations cover the entire downtown area without being too close to each other to confuse the user. In addition, these 25 locations were chosen without regard to actual transit routes in order to make sure that there would be both locations that would be relatively difficult for a traveler to plan a route to and ones that would be relatively easy. Once the 25 locations were chosen, each of them was labeled with a fictional name to signify what type of location they were. These included such places as shopping malls,

restaurants, and movie theaters. Along with their labels, these locations are highlighted on the display by a blue circle with a diameter larger than street or transit route widths.

A Theme of Street Map

For our base layer, we used a pre-coded map of downtown San Francisco, which came with the Arcview'. This map contains every street in the downtown and its adjoining areas. Each street segment (i.e., between every intersection) is coded with attribute data including the length of the segment, the name of the street, and the direction(s) traffic is allowed to travel on that segment, The streets are categorized as being "highway", "major street", or "minor street'' and can be differentiated as such on the map.

A Theme of Transit Network

A network covering both transit lines and walk paths is then created and overlays the street map theme. The first task is to select appropriate transit lines to be included in the Planner. Various transit lines in the San Francisco area follow identical routes in the downtown sector, even though they may deviate once they leave the downtown area. In such cases, only one route was chosen to avoid redundancy. Some of the other routes included in the network just traverse through or loop around the downtown area and never leave the Planner area. There are five MUNI bus lines plus one cable car line included in the Planner. BART subway trains are not included since no one is likely to use BART and still remain in the downtown area.

Once appropriate transit lines are selected, they need to be geo-coded into the Planner. The coding went beyond just laying down a series of crisscrossing transit lines. The difficulty lies in how to represent transfer cost in the map. When a transfer (from one bus to another) occurs, there is a cost in both time and perceived hassle to the traveler. To make sure that the user has to spend an average amount of time waiting for a bus to come, transfer segments are created throughout the network.

Transit lines were laid down using a mouse cursor in ArcView on the transit "theme." Transit line segments closely followed the streets on the street map theme. These transit line segments connected 'bus stops' to one another. The information attached to each theme's attribute table included the name of the transit line, what its two endpoints were, and whether it was one-way or both ways. Travel time on the segment was determined by the length of the segment in miles divided by the average speed for that particular transit line. The average speed was calculated using the official transit schedules.

The bus stops are stops where a traveler might transfer between transit lines or where he or she might get off or on to walk to or from one of the 25 locations. Adjoining segments

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

of the same transit line are "snapped" together with a computer tool so that computer will view the transit line as one continuous line with different points at which a traveler may leave the line.

The point where two transit lines cross is not viewed by the computer as a bus stop since the transit lines are set up as an "overpass." It is set up as "overpass" such that the Planner will not be able to select a route that changes transit lines at this point; this corresponds to the idea that the user must get off at a bus stop to transfer. Instead, very near to an intersection of two transit lines, two transfer lines are put in and they are snapped to both lines and are programmed to not create an overpass. These transfer lines are inserted so close to the actual intersection that a single bus stop is considered to be at the intersection; the computer however has to route the traveler along a transfer line in order to change transit lines. The transfer lines are one-way and are programmed with a waiting time of half the average daytime headway time of the transit route to which they are going.

For instance, assume that the mean headway time of Route **A** is 12 minutes and that of Route B is 15 minutes. The transfer line of Route **A** to Route B (which may, scaled on the map, correspond to only a matter of centimeters) would have a travel time of 7.5 minutes, which is called the waiting time. Transferring the other way would have a waiting time of ***6*** minutes. **A** traveler going along Route **A** would near the physical intersection of Routes **A** and B. When this traveler reaches the point where the transfer line exits Route **A** he or she is diverted to that transfer segment which has a waiting time of 7.5 minutes. The actual user of the planner would be given instructions to get off at this bus stop and then wait a projected 7.5 minutes for the Route B bus to arrive. If going the other way the computer would send the traveler along the other one-way transfer segment.

Creating these transfer lines divides the original transit lines into smaller segments. With several lines all meeting at a certain intersection, transfer lines might mean that a single block of a transit line is made up of a dozen individual line segments. This poses no difficulty for the computer however, which treats it all as one line segment with different entrance/exit points.

The walk lines are coded to connect the 25 locations with the transit lines. They are two-way lines and go from each of the 25 points to the nearest access point of the closest transit lines. The total travel time for **a** walk segment is determined by dividing the length of that segment by its speed (1.5 miles per hour in our study) and is called the walking time. If more than one transit line is close enough to a point for walking to be at all feasible, then there will be multiple walking routes to that point which will are snapped together at their terminuses. This allows a traveler to walk from one transit line to a location and then walk back to a different transit line if it improves their overall route.

4.6. Documentation - Planner Executor

Written in C language, the Planner Executor performs the following tasks:

1) Conduct exhaustive search of all possible sequences given a set of locations to visit,
2) Conduct feasibility check of all possible sequences constraints supplied by the user, and
3) Apply initial weights for the essential arguments (e.g., travel time and travel cost) in the utility function and select two routes with highest and second-highest utility, and iv) update weights in the utility function and re-select optimal routes.

The Planner Executor includes two main programs: COD1.CPP and COD2.CPP as well two subroutines OPT.CPP and ALLSEQ.CPP. These programs are documented below. Codes of these four programs are provided in Appendix.

Name: **COD1.CPP**

Tasks:

- Receive the input data from ARCVIEW component,
- Re-organize data structure
- Conduct exhaustive search of all possible activity location sequences, and
- Conduct feasibility check of possible using the constraints, optimization.

Input files: COST.DAT: the transportation network attribute data. Variables include: origin, destination, meter, miles, cost($), transit travel time(minutes), wait time(minutes), walk time(minutes), number of transfers, and cable car travel time(minutes).
INPUT.DAT: user's input file. Variables include: address, earliest arrival time(hr), latest arrival time(hr), minimum duration(min.), maximum duration(min.).
SEQ.DAT: user's input file. Variables include: current location, the location before which current location must be visited.
TOTAL.DAT: the total number of the activity locations without including the origin and the destination locations.

Output file: OPT_OUT.TEMP000: the best and the second-best routes devised by the Planner. Output information include: location sequence, cost($), transit travel time(minutes), wait time(minutes), walk time(minutes), number of transfers, cable car travel time(minutes).

F_SEQ.TEMP000: the temporary file including all feasible sequences and the total number of the feasible sequences.

Name: **COD2.CPP**

Tasks:

- Update weights for the argument in the utility function according to user's preference and
- Select two routes with highest and second-highest utility.

Input files: COST.DAT: the transportation network attribute data. Variables include: origin, destination, meter, miles, cost($), transit travel time(minutes), wait time(minutes), walk time(minutes), number of transfers, cable car travel time(minutes).
F_SEQ.TEMP000: the temporary file obtained from COD1.CPP including all feasible sequences.
USERPREFER.TEMP000: user's input indicating which route he/she prefers. This input is used by the Planner to update the weights of various arguments in the utility function.
- OPT–ROUTE–1ST_2ND.TEMP000: the temporary file including the previous optimal and secondary optimal routes.

Output file: OPT_OUT.TEMP000: the best and second-best routes devised by the Planner. Variables include: location sequence cost($), transit travel time(minutes), wait time(minutes), walk time(minutes), number of transfers, cable car travel time(minutes).

Name: **OPT.CPP (subroutine)**

Tasks:

- Compute the two routes with highest and second-highest values of the value function among all feasible sequences.

Name: **ALLSEQ.CPP (subroutine)**

Tasks:

- Calculate all possible sequences

5. Conclusion

The Planner prototype developed so far is one of the first itinerary planners that incorporate multiple destinations, multiple objectives, timing and sequencing constraints, and multiple modes of travel. It is also one of the first itinerary planners that bring the user into the decision making process and employ a user preference function.

In the study the concept of the Itinerary Planner was developed, appropriate algorithms identified and implemented, and user interface developed. In particular, an interactive programming approach is adopted to measure the user's preferences in the course of itinerary development such that the function used to evaluate each itinerary will best represent the user's preferences.

Despite the novel features, the developed Planner is still in its prototype stage and unavoidably exhibits many limitations. The ideas and algorithms that have been implemented in the Planner prototype need to be tested and refined. The user-interface needs to be improved such that the user is free to indicate any locations on the map; the transit network needs to be made more complex and closer to reality; and the algorithms used to update the function that represents the user's preferences need to be further tested and refined. Despite the limitations that are typical of an information system that is based on a new concept, the project has points to the potential the Planner has in making travelers' movement more efficient and encouraging transit use by showing travelers how public transit can be used in complex tours.

References:

1. Gendreau, Hertz, and Laporte "New Insertion and Post-Optimization Procedures for the Traveling Salesman Problem". Operations Research 40, 1992.

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

3. 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.

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

5. Laporte, Gilbert "An Overview of Exact and Approximate Algorithms". European Journal of Operational Research 59, 1992.

6. 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.

7. Lawler, E. L. et. al. The Traveling Salesman Problem: **A** Guided Tour of Combinatorial Optimization. Wiley, 1985.

8. Miller and Pekny "Exact Solution of Large Asymmetric Traveling Salesman Problems" Science 251, 1991.

9. Potvin, Jean-Yves "The Traveling Salesman Problem: A Neural Network Perspective" Universite de Montreal, Centre de recherche sur les transports, 1995.

