INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

A CONCURRENT ARCHITECTURE FOR A TRAVEL PLANNING APPLICATION

ERIC SMITH

A THESIS

IN

THE DEPARTMENT

OF

COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE AT

CONCORDIA UNIVERSITY

MONTRÉAL, QUÉBEC, CANADA

OCTOBER 1999

Abstract

A Concurrent Architecture for a Travel Planning Application

Eric Smith

In this thesis, we present a concurrent architecture for a travel planner application. The main idea behind this architecture is the division of the travel planning problem into task and subtask objects that solve distinct aspects of the problem in parallel. These objects collaborate with one another, eliminating non-optimal solutions and producing a coherent solution consisting of partial solutions from the subtasks. Groups of these objects solve different optimization problems within the travel planning problem. The architecture is similar to that of a production system, particularly a blackboard architecture, except that we insist on a distributed control system and use procedural tasks and subtasks rather than declarative knowledge sources. Unlike blackboard systems, the solutions that are created are distributed amongst the objects rather than centralized, so some communication between objects is necessary. We discuss our Java^{TM} implementation of the travel planner and some sample results of the system. The travel plans output are sometimes not good approximations of the optimal solution, because a bounded, breadth-first search strategy is used. The system also loses on performance because of additional overhead incurred by object collaboration. On the other hand, our architecture offers potential performance enhancements achievable through concurrency as well as good design principles such as design for change, separation of concerns, and code reuse. We also demonstrate how our architecture could be used to solve other optimization problems. We conclude that our architecture could be used as a basis for a more efficient travel planner implementation.

Acknowledgments

I first wish to thank Dr. Peter Grogono. It was your idea of "collaborating objects" that inspired this research. Thank you for giving me the freedom to explore (or perhaps wander is the better word) at the beginning while, at the same time, providing the guidance I needed throughout this project. Thank you for your suggestions for the design, the focus of this thesis, and for your comments on the drafts. Where I decided to follow your advice, I succeeded. I failed where I did not heed it. I want to thank my family for enabling me to reach this point of my studies by nurturing me at the beginning. Thanks to my friend Nina Geise for your encouragement throughout this work, and for putting a $\smile$ in my heart (and my e-mail) when there was a $\frown$ there (2 Cor. 9:8). Thank you Lord Jesus for being my forever friend and giving me the ability to complete this work. I love you Jesus.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivating Factors of this Research

This research was partly inspired by Dr. Peter Grogono's research project on "Systems of Collaborating Objects." The goal of the project is to examine the behavior of software systems comprised of many objects that perform the same task in specialized ways. There is a default object which first tries to complete the task on its own, and when the system identifies that the task is too complex for the default object to manage, control is passed on to other more specialized objects. The specialized objects can collaborate with each other in some way as they each work on the task, helping each other progress. It is assumed that each specialized object runs concurrently with the others. The first object that completes the task preempts the others. The idea of "collaborating objects" can be seen as a concurrent problem solving paradigm.

A second motivating factor of this research was to develop an alternative, object-oriented architecture to the blackboard architecture used in the field of Artificial Intelligence (AI). Blackboard is a popular problem solving paradigm. However, the blackboard architecture is not object-oriented in the sense that the data (solutions written on the blackboard data structure) is not encapsulated with the procedures that operate on it. The data is publicly visible to all knowledge sources, so care

must be taken to preserve its consistency. Although the blackboard model is expressed as a concurrent architecture, blackboard implementations have not been able to make use of concurrency to the degree hoped due to the blackboard bottleneck. We will overview the blackboard architecture in Chapter 2.

In this research, we set out to develop an architecture to solve a particular problem that encapsulates the partial solutions of the problem with the procedures that produce these solutions, and exploits any concurrency inherent in the problem to the greatest degree possible. The encapsulation of the data with the procedures provides a greater measure of security that the partial solutions are not incorrectly accessed and modified. Exploiting concurrency allows for potential improvements in software performance when unrelated parts of the software can run independently on different connected processors. This thesis demonstrates that collaborating objects, running concurrently, can effectively solve a multi-dimensional optimization problem.

1.2 Summary of Research

As a starting point, I decided to design a travel planning software application to solve a general travel planning problem (TPP) by dividing the software into tasks and subtasks that could work on separate parts of the problem in parallel and collaborate with each other in some way to produce the required result. In this thesis, we discuss the architecture of our Travel Information Planning System (TIPS) software. We first discuss problem solving in general and other related problem solving techniques, contrasting them to the TIPS architecture. The TIPS design incorporates some elements of "collaborating objects," such as having concurrent objects work on the problem while collaborating with each other. These objects do not specialize in solving different instances of the problem, however, but rather, solve a different subset of the problem, applying different optimization policies.

We discuss our solution to the TPP and implementation issues that arose during the development of the software. We also examine the behavior of TIPS in terms of the kinds of solutions that it

produces. We then present the advantages and disadvantages of using our architecture, including for solving similar optimization problems. Finally, we propose improvements to the design. We claim that the TIPS design exploits concurrency inherent in the problem as compared to serial approaches for solving the TPP. Also, the design facilitates code reuse, design for change, and allows the designer to separate concerns easily.

1.3 Definition of the Travel Planning Problem

We first define the TPP that must be solved by the travel planning application as follows:

Given:

- a traveller's departure city,
- a set of destination cities that the traveller wishes to visit,
- a travel time window for the entire trip,
- the number of travellers making this trip,
- a spending limit for the trip,
- a set of other requirements and preferences for the trip,
- a set of optimization strategies to employ in the search, and
- a number, n, specifying the maximum number of travel plans to be presented to the traveller(s),

determine no more than n travel plans for a trip starting from the departure city and visiting each destination city once before returning to the departure city for the number of travellers specified that falls within the travel time window and costs less than the spending limit, while satisfying the traveller's other requirements and preferences and employing the selected optimization strategies. A travel plan should include:

1. one means of transportation between each city travelled to,

2. one accommodation stay for each destination, and

3. one activity that the traveller(s) can participate in at each destination.

The user should be able to enter the travel constraints, requirements, and preferences in a window on the display and then start the search process for a travel plan. The user should also be able to stop the search at any point. Important events that occur during the search should be recorded in a log file and displayed on a panel of the screen. Travel plans should be displayed in separate windows as they are found so that the user can view them separately. Also, a user should be able to save the travel plan(s) to a file. The use case diagram of the travel planner application is shown in Figure 1.

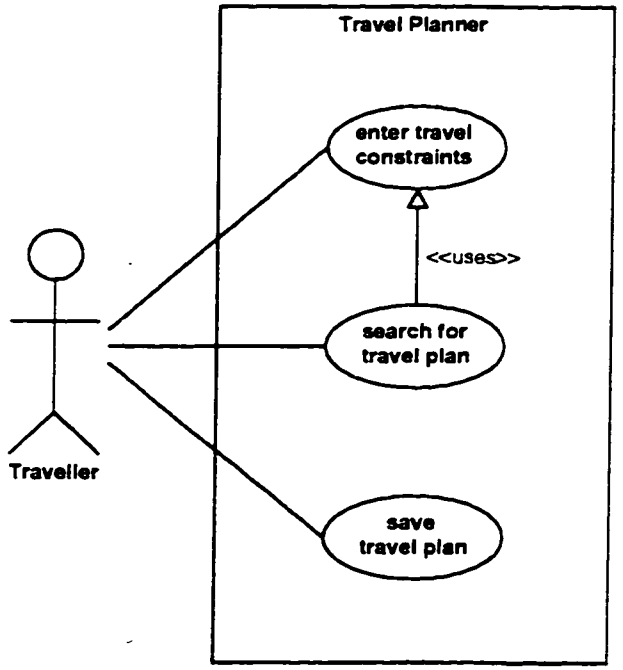

Figure 1: Use Case Diagram for the Travel Planner

In this problem, we are not concerned with giving the user the option of accepting or rejecting travel plans and making reservations if a travel plan is accepted. We are merely concerned with presenting suggested travel plans that meet the user's requirements and preferences and contain

enough information so that the user can find out more information for themselves, if desired, and take appropriate action. We have chosen to focus our research on the concurrent problem solving aspect of the TPP rather than on that of ticket and facility reservation, although we do check for the availability of tickets and facilities when searching for solutions.

1.4 Outline of the Travel Planner Architecture

We now provide an overview of the architecture of our software, which will be described at length in Chapter 3.

1.4.1 Dividing the Problem into Concurrent Tasks and Subtasks

Essentially, the TPP is solved by breaking the problem into task and subtask objects that each work on different aspects of the problem concurrently and collaborate with each other. There is a group of tasks and subtasks for each optimization strategy that is to be employed. Each group of task and subtask objects has a controller object (strategy controller) that combines the transportation, accommodation, and activity solutions produced by their task objects and reports one best resulting combination to the main application controller object, according to the optimization policy that is to be employed.

Each strategy is divided into three task objects that find partial solutions for their respective aspect of the problem, namely, transportation, accommodation, and activities. These tasks combine the solutions produced by their subtasks and return the best combined solutions to the strategy controller, according to their optimization strategy. The subtask objects produce specific transportation, accommodation, and activity solutions, optimizing according to the constraint of their strategy. The transportation subtask objects used in our design are: bus, plane, and train. The accommodation subtasks are: hotel, motel, and bed & breakfast. The activity subtasks are: dining, sightseeing, and events.

Note that strategy controller, task, and subtask objects work in parallel to each other. Each

solution produced by each task and subtask is a complete solution for its aspect of the problem that covers the entire duration of the trip and satisfies locally applicable constraints, requirements, and preferences.

1.4.2 Object Collaboration

The task and subtask objects collaborate with each other in two different ways. First, subtasks within a particular strategy collaborate to initially eliminate parts of solutions, e.g. flights, hotels, restaurants, etc., that may not match well with the parts of solutions belonging to other subtasks. For example, when the bus subtask finds a bus trip that it can use to build part of a solution, it first queries the other accommodation and activity subtasks to determine the degree to which this bus trip fits well, in terms of date, time, and location, with their solution parts. If it does not fit well, according to some acceptability rating factor, it is discarded.

In the second kind of collaboration, objects within a particular strategy collaborate to determine a common ordering of the destinations to be visited and the time intervals for each stay in order to produce coherent solutions. The first type of collaboration described above is non-essential to the problem solving process while the second one is essential. The former merely provides focus to the construction of solutions in the early stages of problem solving, while the latter ensures that each solution part matches the others.

1.4.3 Problem Solving

The subtask objects start by constructing solution parts that meet the locally applicable user constraints. Some of these solution parts are discarded because they do not match well with other subtask solution parts, in an attempt to direct the search to more promising solutions, as we have mentioned. Once all of the solution parts are created, the subtasks build subtask solutions using this set of solution parts. Due to memory constraints, an upper-bound is imposed on the number of solutions that may be constructed. The rationale of using this search strategy is discussed in Section 3.5.7. The subtasks then collaborate with each other to eliminate some of their solutions by

"agreeing" on the order to visit destination cities, based on their optimization strategy. The subtasks then send those solutions that contain that ordering to their parent task.

Each task then combines the subtask solutions it receives to make new solutions. Next, the transportation task determines the best stay time intervals for each destination according to its optimization strategy and informs the other tasks of these intervals. All of the tasks then send the solutions they possess that satisfy the best stay time intervals to the strategy controller. Finally, the strategy controller combines together the solutions it receives from its tasks into travel plans and sends the travel plan that best meets the strategy-specific optimization constraint to the application controller.

Chapter 2

Survey of Problem Solving Paradigms

As the architecture described in this thesis most closely resembles that of a blackboard system, we will start with a discussion of the more general production system problem solving paradigm contributed by the AI field of research, from which blackboard systems are derived. Later, we will describe blackboard systems in detail. Our architecture can be seen as an alternative to blackboard architectures. Blackboard systems were first developed for solving problems like voice recognition and perception problems in the AI domain. We will also contrast production systems and blackboard systems with the TIPS architecture.

Nilsson [7, p. 11–15] categorizes nine general AI application domain problems. The TPP that we propose here naturally falls into the category he calls, "combinatorial and scheduling problems." The TPP is an extension of the Travelling Salesperson Problem (TSP), which also belongs to this category of AI problems. Whereas, the TSP is concerned with finding the minimum distance trip starting at a departure city, visiting each destination city once, and returning to the departure city, solving the TPP involves scheduling the trip based on more than one criterion. Other criteria that must be considered, besides distance travelled, include the availability of accommodation and activities at particular times and places, and the total cost of the trip. So, the TPP is a multi-dimensional problem and cannot be tackled simply by finding the minimum cost path over the arcs of a graph connecting nodes representing the cities where the arcs are labelled with the cost of the path.

2.1 Problem Solving Using Production Systems

Nilsson [7, p. 17] points out that most AI problems are solved by decomposing a system into data, operations, and control components. This is, in essence, the structure of a production system. Production systems are appropriate for solving the TPP. We will discuss how our TIPS implementation can be represented as a production system in Section 2.1.2.

A production system consists of a global database, a set of production rules, and a control system. The database represents the current state of problem solving at a given time in some form. It is the data structure to which the production rules are applied. Production rules are transformations that can be applied to the database when an associated precondition is satisfied by the database in order to bring the problem state one step closer to the solution[1] or goal state. A production rule is said to be applicable if the global database satisfies the precondition of the production rule. So, a production rule can be stated as a logical antecedent–consequent formula or an IF–THEN statement. When an applicable production rule is applied to the global database, it is said to have been fired.

The control system determines the sequence in which the production rules are fired, applying them until the goal solution is obtained. The function of the control system is that of searching–applying different production rules by trial and error until the solution is found. It is often the case with AI problems that the control system must decide between more than one production rule to apply next at any given stage of problem solving. Firebaugh [5] presents the steps of a general control system in production rule selection, which we summarize here:

1. If the global database satisfies the goal state, stop. The solution has been found. Otherwise, go on to the next step.

2. Determine all applicable production rules based on the current state of the global database.

3. Ignore all applicable production rules that would add no new knowledge about the solution to the global database if they were fired.

[1] A problem may have more than one equally acceptable solution. This is often the case with optimization problems.

4. If there are no applicable production rules left to consider, stop. The search fails. Otherwise, continue with the next step.

5. Choose the best production rule to apply from the remaining production rule(s) according to some criteria and fire it.

6. Go back to step 1.

In step 5, there are several methods that can be used to determine the best rule to apply when there is more than one to choose from. Shirai [9] lists some conflict resolution methods for selecting rules, some of which we include here:

- Assign priorities to the production rules and fire the one with the highest priority.
- Fire the rule with the most strict antecedent.
- Fire the rule that was last used.
- Estimate the execution time of each production rule on the database and fire the one with the lowest expected execution time.
- Fire all applicable rules one after the other.

We have added the last conflict resolution technique to this list, because it may be sufficient for cases in which the number of rules to apply is not so large. In our travel planner implementation, we fire all applicable rules in addition to using a kind of heuristic, which we will describe in Section 2.1.1.3.

In summary, problem solving using production systems is comprised of three things: finding an appropriate representation of the problem in the global database, determining a non-redundant set of production rules that can be applied to the global database, and implementing an efficient control system for applying those production rules. The issues of knowledge representation and selection of appropriate production rules are specific to the problem at hand. We will discuss these issues in relation to the TPP in Section 2.1.2. Now, we turn our focus to an integral part of production systems: search strategies commonly employed by the control system in finding solutions.

2.1.1 Search Strategies for Production Systems

Nilsson [7, p. 21–27] describes three distinct types of control or search strategies that can be used by production systems: irrevocable, backtracking, and graph search. The latter two strategies can be more broadly classified as tentative strategies. Irrevocable search strategies, such as hill-climbing, apply production rules one after the other without providing for the possibility of reverting problem solving to a previously applied production rule and continuing from there. Tentative search strategies do allow for this possibility.

Two tentative search strategies, backtracking and graph search, are similar in that while exploring the solution space or set of all possible solutions for the goal solution, a graph or tree data structure is created that "remembers" the sequence of production rules applied. The results of the rule applications at each stage (state of the global database) are stored in the nodes of the data structure and the production rules applied to transform the database from one state to another are represented by the arcs connecting the nodes. With this information, the problem solving process can revert to a previous state. The difference between these two tentative search strategies is that a backtracking search only maintains information on the current solution being explored, while graph search maintains a tree structure of all solutions that were constructed during the search process. Irrevocable search strategies do not need to maintain this information. We now discuss concrete search strategies for production systems.

2.1.1.1 Uninformed Search Strategies

The first kind of search strategies that we outline here are uninformed search strategies. They are uninformed in the sense that production rules are applied in an arbitrary order. No knowledge of the problem is used in order to direct the search to the solution more quickly. As such, it is usually the case that more applications of production rules are needed in order to obtain the goal solution than if some knowledge of the problem were used to direct the selection of production rules.

The strategies discussed in this section are all tentative search strategies. As we have mentioned

above, we can represent them as tree structures. The nodes of the tree represent the states of the global database at each stage in the search and a directed edge from node n_1 to node n_2 that has an associated label R_1 represents the application of production rule R_1 to the global database in state n_1 that results in a global database in state n_2. The root node of the tree is the initial state of the global database and the leaf nodes of the tree represent either states that can be modified further by applying other production rules, unpromising states that cannot lead to a goal solution, or goal solutions for the problem being solved. For each of the uninformed search strategies that we discuss here, the search tree is built, starting at the root node, and creating children of the current node by applying all applicable production rules. We call this process "expanding the nodes," and it is repeated for all subsequently generated child nodes until a goal solution is found or all applicable production rules have been fired for all child nodes without finding a solution.

2.1.1.1.1 Backtracking Search With the backtracking search technique, the nodes are expanded one at a time and the deepest (highest level) nodes in the search tree are expanded first. For example, if during a particular stage of the search, nodes at level l and $l+1$ of the tree are not yet expanded, the nodes at level $l+1$ will be expanded first, where the root node is at the lowest level and the leaf nodes are at the highest level.

As we have already stated, the backtracking search is a tentative search strategy. When the current node that is being examined cannot be expanded any further, i.e. no production rule can be applied that will result in creating a new node that is not found in the path from the root to the leaf node, or is unpromising in that it cannot possibly lead to a goal solution, the search strategy backtracks to the next deepest-level node in the tree and expands it.

The control system only keeps track of the nodes in the path from the root to the current node of the search tree at any time. Nodes that are left because of backtracking are not kept in working memory. Thus, backtracking search conserves on memory space used. However, it is inefficient in terms of computational power required. In the worst case, it may require that all nodes be expanded by firing all possible combinations of applicable production rules before a solution is found.

2.1.1.1.2 Depth-first Search The depth-first search is a graph search strategy. A depth-first search involves expanding the deepest nodes in the tree one at a time, as in the backtracking search, however, unlike the backtracking search, none of the nodes are discarded when they prove to be unpromising. The nodes that are left because they are unpromising are kept in working memory. Depth-first search suffers both from memory storage and computational power inefficiency, because, it can do not better than backtracking search, and it keeps track of all the nodes traversed in the search tree.

2.1.1.1.3 Breadth-first Search Breadth-first search is also a graph search strategy. As in the depth-first strategy, tree nodes are expanded one at a time and are not discarded when they prove to lead to dead ends. However, all of the nodes at the same level of the tree are expanded first, before moving on to expand nodes at a deeper level in the tree. As is the case with depth-first search, breadth-first search is inefficient in terms of both memory storage and computational power required. It does, however, provide a broader scan of the solution space early on in the search process than does depth-first search. A breadth-first search strategy was employed in our TIPS architecture.

2.1.1.2 Heuristic Search Strategies

The main problem with the uninformed search strategies described in Section 2.1.1.1 is that they are inefficient due to the number of production rules that must be applied in order to find a goal solution. In the worst case, all possible combinations of rules must be fired in order to find a goal solution. This may be unacceptable for problems that have large solution spaces.

Heuristic search strategies use some information about the problem in order to selectively apply those production rules that are most likely to lead to a goal solution. This helps reduce the number of production rules that are applied, in general, and so, improves the computational efficiency of the search. In this section, we outline some heuristic search methods.

2.1.1.2.1 Hill-climbing As mentioned above, hill-climbing is an irrevocable search strategy. Production rules are applied sequentially and once a rule is applied, its effects are never undone. It is an heuristic search strategy, because at each step in the search process, the "best" production rule is selected to be applied based on some knowledge of the problem domain. In order to apply a hill-climbing search strategy, it is necessary to determine a hill-climbing function over the domain of the global database. One production rule at a time is selected to act on the database based on the value of the hill-climbing function applied to the database state that would result after applying this production rule. The production rule that yields the highest value of the hill-climbing function on the resulting database state is selected for firing.

The search process stops when either no applicable production rule exists such that, when applied, it would yield a hill-climbing function value greater than or equal to the last value computed (terminating in failure), or a goal solution is found (terminating successfully). One problem with the hill-climbing approach is that it is not guaranteed to find the optimal solution, because the search process may lead to a sub-optimal solution that yields a local maximum hill-climbing function value, and miss the solution that yields the global maximum value of the hill-climbing function.

2.1.1.2.2 Heuristic Evaluation Functions Heuristic evaluation functions can be used to complement the graph search strategies we have mentioned in Section 2.1.1.1 by directing the search to a solution more quickly. At each stage of the search, the evaluation function is computed on each unexpanded node database state and the unexpanded nodes are sorted for expansion based on this statistic.

The difficulty with this approach is to find a suitable evaluation function. One pitfall is to select a function that always leads the search to unpromising solutions at each step of the search. Nilsson [7, p. 88] lists three factors to consider when choosing an heuristic evaluation function:

1. the average length of the path in the search tree from the root node to the solution node obtained when using the evaluation function,

2. the average number of nodes that are expanded to find the solution node when using the

evaluation function, and

3. the computing power required to compute the evaluation function on a node.

The first factor listed above is of concern if it is important to find a goal solution that can be obtained by applying the fewest number of production rules. For the TPP, as we will see in Section 2.1.2, the number of production rules that are applied to obtain a goal solution is fixed, so this factor is of no importance to us. The other two factors are of importance, however, in relation to the efficiency of the search algorithm.

2.1.1.3 Other Search Algorithms

Nilsson [7, p. 88–91] describes other production system search algorithms that can be used as alternatives to the ones mentioned above. One alternative to the search methods mentioned is a staged graph search in which an heuristic evaluation function is used to reduce the size of the search tree when the remaining available memory space is limited. At certain stages in the search, the heuristic evaluation function is computed on the unexpanded nodes and those nodes that yield the lowest value are removed from the tree entirely. This does limit the search and is useful for dealing with limited memory space, but some goal solutions may no longer be considered as candidates in the search as a result of applying this method.

The limitation of successors method is much the same as the staged search, except that the heuristic evaluation function is used to eliminate unexpanded child nodes after every node expansion. Although this may also lead to the elimination of goal solutions, it may be acceptable to do this, depending on the application. This method may be used to solve a problem where an approximation to the goal solution is acceptable. We used the limitation of successors method in the implementation of TIPS.

2.1.2 Solving the Travel Planning Problem Using a Production System

We will now describe the elements of production systems present in our TIPS architecture for solving the TPP. These elements can be used to build other, not necessarily concurrent, travel planner implementations.

The initial subtask solutions that are constructed by TIPS are built using a production system. We represent each solution as a list of trips, stays, or activities, depending on which type of solution we are talking about. Generically, we will call these trips, stays, and activities sub-solutions. Each subtask initially constructs a set of sub-solutions, such as airline flights in the case of the plane subtask, that satisfy the user's travel constraints. In order to conserve memory space, we immediately delete sub-solutions that do not match well with other subtask sub-solutions. This is equivalent to the limitation of successors method described in Section 2.1.1.3. Each subtask has its own database that consists of the set of sub-solutions and complete solutions that are constructed. Each database is not global, because only the subtask that owns it can read and write to it.

All possible solutions that satisfy the user's travel constraints are built using a breadth-first search as described in Section 2.1.1.1.3. Only a fixed maximum number of solutions can be built, however, because of memory constraints. Thus, it is a bounded breadth-first search. The best solutions are then determined after collaborating with the other subtasks and are sent to the parent task. In Section 3.5.7, we provide the rationale for using a breadth-first search strategy in combination with the limitation of successors method for finding subtask solutions.

The production rules for creating the initial solutions differ for each subtask. In the case of transportation subtasks, we first apply the following production rule to each trip t in the set of sub-solutions created to add the first trip leg to the solutions:

IF departureCity(t) = *depCity* THEN add(cons(*null*, t))

where

- *depCity* represents the departure city of travels,
- departureCity is a function that returns the departure city of trip t,

- cons is a function that constructs a new solution by appending sub-solution t to the end of a solution (in this case the *null* or empty solution), and
- add is a function that adds a solution to the database.

Then, for each solution s and trip t constructed in the database, we apply the following production rules in a breadth-first order until either we have applied them to all combinations of s and t, or we have created the maximum number of solutions that we can create in the database:

1. IF (numTrips(s) $<$ *numDests*) $\wedge$ (destinationCity(t) $\neq$ *depCity*) $\wedge$
   (departureCity(t) = lastCity(s)) $\wedge$ (arrivalDate(lastTrip(s)) $<$ departureDate(t))
   THEN add(cons(s, t))
2. IF (numTrips(s) = *numDests*) $\wedge$ (destinationCity(t) = *depCity*) $\wedge$
   (departureCity(t) = lastCity(s)) $\wedge$ (arrivalDate(lastTrip(s)) $<$ departureDate(t))
   THEN add(cons(s, t))

where

- *numDests* represents the number of destination cities selected by the user,
- numTrips is a function that returns the number of trips in a solution s,
- destinationCity is a function that returns the destination city of a trip t,
- lastCity is a function that returns the destination city of the last trip listed in a solution s,
- lastTrip is a function that returns the last trip in a solution s, and
- departureDate and arrivalDate are functions that return the departure and arrival dates of a trip t, respectively.

The first rule creates the middle legs of the trip solution and the second rule creates the final leg of the trip solution. Note that for each solution s, only one of the two preceding production rules is

applicable, because the first conjuncts of each rule are disjoint. After combining a solution s with each trip t, we remove s from our database, as it is no longer needed.

In the case of the accommodation and activity subtasks, we initially apply the following production rule for each sub-solution u in the set of sub-solutions to add the accommodation or activity for the first destination to the solutions:

IF cons(*null*, u) $\notin$ *solutions* THEN add(cons(*null*, u))

where *solutions* represents the set of solutions created.

Next, for each sub-solution u and solution s in the database, we apply the following production rule until either we have exhausted all combinations of u and s or we have created the maximum number of solutions that we can create in the database:

IF $\neg$ contains(s, location(u)) THEN add(cons(s, u))

where location is a function that returns the city in which sub-solution u is located, and contains is a function that returns *true* if solution s contains a sub-solution that is located in the given city. Otherwise, it returns *false*. This rule adds a sub-solution u to a solution s if there is not already a sub-solution for the same destination city present in the solution. After combining a solution s with each sub-solution u, we remove s from our database, as it is no longer needed.

2.2 Overview of the Blackboard Architecture

Blackboard systems, which are instances of production systems, have met with some success in the field of AI. The blackboard architecture is useful for systems in which there is no predefined algorithm to obtain a complete solution to the problem. It is categorized as an opportunistic problem solving strategy. That is, it employs a data-directed control regime. The behavior of the system is determined at each step by examining the state of the data, or solution that is being built to solve the problem.

Buschmann et al. [2] presents an object-oriented description of the blackboard architecture. Figure 2 on page 19 shows a class diagram for this architecture. The data, in the form of partial and

complete solutions, is read from and written to a blackboard object by knowledge source objects. The blackboard may also store the solution to the control problem, i.e. the order in which knowledge sources should be selected to execute. Each knowledge source object tries to solve a certain aspect of the problem. The control object orders the execution sequence of knowledge sources.

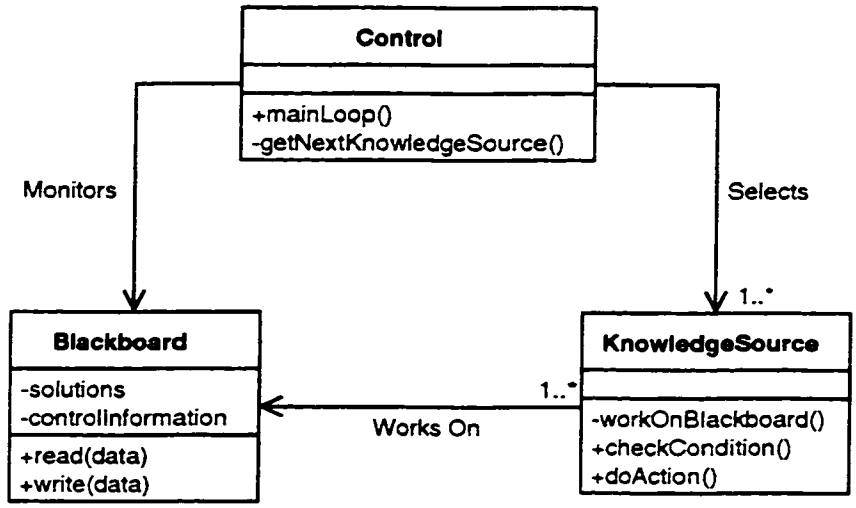

Figure 2: Class Diagram of the Blackboard Architectural Style

At each step of the execution of the blackboard system, the control object first reads any necessary control information from the blackboard. The control object then requests each knowledge source to examine the current state of the solution on the blackboard in order to determine if it can add anything to the solution. The control object chooses the best knowledge source to execute based on these results and lets it run. The selected knowledge source reads the data on the blackboard and modifies the solution in some way so as to bring the solution one step closer to the goal solution. This process is repeated until either a goal solution is found, or no knowledge source can add anything to the solution. In the latter case, the system terminates without finding a solution. The scenario for one execution step of a simple, sequential blackboard system that we have described above is depicted by the collaboration diagram in Figure 3 on page 20.

Blackboard systems are an analogy of human problem solving where human experts (knowledge sources) work on solving a problem together. The problem is solved on a real blackboard that each expert can look at and go up to, and modify the solution on it, bringing the group one step closer to

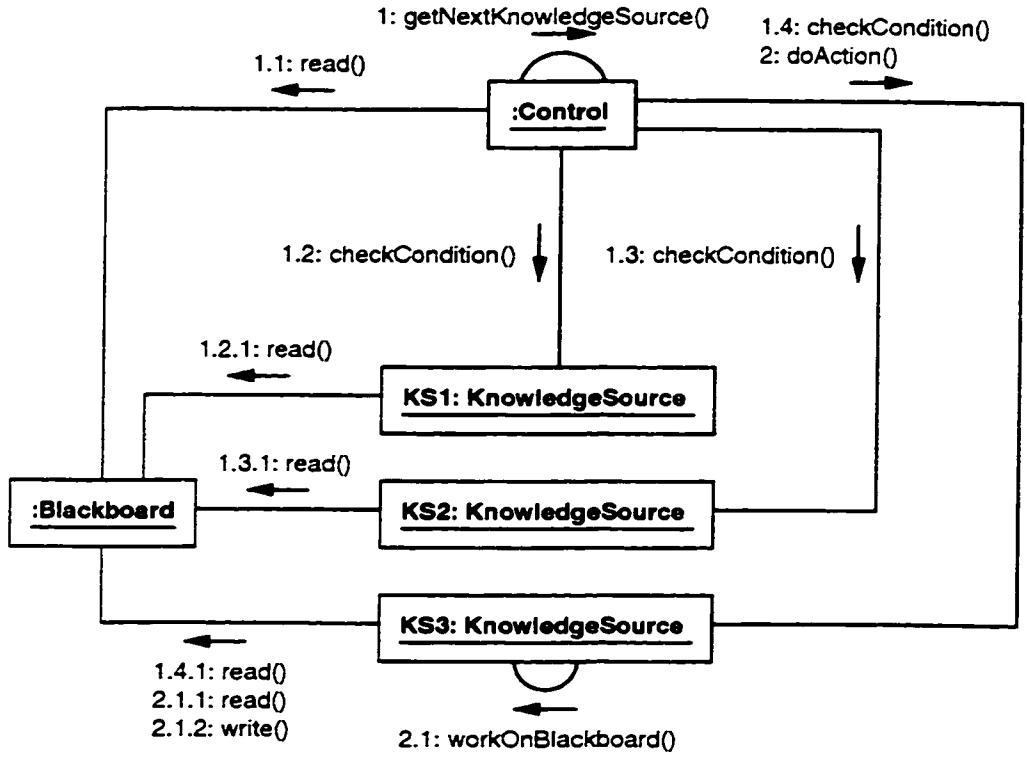

Figure 3: Collaboration Diagram for One Step of a Blackboard System

the goal solution.

2.2.1 Blackboard Data Structure

The blackboard is a data structure that represents the problem domain and it may be partitioned hierarchically. Knowledge sources work on different levels of the hierarchy in which the solution is more abstract or concrete. The blackboard enables knowledge sources to communicate amongst themselves indirectly and cooperate to solve large and complex problems. Any given knowledge source knows neither which other knowledge source wrote a particular entry on the blackboard nor which knowledge source will make use of the entries it places on the blackboard.

2.2.2 Knowledge Sources

The ideal characteristics of knowledge sources as specified by Nii et al. [4, p. 481] are that they should have access to the entire blackboard, they can each write their contributions to the blackboard

asynchronously without hampering the progress of other knowledge sources, and they will not fail due to attempting to write on the blackboard. These properties of knowledge sources characterize them as concurrent and contention-free.

The knowledge encapsulated in the knowledge sources is declarative rather than procedural. By this, we mean that the actions and knowledge of when it is appropriate to take these actions which is contained in each knowledge source is decoupled from the data upon which the knowledge source acts on the blackboard. Procedural knowledge is the data and the actions that are to be performed on that data encapsulated together. The knowledge source "knowledge" is usually in the form of production rules that we have described above (see Section 2.1).

2.2.3 Control

Although the blackboard architecture that we have described so far is serial, parallel implementations which align themselves more closely with the ideal of concurrently executing knowledge sources laid out by Nii have been developed. One component of blackboard systems that can be modified to obtain more concurrent blackboard systems is the control component. There are two main types of control in blackboard architectures: centralized and distributed.

If control of the execution of the knowledge sources is centralized, then we have a separate control component as depicted in Figure 2 on page 19 which directs problem solving by determining which knowledge source is most appropriate to run next based on the current state of the blackboard. This design decision lends itself more easily to a serial implementation of the system.

If control is distributed, each knowledge source has an associated controller that individually determines when it is appropriate to execute its knowledge source. In this case, the knowledge source controllers each know when it is appropriate for their knowledge sources to execute. The choice of having distributed control facilitates the development of a more concurrent blackboard system.

2.3 Concurrent Blackboard Architectures

Nii et al. [4, p. 480] lists four potential sources of concurrency in the blackboard architecture:

1. running knowledge sources concurrently,
2. pipelining the execution of the knowledge sources,
3. allowing the solutions on the blackboard to be accessed concurrently, and
4. permitting the control system(s) to execute concurrently with the problem solving.

Concurrent knowledge source execution (the first concurrency source listed above) is the most natural source of concurrency of those listed above in terms of most accurately following the blackboard metaphor. It is also the source of concurrency used for subtasks and tasks in our TIPS architecture. For these reasons, we will focus exclusively on this source of concurrency in blackboard systems.

Corkill [3] presents three design alternatives for concurrent blackboard systems, namely the shared-memory blackboard (SMBB), distributed blackboard (DBB), and blackboard server (BBS) approach. These architectures are all based on concurrent knowledge source execution. We will outline these approaches since they most closely resemble the work we have done with TIPS.

In each approach, the control component of the blackboard system is distributed rather than centralized. We assume that there is one controller for each processor that schedules one knowledge source to run at any given time. Although it may appear from the following diagrams that a given knowledge source must run on a fixed processor, this need not be the case. The knowledge sources may interact with the blackboard via the controller or directly, even though this is not explicitly shown in the diagrams that follow.

2.3.1 Shared-memory Blackboard Approach

Figure 4 on page 23 demonstrates the interaction between the concurrently executing knowledge sources and the blackboard in the SMBB approach. This approach can only be used on a shared-memory multi-processor, because the blackboard is shared. Each knowledge source interacts with

the same blackboard instance. Thus, access to the blackboard must somehow be synchronized when the same part of the blackboard is accessed by two different knowledge sources.

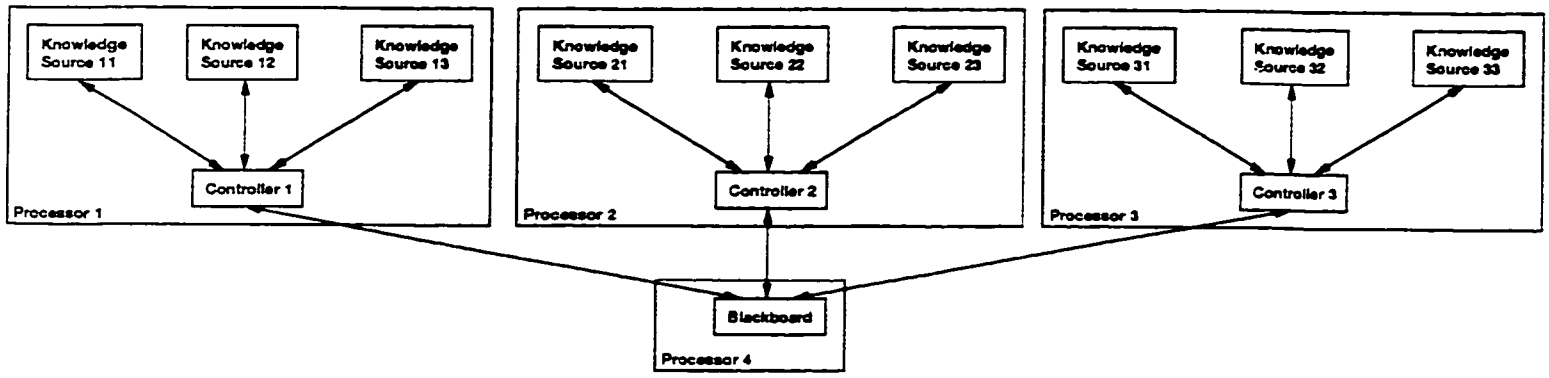

Figure 4: The Shared-memory Blackboard Approach

2.3.2 Distributed Blackboard Approach

In Figure 5 we show the DBB approach. Our use of the term distributed need not imply that the processors are not linked via a shared-memory, although they may be separate processors connected by a network. Here, the blackboard is partitioned with each processor maintaining responsibility for a partition. The partitions may be disjoint, partially overlapping, or completely overlapping. In order to maintain consistency of the data on overlapping blackboards, there must be some method of communication between the blackboards to update each other at the appropriate time.

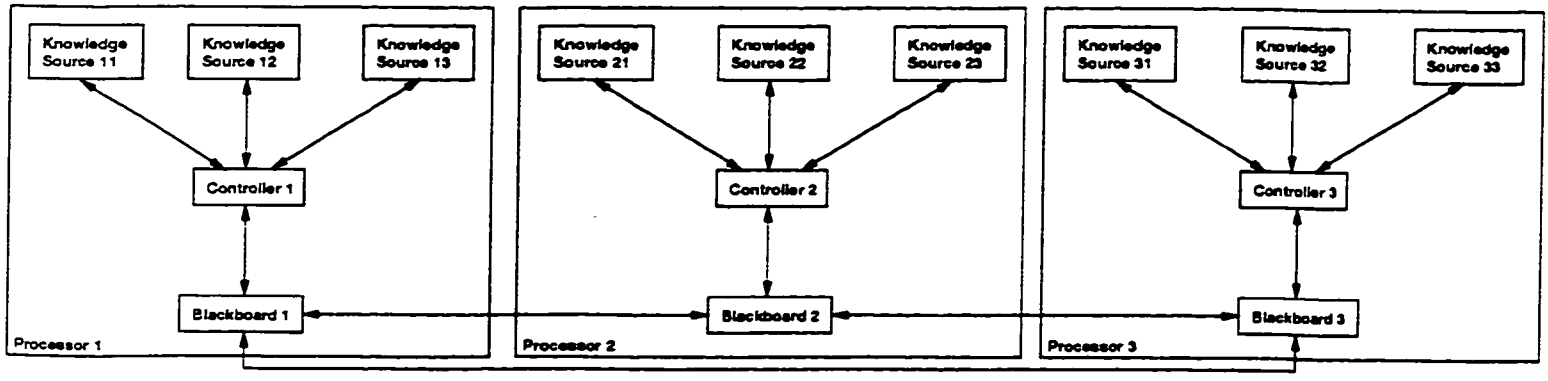

Figure 5: The Distributed Blackboard Approach

2.3.3 Blackboard Server Approach

Another alternative concurrent blackboard architecture is the BBS approach (see Figure 6). Here, there is one blackboard as in the SMBB approach, but the knowledge sources do not interact directly with it. All blackboard requests are funnelled through a blackboard server which we have chosen to be one of the knowledge source controllers in Figure 6. Again, the processors may be linked via a shared memory. In this approach, Controller 2 is responsible for maintaining the consistency of the data on the blackboard through synchronization, and accessing the blackboard can become a bottleneck.

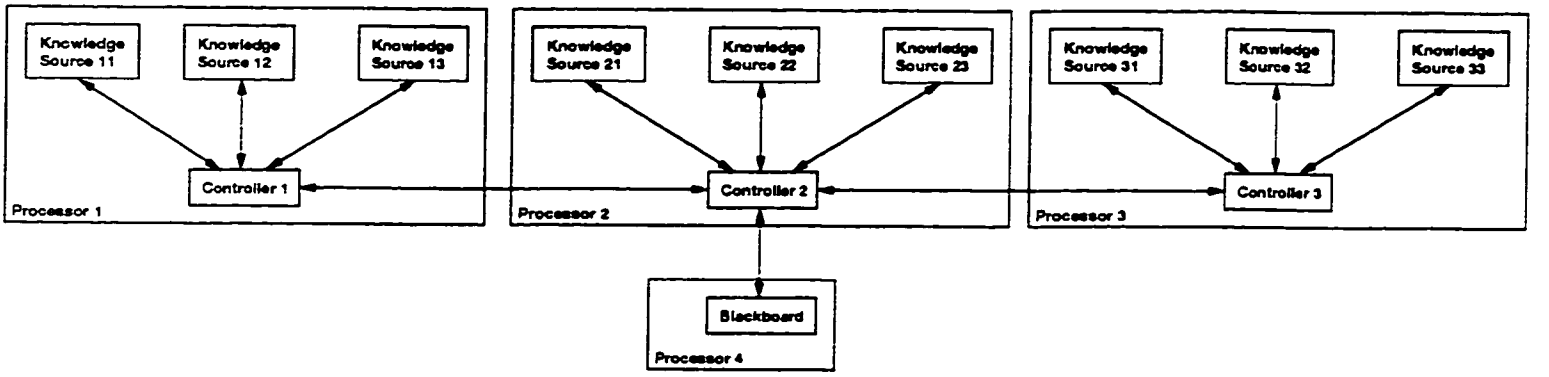

Figure 6: The Blackboard Server Approach

2.4 Travel Planner Architecture vs. Concurrent Blackboard Architectures

The TIPS architecture is similar to the general sequential blackboard architecture presented in Section 2.2 in that we have knowledge sources, which in our case are called tasks and subtasks, that have specialized knowledge of the problem we are attempting to solve. As we showed in Section 2.1.2, each task and subtask has production rules that are applied to the database in order to solve the problem, as is the case with blackboard systems.

The TIPS architecture also shares similarities with concurrent blackboard architectures that feature concurrently executing knowledge sources with distributed control such as those presented in

Section 2.3. Each task and subtask in TIPS has its own thread of control. Thus, the subtasks can execute their production rules on their local databases independently of each other, allowing the search to proceed more quickly than is possible in a serial implementation on a multi-processor.

The main difference between our architecture and the blackboard architecture is that there is no global blackboard data structure. Even in the case of the DBB approach, each blackboard is shared by all knowledge sources that execute on the same processor and data on one blackboard may need to be accessed or modified by knowledge sources on other processors. The TIPS data structure, on the other hand, is partitioned according to tasks and subtasks and only the owner of a partition may read and write to it. Another task or subtask may indirectly read it via the synchronized interface only.

Another important difference between the two architectures is that there is collaboration between the TIPS subtasks and tasks, whereas there is no direct communication between knowledge sources in the blackboard approach. Tasks and subtasks communicate with each other directly by sending query messages to each other. Knowledge sources communicate with each other by writing to and reading from the blackboard data structure only.

The knowledge that TIPS tasks and subtasks possess is procedural as compared to the declarative knowledge of blackboard knowledge sources. The tasks do apply production rules, but the actions of the tasks are intertwined with and dependent upon the data structures that they are working on. Blackboard systems, on the other hand, separate the representation of knowledge (blackboard data structure) from the operations that act on that data (knowledge sources). Knowledge source production rules modify data via the well-defined interface of the blackboard data structure.

The advantage of the TIPS architecture over the blackboard architecture is that the coherency of the local data structure is preserved and there is less danger of two tasks accidentally interleaving their access to the data and corrupting it. On the other hand, the disadvantage of choosing local data structures for solving the TPP is that tasks must at some time have access to each other's local data in order for the system to produce a coherent travel plan. Thus, some information about the local data must be provided to the other tasks via an inter-task query mechanism. This could potentially

lead to increased overhead due to the increased amount of inter-task communication required in order to solve the problem. In the worst case where each knowledge source must access some local data of all n knowledge sources, whereas with the blackboard system the query could involve only one procedure call to the global blackboard data structure, the TIPS architecture requires that each knowledge source call n procedures—one for each knowledge source. Thus, with our TIPS architecture, we wish to limit the amount of inter-task communication as much as possible without hindering our search for a solution to the problem.

Our design is appropriate for this application, because in the TPP, the data can be partitioned and the problem solved in such a way that a subtask or task need only modify the local data. A subtask or task may, during certain stages of the problem solving, only need have certain knowledge of the solutions local to other subtasks and tasks. Because of this reduced need for inter-task communication, the loss of efficiency due to synchronization of access to data is acceptable for relatively small problem instances. We will examine sample results and execution times of TIPS in Chapter 5.

Chapter 3

Detailed Design of the Travel Planner Application

3.1 Introduction

In this chapter, we describe the detailed design of the TIPS software which provides the user with a travel plan that meets the user's travelling constraints, requirements, and preferences. We first present an overview of the entire application architecture and then we focus exclusively on the design of the problem solving part of the application, as we are primarily interested in how we have solved the TPP in a concurrent fashion. Note that this design has been fully implemented in a working version of TIPS using the Java programming language [10]. All design diagrams presented in this chapter are drawn according to the Unified Modelling Language^{TM} (UML) standard notation [8].

3.2 Overview of Architecture

We now present an overview of the entire TIPS application architecture.