10. Solomon and Desrosiers "Time Window Constrained Routing and Scheduling Problems" Transportation Science, Vol. 22, No. 1, 1988.

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

APPENDIX

***Code** - cod1.cpp*

```
/*      Filename:       cod1.c
        Author: Jiayu Chen
        Date Created    10-05-97
        Date Updated    11-15-97

        Purpose:
        Input:
        output:

        Variables:
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <sys/types.h>
#include <math.h>
#define ROWS  41000
#define COLS  8
#include "allseq.cpp"
#include "opt.cpp"

float   cost[25][25][11], u[ROWS];
unsigned char p[ROWS][COLS], transit_fseq[ROWS][COLS+2], taxi_fseq[ROWS][COLS+2],
walk_fseq[ROWS][COLS+2];
int             number, transit–n–seq, taxi–n–seq, walk–n–seq;

void main()
{
        unsigned char   s[COLS+2], input[50][5];
        char    line[255];
        FILE    *fp1, *fp2, *fp3, *fp4, *fptemp, "fptransitfseq, *fptaxifseq, *fpwalkfseq, *fpweight;
        int             seq[100][2];
        int             i, j, k, kk, m, row, flagl, flag2;
```

```
        int             node1, node2;
        int             j1, j2, j3, j4, j5, j6;
        int             route–lst, route_2nd, first–route–mode, second–route–mode;
        float   a[7], total–cost;
        float   f1, f2, f3, f4, f5, f6, f7, f8, f9, f10, f11;

/* input operation ...*/
        printf("open files ...\n");
        fp1 = fopen("input.dat", "rt");
        fp2 = fopen("cost.dat", "rt");
        fp3 = fopen("seq.dat", "rt");
        fp4 = fopen("total.dat", "rt");
        fpweight = fopen("weight.dat", "wt");
        fptemp = fopen("opt_out.temp000", "wt");
        fptransitfseq = fopen("tansitf_seq.temp000", "wt");
        fptaxifseq = fopen("taxif_seq.temp000", "wt");
        fpwalkfseq = fopen("walkf_seq.temp000", "wt");

        printf("Reading ...\n");
        row = 0;
        while (fgets(line, 255, fp1) != NULL) {
                if (line[0] != '#') {
                        sscanf(line, "%d,%d,%d,%d,%d", &j1,&j2,&j3,&j4,&j5);
                        input[row][0] = j 1;
                        input[row][1] = j2;
                        input[row][2] = j3;
                        input[row][3] = j4;
                        input[row][4] = j5;
                        row++;
                }
        }

        while (fgets(line, 255, fp2) != NULL) {
                if (line[0] != '#') (
                        sscanf(line, "%d,%d,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f",
&i,&j,&f1,&f2,&f3,&f4,&f5,&f6,&f7,&f8,&f9,&f10,&f11);
                        cost[i][j][0] = f1;
```

```
                    cost[i][j][1] = f2;
                    cost[i][j][2] = f3;
                    cost[i][j][3] = f4;
                    cost[i][j][4] = f5;
                    cost[i][j][5] = f6;
                    cost[i][j][6] = f7;
                    cost[i][j][7] = f8;
                    cost[i][j][8] = f9;
                    cost[i][j][9] = f10;
                    cost[i][j][10] = f11;
                }
            }

            while (fgets(line, 255, fp4) != NULL) {
                if (line[0] != '#') sscanf(line, "%d\n", &number);
            }

/* seq[0]{0} = origin, seq[number+1][0] = destination */
            row = 0;
            while (fgets(line, 255, fp3) != NULL) {
                if (line[0] != '#') {
                    sscanf(line, "%d,%d", &j1, &j2);
                    seq[row][0] = j1;
                    seq[row][1] = j2;
                    row++;
                }
            }

            printf("reading done\n\n");

/* generate all possible sequences of n-locations */
            allseq(number);
            printf("seq done\n\n");

/* feasibility check: check sequence and time constraints */
            m = 1;
            for (j = 1;j < (number+1); j++) m *= j;
```

```
        transit–n–seq = 0;
        taxi–n–seq = 0;
        walk–n–seq = 0;

/* feasibility check */
        for (i = 0; i < m; i++) (
        /* check if seq p[i] is satisfying for seq.dat -- sequence constraint*/
                s[0] = input[0][0];
                for (j = 1; j < (number+1); j++) s[j] = input[p[i][j-1]][0];
                s[number+1] = input[number+1][0];
                flagl = 1;
                for (j1 = 0; j1 < (number+1); j1++) {
                        if (seq[j1][1] < 50) {
                                j5 = 0;
                                j6 = 0;
                                for (j2 = 0; j2 < (number+1); j2++) {
                                        if (s[j2] == seq[j1][0]) j5 = j2;
                                        if (s[j2] == seq[j1][1]) j6 = j2;
                                }
                                if(j5 > j6) {
                                        flagl = 0;
                                        break;
                                }
                        }
                }

        /*  check the time constraint of transit-travel mode */
                if (flagl == 1) {
                        flag2 = 1;

                        for (j = 0; j < (number+1); j++) {
                                for (k = 0; k < number+1; k++) {
                                        if (input[k][0] == s[j]) (
                                                f1 = input[k][1];                    /*
earliest arrival time */
                                                f2 = (float) (input[k][3]/60.0);/* min. duration */
                                                break;
```

```
                    }
                }

                f3 = cost[s[j]][s[j+1]][3];  /* transit travel time */
                f4 = cost[s[j]][s[j+1]][7];  /* cable travel time */
                f5 = cost[s[j]][s[j+1]][4];  /* waiting time */
                f6 = cost[s[j]][s[j+1]][5];  /* walking time */
                f7 = f1 + f2 + f3 + f4 + f5 + f6;

                for (k = 0; k < number+1; k++) (
                    if (input[k][0] == s[j+1]) {
                        f8 = input[k][2];  /* latest arrival time */
                        break;
                    }
                }
/*
                printf("%d,%d\n",s[j], s[j+1]);
                printf("earlistT,tmin
dur.,transitT,cabletravT,waitingT,walkingT,summmary,latestT\n");
                printf("%f,%f,%f,%f,%f,%f,%f,%f\n",f1,f2,f3,f4,f5,f6,f7,f8);
*/
                if (((int) f8 != 0) && (f7 > f8)) {
                    flag2 = 0;
                    break;
                }
            }

            if (flag2 == 1) (
                for (j5 = 0; j5 < (number+1); j5++) {
                    transit_fseq[transit_n_seq][j5] = s[j5];
/*                  printf("%d,", s[j5]);  */
                }
                transit_fseq[transit_n_seq][number+1] = s[number+1];
/*              printf("%d\n\n\n", s[number+1]);  */
                transit_n_seq++;
            }
```

```
                }

        /*  check the time constraint of taxi-travel mode */
                if (flag1 == 1) (
                        flag2 = 1;

                        for (j = 0; j < (number+1); j++) (
                                for (k = 0; k < number+1; k++) (
                                        if (input[k][0] == s[j]) {
                                                f1 = input[k][1];                       /*
earliest arrival time */
                                                f2 = (float) (input[k][3]/60.0);/* min. duration */
                                                break:
                                        }
                                }

                                f3 = cost[s[j]][s[j+1]][9];   /* taxi travel time */
                                f7 = f1 + f2 + f3;

                                for (k = 0; k < number+1; k++) {
                                        if (input[k][0] == s[j+1]) {
                                                f8 = input[k][2];  /* latest arrival time */
                                                break;
                                        }
                                }
/*
                                printf("%d,%d\n",s[j], s[j+1]);
                                printf("earlistT,tmin dur.,taxiT,summmary,latestT\n");
                                printf("%f,%f,%f,%f,%f\n",f1,f2,f3,f7,f8);
*/
                                if (((int) f8 != 0) && (f7 > f8)) {
                                        flag2 = 0;
                                        break;
                                }
                        }

                        if (flag2 == 1) {
```

```
                        for (j5 = 0; j5 < (number+1); j5++) (
                                taxi_fseq[taxi_n_seq][j5] = s[j5];
/*                              printf("%d,", s[j5]);  */
                        }
                        taxi_fseq[taxi_n_seq][number+1] = s[number+1];
/*                      printf("%d\n\n\n", s[number+1]); */
                        taxi_n_seq++;
                }

        }

    /*  check the time constraint of walk-travel mode */
        if (flag1 == 1) (
                flag2 = 1;

                for (j = 0;j < (number+1); j++) {
                        for (k = 0; k < number+1; k++) {
                                if (input[k][0] == s[j]) (
                                        f1 = input[k][1];                       /*
earlest arrival time */
                                        f2 = (float) (input[k][3]/60.0);/* min. duration */
                                        break;
                                }
                        }

                        f3 = cost[s[j]][s[j+1]][10]; /* walk travel time */
                        f7 = f1 + f2 + f3;

                        for (k = 0; k < number+1; k++) (
                                if (input[k][0] == s[j+1]) (
                                        f8 = input[k][2];  /* latest arrival time */
                                        break;
                                }
                        }
/*
                        printf("%d,%d\n",s[j], s[j+1]);
                        printf("earlistT,tmin dur.,walkT,summmary,latestT\n");
```

```
                                printf("%f,%f,%f,%f,%f\n",f1,f2,f3,f7,f8);
*/
                                if (((int) f8 != 0) && (f7 > f8)) (
                                        flag2 = 0;
                                        break;
                                }
                        }

                        if (flag2 == 1) (
                                for (j5 = 0; j5 < (number+1); j5++) {
                                        walk_fseq[walk_n_seq][j5] = s[j5];
/*                                      printf("%d,", s[j5]); */
                                }
                                walk_fseq[walk_n_seq][number+1] = s[number+1];
/*                              printf("%d\n\n\n", s[number+1]); */
                                walk_n_seq++;
                        }

                }

        }

/* save the tarnsit feasible sequences as file transitf_seq.temp000 */
        printf("Number of transit feasiable sequences: %d\n", transit–n–seq);
        if (transit–n–seq == 0) printf("No trasit feasiable sequence.");
        for (k=0; k < transit–n–seq; k++) {
                for (j5 = 0; j5 < (number+1); j5++) printf("%d,",transit_fseq[k][j5]);
                printf("%d\n", transit_fseq[k][number+1]);
        }
        printf("\n");

        fprintf(fptransitfseq, "%d\n", transit–n–seq);
        for (j = 0; j < transit–n–seq; j++) {
                for (k = 0; k < number+1; k++) fprintf (fptransitfseq,"%d\n", transit_fseq[j][k]);
                fprintf(fptransitfseq, "%d\n", transit_fseq[j][number+1]);
        }
```

```
/* save the taxi feasible sequences as file taxif_seq.temp000 */
        printf("Number of taxi feasiable sequences: %d\n", taxi_n_seq);
        if (taxi_n_seq == 0) printf("No taxi feasiable sequence.");
        for (k=0; k < taxi_n_seq; k++) {
                for (j5 = 0; j5 < (number+1); j5++) printf("%d,",taxi_fseq[k][j5]);
                printf("%d\n", taxi_fseq[k][number+1]);
        }
        printf("\n");

        fprintf(fptaxifseq, "%d\n", taxi_n_seq);
        for (j = 0; j < taxi_n_seq; j++) {
                for (k = 0; k < number+1; k++) fprintf (fptaxifseq, "%d\n", taxi_fseq[j][k]);
                fprintf(fptaxifseq, "%d\n", taxi_fseq[j][number+1]);
        }

/* save the walk feasible sequences as file walkf_seq.temp000 */
        printf("Number of walk feasiable sequences: %d\n", walk_n_seq);
        if (walk_n_seq == 0) printf("No walk feasiable sequence.");
        for (k=0; k < walk_n_seq; k++) {
                for (j5 = 0; j5 < (number+1); j5++) printf("%d,",walk_fseq[k][j5]);
                printf("%d\n", walk_fseq[k][number+1]);
        }
        printf("\n");

        fprintf(fpwalkfseq, "%d\n", walk_n_seq);
        for (j = 0; j < walk_n_seq; j++) {
                for (k = 0; k < number+1; k++) fprintf (fpwalkfseq, "%d\n", walk_fseq[j][k]);
                fprintf(fpwalkfseq, "%d\n", walk_fseq[j][number+1]);
        }

/* set and save initial weights of the objective function:          */
        a[0] = (float) (0.36);          /* cost weight                              */
        a[1] = (float) (0.0189);        /* transit-travel time weight       */
        a[2] = (float) (0.0458);        /* waiting time weight                  */
        a[3] = (float) (0.0121);        /* walking time weight                  */
        a[4] = (float) (0.0093);        /* transfer number weight           */
        a[5] = (float) (0.0120);        /* cable-travel time weigh          */
```

```
        a[6] = (float) (0.0463);            /* taxi-travel time weight          */
        fprintf(fpweight, "#Weights of the objective function:\n");
        for (k = 0; k < 6; k++) fprintf (fpweight, "%f,", a[k]);
        fprintf (fpweight, "%f", a[6]);

/* initial optimization      */
        opt(&a[0], &route_lst, &route_2nd, &first_route_mode, &second_route_mode);

/* output the optimization results to ArcView user interface */
/* output the total and term cost of the 1st optimized route */
        total_cost = (float) (0);
        f1 = (float) (0);   /* cost argument                              */
        f2 = (float) (0);   /* transit-travel time argument       */
        f3 = (float) (0);   /* waiting time argument              */
        f4 = (float) (0);   /* walking time argument              */
        f5 = (float) (0);   /* transfer number argument                   */
        f6 = (float) (0);   /* cable-travel time argument         */
        f7 = (float) (0);   /* taxi-travel time argument          */

        if (first_route_mode == 1) {
                for (j = 0;j < number+1; j++) {
                        node1 = transit_fseq[route_1st][j];
                        node2 = transit_fseq[route_1st][j+1];
                        f1 = f1 + cost[node1][node2][2];
                        f2 = f2 + cost[node1][node2][3];
                        f3 = f3 + cost[node1][node2][4];
                        f4 = f4 + cost[node1][node2][5];
                        f5 = f5 + cost[node1][node2][6];
                        f6 = f6 + cost[node1][node2][7];
                }
                total_cost = a[0]*f1 + a[1]*f2 + a[2]*f3 + a[3]*f4 + a[4]*f5 + a[5]*f6;

                printf("The travel mode with the minimum cost: %d\n", first_route_mode);
                printf("The sequence with the minimum cost:\n");
                for (k=0; k < number+1; k++) printf ("%d ", transit_fseq[route_1st][k]);
                printf("%d\n", transit_fseq[route_1st][number+1]);
```

```
            printf("transit_travelcost($),transit_traveltime(hr),waitime(hr),walktime(hr),transfer-
num,cable_traveltime(hr)\n");
            printf("%f\t%f\t%f\t%f\t%f\t%f\n", f1, f2, f3, f4,f5, f6);
            printf("The minimum total cost:\n");
            printf("%f\n\n", total-cost);

            fprintf(fptemp,"#The travel mode with the minimum cost:\n");
            fprintf(fptemp,"%d\n", first-route-mode);
            fprintf(fptemp, "#The sequence with the minimum cost:\n");
            for (k=0; k < number+1; k++) fprintf (fptemp, "%d ", transit_fseq[route_1st][k]);
            fprintf(fptemp,"%d\n", transit_fseq[route_1st][number+1]);

        fprintf(fptemp,"#transit_travelcost($),transit_traveltime(hr),waitime(hr),walktime(hr),transfer-
num,cable_traveltime(hr)\n");
            fprintf(fptemp,"%f\t%f\t%f\t%f\t%f\t%f\n", f1, f2, f3, f4, f5, f6);
            fprintf(fptemp, "#The minimum cost:\n");
            fprintf(fptemp, "%f\n\n", total-cost);
        }

        if (first-route-mode== 2) {
            for (j = 0;j < number+1 ;j++) {
                node1 = taxi_fseq[route_1st][j];
                node2 = taxi_fseq[route_1st][j+1];
                f1 = f1 + cost[node1][node2][8];
                f7 = f7 + cost[node1][node2][9];
            }
            total-cost = a[0]*f1 + a[6]*f7;

            printf("The travel mode with the minimum cost: %d\n", first-route-mode);
            printf("The sequence with the minimum cost:\n");
            for (k=0; k < number+1; k++) printf ("%d ", taxi_fseq[route_1st][k]);
            printf("%d\n", taxi_fseq[route_1st][number+1]);
            printf("taxi_travelcost($),taxi_traveltime(hr)\n");
            printf("%f\t%f\n", f1, f7);
            printf("The minimum total cost:\n");
            printf("%f\n\n", total-cost);

            fprintf(fptemp,"#The travel mode with the minimum cost:\n");
```

```
            fprintf(fptemp,"%d\n", first−route−mode);
            fprintf(fptemp, "#The sequence with the minimum cost:\n");
            for (k=0; k < number+1; k++) fprintf (fptemp, "%d ", taxi_fseq[route_1st][k]);
            fprintf(fptemp,"%d\n", taxi_fseq[route_1st][number+1]);
            fprintf(fptemp,"#taxi_travelcost($),taxi_traveltime(hr)\n");
            fprintf(fptemp,"%f\t%f\n", f1, f7);
            fprintf(fptemp, "#The minimum cost:\n");
            fprintf(fptemp, "%f\n\n", total−cost);
}

if (first−route−mode== 3) {
            for (j = 0; j < number+1 ;j++) {
                        node1 = walk_fseq[route_1st][j];
                        node2 = walk_fseq[route_1st][j+1];
                        f4 = f4 + cost[node1][node2][10];
            }
            total−cost = a[3]*f4;

            printf("The travel mode with the minimum cost: %d\n", fist−route−mode);
            printf("The sequence with the minimum cost:\n");
            for (k=0; k < number+1; k++) printf ("%d ", walk_fseq[route_1st][k]);
            printf("%d\n", walk_fseq[route_1st][number+1]);
            printf("walk_traveltime(hr)\n");
            printf("%f\n", f4);
            printf("The minimum total cost:\n");