3.2.1 Layers

The application is organized into three layers, namely, the ProblemSolving, InputOutput, and DataStructures layers. The ProblemSolving layer is responsible for searching for and displaying the travel plan(s) that meet the user's constraints. The InputOutput layer is responsible for retrieving the travel data to be used in the search process. The DataStructures layer provides the other layers with user-defined data structures that can be used manipulate the data. The relationships between these layers is shown in Table 1. In the topology tables, a client/server relationship is denoted by "C/S" and a peer-to-peer relationship is denoted by "P-P."

	PS	I/O	D
ProblemSolving (PS)		C/S	C/S
InputOutput (I/O)			C/S
DataStructures (D)			

Table 1: Topology of Architecture Layers

3.2.2 Subsystems

The application layers are comprised of seven main subsystems: Controller, UserInterface, Task, Sorting, File, DateTime, and DataStructures. The Task subsystem is further sub-divided into the NonLocalTransportation, Accommodation, and Activities subsystems. The NonLocalTransportation subsystem is again sub-divided into the Bus, Plane, and Train subsystems. The Accommodation subsystem is further sub-divided into the Hotel, Motel, and BedBreakfast subsystems. The Activities subsystem is further sub-divided into the Dining, Sightseeing, and Events subsystems. The responsibility of each subsystem and the layer to which each one belongs is outlined in Table 2 on page 29. The relationships between the seven main subsystems are shown in Table 3 on page 30.

Subsystem	Layer	Responsibility
Controller	PS	Provides control of the problem solving activities at the system-wide and strategy-wide levels
UserInterface	PS	Retrieves travel constraints, requirements, and preferences from the user and displays suggested travel plans to the user
Task	PS	Determines solutions for different aspects of the TPP that match the user's travel constraints
NonLocalTransportation	PS	Determines the best-suited inter-city transportation solutions
Accommodation	PS	Determines the best-suited accommodation solutions
Activities	PS	Determines the best-suited activity solutions
Bus	PS	Finds acceptable bus transportation solutions
Plane	PS	Finds acceptable plane transportation solutions
Train	PS	Finds acceptable train transportation solutions
Hotel	PS	Finds acceptable hotel accommodation solutions
Motel	PS	Finds acceptable motel accommodation solutions
BedBreakfast	PS	Finds acceptable bed and breakfast accommodation solutions
Dining	PS	Finds acceptable solutions for restaurants to dine at
Sightseeing	PS	Finds acceptable solutions for sightseeing activities
Events	PS	Finds acceptable solutions for limited-time events
File	I/O	Returns travel data from files that can be used to construct travel plans
Sorting	D	Provides sorting and ranking routines for lists of data
DateTime	D	Provides data structures used to manipulate dates and times contained in travel data
DataStructures	D	Provides data structures used to manipulate travel planning solutions

Table 2: Subsystem Layers and Responsibilities

	C	UI	T	S	F	DT	DS
Controller (C)		P-P	P-P	C/S	C/S	C/S	C/S
UserInterface (UI)			C/S		C/S	C/S	C/S
Task (T)				C/S	C/S	C/S	C/S
Sorting (S)							C/S
File (F)							C/S
DateTime (DT)							
DataStructures (DS)							

Table 3: Topology of Subsystems

3.2.3 Classes

The responsibilities of each class can be found in the Data Dictionary (see Appendix A). The entries are ordered alphabetically, first by subsystem name, then by class and interface name, and then by operation and attribute name. The type of each entry (class, interface, operation, or attribute) is indicated in italics between parentheses followed by a short description of the entry. The name of each entry is set in boldface, consisting of the subsystem name, class or interface name, and, optionally, operation or attribute name, in that order, each separated by double colons.

Figure 7 on page 31 shows all of the classes and the subsystems to which they belong. Inter-subsystem class dependencies, which are uses dependencies, as well as dependencies between subsystems, which are imports dependencies, are shown. In the remaining part of this chapter, we will examine the relationships between the classes contained in the two most important subsystems in terms of problem solving: the Task and Controller subsystems.

3.3 Object Model

We now present the object model of the Task and Control subsystem objects which are the important subsystems in terms of the problem solving aspect of the software system.

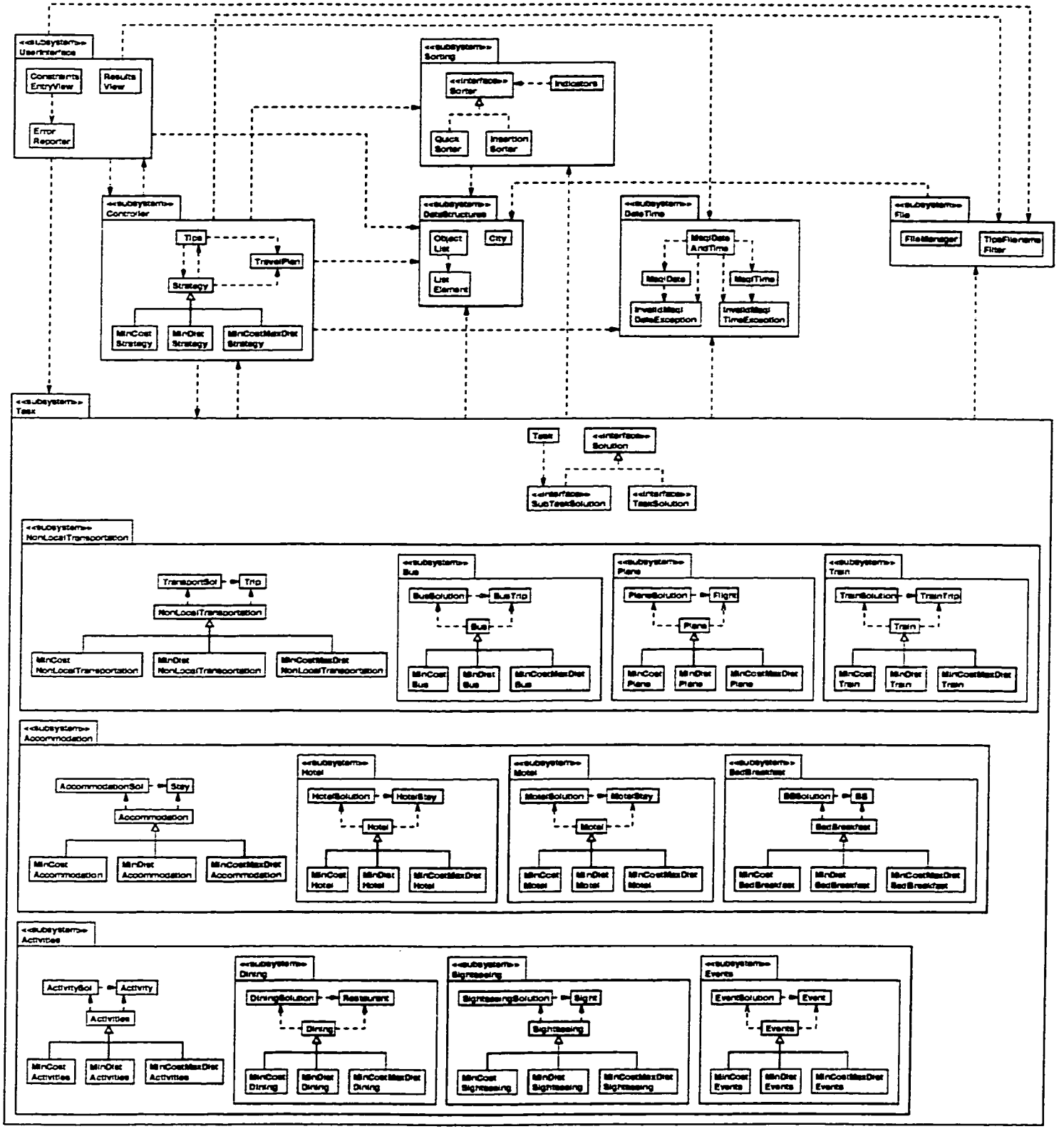

Figure 7: All Subsystems and Classes

3.3.1 Inheritance

Figure 8 is a class diagram of the TIPS Control and Task subsystem objects showing only the inheritance hierarchy. We defer the presentation of associations, attributes, and operations to subsequent

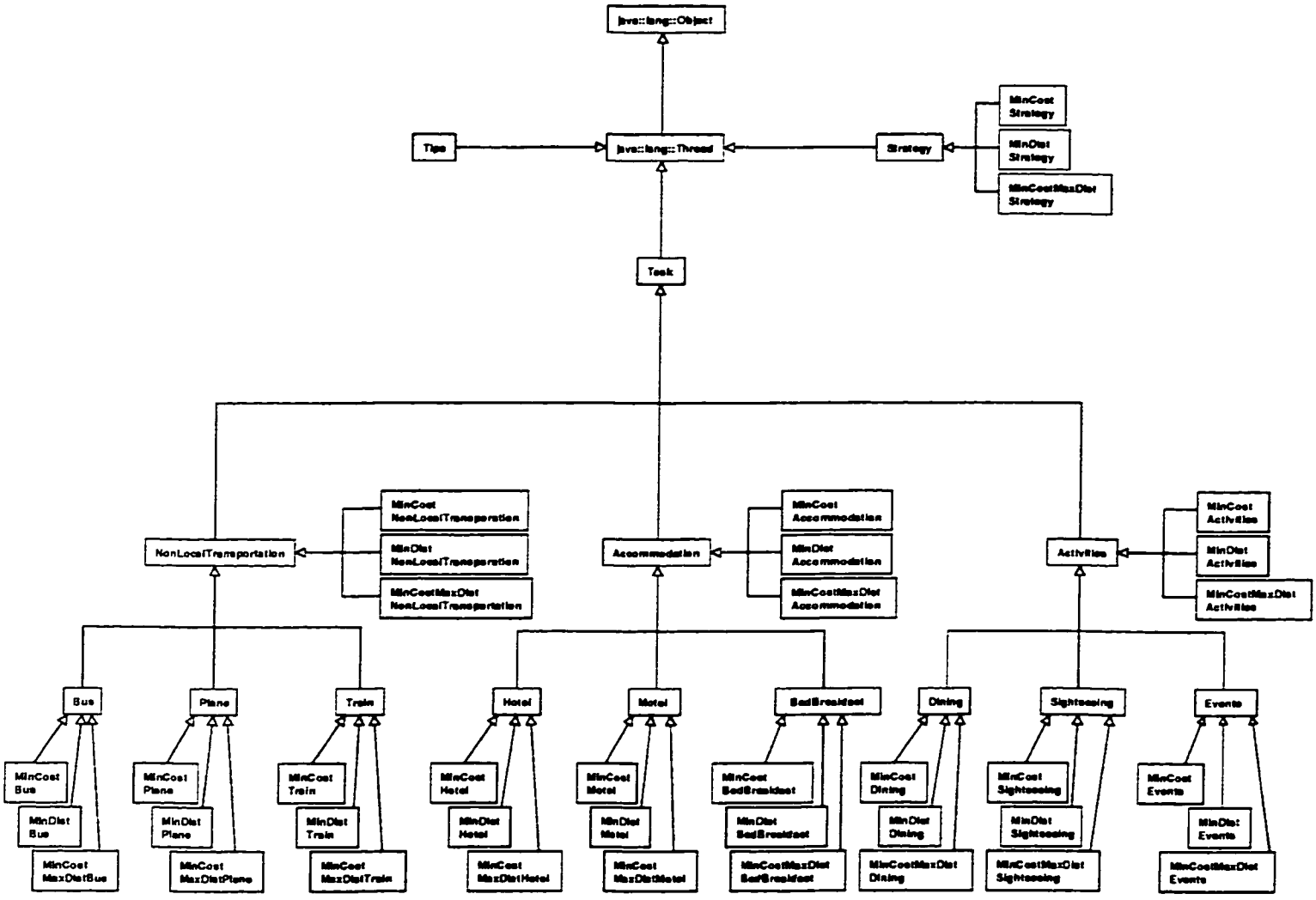

Figure 8: Class Diagram of Problem Solving Objects Inheritance

class diagrams. The Tips object is responsible for system-wide control operations. The three optimization strategies for our application are:

1. Minimize Cost (MinCost): Find a travel plan by minimizing the total cost of the trip.

2. Minimize Distance Travelled (MinDist): Find a travel plan by minimizing the total distance

travelled.

3. Minimize Cost and Maximize Distance Travelled (MinCostMaxDist): Find a travel plan of minimum total cost while, at the same time, of maximum total distance travelled.

The three controller objects corresponding to each of these strategies are: MinCostStrategy, MinDistStrategy, and MinCostMaxDistStrategy, respectively. These objects are responsible for combining the partial travel plan solutions from the task objects for their strategy and then sending one optimal solution to the Tips object to be displayed to the user, e.g. the solution of lowest total cost in the case of the MinCost strategy. The three strategy objects extend the abstract Strategy class and each one runs in its own separate thread of control.

The TPP is broken down into three tasks:

1. Non-Local Transportation: Plan the inter-city transportation.

2. Accommodation: Find appropriate accommodations for each destination of the trip.

3. Activities: Determine suggested activities (one per destination) to do during the trip.

Note that the intra-city transportation aspect of the problem is not handled by the current design. The abstract classes for the three tasks mentioned above are NonLocalTransportation, Accommodation, and Activities, respectively. The concrete strategy-specific task objects extend these abstract classes. Their role is to collect and combine the subtask solutions provided by their subtask objects and provide a fixed number of best task solutions to their corresponding strategy controller object. By best solutions we mean the solutions which best meet the particular goal of the strategy, e.g. the least costly solutions in the case of the MinCost strategy.

Finally, we have chosen nine subtasks which each provide the best transportation, accommodation, or activity subtask solutions according to the part of the TPP that they are attempting to solve, namely, Bus, Plane, Train, Hotel, Motel, BedBreakfast, Dining, Sightseeing, and Events. The classes whose names correspond to these subtasks are abstract classes and the strategy-specific concrete objects extend these classes.

Note that each concrete task and subtask object is a descendent of the Task abstract class, which extends the Java Thread class. Each concrete problem solving object runs in its own thread in order to support as much concurrency as possible and to allow for non-determinacy of the order in which each strategy terminates.

3.3.2 Associations

In Figure 9 we show the inheritance hierarchy and associations for the MinDist strategy objects. The Tips object has a bidirectional link to each concrete strategy controller object which it con-

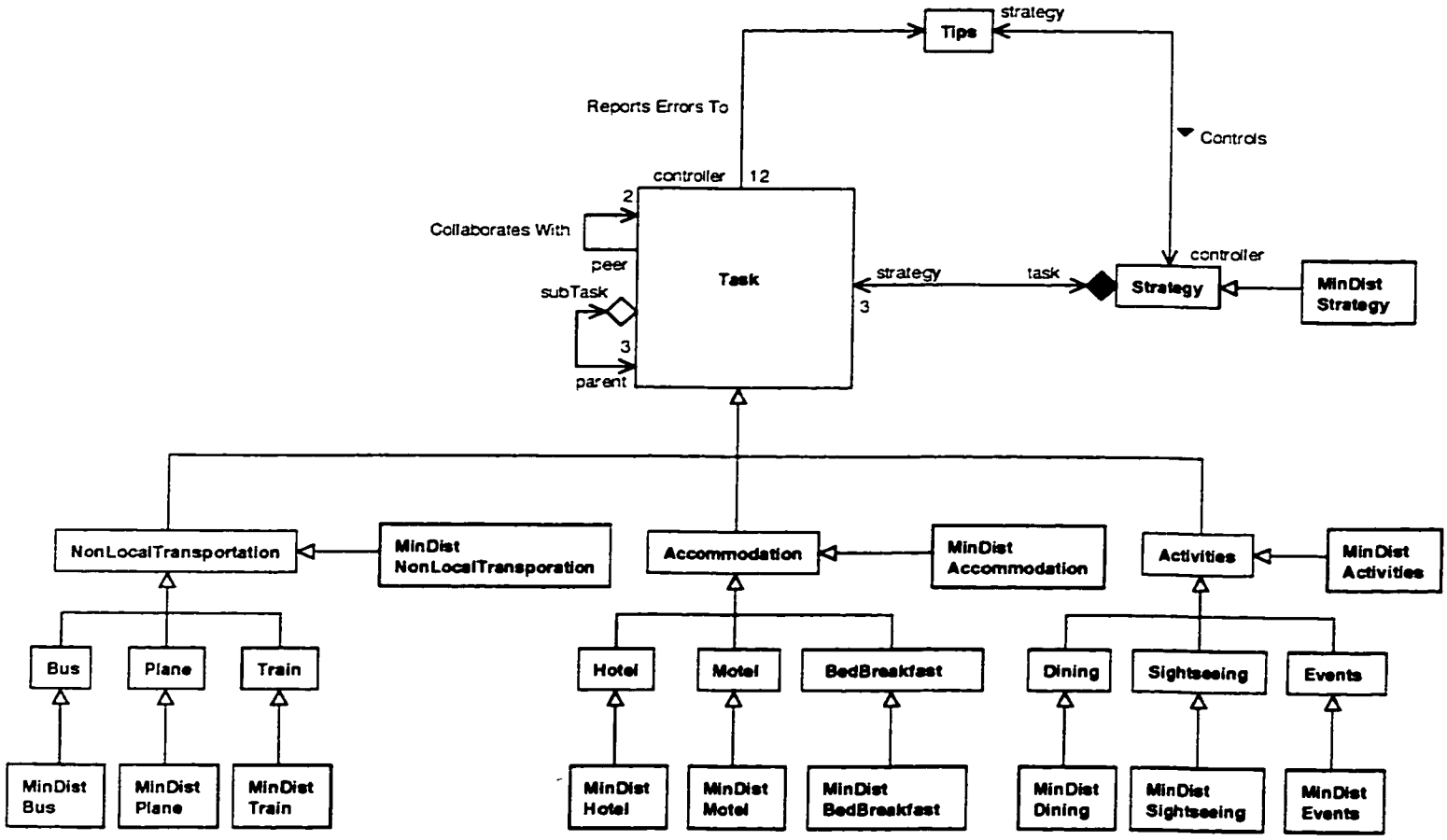

Figure 9: Class Diagram of MinDist Strategy Inheritance and Associations

trols by creating it, starting its execution, and stopping it, indicated by the "Controls" association. Each object derived from the Strategy class has bidirectional links to each of its three task objects

(MinDistNonLocalTransportation, MinDistAccommodation, and MinDistActivities in the case of MinDistStrategy). The strategy controller objects create, start, and stop their task objects, while the task objects return solutions to their strategy controller.

Each concrete task object derived from the Task abstract class has bidirectional links between itself and its three subtasks. For example, there is a bidirectional link between MinDistActivities and its three subtasks: MinDistDining, MinDistSightseeing, and MinDistEvents. The parent task creates, starts, and stops its subtasks, while the subtasks query their parent task for information about other tasks' sub-solutions and send their solutions to their parent task. Each task object has a link to its two other peer task objects, shown by the "Collaborates With" association. For example, MinDistNonLocalTransportation has a link to MinDistAccommodation and MinDistActivities and vice-versa. Peer tasks collaborate with each other by querying one another for information about their peers' sub-solutions. All task and subtask objects have unidirectional links to the Tips controller object in order to report errors, shown by the "Reports Errors To" association.

The class diagrams for the other strategies are similar to the one presented in Figure 9 on page 34. Although, all of the links are stored in the Task abstract class and are propagated to all classes derived from Task, each link is only actually used by those derived objects that we have mentioned above.

3.3.3 Attributes and Operations

We now present the complete class diagrams of the problem solving classes in the Controller and Task subsystems for the MinDist strategy, including attributes and operations. The most important attributes and operations are described in the Data Dictionary (see Appendix A). Figure 10 on page 36 shows the Task, Strategy, MinDistStrategy, and Tips classes. Figures 11, 12, and 13 on pages 37, 38, and 39 show the subclasses of the NonLocalTransportation, Accommodation, and Activities abstract classes, respectively.

The user's travel constraints, preferences, and requirements are passed down the inheritance hierarchy to the subtask constructor methods. The parameters are not shown (replaced by an ellipsis)

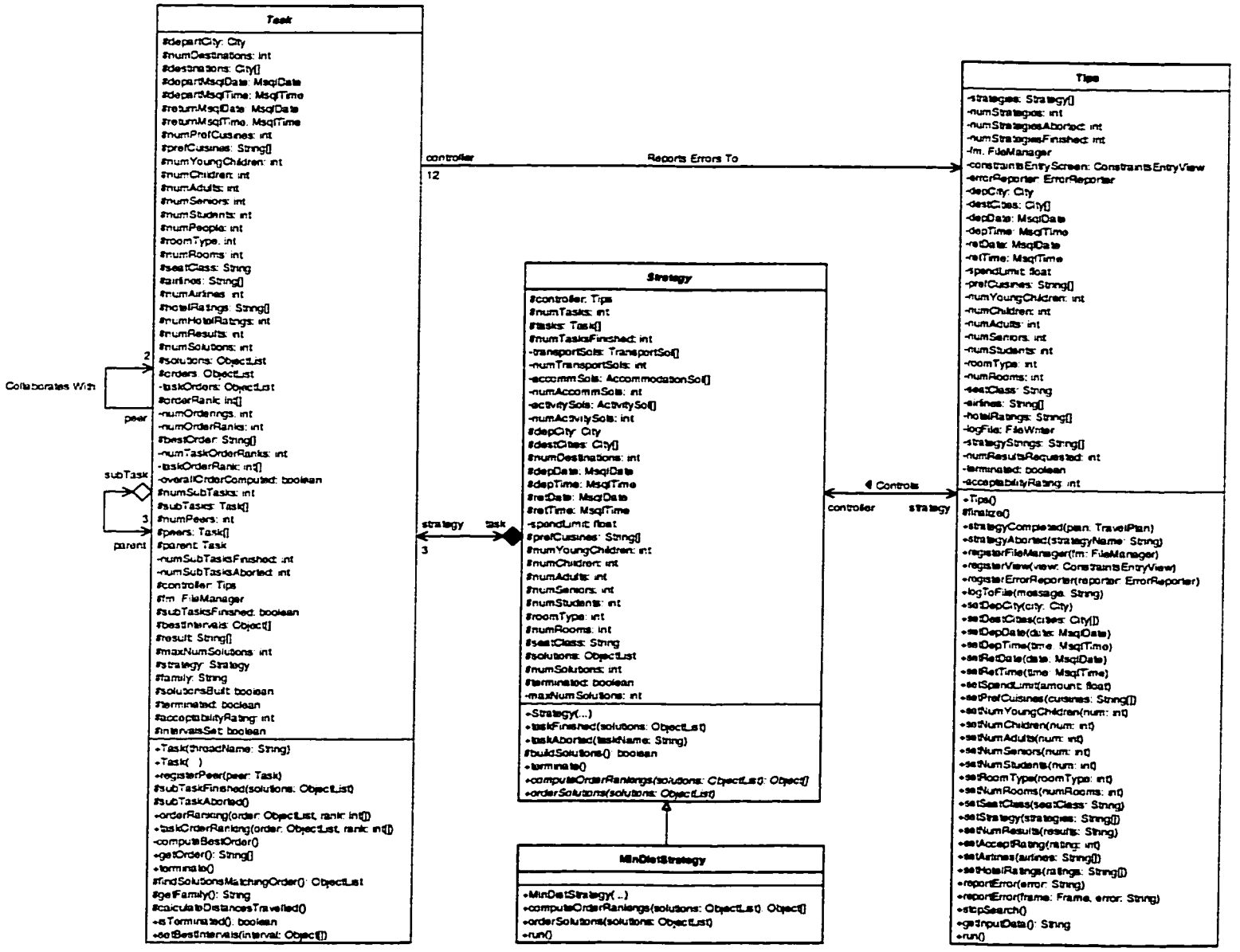

Figure 10: Task, Strategy, MinDistStrategy, and Tips Classes Attributes and Operations