            printf("%f\n\n", total−cost);

            fprintf(fptemp,"#The travel mode with the minimum cost:\n");
            fprintf(fptemp,"%d\n", first−route−mode);
            fprintf(fptemp, "#The sequence with the minimum cost:\n");
            for (k=0; k < number+1; k++) fprintf (fptemp, "%d ", walk_fseq[route_1st][k]);
            fprintf(fptemp,"%d\n", walk_fseq[route_1st][number+1]);
            fprintf(fptemp,"#walk_traveltime(hr)\n");
            fprintf(fptemp,"%f\n", f4);
            fprintf(fptemp, "#The minimum cost:\");
            fprintf(fptemp, "%f\n\n", total−cost);
```

```
        }


/* output the total and term cost of the 2nd optimized route */
        total–cost = (float) (0);
        f1 = (float) (0);   /* cost argument                          */
        f2 = (float) (0);   /* transit-travel time argument       */
        f3 = (float) (0);   /* waiting time argument              */
        f4 = (float) (0);   /* walking time argument              */
        f5 = (float) (0);   /* transfer number argument               */
        f6 = (float) (0);   /* cable-travel time argument         */
        f7 = (float) (0);   /* taxi-travel time argument          */

        if (second–route–mode== 1) {
                for (j = 0; j < number+1; j++) {
                        node1 = transit_fseq[route_2nd][j];
                        node2 = transit_fseq[route_2nd][j+1];
                        f1 = f1 + cost[node1][node2][2];
                        f2 = f2 + cost[node1][node2][3];
                        f3 = f3 + cost[node1][node2][4];
                        f4 = f4 + cost[node1][node2][5];
                        f5 = f5 + cost[node1][node2][6];
                        f6 = f6 + cost[node1][node2][7];
                }
                total–cost = a[0]*f1 + a[1]*f2 + a[2]*f3 + a[3]*f4 + a[4]*f5 + a[5]*f6;

                printf("The travel mode with the second cost: %d\n", second–route–mode);
                printf("The sequence with the second cost:\n");
                for (k=0; k < number+1; k++) printf ("%d ", transit_fseq[route_2nd][k]);
                printf("%d\n", transit_fseq[route_2nd][number+1]);
                printf("transit_travelcost($),transit_traveltime(hr),waitime(hr),walktime(hr),transfer-
num,cable_traveltime(hr)\n");
                printf("%f\t%f\t%f\t%f\t%f\t%f\n", f1, f2, f3, f4, f5, f6);
                printf("The second total cost:\n");
                printf("%f\n\n", total–cost);

                fprintf(fptemp,"#The travel mode with the second cost:\n");
```

```
                fprintf(fptemp,"%d\n", second−route−mode);
                fprintf(fptemp, "#The sequence with the second cost:\n");
                for (k=0; k < number+1; k++) fprintf (fptemp, "%d ", transit_fseq[route_2nd][k]);
                fprintf(fptemp,"%d\n", transit_fseq[route_2nd][number+1]);

        fprintf(fptemp,"#transit_travelcost($),transit_traveltime(hr),waitime(hr),walktime(hr),transfer-
num,cable_traveltime(hr)\n");
                fprintf(fptemp,"%f\t%f\t%f\t%f\t%f\t%f\n", f1, f2, f3, f4, f5, f6);
                fprintf(fptemp, "#The second cost:\n");
                fprintf(fptemp, "%f\n\n", total−cost);


        if (second−route−mode == 2) {
                for (j = 0;j < number+1; j++) (
                        node1 = taxi_fseq[route_2nd][j];
                        node2 = taxi_fseq[route_2nd][j+1];
                        f1 = f1 + cost[node1][node2][8];
                        f7 = f7 + cost[node1][node2][9];
                }
                total−cost = a[0]*f1 + a[6]*f7;

                printf("The travel mode with the second cost: %d\n", second−route−mode);
                printf("The sequence with the second cost:\n");
                for (k=0; k < number+1; k++) printf ("%d ", taxi_fseq[route_2nd][k]);
                printf("%d\n", taxi_fseq[route_2nd][number+1]);
                printf("taxi_travelcost($),taxi_traveltime(hr)\n");
                printf("%f\t%f\n", f1, f7);
                printf("The second total cost:\n");
                printf("%f\n\n", total−cost);

                fprintf(fptemp,"#The travel mode with the second cost:\n");
                fprintf(fptemp,"%d\n", second−route−mode);
                fprintf(fptemp, "#The sequence with the second cost:\n");
                for (k=0; k < number+1; k++) fprintf (fptemp, "%d ", taxi_fseq[route_2nd][k]);
                fprintf(fptemp,"%d\n", taxi_fseq[route_2nd][number+1]);
                fprintf(fptemp,"#taxi_travelcost($),taxi_traveltime(hr)\n");
                fprintf(fptemp,"%f\t%f\n", f1, f7);
                fprintf(fptemp, "#The second cost:\n");
```

```
            fprintf(fptemp, "%f\n\n", total–cost);
        }

        if (secondroute–mode == 3) {
            for (j = 0;j < number+1 ;j++) {
                nodel  = walk_fseq[route_2nd][j];
                node2 = walk_fseq[route_2nd][j+1];
                f4 = f4 + cost[node1][node2][10];
            }
            total–cost = a[3]*f4;

            printf("The travel mode with the second cost: %d\n", second–route–mode);
            printf("The sequence with the second cost:\n");
            for (k=0; k < number+1; k++) printf ("%d ", walk_fseq[route_2nd][k]);
            printf("%d\n", walk_fseq[route_2nd][number+1]);
            printf("walk_traveltime(hr)\n");
            printf("%f\n", f4);
            printf("The second total cost:\n");
            printf("%f\n\n", total–cost);

            fprintf(fptemp,"#The travel mode with the second cost:\n");
            fprintf(fptemp,"%d\n", second–route–mode);
            fprintf(fptemp, "#The sequence with the second cost:\n");
            for (k=0; k < number+1; k++) fprintf (fptemp, "%d ", walk_fseq[route_2nd][k]);
            fprintf(fptemp,"%d\n", walk_fseq[route_2nd][number+1]);
            fprintf(fptemp,"#walk_traveltime(hr)\n");
            fprintf(fptemp,"%f\n", f4);
            fprintf(fptemp, "#The second cost:\n");
            fprintf(fptemp, "%f\n\n", total–cost);

        }


/* output the total and term cost of all feasible routes */
for (kk = 0; kk < transit–n–seq; kk++) {
        total–cost = (float) (0);
        f1 = (float) (0);
```

```
        f2 = (float) (0);
        f3 = (float) (0);
        f4 = (float) (0);
        f5 = (float) (0);
        f6 = (float) (0);
        f7 = (float) (0);
        for (j = 0; j < number+1; j++) {
                nodel = transit_fseq[kk][j];
                node2 = transit_fseq[kk][j+1];
                f1 = f1 + cost[node1][node2][2];
                f2 = f2 + cost[node1][node2][3];
                f3 = f3 + cost[node1][node2][4];
                f4 = f4 + cost[node1][node2][5];
                f5 = f5 + cost[node1][node2][6];
                f6 = f6 + cost[node1][node2][7];
        }
        total-cost = a[0]*f1 + a[1]*f2 + a[2]*f3 + a[3]*f4 + a[4]*f5 + a[5]*f6 + a[6]*f7;

        printf("The transit sequence:\n");
        for (k=0; k < number+1; k++) printf ("%d ", taxi_fseq[kk][k]);
        printf("%d\n", transit_fseq[kk][number+1]);
        printf("cost($),transit_traveltime(hr),waitime(hr),walktime(hr),transfer-
num,cable_traveltime(hr),taxi_travel time(hr)\n");
        printf("%f\t%f\t%f\t%f\t%f\t%f\t%f\n", f1, f2, f3, f4, f5, f6, f7);
        printf("The total cost:\n");
        printf("%f\n\n", total-cost);

}

for (kk = 0; kk < taxi-n-seq; kk++) {
        total-cost = (float) (0);
        f1 = (float) (0);
        f2 = (float) (0);
        f3 = (float) (0);
        f4 = (float) (0);
        f5 = (float) (0);
        f6 = (float) (0);
```

```
        f7 = (float) (0);
        for (j = 0;j < number+1; j++) {
                nodel  = taxi_fseq[kk][j];
                node2 = taxi_fseq[kk][j+1];
                f1 = f1 + cost[node1][node2][8];
                f7 = f7 + cost[node1][node2][9];
        }
        total–cost = a[0]*f1 + a[1]*f2 + a[2]*f3 + a[3]*f4 + a[4]*f5 + a[5]*f6 + a[6]*f7;

        printf("The taxi sequence:\n");
        for (k=0; k < number+1; k++) printf ("%d ", taxi_fseq[kk][k]);
        printf("%d\n", taxi_fseq[kk][number+1]);
        printf("cost($),transit_traveltime(hr),waitime(hr),walktime(hr),transfer-
num,cable_traveltime(hr),taxi_travel time(hr)\n");
        printf("%f\t%f\t%f\t%f\t%f\t%f\t%f\n", f1, f2, f3, f4, f5, f6, f7);
        printf("The total cost:\n");
        printf("%f\n\n", total–cost);

}

for (kk = 0; kk < walk–n–seq; kk++) {
        total–cost = (float) (0);
        f1 = (float) (0);
        f2 = (float) (0);
        f3 = (float) (0);
        f4 = (float) (0);
        f5 = (float) (0);
        f6 = (float) (0);
        f7 = (float) (0);
        for (j = 0;j < number+1 ;j++) {
                nodel  = walk_fseq[kk][j];
                node2 = walk_fseq[kk][j+1];
                f4 = f4 + cost[node1][node2][10];
        }
        total–cost = a[0]*f1 + a[1]*f2 + a[2]*f3 + a[3]*f4 + a[4]*f5 + a[5]*f6 + a[6]*f7;

        printf("The walk sequence:\n");
```

```
        for (k=0; k < number+1; k++) printf ("%d ", walk_fseq[kk][k]);
        printf("%d\n", walk_fseq[kk][number+1]);
        printf("cost($),transit_traveltime(hr),waitime(hr),walktime(hr),transfer-
num,cable_traveltime(hr),taxi_travel time(hr)\n");
        printf("%f\t%f\t%f\t%f\t%f\t%f\t%f\n", f1, f2, f3, f4, f5, f6, f7);
        printf("The total cost:\n");
        printf("%f\n\n", total_cost);

}

}
```