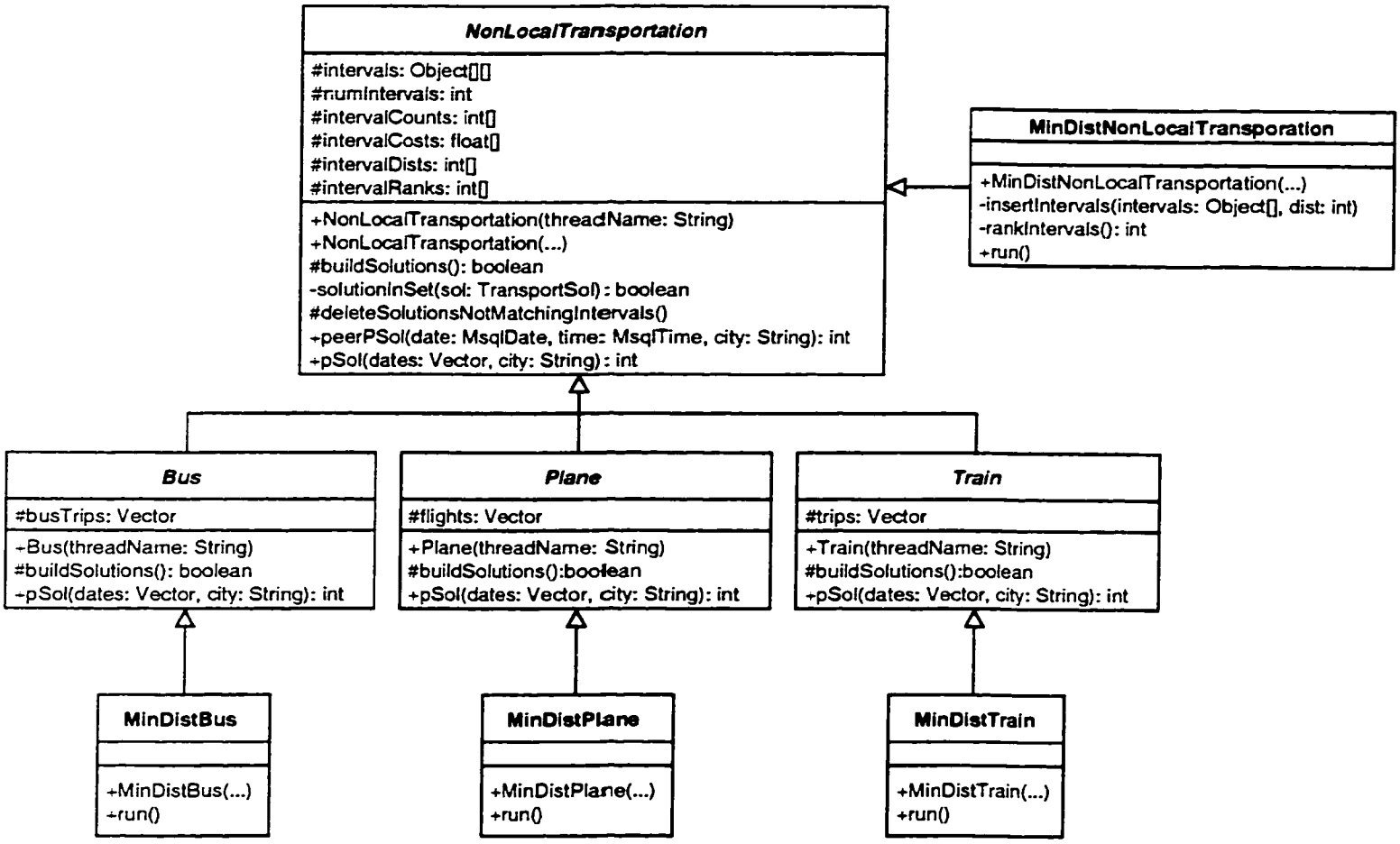

Figure 11: NonLocalTransportation Classes Attributes and Operations for MinDist Strategy

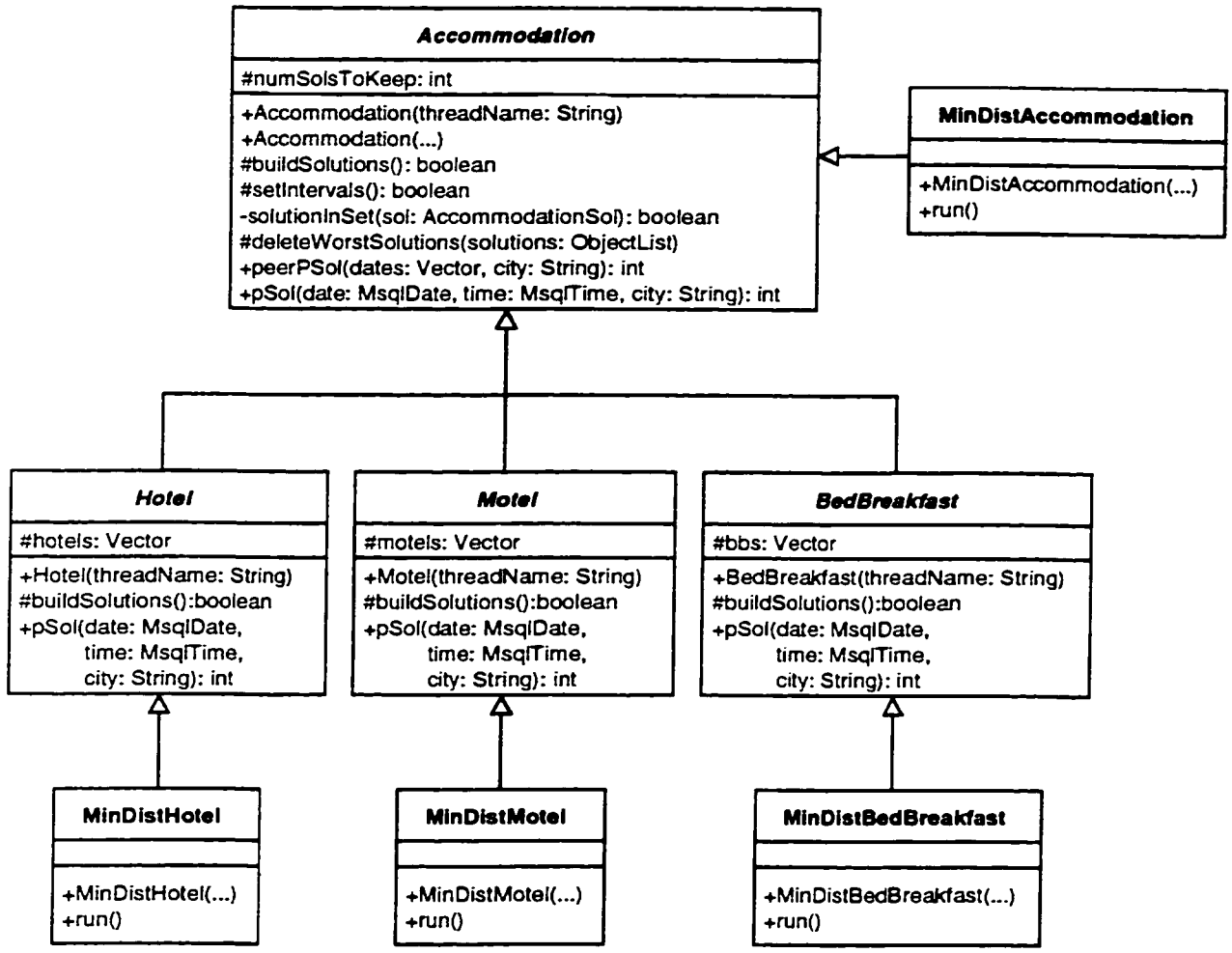

Figure 12: Accommodation Classes Attributes and Operations for MinDist Strategy

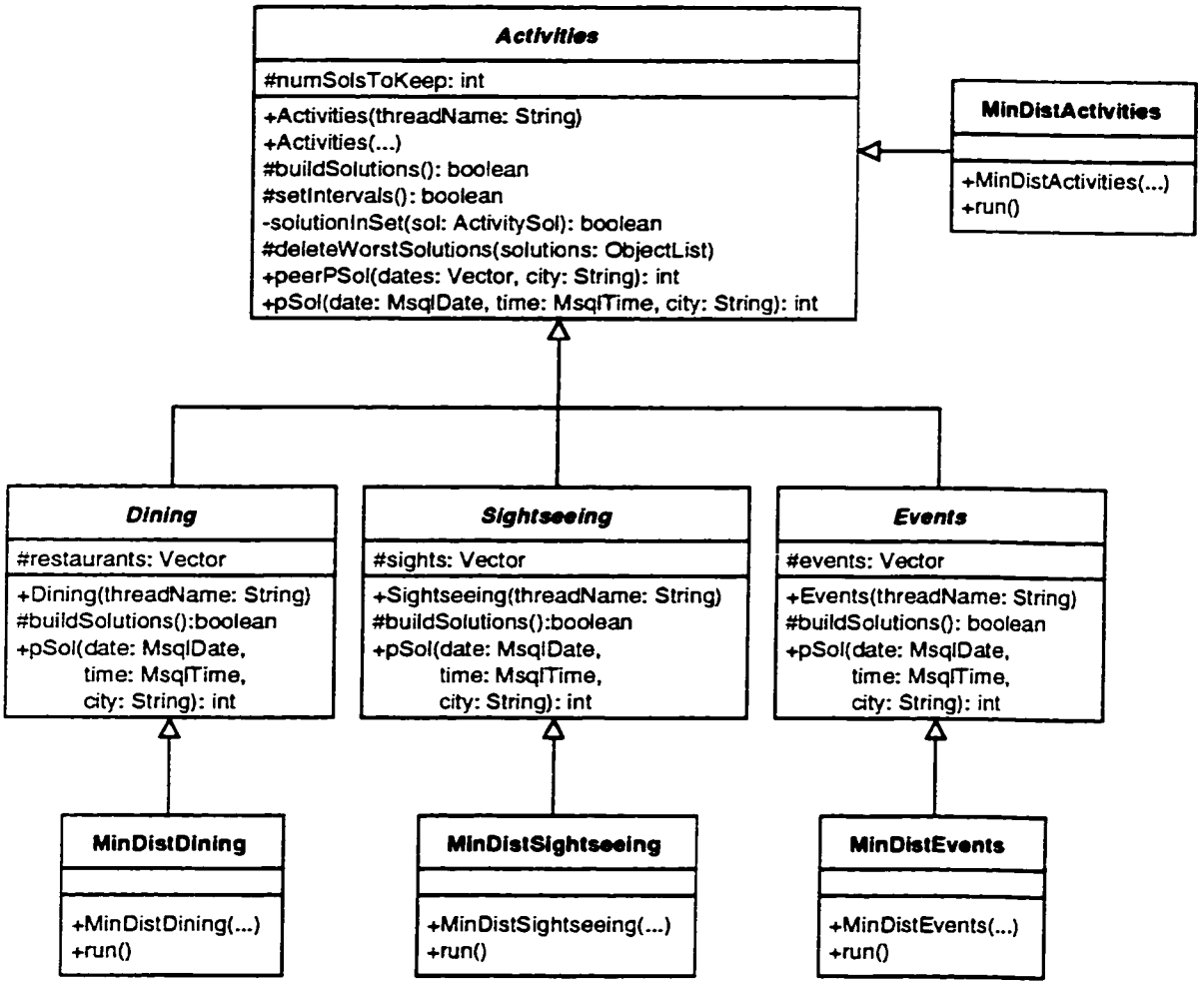

Figure 13: Activities Classes Attributes and Operations for MinDist Strategy

because they are many. The run method in each concrete object overrides the run method in the Java Thread class and contains the code which is executed inside each thread. The buildSolutions methods in the subtask objects create new subtask solutions from the travel data available, whereas, the buildSolutions methods in the task and strategy objects combine subtask solutions that are passed to them from their subtasks and and tasks, respectively, to produce new solutions. The peerPSol and pSol methods in the task and subtask objects are used to determine the percentage of sub-solutions produced by other subtasks that match a given subtask solution. In Section 3.4.5, we will describe the sequence of operations that accomplishes this.

The system controller (Tips) contains "register" methods which are used by different objects to link themselves to the controller. The "set" methods are used by the ConstraintsEntryView user interface object to send the user input to the controller so that the problem solving objects can be passed this information. The reportError and logToFile methods are called by the problem solving objects to communicate errors and important events to the controller. The strategyCompleted and strategyAborted methods are called by the strategy controller objects to indicate their completion and send the travel plan they obtained to the strategy controller, or to indicate failure to find an acceptable travel plan, respectively.

The Strategy abstract class contains attributes for the user constraints and the final strategy solutions. It contains methods for building these solutions (buildSolutions), managing the termination of its tasks (taskFinished and taskAborted), and two abstract methods (computeOrderRankings and orderSolutions). computeOrderRankings is called by a subtask to calculate the rankings of the city destination orders based on the optimization constraints employed by the strategy the subtask belongs to. orderSolutions is called by a task to sort its solutions by the optimization constraints used by the task's strategy in order to delete the worst solutions.

The Task abstract class mainly contains attributes for the user's travel constraints as well as the solutions. The important operations of this object are operations used to compute the best destination ordering (orderRanking, taskOrderRanking, computeBestOrder, and getOrder) and operations that handle the completion of the subtasks (subTaskFinished and subTaskAborted). Much code

reuse is achieved by placing attributes and operations used by tasks and subtasks in the Task class.

3.3.4 Solutions

An important detail in this design is the structure of the solutions that are passed between levels of the task hierarchy. The subtask objects work on different abstractions of the solutions than the task objects do. Figure 14 is a class diagram showing the inheritance hierarchy and associations of the solutions. There is a separate concrete class for each subtask and task solution. Each subtask and task solution is an aggregation of sub-solutions. For example, a PlaneSolution is an aggregation of Flight sub-solution objects. In Figure 14, the maximum number of sub-solutions a solution can have is d for non-transportation solutions and $d + 1$ for transportation solutions, where d represents the number of destinations selected by the user. This is so, because a transportation solution must also include an extra trip from the last destination back to the departure city. A TravelPlan object encapsulates a complete TPP solution and is composed of one TransportSol, AccommodationSol, and ActivitySol object.

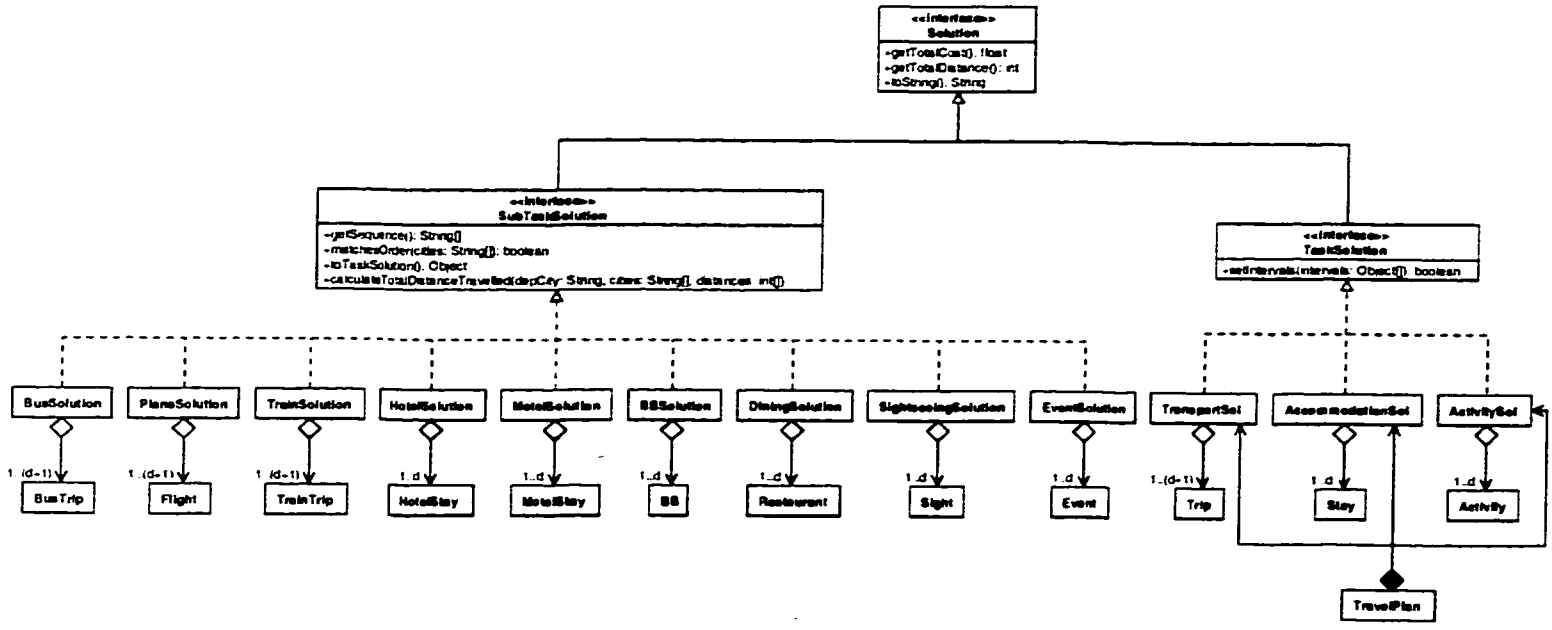

Figure 14: Class Diagram of the Solutions

Each solution object implements an appropriate interface. Subtask solution objects implement

the SubTaskSolution interface, and task solution objects implement the TaskSolution interface. Both of these interfaces extend the Solution interface. The reason this was done was to allow for the manipulating of the solutions by generic code. For example, the computeOrderRankings and orderSolutions methods of the Strategy class receive as parameters a list of a particular kind of solutions which they must assign rankings to or sort, respectively. They need to be able to operate on these solutions independently of their type. This is made possible by using these interfaces.

When a subtask passes one of its solution objects to its parent task, it first must convert it to the parent task solution type using the toTaskSolution method. A task need not distinguish the type of the solutions it receives from its subtasks. It can manipulate a subtask solution independently of its type. A parent task does not, nor should it have any knowledge of the structure of a subtask solution. Because of this fact, additional subtasks can be added to the system without necessitating modification of the task code. A subtask must only know how to convert a subtask solution to the appropriate task solution.

3.4 Dynamic Model

Let us now look in detail at what actions are performed by the strategy controllers, tasks, and subtasks by examining sample state diagrams of some of the objects. First, we will look at each object individually and later, in Section 3.4.4, we will combine their actions all together into one collaboration diagram.

3.4.1 Subtasks

In order to illustrate what actions are performed by the subtask objects, we have chosen to show the state diagram of the MinCostMaxDistBus object (see Figure 15 on page 43). Other subtask objects for other tasks and strategies will be very similar to this one.

In the Building Solutions state, the MinCostMaxDistBus object first creates a list of all bus trips that meet the locally-applicable travel constraints. Figure 16 on page 43 expands the Building