Code-cod2.cpp

```
/*      Filename:               cod2.c
        Author:         Jiayu Chen
        Date Created    10-05-97
        Date Updated    11-15-97

        Purpose:
        Input:
        output:

        Variables:
*/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <sys/types.h>
#include <math.h>
#define  ROWS   41000
#define  COLS    8
#include "opt.cpp"
```

```
float     cost[25][25][11], u[ROWS];
unsigned  char  p[ROWS][COLS],  transit_fseq[ROWS][COLS+2],  taxi_fseq[ROWS][COLS+2],
walk_fseq[ROWS][COLS+2];
int               number, transit–n–seq, taxi–n–seq, walk–n–seq;

void main()
{
        FILE   *fpcost, *fpnumber, *fpweight, "fptransitfseq, *fptaxifseq, *fpwalkfseq, *fpprefer,
*fpopt_route, *fpopt_out;
        unsigned char prefer–route;
        char     line[255];
        int              i, j, k, kk, kkk;
        int              nodel, node2, key, key1, key2, model, mode2;
        int              route–lst, route_2nd, first–route–mode, second–route–mode;
        float    preferjndex, sum, total–cost;
        float    f1, f2, f3, f4, f5, f6, f7, f8, f9, f10, f11;
        float    a[7];
        float    route_value[2][7];

/* data loading */
        printf("open files ...\n");
        fpcost = fopen("cost.dat", "rt");
        fpnumber = fopen("total.dat", "rt");
        fpweight = fopen("weight.dat", "r+");
        fptransitfseq = fopen("tansitf_seq.temp000", "rt");
        fptaxifseq  = fopen("taxif_seq.temp000", "rt");
        fpwalkfseq = fopen("walkf_seq.temp000", "rt");
        fpprefer = fopen("userprefer.temp000", "rt");
        fpopt–route = fopen("opt_route_1st_2nd.temp000", "r+");
        fpopt–out = fopen("opt_out.temp000", "wt");

        printf("Reading ..\n");

/* read traveling network attributes data file cost.dat */
        while (fgets(line, 255, fpcost) != NULL) {
                if (line[0] != '#') {
                        sscanf(line,                "%d,%d,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f,%f",
&i,&j,&f1,&f2,&f3,&f4,&f5,&f6,&f7,&f8,&f9,&f10,&f11);
```

```
                    cost[i][j][0] = f1;
                    cost[i][j][1] = f2;
                    cost[i][j][2] = f3;
                    cost[i][j][3] = f4;
                    cost[i][j][4] = f5;
                    cost[i][j][5] = f6;
                    cost[i][j][6] = f7;
                    cost[i][j][7] = f8;
                    cost[i][j][8] = f9;
                    cost[i][j][9] = f10;
                    cost[i][j][10] = f11;
                }
        }

/* read total number of all possible sequences */
        while (fgets(line, 255, fpnumber) != NULL) {
                if (line[0] != '#') sscanf(line, "%d\n", &number);
        }

/* read feasible sequences of transit mode */
        fgets(line, 255, fptransitfseq);
        sscanf(line, "%d", &transit_n_seq);
        for (k = 0; k < transit_n_seq; k++) {
                for (j = 0; j < number+2; j++) {
                        fgets(line, 255, fptransitfseq);
                        sscanf(line, "%d", &transit_fseq[k][j]);
                }
        }

/* read feasible sequences of taxi mode */
        fgets(line, 255, fptaxifseq);
        sscanf(line, "%d", &taxi_n_seq);
        for (k = 0; k < taxi_n_seq; k++) {
                for (j = 0; j < number+2; j++) {
                        fgets(line, 255, fptaxifseq);
                        sscanf(line, "%d", &taxi_fseq[k][j]);
                }
```

```
        }

/* read feasible sequences of walk mode */
        fgets(line, 255, fpwalkfseq);
        sscanf(line, "%d", &walk_n_seq);
        for (k = 0; k < walk–n–seq; k++) {
                for (j = 0; j < number+2; j++) {
                        fgets(line, 255, fpwalkfseq);
                        sscanf(line, "%d", &walk_fseq[k][j]);
                }
        }

/* read objective function weights */
        while (fgets(line, 255, fpweight) != NULL) {
                if (line[0] != '#') sscanf(line, "%f,%f,%f,%f,%f,%f,%f",
&a[0],&a[1],&a[2],&a[3],&a[4],&a[5],&a[6]);
        }
        rewind(fpweight);

/* reading for previous opt–route */
        fgets(line, 255, fpopt–route);
        sscanf(line, "%d,%d", &key1, &mode1);
        fgets(line, 255, fpopt–route);
        sscanf(line, "%d,%d", &key2, &mode2);
        rewind(fpopt_route);

/* getting prefer–index */
        fgets(line, 255, fpprefer);
        sscanf(line, "%d", &prefer–route);
        if (prefer–route == 1) prefer–index = float(1.0);
        if (prefer–route == 2) preferjndex = -float(1.0);

        printf("reading done\n\n");

/* Computing the route attributes of 1st and 2nd route */
        for (k = 0; k < 2; k++) {
                for (kkk = 0; kkk < 7; kkk++) route_value[k][kkk] = (float)(0);
```

```
if (k == 0) key = key1;
if (k == 1) key = key2;
for (j = 0; j < number+1; j++) {
        if (k == 0) {
                if (model == 1) {
                        nodel = transit_fseq[key][j];
                        node2 = transit_fseq[key][j+1];
                        f1 = cost[node1][node2][2];
                        f2 = cost[node1][node2][3];
                        f3 = cost[node1][node2][4];
                        f4 = cost[node1][node2][5];
                        f5 = cost[node1][node2][6];
                        f6 = cost[node1][node2][7];
                        f7 = (float) (0.);
                }
                if (model == 2) (
                        nodel = taxi_fseq[key][j];
                        node2 = taxi_fseq[key][j+1];
                        f1 = cost[node1][node2][8];
                        f2 = (float)(0.);
                        f3 = (float)(0.);
                        f4 = (float)(0.);
                        f5 = (float)(0.);
                        f6 = (float)(0.);
                        f7 = cost[node1][node2][9];
                }
                if (model == 3) (
                        nodel = walk_fseq[key][j];
                        node2 = walk_fseq[key][j+1];
                        f1 = (float)(0.);
                        f2 = (float)(0.);
                        f3 = (float)(0.);
                        f4 = cost[node1][node2][10];
                        f5 = (float)(0.);
                        f6 = (float)(0.);
                        f7 = (float)(0.);
                }
```

```
}
if (k == 1) {
        if (mode2 == 1){
                nodel =transit_fseq[key][j];
                node2 =transit_fseq[key][j+1];
                f1 =cost[node1][node2][2];
                f2 = cost[node1][node2][3];
                f3 =cost[node1][node2][4];
                f4 =cost[node1][node2][5];
                f5 =cost[node1][node2][6];
                f6 =cost[node1][node2][7];
                f7 = (float) (0.);
        }
        if (mode2 == 2) {
                nodel =taxi_fseq[key][j];
                node2 =taxi_fseq[key][j+1];
                f1 =cost[node1][node2][8];
                f2 =(float)(0.);
                f3 =(float)(0.);
                f4 =(float)(0.);
                f5 =(float)(0.);
                f6 =(float)(0.);
                f7 =cost[node1][node2][9];
        }
        if (mode2 == 3) {
                nodel = walk_fseq[key][j];
                node2 = walk_fseq[key][j+1];
                f1 =(float)(0.);
                f2 =(float)(0.);
                f3 =(float)(0.);
                f4 = cost[node1][node2][10];
                f5 =(float)(0.);
                f6 =(float)(0.);
                f7 =(float)(0.);
        }
}
route_value[k][0] =route_value[k][0] + f1;
```

```
                route_value[k][1] = route_value[k][1] + f2;
                route_value[k][2] = route_value[k][2] + f3;
                route_value[k][3] = route_value[k][3] + f4;
                route_value[k][4] = route_value[k][4] + f5;
                route_value[k][5] = route_value[k][5] + f6;
                route_value[k][6] = route_value[k][6] + f7;
        }
printf("route_value:\n");
for (kkk = 0; kkk < 6; kkk++) printf ("%f,", route_value[k][kkk]);
printf ("%f\n", route_value[k][6]);
}

if(route_value[0][0] <route–value[1][0]) a[0] = a[0]*(float)(pow(2,prefer_index));
if(route_value[0][0] > route_value[1][0]) a[0] = a[0]/(float)(pow(2,prefer_index));
if(route_value[0][1] < route–value[1][1]) a[1] = a[1]*(float)(pow(2,prefer_index));
if(route_value[0][1] > route–value[1][1]) a[1] = a[1]/(float)(pow(2,prefer_index));
if(route_value[0][2] < route_value[1][2]) a[2] = a[2]*(float)(pow(2,prefer_index));
if(route_value[0][2] > route_value[1][2]) a[2] = a[2]/(float)(pow(2,prefer_index));
if(route_value[0][3] < route–value[1][3]) a[3] = a[3]*(float)(pow(2,prefer_index));
if(route_value[0][3] > route–value[1][3]) a[3] = a[3]/(float)(pow(2,prefer_index));
if(route_value[0][4] < route–value[1][4]) a[4] = a[4]*(float)(pow(2,prefer_index));
if(route_value[0][4] > route–value[1][4]) a[4] = a[4]/(float)(pow(2,prefer_index));
if(route_value[0][5] < route_value[1][5]) a[5] = a[5]*(float)(pow(2,prefer_index));
if(route_value[0][5] > route–value[1][5]) a[5] = a[5]/(float)(pow(2,prefer_index));
if(route_value[0][6] <route_value[1][6]) a[6] = a[6]*(float)(pow(2,prefer_index));
if(route_value[0][6] > route–value[1][6]) a[6] = a[6]/(float)(pow(2,prefer_index));
sum = a[0] + a[1] + a[2] + a[3] + a[4] + a[5]+ a[6];
a[0] = a[0]/sum;
a[1] = a[1]/sum;
a[2] = a[2]/sum;
a[3] = a[3]/sum;
a[4] = a[4]/sum;
a[5] = a[5]/sum;
a[6] = a[6]/sum;
printf("the weights of the objective function: \n");
printf("%f\t%f\t%f\t%f\t%f\t%f\t%f\n\n", a[0],a[1],a[2],a[3],a[4],a[5],a[6]);
```

```
/* update the objective function weights */
        fprintf(fpweight, "#Weights of the objective function:\n");
        for (k = 0; k < 6; k++) fprintf (fpweight, "%f,", a[k]);
        fprintf (fpweight, "%f", a[6]);

/* optimization  */
        opt(&a[0], &route-lst, &route_2nd, &first-route-mode, &second-route-mode);
        if (route_1st != keyl) {
                route_2nd = keyl;
                secondroute-mode = mode1;
        }

/* output the optimization results to ArcView user interface */
/* output the total and term cost of the 1st optimized route */
        total-cost = (float) (0);
        f1 = (float) (0);   /* cost argument                          */
        f2 = (float) (0);   /* transit-travel time argument       */
        f3 = (float) (0);   /* waiting time argument              */
        f4 = (float) (0);   /* walking time argument              */
        f5 = (float) (0);   /* transfer number argument               */
        f6 = (float) (0);   /* cable-travel time argument         */
        f7 = (float) (0);   /* taxi-travel time argument          */

        if (first-route-mode == 1) (
                for (j = 0; j < number+1 ; j++) {
                        node1 = transit_fseq[route_1st][j];
                        node2 = transit_fseq[route_1st][j+1];
                        f1 = f1 + cost[node1][node2][2];
                        f2 = f2 + cost[node1][node2][3];
                        f3 = f3 + cost[node1][node2][4];
                        f4 = f4 + cost[node1][node2][5];
                        f5 = f5 + cost[node1][node2][6];
                        f6 = f6 + cost[node1][node2][7];
                }
                total-cost = a[0]*f1 + a[1]*f2 + a[2]*f3 + a[3]*f4 + a[4]*f5 + a[5]*f6;

                printf("The travel mode with the minimum cost: %d\n", first-route-mode);
```

```
                printf("The sequence with the minimum cost:\n");
                for (k=0; k < number+1; k++) printf ("%d ", transit_fseq[route_1st][k]);
                printf("%d\n", transit_fseq[route_1st][number+1]);
                printf("transit_travelcost($),transit_traveltime(hr),waitime(hr),walktime(hr),transfer-
num,cable_traveltime(hr)\n");
                printf("%f\t%f\t%f\t%f\t%f\t%f\n", f1, f2, f3, f4, f5, f6);
                printf("The minimum total cost:\n");
                printf("%f\n\n", total-cost);

                fprintf(fpopt_out,"#The travel mode with the minimum cost:\n");
                fprintf(fpopt_out,"%d\n", first-route-mode);
                fprintf(fpopt_out, "#The sequence with the minimum cost:\n");
                for (k=0; k < number+1; k++) fprintf (fpopt-out,"%d ", transit_fseq[route_1st][k]);