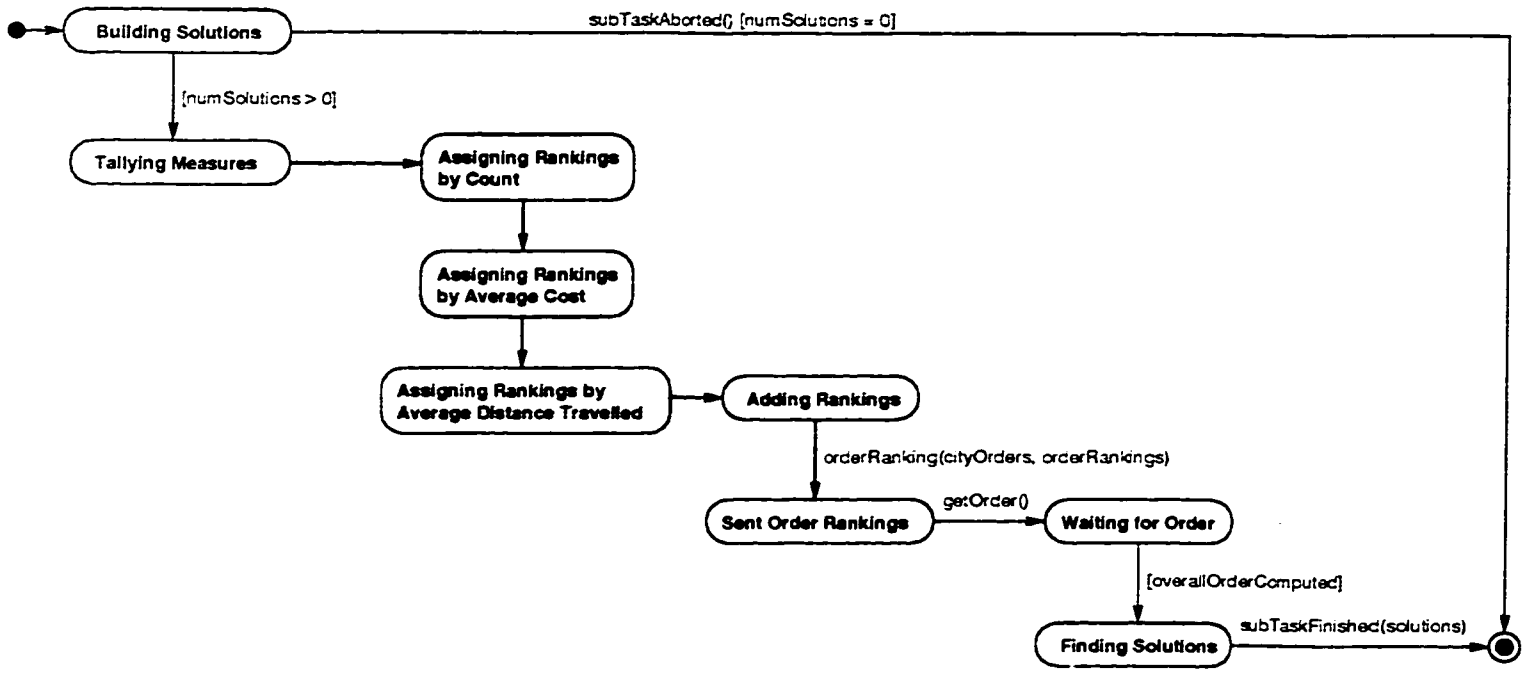

Figure 15: State Diagram of the MinCostMaxDistBus Object

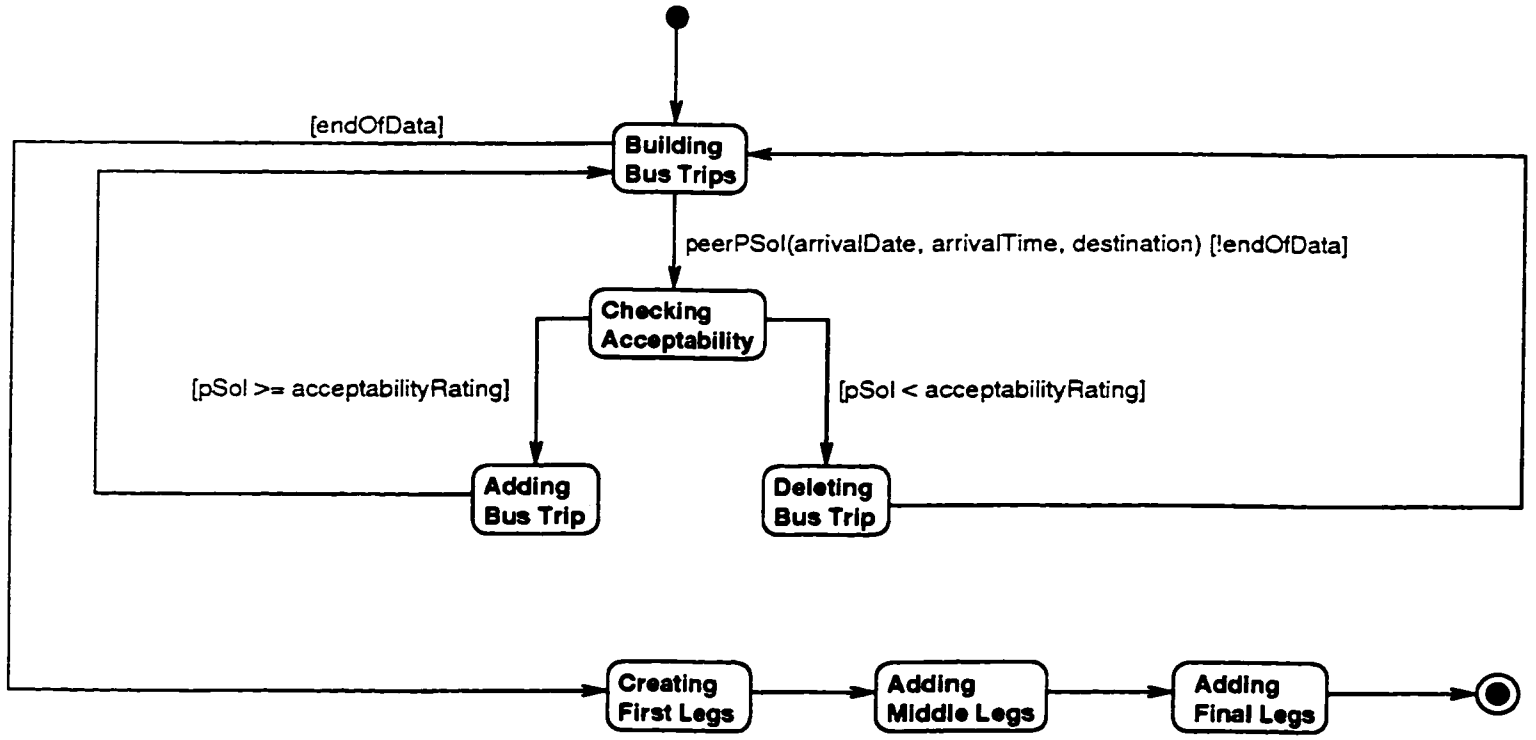

Figure 16: Refinement of the MinCostMaxDistBus Building Solutions Super-State

Solutions super state. While building the bus trips, the object also queries the other subtask objects (via the peerPSol event) to determine if each bus trip matches well with other accommodation and activity subtask sub-solutions such as hotel stays and events. For example, if the user has set the acceptability rating to some value, *acceptabilityRating*, and a particular bus trip matches well with *pSol* percent of the accommodation and activity solutions that are currently built for this strategy, then this bus trip will be kept for future use if $pSol \geq acceptabilityRating$. Otherwise, it is deleted from the database. A bus trip matches well with another sub-solution if the arrival date of the trip at the destination city is prior to at least one date that the accommodation is available or that the activity occurs. Only sub-solutions for cities that are the destination city of the bus trip are considered when computing the value of *pSol*. Likewise, accommodation and activity solutions such as restaurants and events are compared with transportation sub-solutions (bus trips, flights, and train trips) to determine if they meet the acceptability criteria.

When all of the bus trips that can be created from the travel data available have been created and checked for acceptability criteria (indicated by the endOfData condition in Figure 16 on page 43), we go on to build the bus solutions, which are lists of bus trips. The last three states in Figure 16 on page 43 indicate that the bus solutions are built in stages from the set of bus trips. First of all, solutions are built containing one bus trip for the first leg of the trip with the departure city of travels as the departure city of the bus trip. The middle legs of the trip are then added by adding bus trips that link the city last visited of each solution with new destinations from the list of selected destinations. Finally, the last leg of the trip is added by adding a bus trip that links the last city visited of each solution with the departure city of travels. Only a fixed maximum number of solutions can be created, in order to not exceed memory space limitations. If this limit is reached while adding any leg to the trip solutions, we move on to the add the remaining legs to the existing solutions. The time constraints of the trip specified by the user are also taken into consideration when creating these initial solutions.

In the subsequent states of Figure 15 on page 43, the number of bus solutions are reduced by

keeping the solutions that contain the best order of destination cities visited, based on the optimization strategy. In the case of the MinCostMaxDist strategy, this best order takes into account the number of solutions that contain this order, the average cost of all solutions with this order and the average distance travelled of all solutions with this order. If there are some acceptable solutions built in the Building Solutions state, we go on to the Tallying Measures state where we add up the number of solutions, costs of solutions, and distances travelled of solutions for each order of destinations in the solutions. Each order of destinations is then ranked by the greatest number of solutions with this order, least average cost of the solutions with this order, and highest average distance travelled for solutions with this order. Each order is assigned a rank starting with a value of 1 (highest) and increasing in value. Destination orders with the same tallies of these values are assigned the same ranking. The three rankings are then summed up for each destination order and sent to the MinCostMaxDistNonLocalTransportation task (via the orderRanking event) which accumulates the order rankings of all its subtasks and then returns the order with the highest combined ranking (in response to the getOrder event). Once the best order has been determined, the MinCostMaxDistBus object finds all solutions that contain this city ordering and sends them to the MinCostMaxDistNonLocalTransportation object via the subTaskFinished event.

3.4.2 Tasks

We now describe the state diagrams for the three tasks of the MinCostMaxDist strategy. The diagrams for the Accommodation and Activities tasks are essentially the same so we present only the diagram for the Accommodation task.

3.4.2.1 NonLocalTransportation

Figure 17 on page 46 shows the state diagram of the MinCostMaxDistNonLocalTransportation object. Initially, the object stores the order rankings it receives from its own subtasks (via the orderRanking event on the self transition to the Accumulating Order Rankings state) and the combined order rankings from the other strategy tasks (via the taskOrderRanking event on the self transition

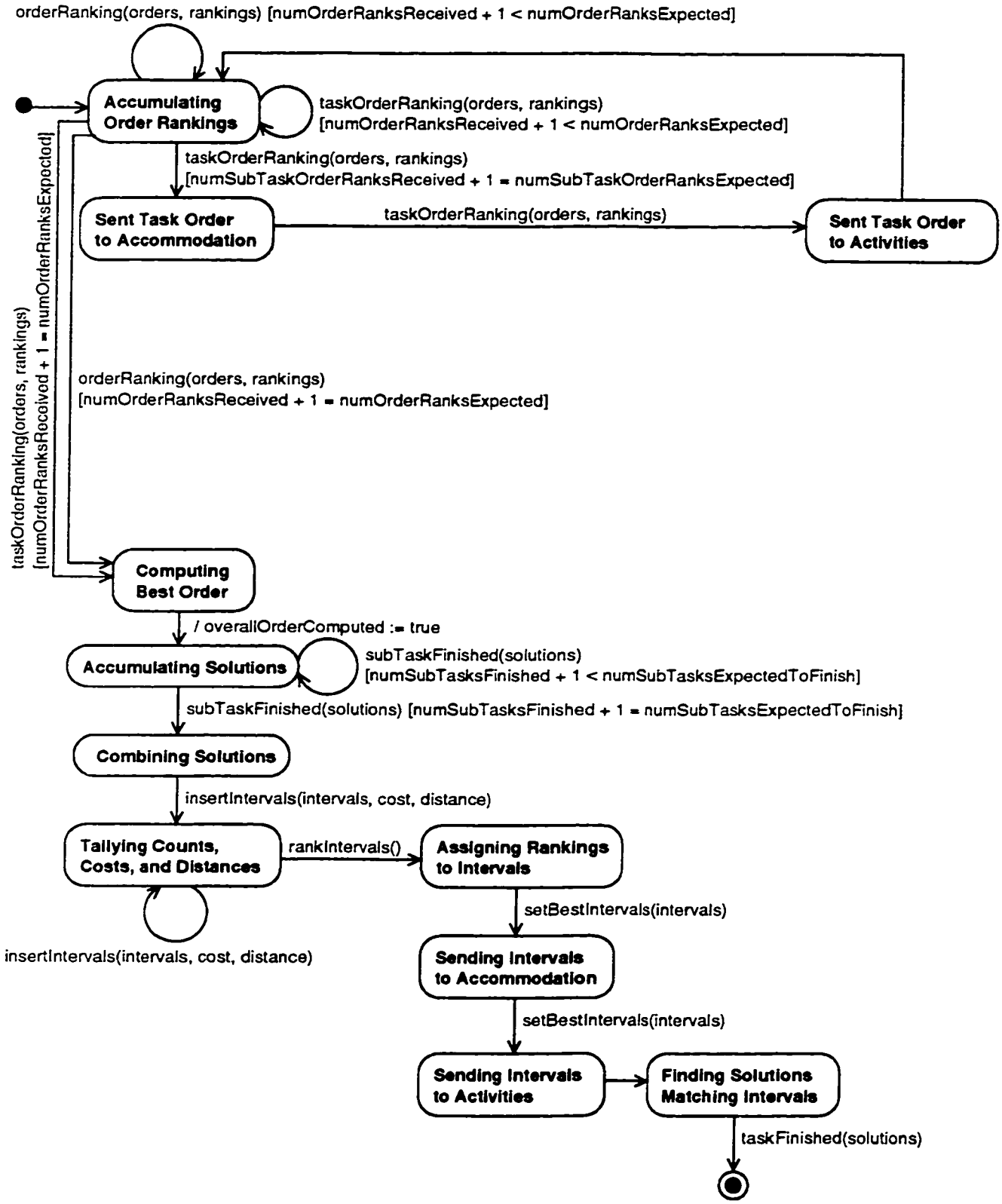

Figure 17: State Diagram of the MinCostMaxDistNonLocalTransportation Object

to the Accumulating Order Rankings state). When all of the order rankings from its own subtasks have been received, it sends the sum of the rankings from its subtasks to both the Accommodation and Activities tasks for this strategy (via the taskOrderRanking events to the Sent Task Order to Accommodation and Sent Task Order To Activities states). When both its own subtask order rankings and the other task order rankings have been received, the object determines the best order by summing up all of the rankings it has received for each order and choosing the one with the highest combined ranking (the one of lowest value) in the Computing Best Order state. This best order is then returned to its subtasks that are waiting on the order when the value of overallOrderComputed is set to true on the transition leaving the Computing Best Order state.

The object then stores its subtasks' solutions as they send the ones with the best order (via the subTaskFinished event). When the last subtask's solutions are received, new solutions are created by attempting to replace each trip leg of each subtask solution with a similar trip leg from another subtask's solution in the Combining Solutions state. A trip can be replaced by another trip t in a solution if the departure and destination cities as well as the departure and arrival times of t match together with those of the previous and next trips in the list of trips where it is to be inserted. As was the case with the subtasks, only a fixed maximum number of new solutions may be created at this stage, due to memory considerations.

The object then seeks to find the best time intervals for visiting each of the destinations by applying the goals of the strategy. Each transportation solution is a list of trips with departure and arrival dates and times which define the time intervals during which the traveller is visiting each destination. For each unique list of time intervals present in the solutions, the object sums up the number of solutions corresponding to these intervals, the total cost of these solutions, and the total distance travelled for these solutions. This is done each time the insertIntervals event fires for each solution. Each unique list of time intervals that is present in the solutions is then assigned a ranking for the greatest number of solutions, least average cost of the solutions, and maximum average distance travelled of the solutions. The list of time intervals with the highest combined ranking is chosen as the best list of stay intervals. These stay intervals are then sent to the

MinCostMaxDistAccommodation and MinCostMaxDistActivities objects via the setBestIntervals event so that they can use these time intervals to assign dates and times to their solutions. Finally, those transportation solutions which contain the list of time intervals that was chosen are sent to the MinCostMaxDistStrategy object via the taskFinished event.

3.4.2.2 Accommodation and Activities

Figure 18 on page 49 shows the state diagram of the MinCostMaxDistAccommodation object. The states which describe the process of determining the task's destination order rankings and best overall destination order are the same as what we saw in Figure 17 on page 46 except that the order rankings of this task are sent to the NonLocalTransportation and Activities tasks. Once the best overall destination order has been determined, the task accumulates its subtasks' solutions and combines them, creating new solutions in a similar fashion to the way MinCostMaxDistNonLocalTransportation combines solutions. New accommodation solutions can be created by replacing each accommodation stay of each accommodation solution with one of another type for the same destination, e.g. replacing a motel accommodation with a bed and breakfast accommodation for the same destination. As was the case with the NonLocalTransportation object, only a fixed maximum number of new solutions may be created at this stage in order to conserve on memory consumption.

Upon receiving the setBestIntervals event from MinCostMaxDistNonLocalTransportation, the object stores the best list of time intervals and assigns check-in and check-out times to the accommodation solutions based on these intervals in the Assigning Intervals state. If a solution cannot be used during these time intervals, it is deleted.

Then the total costs and distances travelled for each solution are determined and each solution is assigned a ranking based on these measures by least cost and maximum distance travelled. The rankings are summed up for each solution and the solutions are ordered by highest ranking (in the Ordering Solutions state) and a fixed number of lowest-ranked solutions are deleted (in the Deleting Solutions state). The remaining solutions are sent to the MinCostMaxDistStrategy object via the taskFinished event.

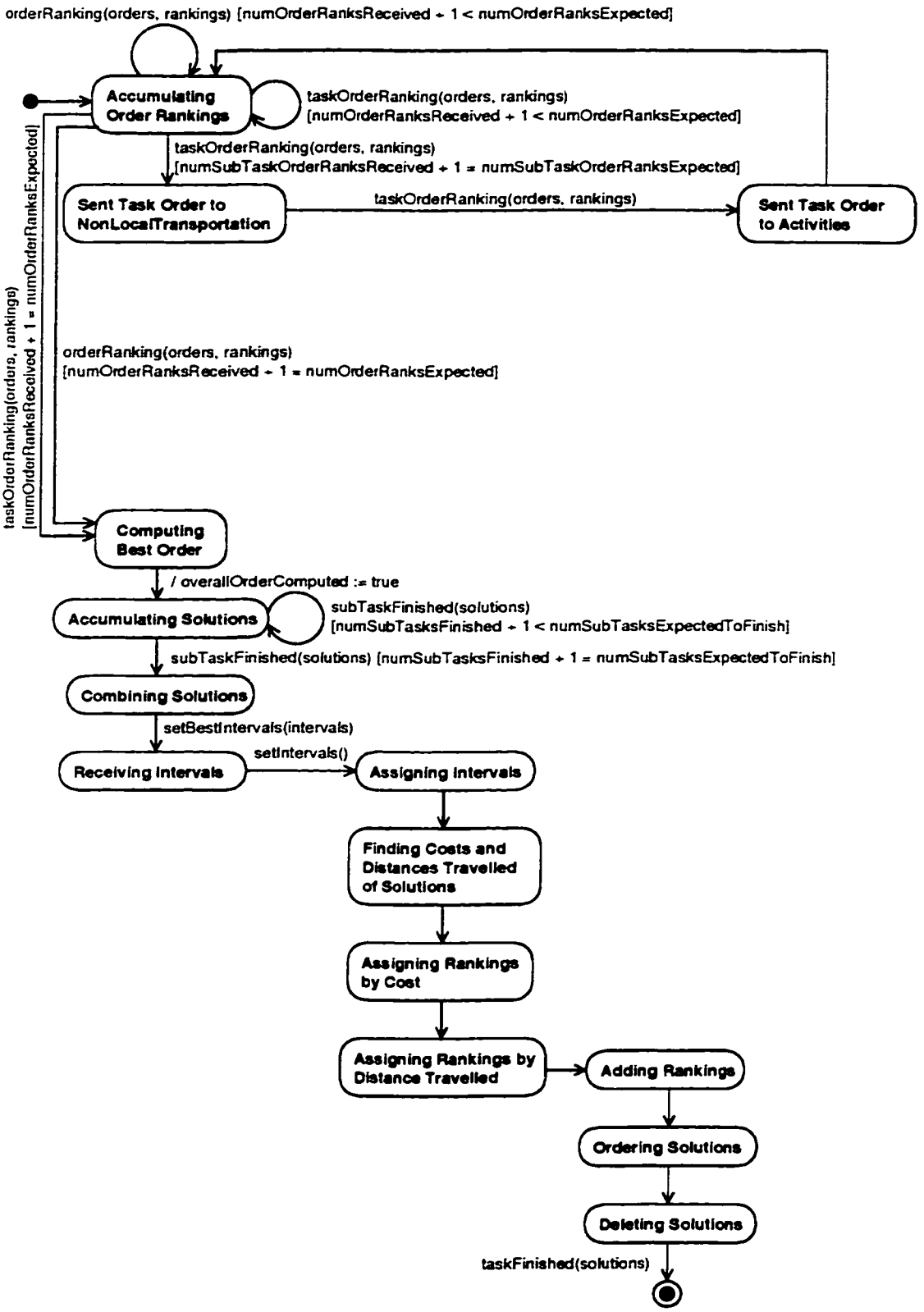

Figure 18: State Diagram of the MinCostMaxDistAccommodation Object

3.4.3 Strategy Controller

A state diagram for the MinCostMaxDistStrategy object is shown in Figure 19. Solutions are stored as they are sent by the three strategy tasks described in Sections 3.4.2.1 and 3.4.2.2 in the Accumulating Solutions state. When the last task returns its solutions, travel plans are created by forming all possible combinations of the solutions obtained from the three tasks in the Making Travel Plans state. A travel plan consists of one solution from each task. Only a fixed maximum number of travel plans may be created in this state in order to not exceed memory storage limits. Then the object retrieves the total cost and distance travelled of each solution which it uses to rank the travel plans. The object then finds the travel plan with the best combined ranking in the Finding Best Travel Plan state and sends it to the Tips object via the strategyCompleted event.

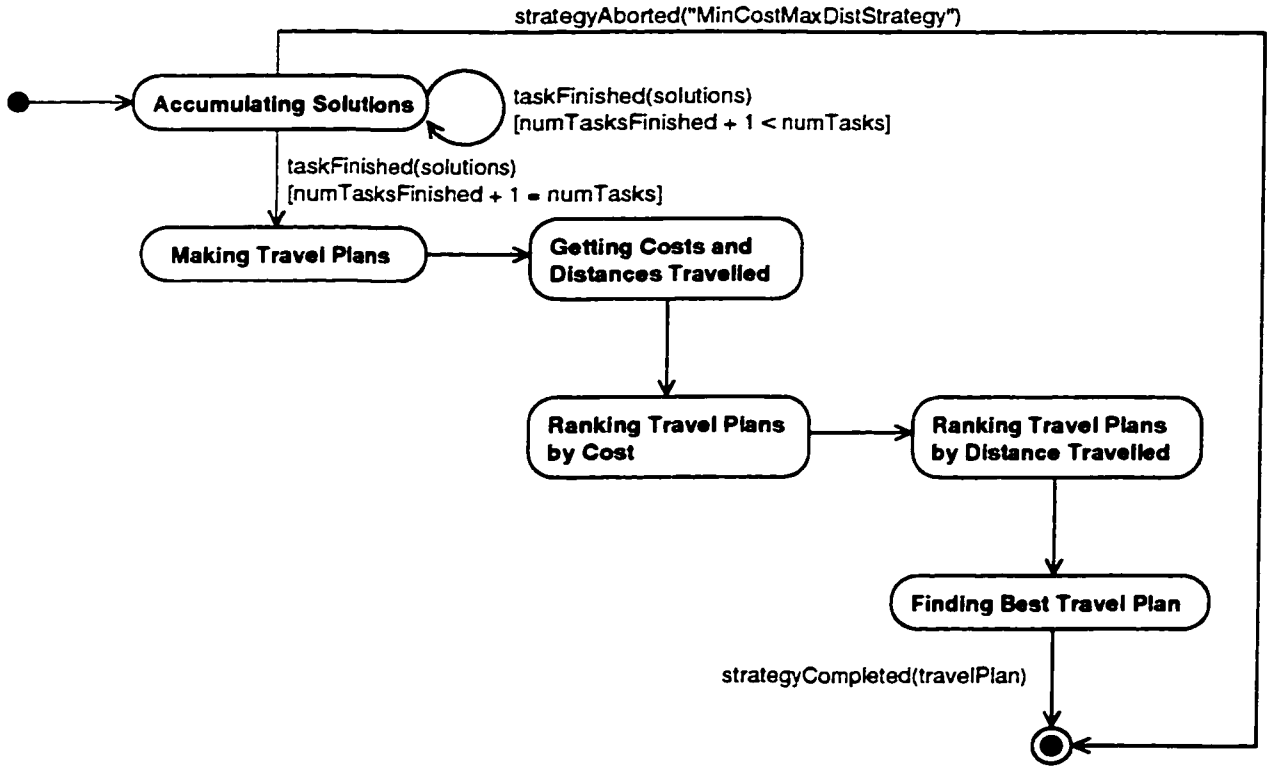

Figure 19: State Diagram of the MinCostMaxDistStrategy Object

3.4.4 Collaboration Between Problem Solving Objects

Figure 20 shows a collaboration diagram for the objects participating in the MinCost strategy. It shows only the external messages that are sent between each of these objects which are all running

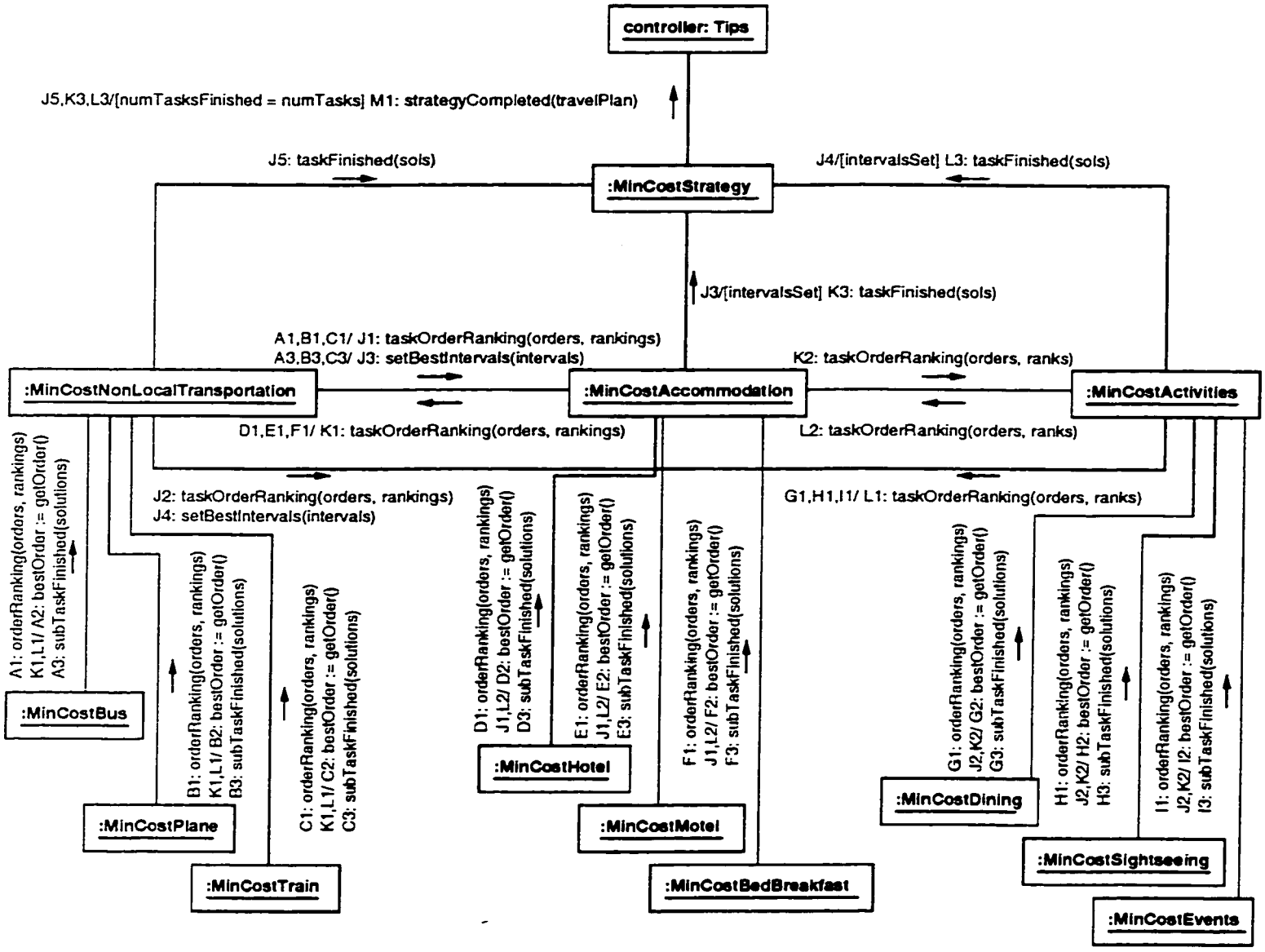

Figure 20: Collaboration Diagram for the MinCost Strategy

in their own separate Java threads. It does not show the sequence of messages that are sent to determine the acceptability of sub-solutions. We describe this aspect of subtask collaboration in Section 3.4.5. The names of messages that must be sent before a message can be sent precede the

forward slash character in the message label. Conditions that must be met in order for a message to be sent are indicated between brackets. The name of a message precedes the colon on the message label and the message signature follows after the colon.

Each subtask first sends its destination order rankings to its parent task via the orderRanking message. Once the cumulative task order rankings from the other tasks have been sent (via the taskOrderRanking message) to its parent task, a subtask then retrieves the best destination order via the getOrder message. It then sends its final solutions matching this destination order to the parent task via the subTaskFinished message.

Once it has received all of its subtask solutions and made combinations of them, the MinCostNonLocalTransportation object calculates and sends the best time intervals of these solutions to the other strategy tasks with the setBestIntervals message. Once each task has all of its final solutions that have been assigned dates and times corresponding to the best stay time intervals, it sends them in the taskFinished message to the MinCostStrategy object.

Finally, the MinCostStrategy object sends the best travel plan to the Tips object in the strategyCompleted message, after having received all of its tasks' solutions. The messages that are exchanged ensure that the solutions that are produced by each object at each stage are compatible in that the order of cities visited and time stay intervals match.

3.4.5 Limitation of Sub-solutions Collaboration

Figure 21 on page 53 illustrates the sequence of messages that are exchanged between the problem solving objects to determine the degree to which a subtask's sub-solution matches with the sub-solutions of other subtasks. In this diagram, we show the MinCostPlane subtask querying the accommodation and activity subtasks about the acceptability of one of its flights. The MinCostPlane object sends a peerPSol message to its parent task, MinCostNonLocalTransportation, asking it to determine the percentage of sub-solutions belonging to its peer tasks (MinCostAccommodation and MinCostActivities) that match with its flight sub-solution. The date and time sent in the message is

the arrival date and time of the flight in the specified destination city. MinCostNonLocalTransportation then queries each of its peer tasks for the percentage of their sub-solutions that match with its sub-solution and returns the average percentage to MinCostPlane. Based on whether this percentage is greater or less than the acceptability rating percentage set by the user, MinCostPlane will retain or delete the flight from its database, respectively.

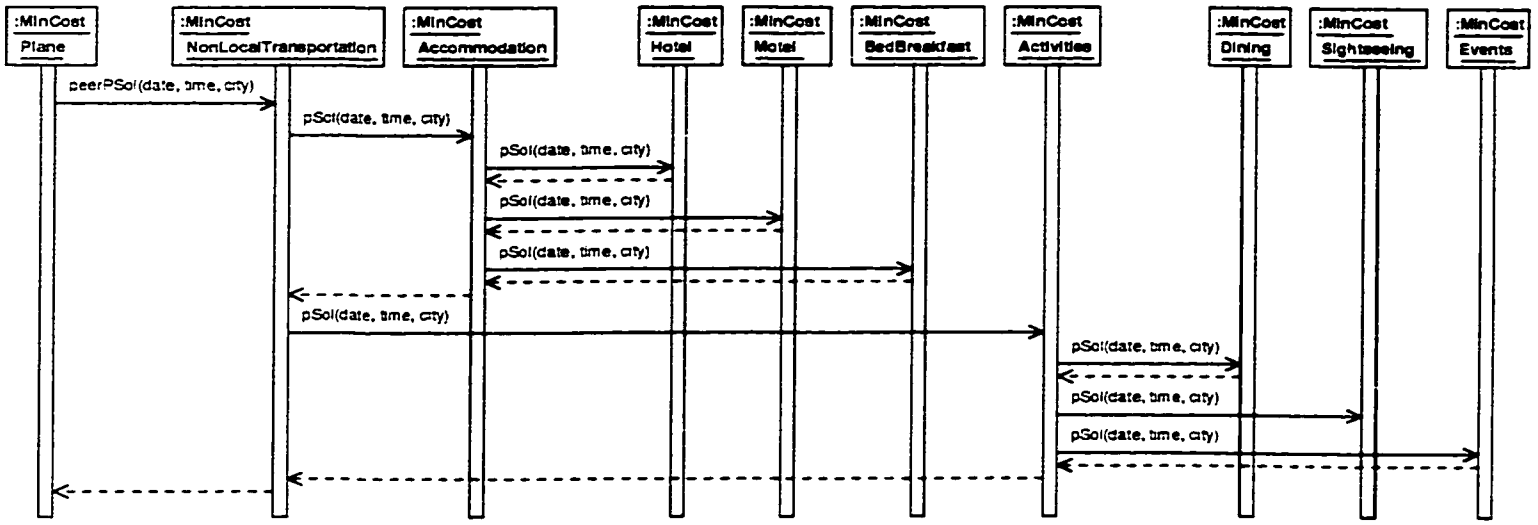

Figure 21: Sequence Diagram of the Limitation of Sub-solutions for the MinCost Strategy

In response to the pSol message, each peer task of MinCostNonLocalTransportation queries its subtasks for the percentage of sub-solutions that match the flight arrival date, time, and city and return the average percentage to MinCostNonLocalTransportation. When calculating the percentage, subtasks consider only those accommodation stays or activities that are located in the destination city of the flight. A subtask sub-solution matches the flight if the accommodation stay or activity can be booked later in time than the flight arrival date and time.

This sequence of messages is sent once for each flight when it is created and the percentage that is returned is entirely dependent upon the state of the problem solving in the other subtasks at the time the message is sent, since the messages are sent asynchronously. If a given subtask has not computed any sub-solutions at the time the pSol message is received, it will return a percentage of 100%, so as not to presumptuously favor the deletion of the given flight.

In the case of the limitation of sub-solution queries sent by the other subtasks, the principle is essentially the same, except for the following exceptions. Accommodation and activity subtasks do not query each other with regards to the acceptability of sub-solutions. They only query the NonLocalTransportation subtasks. They do so by sending a list of possible dates on which the accommodation stay or activity can be booked and a city rather than a specific date and time and a city, because dates and times of Accommodation and Activities solutions is not determined until later on in the problem solving process. This is different from the NonLocalTransportation subtasks which assign a specific date and time to their sub-solutions immediately upon their creation. An Accommodation or Activity subtask sub-solution matches a given NonLocalTransportation sub-solution with a destination equal to the city specified if the arrival date and time of the NonLocalTransportation sub-solution is earlier in time than at least one of the dates in the list of dates specified.

3.5 Design Rationale

We now present the rationale for the important design decisions discussed in this chapter.

3.5.1 Packaging

Figure 7 on page 31, the Tips, strategy controller classes, and TravelPlan class are placed in the Controller subsystem. This was done because there is a strong relationship between the Tips and Strategy objects in that Tips creates the strategies and the strategies provide travel plans to the Tips controller. The TravelPlan object is the solution type used by these controllers.

The other problem solving objects were grouped according to task and subtask functionality. The solutions and sub-solutions used by these objects were also grouped together. The objects are grouped by task and subtask functionality rather than by strategy functionality, because there is more coherence between objects that perform the same task than between objects that belong to the same strategy.

3.5.2 Choice of Problem Solving Objects

The Task object was chosen to represent a generic aspect of problem solving for the TPP. Different kinds of tasks and subtasks can be manipulated interchangeably via the Task interface. The abstract task classes (NonLocalTransportation, Accommodation, and Activities) and abstract subtask classes, e.g. Bus, Motel, and Sightseeing, were chosen to provide operations that can be re-used by their concrete derived classes. These derived classes were then used to implement the particular strategy for each of these tasks and subtasks.

3.5.3 Choice of Control Objects

The Tips object was chosen to act as a controller for the entire system. It controls which strategies to run and receives the resulting travel plans produced by those strategies. When it has received the number of results requested by the user, it terminates any strategies that are still running, because they are no longer needed. Also, it receives all errors reported by the problem solving objects and reports them to the user. In the case where all of the strategies abort, it must also report this fact to the user. The selection of strategies to run, the collection of travel plans produced by the strategies, and the reporting of errors are the main factors that necessitate this system-wide control object.

The strategy controller objects are partly responsible for control of the objects of a strategy and partly responsible for problem solving at the strategy-level when combining solutions produced by tasks. However, we see them primarily as controller objects. The strategy controller objects were designed using the Strategy design pattern [6]. They comprise strategy-dependent control operations than can be used by all tasks and subtasks. By placing strategy-specific code that is used in many task or subtask objects belonging to the same strategy in one strategy controller class, much code reuse is achieved. The strategy controllers were created to start the strategy-specific tasks, to collect their solutions, and to handle their termination. If a task terminates without producing any solutions, the strategy controller must halt the execution of the other tasks and subtasks and abort the entire strategy, because complete TPP solutions cannot be produced by the strategy in this event. This

required functionality necessitates the existence of the strategy controllers.

3.5.4 Arrangement of Inheritance

The Task abstract class represents a subdivision of work for solving the TPP. As such, each subclass of Task can be seen as a specialization of Task. The NonLocalTransportation, Accommodation, and Activities abstract classes represent the transportation, accommodation, and activity sub-divisions of work, respectively, and are concerned with producing appropriate solutions for their particular sub-division of the TPP. If we look at the Accommodation hierarchy as representative of all of the tasks, for example, MinDistAccommodation inherits from Accommodation, because it works on the accommodation part of the problem, specializing in finding solutions of minimum distance travelled. The Hotel, Motel, and BedBreakfast abstract classes also inherit from Accommodation, because they also work on the accommodation problem, focusing on hotel, motel, and bed and breakfast accommodation solutions, respectively. MinDistHotel, MinDistMotel, and MinDistBedBreakfast are subclasses of Hotel, Motel, and BedBreakfast, respectively, because they each work on the same aspect of the problem while specializing in finding solutions of minimum distance. The reasoning for the inheritance hierarchy of the other task and subtask objects is similar.

The inheritance of the Task subsystem was arranged in such a way as to increase code reuse. The attributes used by all task and subtask objects are stored in the Task abstract class. Also, there are many operations common to derived classes that are stored in the abstract base classes. Furthermore, it would be relatively easy to add new groups of classes to handle other subtasks and other optimization strategies not implemented by the travel planner we have presented here. For example, a new CarRental subtask abstract class could extend NonLocalTransportation that would search for car rental transportation solutions. By extending the Strategy class and the task and subtask abstract classes with strategy-specific derived classes, we can implement new optimization strategies. For example, a new MinTravelTimeStrategy class can be implemented to extend Strategy, which implements operations for a strategy that minimizes the amount of time spent travelling between cities. Other classes such as MinTravelTimeAccommodation and MinTravelTimeTrain

would also be implemented to find accommodation and train solutions that minimize on time spent travelling.

New subtask abstract classes can be added to the system with little modification to the existing code, because each subtask is handled through the Task class interface. New strategies can be added easily, because strategy-specific, task-independent operations are provided via the Strategy abstract class interface, which is extended when we implement a new concrete strategy controller object. These facts demonstrate the provision for design for change in our architecture.

3.5.5 Design of Associations

Figure 9 on page 34 presented the associations for the problem solving objects. The association between the Strategy and Tips classes is bidirectional, because Tips must start and stop the strategies from running and the strategies must report their solutions to Tips. The association from Task to Tips is unidirectional, because Tips does not directly create tasks and subtasks. That is left up to the strategy controller objects. Tasks and subtasks must be able to report errors to Tips, on the other hand. The association between Task and Strategy is a composition, because a strategy ceases to run when one of its tasks aborts. It is bidirectional, because a strategy controller must start and stop tasks and a task must send its solutions to its strategy controller. The "Collaborates With" association between tasks is necessary in order to allow tasks to query other tasks for information on their solutions and distribute this information amongst themselves. The association between tasks and subtasks is necessary, because a task must be able to create, start, and stop its subtasks and a subtask must report its solutions to its parent task. The association is an aggregation rather than a composition, because a task can continue to exist if one of its subtasks aborts, as long as it has at least one other subtask still running.

Figure 14 on page 41 presented the associations for the solution objects. The associations between the solution and sub-solution objects is a unidirectional aggregation, because the solution consists of sub-solutions that may be added or removed throughout the problem solving search and sub-solutions objects need not access the solution object. The association between TravelPlan and

its components is a unidirectional composition, because a travel plan must have one transportation, accommodation, and activity solution and the three parts of the travel plan need not access the TravelPlan object.

3.5.6 Choice of Data Structures

We chose to represent the solutions produced each task and subtask as a list of generic objects. Each object in the list represents a solution, which, in turn, is represented as a list of generic objects. Each of these objects represents a sub-solution. By choosing to represent the solutions as a list of generic objects, we can place the attribute that represents the solutions of the tasks and subtasks in the Task abstract class. Also, it requires us to provide only one method for operations that must be passed a list of solutions as the parameter and must be called by different tasks and subtasks. The method can access the particular solution it receives via one of the solution interfaces described in Section 3.3.4.

3.5.7 Choice of Search Algorithm

The search algorithm used by each subtask is a combination of the breadth-first search and the limitation of successors method. The breadth-first search strategy was chosen because it is a simple search algorithm to implement and describe and works efficiently for TPPs in which the number of destination cities selected is small, the travel interval is short, and the solution space is small. In our implementation, the set of travel data used is rather small, so the solution space contains a manageable number of solutions. With this knowledge at hand, it seemed that the breadth-first search would be satisfactory for our purposes. However, if the number of destinations selectable by the user were to increase, longer travel intervals are requested, and the solution space grew accordingly, a breadth-first search would not be amenable to the TPP. The breadth-first search was also chosen because it produces solutions with a broader range of destination cities earlier in the search than the depth-first or backtracking search does.

By bounding the breadth-first search strategy and combining it with the limitation of successors method, the number of solutions explored can be reduced so as not to exceed memory constraints.

Putting a bound on the number of solutions that are created does mean that the entire solution space is not explored. However, it is required when faced with a large solution space and limited memory storage space.

Discarding a sub-solution because it does not match with a given percentage of other sub-solutions as we do with the limitation of successors method may be premature, since it might actually be part of the optimal solution. In order to avoid this possibility, the acceptability rating can be set to a lower value. Although, the optimal solution has a chance of being discarded, using the limitation of successors method does provide more focus to the problem solving and does eliminate solutions that probably will not lead to an optimal solution so that memory and computing resources are not over-taxed. It does complement the breadth-first search strategy in this regard.

3.5.8 Implementation of Control

The key concept that is maintained throughout this design is the decentralization or distribution of control using concurrent tasks. No one strategy controller, task, or subtask object has sole responsibility for control decisions. Within a strategy, each object working on a particular sub-division of the TPP has the appropriate local knowledge to make its own control decisions to solve its part of the problem. Without this decentralization of control, much concurrency would not be possible with the TPP.

The natural order of computation in a sequential design of a travel planning application would be to first compute the order in which the cities will be visited, then find an appropriate transportation solution for travelling to those cities and then determine accommodation and activity solutions that complement the transportation solution. With our design, we have attempted to counteract the natural tendency to solve this problem in a sequential fashion by introducing as much concurrency as possible in the problem solving process in order to make computational gains afforded by multi-processors.

The Tips, strategy controller, concrete task, and concrete subtask objects are each given their own separate threads of control. Some measure of concurrency can be realized by having the task

and subtask objects work on finding solutions to their own part of the TPP. Tasks and subtasks belonging to different strategies can execute in parallel throughout their lifetime.

Subtasks of the same strategy can work independently, except for periods of collaboration, when a subtask must query other subtasks for the acceptability of its solutions and when it must wait for the results of the other subtasks when determining the order in which the destination cities are to be visited. As mentioned in Section 3.5.7, the former kind of collaboration is necessary in order to reduce the number of solutions explored. The latter kind of subtask collaboration is necessary in order to produce coherent solutions. At some point in the problem solving process, the order in which the cities are visited must be fixed in order to produce complete and compatible solutions in this distributed problem solving framework. This decision is made jointly by the subtasks.

The Accommodation and Activities task objects do not execute concurrently with the NonLocalTransportation objects until the NonLocalTransportation task has determined the time intervals for staying at each destination. Again, at some point in the search process, the tasks must agree upon common time intervals for the stays at each destination. One alternate method of determining the time stay intervals would be to have the Accommodation and Activities tasks create solutions that are assigned specific time intervals. Then the tasks could collaborate together to find the best time interval in a manner similar to how the destination city order is determined through subtask collaboration. However, the assignment of time intervals would then be arbitrary. For example, in the case of hotel accommodation, a stay at a particular hotel can be assigned a length of any number of days, provided that it falls with the travel interval. Plane flights, on the other hand, only take off at specific times. Thus, it was deemed inappropriate to speculate on the time intervals of the accommodation and activity solutions, and allow the NonLocalTransportation task to handle the determination of the stay time intervals.

Alternatively, the destination order and stay time intervals could have both been determined at the subtask level of control. Then the task objects would only have been concerned with combining the solutions received from their different subtasks. This approach was decided against in order to give the tasks some degree of control and distribute the application of the optimization strategy

more uniformly across all problem solving objects.

The strategy controllers can also take advantage of concurrency since the solutions they produce are completely independent of one another. The Tips object was given its own thread of control to allow for the future possibility of having it manage system-wide processing, such as operations for inter-strategy cooperation.

3.5.9 Distribution of Data

A given problem solving object does not and should not have knowledge about the type of the solutions produced by other objects so as not to create dependencies between problem solving objects. In this design, each problem solving object only has knowledge of the type of the solutions it produces. This is achieved by representing the solutions for each task and subtask as a distinct solution class. When passing its solutions to a higher level of problem solving, an object converts its solution to that of the higher level solution type. In particular, subtasks convert their solutions to task solutions of the appropriate type before sending them to their parent tasks and the strategy controllers combine their task solutions into a TravelPlan object before sending them to the Tips object. Thus, each problem solving object knows only about the type of solutions it produces and how to convert them to the type of solutions of the next higher layer of problem solving.

However, it is desirable to have some distribution of knowledge of the data itself, as in the blackboard architecture, in order to facilitate problem solving among the concurrent tasks. In the TIPS architecture, this is achieved through focussed messages passed amongst the problem solving objects. Subtasks know how to send messages to query other subtasks about the compatibility of their sub-solutions with those of other subtasks. This is more efficient and safer than sending their complete solutions to each other, because only the minimal amount of information needs to be sent and subtasks cannot inadvertently modify each other's solutions.

Chapter 4

Implementation Issues

In this chapter, we discuss the implementation of our design, the difficulties encountered, and our experience with the Java programming language that we used to implement TIPS. TIPS was developed on a Sun 4 workstation running the Solaris^{TM} 7 operating system. Alternatively, an Intel Pentium^{TM} machine running Windows NT^{TM} 4.0 could have been used. We decided to implement TIPS on the Sun workstation because the total number of user threads created when the three optimization strategies are run is equal to 40 and we were concerned that the performance of TIPS on the Intel machine would be unsatisfactory.

4.1 Travel Planner Implementation

The initial window that displays the travel constraints, requirements, and preferences a user can set and allows the user to start the search for a travel plan is shown in Figure 22 on page 63. At the top of the screen, the user must select the departure city and one or more destination cities. Below that, the user must set the departure and return date and time of the entire travel period which will be used to limit the travel interval when constructing travel plans. On the left hand side of the screen, the user should indicate the number of young children, older children, adults, seniors, and students not already included in the above number, that are going on the trip. In the center of the screen,

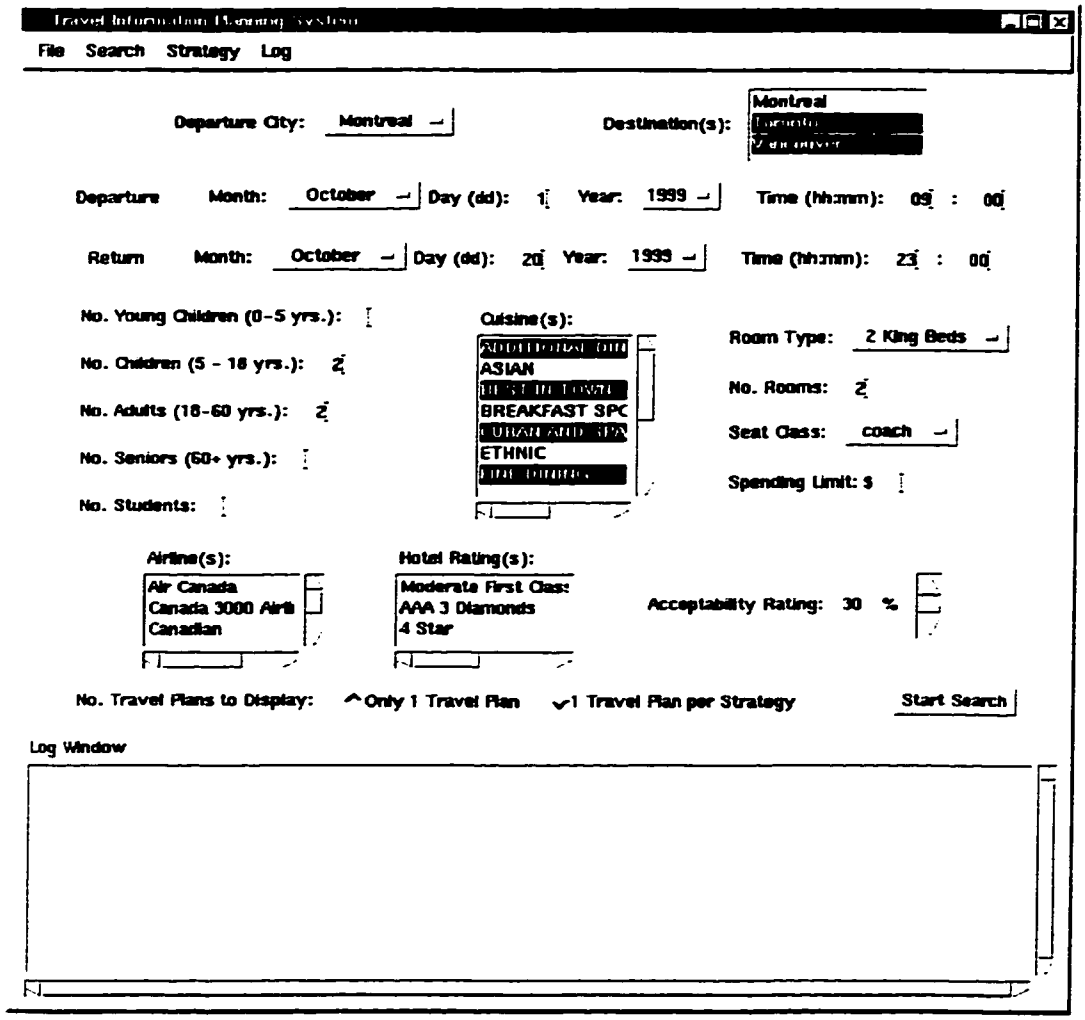

Figure 22: The TIPS Travel Constraints Entry Screen

the user can optionally set one or more of the travellers' preferred kind of restaurants to visit, in the *Cuisine(s)* field. If no cuisine is selected, any listed cuisine may be used for travel plans, otherwise, only the selected cuisine(s) will be used. To the right of the cuisine list, the number and size of beds required in each accommodation room must be set. The number of rooms required must also be set. Only rooms with the same number and size of beds can be requested. A transportation seat class of either coach, business, or first class must also be set. Optionally, a user may set an upper-limit on the total cost of a travel plan presented to the user in the *Spending Limit* field. If the user so chooses, one or more preferred airlines may be selected in the *Airline(s)* field. If any airlines are selected, only flights with those airlines will be presented to the user. Otherwise, all airlines may be used for airline flights. The user can also specify one or more ratings of hotels he/she wishes to stay in. Again, if any ratings are selected, only those hotels that have those ratings will be presented to the user. Otherwise, all hotels may be used for travel plans.