                fprintf(fpopt_out,"%d\n", transit_fseq[route_1st][number+1]);

        fprintf(fpopt_out,"#transit_travelcost($),transit_traveltime(hr),waitime(hr),walktime(hr),transfer-
num,cable_traveltime(hr)\n");
                fprintf(fpopt_out,"%f\t%f\t%f\t%f\t%f\t%f\n", f1, f2, f3, f4, f5, f6);
                fprintf(fpopt_out, "#The minimum cost:\n");
                fprintf(fpopt_out, "%f\n\n", total-cost);
        }

        if (first-route-mode == 2) {
                for (j = 0;j < number+1;j++) {
                        node1 = taxi_fseq[route_1st][j];
                        node2 = taxi_fseq[route_1st][j+1];
                        f1 = f1 + cost[node1][node2][8];
                        f7 = f7 + cost[node1][node2][9];
                }
                total-cost = a[0]*f1 + a[6]*f7;

                printf("The travel mode with the minimum cost: %d\n", first-route-mode);
                printf("The sequence with the minimum cost:\n");
                for (k=0; k < number+1; k++) printf ("%d ", taxi_fseq[route_1st][k]);
                printf("%d\n", taxi_fseq[route_1st][number+1]);
                printf("taxi_travelcost($),taxi_traveltime(hr)\n");
                printf("%f\t%f\n", f1, f7);
                printf("The minimum total cost:\n");
```

```
            printf("%f\n\n", total–cost);

            fprintf(fpopt_out,"#The travel mode with the minimum cost:\n");
            fprintf(fpopt_out,"%d\n", first–route–mode);
            fprintf(fpopt_out, "#The sequence with the minimum cost:\n");
            for (k=0; k < number+1; k++) fprintf (fpopt–out, "%d ", taxi_fseq[route_1st][k]);
            fprintf(fpopt_out,"%d\n", taxi_fseq[route_1st][number+1]);
            fprintf(fpopt_out,"#taxi_travelcost($),taxi_traveltime(hr)\n");
            fprintf(fpopt_out,"%f\t%f\n", f1, f7);
            fprintf(fpopt_out, "#The minimum cost:\n");
            fprintf(fpopt_out, "%f\n\n", total–cost);
}

if (first–route–mode == 3) {
            for (j = 0; j < number+1; j++) {
                        node1 = walk_fseq[route_1st][j];
                        node2 = walk_fseq[route_1st][j+1];
                        f4 = f4 + cost[node1][node2][10];
            }
            total–cost = a[3]*f4;

            printf("The travel mode with the minimum cost: %d\n", first–route–mode);
            printf("The sequence with the minimum cost:\n");
            for (k=0; k < number+1; k++) printf ("%d ", walk_fseq[route_1st][k]);
            printf("%d\n", walk_fseq[route_1st][number+1]);
            printf("walk_traveltime(hr)\n");
            printf("%f\n", f4);
            printf("The minimum total cost:\n");
            printf("%f\n\n", total–cost);

            fprintf(fpopt_out,"#The travel mode with the minimum cost:\n");
            fprintf(fpopt_out,"%d\n", first–route–mode);
            fprintf(fpopt_out, "#The sequence with the minimum cost:\n");
            for (k=0; k < number+1; k++) fprintf (fpopt–out, "%d ", walk_fseq[route_1st][k]);
            fprintf(fpopt_out,"%d\n", walk_fseq[route_1st][number+1]);
            fprintf(fpopt_out,"#walk_traveltime(hr)\n");
            fprintf(fpopt_out,"%f\n", f4);
```

```
                fprintf(fpopt_out, "#The minimum cost:\n");
                fprintf(fpopt_out, "%f\n\n", total-cost);

        }

/* output the total and term cost of the 2nd optimized route */
        total-cost = (float) (0);
        f1 = (float) (0);   /* cost argument                          */
        f2 = (float) (0);   /* transit-travel time argument    */
        f3 = (float) (0);   /* waiting time argument           */
        f4 = (float) (0);   /* walking time argument           */
        f5 = (float) (0);   /* transfer number argument                */
        f6 = (float) (0);   /* cable-travel time argument      */
        f7 = (float) (0);   /* taxi-travel time argument       */

        if (second-route-mode== 1) (
                for (j = 0; j < number+1; j++) {
                        node1 = transit_fseq[route_2nd][j];
                        node2 = transit_fseq[route_2nd][j+1];
                        f1 = f1 + cost[node1][node2][2];
                        f2 = f2 + cost[node1][node2][3];
                        f3 = f3 + cost[node1][node2][4];
                        f4 = f4 + cost[node1][node2][5];
                        f5 = f5 + cost[node1][node2][6];
                        f6 = f6 + cost[node1][node2][7];
                }
                total-cost = a[0]*f1 + a[1]*f2 + a[2]*f3 + a[3]*f4 + a[4]*f5 + a[5]*f6;

                printf("The travel mode with the second cost: %d\n", second-route-mode);
                printf("The sequence with the second cost:\n");
                for (k=0; k c number+1; k++) printf ("%d ", transit_fseq[route_2nd][k]);
                printf("%d\n", transit_fseq[route_2nd][number+1]);
                printf("transit_travelcost($),transit_traveltime(hr),waitime(hr),walktime(hr),transfer-
num,cable_traveltime(hr)\n");
                printf("%f\t%f\t%f\t%f\t%f\t%f\n", f1, f2, f3, f4, f5, f6);
                printf("The second total cost:\n");
                printf("%f\n\n", total-cost);
```

```
                fprintf(fpopt_out,"#The travel mode with the second cost:\n");
                fprintf(fpopt_out,"%d\n", second–route–mode);
                fprintf(fpopt_out, "#The sequence with the second cost:\n");
                for (k=0; k < number+1; k++) fprintf (fpopt–out, "%d ", transit_fseq[route_2nd][k]);
                fprintf(fpopt_out,"%d\n", transit_fseq[route_2nd][number+1]);

        fprintf(fpopt_out,"#transit_travelcost($),transit_traveltime(hr),waitime(hr),walktime(hr),transfer-
num,cable_traveltime(hr)\n");
                fprintf(fpopt_out,"%f\t%f\t%f\t%f\t%f\t%f\n", f1, f2, f3, f4, f5, f6);
                fprintf(fpopt_out, "#The second cost:\n");
                fprintf(fpopt_out, "%f\n\n", total–cost);
        }

        if (second–route–mode == 2) {
                for (j = 0; j < number+1 ;j++) {
                        node1 = taxi_fseq[route_2nd][j];
                        node2 = taxi_fseq[route_2nd][j+1];
                        f1 = f1 + cost[node1][node2][8};
                        f7 = f7 + cost[node1][node2][9};
                }
                total–cost = a[0]*f1 + a[6]*f7;

                printf("The travel mode with the second cost: %d\n", second–route–mode);
                printf("The sequence with the second cost:\n");
                for (k=0; k < number+1; k++) printf ("%d ", taxi_fseq[route_2nd][k]);
                printf("%d\n", taxi_fseq[route_2nd][number+1]);
                printf("taxi_travelcost($),taxi_traveltime(hr)\n");
                printf("%f\t%f\n", f1, f7);
                printf("The second total cost:\n");
                printf("%f\n\n", total–cost);

                fprintf(fpopt_out,"#The travel mode with the second cost:\n");
                fprintf(fpopt_out,"%d\n", secondroute–mode);
                fprintf(fpopt_out, "#The sequence with the second cost:\n");
                for (k=0; k < number+1; k++) fprintf (fpopt–out, "%d ", taxi_fseq[route_2nd][k]);
                fprintf(fpopt_out,"%d\n", taxi_fseq[route_2nd][number+1]);
                fprintf(fpopt_out,"#taxi_travelcost($),taxi_traveltime(hr)\n");
```

```
            fprintf(fpopt_out,"%f\t%f\n", f1, f7);
            fprintf(fpopt_out, "#The second cost:\n");
            fprintf(fpopt_out, "%f\n\n", total-cost);
        }

        if (second-route-mode== 3) {
            for (j = 0;j < number+1; j++) {
                node1 = walk_fseq[route_2nd][j];
                node2 = walk_fseq[route_2nd][j+1];
                f4 = f4 + cost[node1][node2][10];
            }
            total-cost = a[3]*f4;

            printf("The travel mode with the second cost: %d\n", second-route-mode);
            printf("The sequence with the second cost:\n");
            for (k=0; k < number+1; k++) printf ("%d ", walk_fseq[route_2nd][k]);
            printf("%d\n", walk_fseq[route_2nd][number+1]);
            printf("walk_traveltime(hr)\n");
            printf("%f\n", f4);
            printf("The second total cost:\n");
            printf("%f\n\n", total-cost);

            fprintf(fpopt_out,"#The travel mode with the second cost:\n");
            fprintf(fpopt_out,"%d\n", second-route-mode);
            fprintf(fpopt_out, "#The sequence with the second cost:\n");
            for (k=0; k < number+1; k++) fprintf (fpopt-out, "%d ", walk_fseq[route_2nd][k]);
            fprintf(fpopt_out,"%d\n", walk_fseq[route_2nd][number+1]);
            fprintf(fpopt_out,"#walk_traveltime(hr)\n");
            fprintf(fpopt_out,"%f\n", f4);
            fprintf(fpopt_out, "#The second cost:\n");
            fprintf(fpopt_out, "%f\n\n", total-cost);

        }

/* output the total and term cost of all feasible routes */
for (kk= 0; kk < transit-n-seq; kk++) {
        total-cost = (float) (0);
```

```
        f1 = (float) (0);
        f2 = (float) (0);
        f3 = (float) (0);
        f4 = (float) (0);
        f5 = (float) (0);
        f6 = (float) (0);
        f7 = (float) (0);
        for (j = 0; j < number+1; j++) {
                node1 = transit_fseq[kk][j];
                node2 = transit_fseq[kk][j+1];
                f1 = f1 + cost[node1][node2][2];
                f2 = f2 + cost[node1][node2][3];
                f3 = f3 + cost[node1][node2][4];
                f4 = f4 + cost[node1][node2][5];
                f5 = f5 + cost[node1][node2][6];
                f6 = f6 + cost[node1][node2][7];
        }
        total_cost = a[0]*f1 + a[1]*f2 + a[2]*f3 + a[3]*f4 + a[4]*f5 + a[5]*f6 + a[6]*f7;

        printf("The transit sequence:\n");
        for (k=0; k < number+1; k++) printf ("%d ", taxi_fseq[kk][k]);
        printf("%d\n", transit_fseq[kk][number+1]);
        printf("cost($),transit_traveltime(hr),waitime(hr),walktime(hr),transfer-
num,cable_traveltime(hr),taxi_travel time(hr)\n");
        printf("%f\t%f\t%f\t%f\t%f\t%f\t%f\n", f1, f2, f3, f4, f5, f6, f7);
        printf("The total cost:\n");
        printf("%f\n\n", total_cost);



for (kk = 0; kk < taxi_n_seq; kk++) {
        total_cost = (float) (0);
        f1 = (float) (0);
        f2 = (float) (0);
        f3 = (float) (0);
        f4 = (float) (0);
        f5 = (float) (0);
```

```
        f6 = (float) (0);
        f7 = (float) (0);
        for (j = 0;j < number+1; j++) {
                node1 = taxi_fseq[kk][j];
                node2 = taxi_fseq[kk][j+1];
                f1 = f1 + cost[node1][node2][8];
                f7 = f7 + cost[node1][node2][9];
        }
        total–cost = a[0]*f1 + a[1]*f2 + a[2]*f3 + a[3]*f4 + a[4]*f5 + a[5]*f6 + a[6]*f7;

        printf("The taxi sequence:\n");
        for (k=0; k < number+1; k++) printf ("%d ", taxi_fseq[kk][k]);
        printf("%d\n", taxi_fseq[kk][number+1]);
        printf("cost($),transit_traveltime(hr),waitime(hr),walktime(hr),transfer-
num,cable_traveltime(hr),taxi_travel time(hr)\n");
        printf("%f\t%f\t%f\t%f\t%f\t%f\t%f\n", f1, f2, f3, f4, f5, f6, f7);
        printf("The total cost:\n");
        printf("%f\n\n", total–cost);

}

for (kk = 0; kk < walk–n–seq; kk++) {
        total–cost = (float) (0);
        f1 = (float) (0);
        f2 = (float) (0);
        f3 = (float) (0);
        f4 = (float) (0);
        f5 = (float) (0);
        f6 = (float) (0);
        f7 = (float) (0);
        for (j = 0;j < number+1; j++) {
                node1 = walk_fseq[kk][j];
                node2 = walk_fseq[kk][j+1];
                f4 = f4 + cost[node1][node2][10];
        }
        total–cost = a[0]*f1 + a[1]*f2 + a[2]*f3 + a[3]*f4 + a[4]*f5 + a[5]*f6 + a[6]*f7;
```

```
        printf("The walk sequence:\n");
        for (k=0; k < number+1; k++) printf ("%d ", walk_fseq[kk][k]);
        printf("%d\n", walk_fseq[kk][number+1]);
        printf("cost($),transit_traveltime(hr),waitime(hr),walktime(hr),transfer-
num,cable_traveltime(hr),taxi_travel time(hr)\n");
        printf("%f\t%f\t%f\t%f\t%f\t%f\t%f\n", f1, f2, f3, f4, f5, f6, f7);
        printf("The total cost:\n");
        printf("%f\n\n", total-cost);

]

]
```