Under the *Strategy* menu, the user must select any combination of optimization strategies to run when searching for travel plans from among the following strategies: Minimize Cost, Minimize Distance Travelled, and Minimize Cost and Maximize Distance Travelled. If all strategies are to be used, the **All** item under the *Strategy* menu can be selected. In the *No. Travel Plans to Display* field, the user must either choose to have one travel plan displayed from the first optimization strategy that completes the search first, or choose to have one travel plan displayed from each optimization strategy that was selected.

In order to adjust the number of sub-solutions that are produced by the system, the user is provided with an *Acceptability Rating* field (see Section 3.4.1 for a discussion of reducing the number of sub-solutions). If the value of this field is set to $n\%$, only sub-solutions that are compatible with at least $n\%$ of sub-solutions in their counterpart subtasks are kept for creating travel plans. All other sub-solutions are discarded. If the value is set to 0%, no sub-solutions will be rejected. Setting the value of n to 100% is, in effect, requiring all sub-solutions to be completely compatible. Smaller values of n generally increase the number of solutions produced, while larger values of n reduce the number of solutions. The *Acceptability Rating* field was added for a user who is a designer or has

some knowledge of the design of the application and is experimenting with it, rather than for a user who is a traveller.

Once the travel constraints have been set, the user can start the search by selecting the **Start Search** button. The search can be halted at any time by selecting the **Stop** item under the *Search* menu. Once a search is stopped, it can only be started again from the beginning of the search process. When the search process is underway, the cursor in the constraints entry screen window is set to a timer. When the search is over, the cursor is returned to its normal state and the user can enter new constraints and start a new search again. To quit the application at any point, the user should select **Exit** under the *File* menu.

While the search is proceeding, various information about the status of the search is presented in the *Log Window* located at the bottom of the screen. The contents of this window may be erased using the **Clear** item under the *Log* menu. The log may be removed from the user's view or re-displayed using the **Close** or **Show** items, respectively, under the *Log* menu.

Once a travel plan has been found, it is displayed in a separate window on the screen such as the one shown in Figure 23 on page 66. The user can scroll the text of the travel plan in this window. The first line of text indicates the optimization strategy that was used to find this travel plan. This is followed by the detailed transportation, accommodation, and activity solution information, including cost and distance travelled, for each destination of the trip, in that order. The last solution is the transportation solution from the last destination back to the departure city. At the end of the text, the total cost and total distance travelled of the entire trip is shown. From the result window, the user may save the result to a file, close the result window only, or quit the application using the **Save**, **Close**, or **Exit** items, respectively under the *File* menu.

4.1.1 Travel Data

We have attempted to use as much realistic travel data as possible for our travel planner. The information was obtained from various travel planning web-sites on the Internet and entered in by hand into travel information files that are used by TIPS for building travel plans. The general

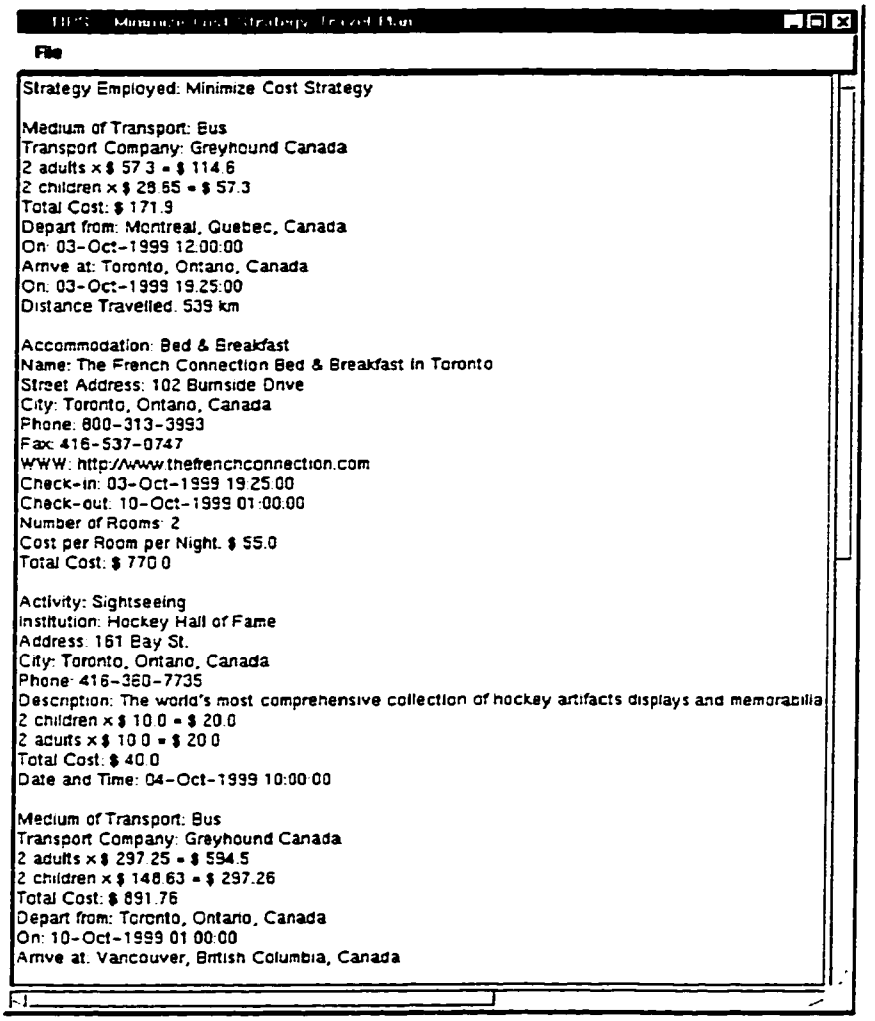

Figure 23: A TIPS Travel Plan Result Screen

institutional information is that of real institutions. The time schedule information is also valid, in as far as that information was made available. The pricing information is valid in as far as it was provided, but, in general, is only valid for the transportation trips and the restaurants. The booking availability data is artificially created. The data for distances between cities is accurate.

4.1.2 Evolution of the Implementation

An earlier version of our travel planner did not limit the number of sub-solutions constructed by a subtask. In that version, once a subtask had constructed its set of sub-solutions, it would send its results to other subtasks belonging to other strategies that were working on the same problem. For example, the MinDistBedBreakfast subtask would send its sub-solutions to the MinCostBedBreakfast subtask, in case it had not yet finished constructing its sub-solutions and could use the ones sent by MinDistBedBreakfast to speed up its progress. Because they both constructed the exact same sub-solutions and did not discard any, it made sense to have this kind of collaboration between strategies.

In order to accomplish this, each subtask would send its sub-solutions to the application controller (Tips) once they had been constructed and the controller would send them to the appropriate subtasks. If a subtask had already built its sub-solutions, it would simply continue whatever it was doing, otherwise the subtask would stop building sub-solutions, make a copy of the sub-solutions received from the controller, and continue by constructing complete solutions from the sub-solutions.

Later, when we decided to have each subtask reject sub-solutions that were not compatible with enough of the sub-solutions of other subtasks in the same strategy, it no longer made sense to have subtasks share their sub-solutions with each other. Each strategy is at different stages of progress and may reject sub-solutions not rejected by other strategies. Therefore, we no longer have subtasks broadcast their sub-solutions to other strategies.

4.2 Difficulties with the Implementation

Originally, we had planned to use a database to manage access to the travel data, rather than files. We started by using the Mini-SQL^{TM} (mSQL) database management system (DBMS), which is based on the client/server model. At first, we used an application programmer's interface (API) that was written in Java to access the database. Initial tests were performed using this DBMS and API by sending queries from a single-threaded application. We got the expected results. However, when the DBMS was queried from a multi-threaded application, exceptions would be raised from time to time in the Java Network API^{TM} code called from the database API. We continued using this DBMS and API, thinking that the problem could be resolved later, but it persisted.

We then decided to try another, more recent API for mSQL. It was a driver designed for the Java Database Connectivity^{TM} (JDBC) API. JDBC is a set of interfaces that are implemented by the JDBC driver classes that provide the connectivity and access to the particular database that it is designed for, in this case mSQL. The application programmer accesses the database through the standard JDBC interfaces. The advantage of using JDBC is that if, at some point during application development, a new DBMS is used, the programmer need only find a JDBC driver for the new DBMS and load it instead of the JDBC driver for the old DBMS. No other changes to the code are required, because the new DBMS is accessed through the same JDBC interfaces as the old DBMS was. The programmer must, of course, create a new database using the new DBMS. We chose to use the JDBC driver for mSQL to see if our problem was due to the API we were using to access the database. Unfortunately, we encountered similar problems using the new JDBC driver.

Upon consultation with an analyst, it was decided that we should try a new client/server DBMS called mySQL^{TM}, which is very similar to mSQL. We thought that the problem might be due to the implementation of the DBMS server. We found a JDBC driver for mySQL and tried our multi-threaded application with it, to no avail. We encountered similar problems with mySQL and the JDBC driver. After talking to the designer of one of the JDBC drivers, we learned that there were, perhaps, bugs in the underlying Java Network API in the version of the Java Development Kit^{TM}

(JDK) that we were using. It was suggested that we use a newer version of the JDK. Unfortunately, the newer version was not yet installed on the platform that we were using at that time and it would have required extra time to transfer to another platform that had the newer JDK, so the decision was taken to store the travel data on the local file system. The FileManager class was created to provide easier access to the data. FileManager does not create a separate thread to handle each read request from different threads in our application. This accounts for the fact that our application runs slower than it would if we used the mySQL DBMS, which is multi-threaded.

Another problem that was encountered was apparently due to running out of memory because a large number of solutions was produced during the search. It was difficult to find out what the cause of the problem actually was, because no exceptions were thrown. At first, it was suspected that the problem was due to deadlock. Deadlock problems were uncovered, but once the deadlock problem was resolved, the problem re-occurred. From looking at the thread dumps when the problem occurred, it was found that one thread's execution would halt in a constructor method. It was attempting to create an out-of-memory exception object, but it appeared that it did not have enough free memory to do so. Java does not require the programmer to explicitly free memory, but, rather, uses garbage collection to reclaim unused memory. As such, the programmer does not have to worry about forgetting to free unused memory, but never knows exactly when used memory will again become available for reuse.

After closely examining the number of solutions produced, we found that more solutions were being produced than was expected, including some duplicate solutions and some solutions that were not suitable. Assuming that our problem was due to running out of memory because of the number of solutions produced, we eliminated duplicate and unsuitable solutions. An example of an unsuitable solution would be a transportation solution which allows less than one full day of visiting a particular destination between the arrival and departure times. Such solutions are probably undesirable to the traveller, and so, they were not created.

TIPS usually creates more transportation solutions than accommodation or activity solutions. There are many different times that bus trips, flights, and train rides can depart at while there are

fewer variations possible for accommodation and activity solutions. Only one accommodation facility or activity is assigned for each destination in a solution. Variation due to length of stay is not possible initially for accommodation and activity solutions, since no date and time is assigned to them until later on during problem solving.

In order to reduce the overall number of solutions created, we first imposed an upper bound on the number of solutions created by each task, subtask, and strategy controller. The value of this limit was determined after experimenting with the system. Secondly, we introduced the mechanism described in Section 3.4.1 to eliminate the number of sub-solutions created by each subtask, and thereby reduce the number of solutions created. This seems to have eliminated the problem with limited memory storage.

4.3 Experience with Java

Java was chosen as the development language for this project because it provides a comprehensive API that allows the developer to write the graphical user interface (GUI), database access, and multi-threaded code all in Java, thereby not necessitating the programmer to implement the system in a patchwork fashion.

The Java GUI API used in this implementation was the Abstract Window Toolkit^{TM} (AWT) rather than the Swing^{TM} API, because Swing was only released in the newer version of the JDK, which we were not using. Conceptually, developing a user interface with Java is not difficult. Creating GUI components is as simple as instantiating an object and invoking a few methods on it. User input events are handled through call-backs to event handlers which are objects that implement an interface defined by the GUI API. Arranging components in a window in a satisfactory way is somewhat hampered with AWT, but improvements have been made with Swing.

As mentioned in Section 4.2, memory management is handled by the Java runtime system, thus easing the burden on the programmer. Garbage collection can also be initiated by the programmer. Another good feature of the language is the omission of explicit pointer types, such as are provided

by the C++ language. By treating all user-defined types as implicit references to memory locations, there is no need for pointer types, and no chance of making pointer arithmetic errors.

Java supports the creation of multi-threaded programs. Creating a thread is accomplished by extending the java::lang::Thread class and providing a run method in the derived class. The thread is started by instantiating this new class and invoking the start method on it. The start method is defined in java::lang::Thread and calls the run method of the subclass. The body of the run method is executed in a separate thread from which the thread was created. The synchronization of access to data is centered around the object. Threads run inside object methods. In order to prevent multiple threads from accessing the same instance variable of an object at the same time, critical regions can be protected with the **synchronized** keyword. For any instance of a class, only one thread can execute inside any method or block of statements that is labelled with the **synchronized** keyword at any one time. So, to ensure the consistency of some attribute accessed by multiple threads, the user must analyze every statement in which it is used and use the **synchronized** keyword, if mutual exclusion is necessary.

In addition, Java provides a mechanism for creating conditional critical regions which control the order of execution of threads inside synchronized methods via the wait and notify methods, which belong to the root object, Object. When a thread needs to wait for some event to occur, it can call wait, releasing its mutual exclusion lock on the object and suspending its execution, allowing another thread that is waiting to enter synchronized code to resume execution. A thread can call notify when it is finished an action and wishes to release the mutual exclusion lock, suspend itself, and allow a thread that was waiting for some event to occur to get ready for execution at the next available opportunity.

In our travel planner, the main data that is accessed by different threads is the sub-solutions and solutions of each subtask. The subtask that owns the solutions will read and write this data, while the other subtasks may want to read this data in order to obtain information about it. We restrict access to this data by creating mutually exclusive blocks of code wherever this data is accessed using the **synchronized** keyword. We also used the wait/notify mechanism to control access to the

file manager in order to permit only one file to be open at a time. If a thread wants to open and read from a file and a file is already open, the thread suspends itself by calling wait. When a thread is finished reading from a file, the thread calls notifyAll to release its lock and to allow all of the other waiting threads to have a chance to open and read from a file.

Brinch Hansen [1] argues that Java does not support monitors, but rather, "a programming style that imitates insecure monitors." He points out that the programmer must explicitly specify which methods are to be synchronized and must protect class variables using the **private** keyword. By default, a Java method is unsynchronized and class variables are public. It is up to the programmer to protect private data from being accessed by two different threads simultaneously with mutual exclusion. A Java thread can directly access another thread's public variables, indirectly modify its variables via unsynchronized methods, or cause deadlock by calling synchronized methods recursively. There is no built-in monitor construct that automatically provides compile-time checking of a multi-threaded program to ensure its correct execution. Consequently, it required significant effort on our part to eliminate problems of deadlock and synchronize access to private variables in our TIPS implementation.

Chapter 5

Output Results of the Travel Planner

In this chapter we will examine the input and output of three sample scenarios for TIPS. For each scenario, we will present the travel constraints submitted to TIPS followed by the travel plan result of the scenario, divided into three parts. Only one optimization strategy result is presented for each scenario in order to simplify our discussion. We will then discuss the quality of the result.

After a discussion of each resulting travel plan, we will present some statistics gathered by TIPS for the scenario and analyze them, in order to get more insight into what actions led to the result that was obtained. These statistics include counts of the numbers of solutions produced at different stages of the search and the elapsed times of different stages of the search for each problem solving component. The stages of the search that we list statistics for are:

1. production of sub-solutions by subtasks (sub-solutions)
2. creation of the initial set of subtask solutions (initial solutions)
3. the entire process of producing subtask solutions (subtask total)
4. combining subtask solutions or task solutions to produce new solutions (mixing solutions)
5. the NonLocalTransportation task finding the best stay time intervals (computing intervals)

6. setting dates and times of solutions according to the best stay time intervals (assigning intervals)

7. the entire process of producing task solutions (task total)

8. determining the best travel plan after combining task solutions (finding best plan)

9. the entire process of strategy problem solving from start to finish (strategy total)

For the *sub-solutions* search stage we report two values in the *No. Solutions* column. The first value represents the number of sub-solutions that were rejected because they did not meet the acceptability rating factor for compatibility with other sub-solutions. The second value represents the number of sub-solutions that were actually created. The other values in this column of the statistics tables represent the number of viable solutions at the end of each search stage.

In a concurrent system, the elapsed times are not particularly meaningful for comparing different problem solving objects with each other, since other threads may have been running for some period of this time, but they are useful for demonstrating the proportion of time each object devotes to a particular stage of problem solving. The elapsed times presented for the *initial solutions* stage includes the time used for the *sub-solutions* stage, because building the sub-solutions is part of building the solutions and the Accommodation and Activities subtasks perform these actions in parallel. The total elapsed time reported for subtasks is the total time the subtask executes. The total elapsed time of tasks measures the time that elapses between when the subtasks are started until when the task sends its solutions to the strategy controller. The total elapsed time of a strategy is the time from when the entire strategy is started until the strategy sends its travel plan to the system controller.

5.1 General Discussion of the Results

Before we present the scenarios and results, a few comments should be made about the travel planner's output in general. In the results presented for each scenario, the check-in and check-out times

assigned to accommodation stays coincide exactly with the arrival and departure times for the transportation to and from the given destination. This, of course, is not realistic in practice. The times were assigned in this fashion because we do not account for time delay for transportation between the point of arrival in a particular city and the location of the accommodation. Had we added a local transportation task to our system to solve this aspect of the problem, these realities would have been reflected in the results. We could have assigned a fixed time delay between arrival times and check-in times, but this would have been completely arbitrary and, in effect, equivalent to allowing for no time delay at all.

The date and times chosen to attend activities is typically on or soon after the time of arrival at the destination. Again, we do not provide solutions for transportation between accommodation facilities and activity locations, and assigning a fixed time delay is arbitrary. The activity date and time is assigned by using the first available time-slot for the activity starting from the arrival date at the destination. Hence, the activity dates and times are usually close to the arrival time in the city.

The time length of the entire trip presented to the user typically does not coincide exactly with the requested travel interval. However, the trip does fall within the boundary specified. The reason why the actual trip interval often does not closely follow the desired travel interval is that the optimization strategies we have chosen to implement in this system optimize exclusively on the constraints of the strategy. In fact, the minimum cost constraint employed in two of the strategies, for example, is diametrically opposed to that of maximum trip duration, because trips of longer duration are usually more expensive.

5.2 Scenario 1

Table 4 on page 76 shows the travel constraints submitted to TIPS for our first scenario. Two destinations are selected, and we run all three strategies, but request only one result. In this scenario, we set the acceptability rating to 0% so we are, in effect, not limiting the number of sub-solutions created.

Departure City	Montreal, Quebec, Canada
Destination(s)	Toronto, Ontario, Canada; Vancouver, British Columbia, Canada
Departure Date & Time	01-Oct-1999 09:00:00
Return Date & Time	31-Oct-1999 22:00:00
No. Young Children	1
No. Children	1
No. Adults	1
No. Seniors	1
No. Students	1
Cuisine(s)	ADDITIONAL DINING EXPERIENCES, BEST IN TOWN, FRENCH, HOT AND TRENDY, ITALIAN, ITALIAN AND FRENCH, LOCAL FAVORITES
Room Type	2 Single Beds
No. Rooms	3
Seat Class	coach
Spending Limit	$ 8000.0
Airline(s)	Air Canada, Canadian
Hotel Rating(s)	Moderate First Class/Superior Tourist Class
Acceptability Rating	0%
Strategies	Minimize Cost, Minimize Distance Travelled, Minimize Cost & Maximize Distance Travelled
No. Travel Plans	1

Table 4: Travel Constraints for Scenario 1

Figures 24, 25, and 26 on pages 78, 79, and 80 contain the travel plan returned by the system, which happened to be produced by the MinCost strategy. The destination order selected by this strategy was Toronto–Vancouver, however both orders were ranked the same overall. The key thing to note from this result is that bed and breakfast and motel solutions were chosen over hotel solutions because they tend to be less expensive. Also, bus solutions were chosen over other transportation solutions, because they are usually less expensive. Low cost sightseeing solutions were also chosen. The total duration of the trip is 16 days out of a possible 31 days maximum. This is to be expected since the optimization strategy was attempting to find the least cost solution, and trips of longer duration tend to cost more. For these reasons, the travel plan result seems to be a good one.

In Table 5 on page 81, we present the statistics produced by TIPS for this scenario. As expected, no sub-solutions were rejected because we set the acceptability rating to 0%. Because we specified a long travel interval, many transportation solutions were produced. All NonLocalTransportation subtasks produced the maximum number of solutions allowed. MinCostTrain aborted because it only produced solutions for the Vancouver–Toronto destination order. The subtasks determined collectively that the best destination order was Toronto–Vancouver and MinCostTrain had no solutions with that destination order. This demonstrates the problem with using a breadth-first search without using the limitation of sub-solutions method. MinCostHotel also aborted because it only produced one sub-solution, and so it was not able to create any initial solutions. The fact that MinCostEvents only produced one solution explains why no event solutions were chosen for the travel plan.