Code-opt.cpp

```
/*      Filename:               opt.cpp
        Author:         Jiayu Chen
        Date Created:   10-05-97
        Date Updated:   10-05-97

        Purpose: optimization -- choose the sequence with the minimum cost
        Input:
        output:

        Variables:
*/

void opt(float *a, int "route-lst, int *route_2nd, int *first-route-mode, int "second-route-mode)
{
        extern float cost[25][25][11];
        extern unsigned char transit_fseq[ROWS][COLS+2];
        extern unsigned char taxi_fseq[ROWS][COLS+2];
        extern unsigned char walk_fseq[ROWS][COLS+2];
        extern int number;
        extern int transit-n-seq;
        extern int taxi-n-seq;
```

```
extern int walk–n–seq;
int             j, k, keyl, key2, model, mode2, nodel, node2;
float    u[10000], min–u, sec–u;
float    f1, f2, f3, f4, f5, f6, f7;
FILE     *fp;

fp = fopen("opt_route_1st_2nd.temp000", "r+");


/* calculate the objective function value and obtain the optimal route in transit–fseq */
for (k = 0; k < transit–n–seq; k++) (
        u[k] = (float) (0);
        for (j = 0;j < number+1; j++) {
                nodel = transit_fseq[k][j];
                node2 = transit_fseq[k][j+1];
                f1 = cost[node1][node2][2];
                f2 = cost[node1][node2][3];
                f3 = cost[node1][node2][4];
                f4 = cost[node1][node2][5];
                f5 = cost[node1][node2][6];
                f6 = cost[node1][node2][7];
                u[k] = u[k] + a[0]*f1 + a[1]*f2 + a[2]*f3 + a[3]*f4 + a[4]*f5 + a[5]*f6;
        }
}

min–u = u[0];
keyl = 0;
model = 1;
sec–u = (float)(1.0E+30);
key2 = 1;
mode2 = 1;

if(transit_n_seq >1) {
        for (k = 1; k < transit–n–seq; k++) {
                if (u[k] < min–u) {
                        sec–u = min–u;
                        key2 = keyl ;
```

```
                    mode2 = 1;
                    min–u = u[k];
                    keyl = k;
                    model = 1;
                }
                else {
                    if (u[k] == min–u) printf("There are more than one sequences having same cost.\n");
                    if ((u[k] > min–u) && (u[k] < sec_u)) {
                        sec–u = u[k];
                        key2 = k;
                        mode2 = 1;
                    }
                }
            }
        }

/* calculate the objective function value and obtain the optimal route in taxi_fseq */
        for (k = 0; k < taxi–n–seq; k++) {
            u[k] = (float) (0);
            for (j = 0;j < number+1 ;j++) {
                node1 = taxi_fseq[k][j];
                node2 = taxi_fseq[k][j+1];
                f1 = cost[node1][node2][8];
                f7 = cost[node1][node2][9];
                u[k] = u[k] + a[0]*f1 + a[6]*f7;
            }
        }

        for (k = 0; k < taxi–n–seq; k++) {
            if (u[k] < min–u) (
                sec–u = min–u;
                key2 = keyl;
                mode2 = mode1;
                min–u = u[k];
                keyl = k;
```

```
                    model = 2;
                }
                else (
                    if (u[k] == min–u) printf("There are more than one sequences having same
cost.\n");
                    if ((u[k] > min–u) && (u[k] < sec–u)) {
                        sec–u = u[k];
                        key2 = k;
                        mode2 = 2;
                    }
                }
            }

/* calculate the objective function value and obtain the optimal route in walk–fseq */
            for (k = 0; k < walk–n–seq; k++) {
                u[k] = (float) (0);
                for (j = 0;j < number+1;j++) {
                    nodel = walk_fseq[k][j];
                    node2 = walk_fseq[k][j+1];
                    f4 = cost[node1][node2][10];
                    u[k] = u[k] + a[3]*f4;
                }
            }

            for (k = 0; k < walk–n–seq; k++) {
                if (u[k] < min–u) {
                    sec–u = min–u;
                    key2 = key 1;
                    mode2 = model;
                    min–u = u[k];
                    key1 = k;
                    model = 3;
                }
                else {
                    if (u[k] == min–u) printf("There are more than one sequences having same
cost.\n");
                    if ((u[k] > min–u) && (u[k] < sec–u)) {
```

```
                                    sec–u = u[k];
                                    key2 = k;
                                    mode2 = 3;
                            }
                    I
            I


            printf("1st min cost: %f\n", min–u);
            printf("first_route_mode: %d\n", model);
            printf("2nd min cost: %f\n", sec–u);
            printf("second_route_mode: %d\n\n", mode2);

/* output opt-routes */
            fprintf(fp,"%d,%d\n", keyl, model);
            fprintf(fp,"%d,%d\n", key2, mode2);
            *route–1st = key 1;
            *route_2nd = key2;
            "first–route–mode= model;
            "second–route–mode= mode2;

            return;
}
```

Code-allseq.cpp

```
allseq(int n)
{
            extern unsigned char p[ROWS][COLS];
            int         j, row;
            int                     j1, j2, j3, j4, j5, j6, j7, j8;


switch (n) {
case 1:
```

```
        row = 0;
        j =0;
        p[0][0] = j+1;
        return 0;
case 2:
        row = -1;
        for (j1 = 0;j1 < n; j1++) {
                for (j2 = 0;j2 < n; j2++) (
                        if cj2 !=j1) {
                                row++;
                                p[row][0] = j1+1;
                                p[row][1] = j2+1;
                                }
                        }
                }
        return 0;
case 3:
        row = 0;
        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)) {
                                                p[row][0] = j1+1;
                                                p[row][1] = j2+1;
                                                p[row][2] = j3+1;
                                                row++;
                                                }
                                        }
                                }
                        }
                }
        return 0;
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++;
                                  p[row][0] = j1+1;
                                  p[row][1] = j2+1;
                                  p[row][2] = j3+1;
                                  p[row][3] = j4+1;
                                  }
                                }
                          }
                      }
                    }
                  }
                }
          return 0;
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++;
                       p[row][0] = j1+1;
                       p[row][1] = j2+1;
                       p[row][2] = j3+1;
                       p[row][3] = j4+1;
                       p[row][4] = j5+1;
                      }
                }
```

```
            }
           }
          }
         }
      }
        }
       }
      return 0;
case 6:
      row = -1;
       for cjl = 0;j1 < n; j1++) {
        for (j2 = 0;j2 < n; j2++) {
         if cj2 !=j1) {
              for cj3 = 0;j3 < n;j3++) {
               if ((j3!=j2) && (j3!=j1)) {
                for (j4 = 0;j4 < n; j4++) {
                 if ((j4 !=j3) &&(j4 !=j2) && cj4 !=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 !=j 1)){
                              row++;
                              p[row][0] =j1+1;
                              p[row][1] =j2+1;
                              p[row][2] =j3+1;
                              p[row][3] =j4+1;
                              p[row][4] =j5+1;
                              p[row][5] =j6+1;
                              }
                          }
                        }
                       }
                      }
                 }
                }
               }
              }
```

```
        }
      }
    return 0;
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 (i4 = 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++;
                              p[row][0] = j1+1;
                              p[row][1] = j2+1;
                              p[row][2] = j3+1;
                              p[row][3] = j4+1;
                              p[row][4] = j5+1;
                              p[row][5] = j6+1;
                              p[row][6] = j7+1;
                            }
                          }
                        }
                      }
                    }
                  }
                }
              }
            }
          }
        }
      }
    return 0;
```

```
            }
           }
          }
        return 0;
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 !=j 1)) {
                    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 !=j.5) && (j7 !=j4) && (j7 !=j3) && (j7 !=j2) && (j7 !=j 1)){
                         for (j8 = 0; j8< n; j8++) {
                          if ((j8 != j7)&& (j8 != j6) && (j8 != j5) && (j8 != j4) && (j8 != J3) && (j8 != j2)
&& (j8 != j1)) {
                                row++;
                                p[row][0] = j1+1;
                                p[row][1] = j2+1;
                                p[row][2] = j3+1;
                                p[row][3] = j4+1;
                                p[row][4] = j5+1;
                                p[row][5] = j6+1;
                                p[row][6] = j7+1;
                                p[row][7] = j8+1;
                                }
                           }
                          }
                         }
                        }
                       }
```

```
                    }
                   }
                  }
                }
               }
              }
             }
            }
           }
          return 0;
 default:
          return 0;
          }
 }
```