As expected, more NonLocalTransportation solutions were produced than Accommodation and Activities solutions, because more variation is possible for NonLocalTransportation solutions than with Accommodation and Activities solutions. MinCostNonLocalTransportation produced the maximum number of solutions for a task. As described in Section 3.4.2.2, the Accommodation and Activities tasks only send up to a fixed number of best solutions to the strategy controller. This fixed limit is twice the number of solutions that they receive from their subtasks. MinCostAccommodation had to eliminate many solutions because they did not satisfy the stay intervals. MinCostActivities, on the other hand, had more solutions to send to the strategy controller than it was allowed to

Strategy Employed: Minimize Cost Strategy

Medium of Transport: Bus
Transport Company: Greyhound Canada
1 adults x $ 57.3 = $ 57.3
1 students x $ 40.13 = $ 40.13
1 seniors x $ 42.99 = $ 42.99
1 children x $ 28.65 = $ 28.65
1 young children x $ 28.65 = $ 28.65
Total Cost: $ 197.71999
Depart from: Montreal, Quebec, Canada
On: 04-Oct-1999 06:00:00
Arrive at: Toronto, Ontario, Canada
On: 04-Oct-1999 16:00:00
Distance Travelled: 539 km

Accommodation: Bed & Breakfast
Name: Jarvis House Downtown Toronto Bed and Breakfast Inn
Street Address: 344 Jarvis Street
City: Toronto, Ontario, Canada
Phone: 416-975-3838
Fax: 416-975-9808
WWW: http://www.jarvishouse.com
Check-in: 04-Oct-1999 16:00:00
Check-out: 10-Oct-1999 01:00:00
Number of Rooms: 3
Cost per Room per Night: $ 55.0
Total Cost: $ 990.0

Activity: Sightseeing
Institution: Hockey Hall of Fame
Address: 161 Bay St.
City: Toronto, Ontario, Canada
Phone: 416-360-7735
Description: The world's most comprehensive collection of hockey artifacts displays and memorabilia including state-of-the-art and interactive exhibits and technology.
1 young children x $ 5.5 = $ 5.5
1 children x $ 10.0 = $ 10.0
1 adults x $ 10.0 = $ 10.0
1 seniors x $ 5.5 = $ 5.5
1 students x $ 10.0 = $ 10.0
Total Cost: $ 41.0
Date and Time: 04-Oct-1999 16:00:00

Figure 24: Travel Plan Result for Scenario 1 (Part 1)

Medium of Transport: Bus
Transport Company: Greyhound Canada
1 adults x $ 297.25 = $ 297.25
1 students x $ 267.53 = $ 267.53
1 seniors x $ 267.53 = $ 267.53
1 children x $ 148.63 = $ 148.63
1 young children x $ 148.63 = $ 148.63
Total Cost: $ 1129.5701
Depart from: Toronto, Ontario, Canada
On: 10-Oct-1999 01:00:00
Arrive at: Vancouver, British Columbia, Canada
On: 13-Oct-1999 21:05:00
Distance Travelled: 4424 km

Accommodation: Motel
Name: Homaway Inns Ltd
Street Address: 1203 Broughton St
City: Vancouver, British Columbia, Canada
Phone: 604-684-7811
Check-in: 13-Oct-1999 21:05:00
Check-out: 17-Oct-1999 06:45:00
Number of Rooms: 3
Cost per Room per Night: $ 54.0
Total Cost: $ 648.0

Activity: Sightseeing
Institution: Stanley Park
Address: Downtown
City: Vancouver, British Columbia, Canada
Phone: (604) 257-8531
Description: Named after Lord Stanley the Governor General of Canada from 1888-93 (also the donator of the original Stanley Cup). One of the finest natural parks in North America. Recreationists can swim golf lawn bowl play tennis jog or bike on nearly 80km of trails and roads contained in the tract. There is also a children's zoo rose garden miniature steam railway and live shows at the Malkin Bowl in July and August.
1 young children x $ 0.0 = $ 0.0
1 children x $ 0.0 = $ 0.0
1 adults x $ 0.0 = $ 0.0
1 seniors x $ 0.0 = $ 0.0
1 students x $ 0.0 = $ 0.0
Total Cost: $ 0.0
Date and Time: 13-Oct-1999 22:00:00

Figure 25: Travel Plan Result for Scenario 1 (Part 2)

Medium of Transport: Bus
Transport Company: Greyhound Canada
1 adults x $ 297.46 = $ 297.46
1 students x $ 267.72 = $ 267.72
1 seniors x $ 267.72 = $ 267.72
1 children x $ 148.73 = $ 148.73
1 young children x $ 148.73 = $ 148.73
Total Cost: $ 1130.36
Depart from: Vancouver, British Columbia, Canada
On: 17-Oct-1999 06:45:00
Arrive at: Montreal, Quebec, Canada
On: 20-Oct-1999 09:20:00
Distance Travelled: 4963 km

Total Cost of Trip = $ 4136.65
Total Distance Travelled = 9926 km

Figure 26: Travel Plan Result for Scenario 1 (Part 3)

so it had to eliminate some by choosing the least cost solutions.

The subtasks all finished in roughly the same amount of time (15 sec). This is to be expected, because all subtasks must wait for the overall destination order to be computed, after which they all send the solutions matching that order to their parent task. All of the tasks finished in about 83 sec. This also is to be expected, because all of the tasks must wait for the stay intervals to be computed, and then send solutions matching those intervals to their strategy controller.

5.3 Scenario 2

The travel constraints for the second scenario are shown in Table 6 on page 82. In this scenario, we choose Toronto as the departure city and we specify a shorter travel time interval than we did in the previous scenario. Note that we have set the acceptability rating to 50% in an attempt to reduce the number of sub-solutions produced somewhat. This time, we run only the MinDist strategy and request one travel plan from the system.

Figures 27, 28, and 29 on pages 83, 84, and 85 show the travel plan that is chosen by TIPS. Note that the destination order does not play a significant role in determining the minimum distance

Subtask/Task/Strategy Controller	Search Stage	No. Solutions	Elapsed Time (ms)
MinCostBus	sub-solutions	0/520	1990
	initial solutions	101	3414
	subtask total	101	15399
MinCostPlane	sub-solutions	0/80	492
	initial solutions	100	745
	subtask total	77	15490
MinCostTrain	sub-solutions	0/180	13238
	initial solutions	101	13998
	subtask total	N/A	N/A
MinCostHotel	sub-solutions	0/1	14563
	initial solutions	N/A	N/A
	subtask total	N/A	N/A
MinCostMotel	sub-solutions	0/6	8762
	initial solutions	18	8766
	subtask total	9	15496
MinCostBedBreakfast	sub-solutions	0/6	12551
	initial solutions	18	12599
	subtask total	9	15472
MinCostDining	sub-solutions	0/8	1039
	initial solutions	14	1044
	subtask total	7	15391
MinCostSightseeing	sub-solutions	0/6	14995
	initial solutions	18	14999
	subtask total	9	15413
MinCostEvents	sub-solutions	0/2	3685
	initial solutions	2	3900
	subtask total	1	15447
MinCostNonLocalTransportation	mixing solutions	400	47814
	computing intervals	400	18195
	task total	1	82368
MinCostAccommodation	mixing solutions	36	363
	assigning intervals	3	3
	task total	3	83452
MinCostActivities	mixing solutions	55	158
	assigning intervals	40	428
	task total	34	83917
MinCostStrategy	mixing solutions	102	6
	finding best plan	1	1
	strategy total	1	91536

Table 5: Statistics for Scenario 1

Departure City	Toronto, Ontario, Canada
Destination(s)	Montreal, Quebec, Canada; Vancouver, British Columbia, Canada
Departure Date & Time	15-Nov-1999 10:00:00
Return Date & Time	30-Nov-1999 20:30:00
No. Young Children	0
No. Children	0
No. Adults	1
No. Seniors	1
No. Students	0
Cuisine(s)	ASIAN, BREAKFAST SPOTS, CUBAN AND SPANISH, ETHNIC, FINE DINING, LATE NIGHT/BREAKFAST
Room Type	1 Single Bed
No. Rooms	2
Seat Class	first
Spending Limit	$ 5100.0
Airline(s)	Air Canada, Canada 3000 Airlines Ltd., Canadian
Hotel Rating(s)	AAA 3 Diamonds, 4 Star
Acceptability Rating	50%
Strategies	Minimize Distance Travelled
No. Travel Plans	1

Table 6: Travel Constraints for Scenario 2

Strategy Employed: Minimize Distance Travelled Strategy

Medium of Transport: Bus
Transport Company: Greyhound Canada
1 adults x $ 57.3 = $ 57.3
1 seniors x $ 42.99 = $ 42.99
Total Cost: $ 100.29
Depart from: Toronto, Ontario, Canada
On: 19-Nov-1999 00:20:00
Arrive at: Montreal, Quebec, Canada
On: 19-Nov-1999 09:20:00
Distance Travelled: 539 km

Accommodation: Bed & Breakfast
Name: Bed & Breakfast Downtown
Street Address: 3458 Laval Ave
City: Montreal, Quebec, Canada
Phone: 514-289-9749
Check-in: 19-Nov-1999 09:20:00
Check-out: 23-Nov-1999 09:20:00
Number of Rooms: 2
Cost per Room per Night: $ 47.3
Total Cost: $ 378.4

Activity: Dining
Institution: Katsura
Address: 2170 De la Montagne
City: Montreal, Quebec, Canada
Phone: 514-849-1172
Description: ASIAN
1 adults x $ 15.0 = $ 15.0
1 seniors x $ 15.0 = $ 15.0
Total Cost: $ 30.0
Date and Time: 19-Nov-1999 12:00:00

Figure 27: Travel Plan Result for Scenario 2 (Part 1)

Medium of Transport: Plane
Transport Company: Canadian
Trip Number: 911
Seat Class: first
Direct Trip: Yes
1 adults x $ 348.5 = $ 348.5
1 seniors x $ 348.5 = $ 348.5
Total Cost: $ 697.0
Depart from: Montreal, Quebec, Canada
On: 23-Nov-1999 09:20:00
Arrive at: Vancouver, British Columbia, Canada
On: 23-Nov-1999 11:45:00
Distance Travelled: 3672 km

Accommodation: Motel
Name: Riviera Motor Inn
Street Address: 431 Robson St.
City: Vancouver, British Columbia, Canada
Phone: 604-685-1301
Check-in: 23-Nov-1999 11:45:00
Check-out: 26-Nov-1999 06:45:00
Number of Rooms: 2
Cost per Room per Night: $ 52.0
Total Cost: $ 312.0

Activity: Sightseeing
Institution: Museum of Anthropology
Address: 6393 N.W. Marine Dr.
City: Vancouver, British Columbia, Canada
Phone: (604) 822-3825
Description: The museum houses a significant collection of artwork from Northwest Coast First Nations. Totem poles canoes and other artifacts from daily aboriginal life can also be viewed as well as other pieces from across the globe.
1 adults x $ 6.0 = $ 6.0
1 seniors x $ 5.0 = $ 5.0
Total Cost: $ 11.0
Date and Time: 23-Nov-1999 12:00:00

Figure 28: Travel Plan Result for Scenario 2 (Part 2)

Medium of Transport: Bus
Transport Company: Greyhound Canada
1 adults x $ 297.25 = $ 297.25
1 seniors x $ 267.53 = $ 267.53
Total Cost: $ 564.78
Depart from: Vancouver, British Columbia, Canada
On: 26-Nov-1999 06:45:00
Arrive at: Toronto, Ontario, Canada
On: 29-Nov-1999 05:50:00
Distance Travelled: 4424 km

Total Cost of Trip = $ 2093.4702
Total Distance Travelled = 8635 km

Figure 29: Travel Plan Result for Scenario 2 (Part 3)

solution for these travel constraints since the departure city is Toronto, and one has to travel through Toronto to travel between both destinations. On the other hand, one might have expected the system to prefer plane solutions over other NonLocalTransportation solutions, since they usually are more direct. A flight is chosen for the second leg of the trip, but not the others. The decision to choose two bus trips for the other legs could be due to the restrictions placed on the airlines or the spending limit, as flights tend to be more expensive and first class flights were requested. The total distance travelled, however, seems to be fairly low (see Figure 29). No hotel solutions were chosen, perhaps because of the restriction on the hotel ratings. Overall, it appears that this travel plan is reasonable for the optimization strategy employed.

As shown in Table 7 on page 86, MinDistBus and MinDistPlane both created the maximum number of solutions allowed, but MinDistTrain aborted. This is because it only found solutions for the Vancouver–Toronto destination order, and the other order was chosen collectively. Setting the acceptability rating to 50% only led to rejecting one motel solution. No sub-solutions of other subtasks were rejected. This could have been because most sub-solutions were compatible or other sub-solutions had not yet been built to compare the sub-solutions against. MinDistEvents aborted because it only built one sub-solution, which is not enough to make a complete solution. The strategy controller had many more travel plans (800) to choose from in this scenario as compared to the

Subtask/Task/Strategy Controller	Search Stage	No. Solutions	Elapsed Time (ms)
MinDistBus	sub-solutions	0/275	3945
	initial solutions	101	4453
	subtask total	68	4766
MinDistPlane	sub-solutions	0/73	2551
	initial solutions	101	3091
	subtask total	60	4770
MinDistTrain	sub-solutions	0/131	2986
	initial solutions	15	3393
	subtask total	N/A	N/A
MinDistHotel	sub-solutions	0/2	1299
	initial solutions	2	1305
	subtask total	1	4790
MinDistMotel	sub-solutions	1/5	4515
	initial solutions	12	4520
	subtask total	6	4869
MinDistBedBreakfast	sub-solutions	0/14	200
	initial solutions	66	209
	subtask total	33	4789
MinDistDining	sub-solutions	0/4	419
	initial solutions	8	423
	subtask total	4	4871
MinDistSightseeing	sub-solutions	0/6	2764
	initial solutions	18	2769
	subtask total	9	4874
MinDistEvents	sub-solutions	0/1	2083
	initial solutions	N/A	N/A
	subtask total	N/A	N/A
MinDistNonLocalTransportation	mixing solutions	400	28315
	computing intervals	400	4995
	task total	1	38263
MinDistAccommodation	mixing solutions	90	1031
	assigning intervals	40	4
	task total	40	38249
MinDistActivities	mixing solutions	25	20
	assigning intervals	20	75
	task total	20	38279
MinDistStrategy	mixing solutions	800	11
	finding best plan	1	38
	strategy total	1	44978

Table 7: Statistics for Scenario 2

first one, because more accommodation solutions were built that satisfied the stay intervals determined collectively. As expected, the execution times were much shorter for this scenario compared to the first one, because only one optimization strategy was run.

5.4 Scenario 3

The travel constraints for the final scenario are presented in Table 8. Here, we choose Vancouver as the departure city and specify a shorter travel interval window than we did in the previous two scenarios. We run the MinCostMaxDist strategy only and set the acceptability rating to 60%.

Departure City	Vancouver, British Columbia, Canada
Destination(s)	Montreal, Quebec, Canada; Toronto, Ontario, Canada
Departure Date & Time	23-Dec-1999 06:00:00
Return Date & Time	03-Jan-2000 21:30:00
No. Young Children	0
No. Children	2
No. Adults	2
No. Seniors	0
No. Students	0
Cuisine(s)	
Room Type	2 Double Beds
No. Rooms	2
Seat Class	business
Spending Limit	$ 5500.0
Airline(s)	
Hotel Rating(s)	
Acceptability Rating	60%
Strategies	Minimize Cost & Maximize Distance Travelled
No. Travel Plans	1

Table 8: Travel Constraints for Scenario 3

Figures 30, 31, and 32 on pages 88, 89, and 90 display the travel plan returned for this scenario. The destination order chosen for this scenario is Toronto–Montreal. This solution doesn't appear to be a very good solution. First of all, three plane flights were chosen. Since flights are usually more expensive and of shorter distance than other methods of transportation, we might have expected

Strategy Employed: Minimize Cost and Maximize Distance Travelled Strategy

Medium of Transport: Plane
Transport Company: United Airlines
Trip Number: 3372
Seat Class: business
Direct Trip: Yes
2 adults x $ 271.0 = $ 542.0
2 children x $ 271.0 = $ 542.0
Total Cost: $ 1084.0
Depart from: Vancouver, British Columbia, Canada
On: 23-Dec-1999 09:00:00
Arrive at: Toronto, Ontario, Canada
On: 23-Dec-1999 16:15:00
Distance Travelled: 3336 km

Accommodation: Hotel
Name: BAY BLOOR EXECUTIVE SUITES
Street Address: 1101 BAY STREET
City: Toronto, Ontario, Canada
Postal/Zip Code: M5S-2W8
Phone: 416-968-3878
Fax: 1-416-968-7385
WWW: www.travelweb.com/TravelWeb/lm/common/lsc.html
Check-in: 23-Dec-1999 16:15:00
Check-out: 31-Dec-1999 11:00:00
Room Type: 2 Double Beds
Number of Rooms: 2
Cost per Room per Night: $ 123.93
Total Cost: $ 1982.88

Activity: Dining
Institution: Zola
Address: 162 Cumberland St.
City: Toronto, Ontario, Canada
Phone: 416-515-1222
Description: FRENCH
2 children x $ 50.0 = $ 100.0
2 adults x $ 50.0 = $ 100.0
Total Cost: $ 200.0
Date and Time: 23-Dec-1999 17:00:00

Figure 30: Travel Plan Result for Scenario 3 (Part 1)

Medium of Transport: Plane
Transport Company: Canadian
Trip Number: 1950
Seat Class: business
Direct Trip: Yes
2 adults x $ 80.5 = $ 161.0
2 children x $ 80.5 = $ 161.0
Total Cost: $ 322.0
Depart from: Toronto, Ontario, Canada
On: 31-Dec-1999 11:00:00
Arrive at: Montreal, Quebec, Canada
On: 31-Dec-1999 12:15:00
Distance Travelled: 504 km

Accommodation: Hotel
Name: Courtyard Montreal Downtown
Street Address: 410 Rue Sherbrooke Ouest
City: Montreal, Quebec, Canada
Postal/Zip Code: H3A 1B3
Phone: 1-514-844-8851
Fax: 1-514-844-0912
WWW: www.courtyard.com
Check-in: 31-Dec-1999 12:15:00
Check-out: 02-Jan-2000 06:30:00
Room Type: 2 Double Beds
Number of Rooms: 2
Cost per Room per Night: $ 124.68
Total Cost: $ 498.72

Activity: Dining
Institution: Les Halles
Address: 1450 Rue Crescent
City: Montreal, Quebec, Canada
Phone: 514-844-2328
Description: BEST IN TOWN
2 children x $ 35.0 = $ 70.0
2 adults x $ 35.0 = $ 70.0
Total Cost: $ 140.0
Date and Time: 31-Dec-1999 18:00:00

Figure 31: Travel Plan Result for Scenario 3 (Part 2)

Medium of Transport: Plane
Transport Company: Air Canada
Trip Number: 485/105
Seat Class: business
Direct Trip: No
2 adults x $ 199.5 = $ 399.0
2 children x $ 199.5 = $ 399.0
Total Cost: $ 798.0
Depart from: Montreal, Quebec, Canada
On: 02-Jan-2000 06:30:00
Arrive at: Vancouver, British Columbia, Canada
On: 02-Jan-2000 10:45:00
Distance Travelled: 3840 km

Total Cost of Trip = $ 5025.6
Total Distance Travelled = 7680 km

Figure 32: Travel Plan Result for Scenario 3 (Part 3)

other NonLocalTransportation solutions to be selected for the travel plan. Secondly, two hotel solutions were chosen. Again, this was not expected since we are trying to minimize cost. Finally, one might have expected cheaper activity solutions to be chosen, rather than two expensive restaurants. The total cost is rather high and the total distance travelled is rather low compared to the travel plans presented for the other scenarios which are for longer travel intervals. We would not expect this to be the case for the MinCostMaxDist strategy. We will now look at the statistics for this scenario in order to try to gain more insight into this result.

Table 9 on page 91 shows the statistics produced for this scenario. The MinCostMaxDistTrain subtask aborted before completely building its initial list of solutions. After closer examination, it was found that this was due to the fact that it did not have any sub-solutions for the final legs of the trip. Some motel and several bed and breakfast solutions were produced as alternatives to hotel solutions. Some activity sub-solutions were rejected because of the acceptability rating setting and this was the reason why the sightseeing and events subtasks aborted and we only saw dining solutions in the result. Looking closely at the number of solutions produced by the NonLocalTransportation, Accommodation, and Activities tasks, we see that the maximum number of travel plans that can be

Subtask/Task/Strategy Controller	Search Stage	No. Solutions	Time (ms)
MinCostMaxDistBus	sub-solutions	0/187	634
	initial solutions	97	1024
	subtask total	21	5054
MinCostMaxDistPlane	sub-solutions	0/52	2198
	initial solutions	82	2386
	subtask total	21	5060
MinCostMaxDistTrain	sub-solutions	0/73	2941
	initial solutions	N/A	N/A
	subtask total	N/A	N/A
MinCostMaxDistHotel	sub-solutions	0/6	1473
	initial solutions	18	1476
	subtask total	9	4900
MinCostMaxDistMotel	sub-solutions	0/6	4724
	initial solutions	18	4730
	subtask total	9	4900
MinCostMaxDistBedBreakfast	sub-solutions	0/14	3402
	initial solutions	66	3409
	subtask total	33	5199
MinCostMaxDistDining	sub-solutions	2/13	2572
	initial solutions	80	2610
	subtask total	40	4900
MinCostMaxDistSightseeing	sub-solutions	4/2	3572
	initial solutions	N/A	N/A
	subtask total	N/A	N/A
MinCostMaxDistEvents	sub-solutions	3/0	4497
	initial solutions	N/A	N/A
	subtask total	N/A	N/A
MinCostMaxDistNonLocalTransportation	mixing solutions	135	1910
	computing intervals	135	699
	task total	1	7868
MinCostMaxDistAccommodation	mixing solutions	153	4599
	assigning intervals	52	5
	task total	52	9962
MinCostMaxDistActivities	mixing solutions	40	27
	assigning intervals	40	30
	task total	40	7859
MinCostMaxDistStrategy	mixing solutions	910	14
	finding best plan	1	656
	strategy total	1	13314

Table 9: Statistics for Scenario 3

built by combining these solutions is $1 \times 52 \times 40 = 2080$ travel plans. However, only 910 travel plans were created by making combinations. This is the maximum number of travel plans that can be created by a strategy controller, which we impose in order to not exceed memory constraints. Thus, not all possible combinations of solutions were examined by the strategy controller. This might explain why no cheaper bus trips were chosen over more expensive plane flights and why cheaper motels and bed and breakfasts were not chosen over more expensive hotels. This again highlights a problem with our travel planner. Too many solutions are explored, even for relatively short travel intervals. Perhaps if the acceptability rating were increased further, more unsuitable sub-solutions would be eliminated initially and all of the remaining sub-solutions could be explored to find a cheaper travel plan.

5.5 Non-determinacy of the Results

As we mentioned in Section 4.1.1, the travel data used by TIPS was created manually and stored in a file. It is static. One might surmise then that given the same travel constraints in two trial runs, the travel planner should produce the same travel plan both times. This is not the case, however, as we will now explain. In the case where only one travel plan is requested and more than one strategy is selected for execution, it cannot be determined which strategy will finish first, because of the concurrent nature of the system. The outcome depends entirely upon the scheduling of execution of the threads by the operating system.

We will now examine the case where we request a result from each strategy. In this case, if the acceptability rating is set to a value greater than 0%, the outcomes for separate executions with the same travel constraints given may not be the same. This is because the compatibility of sub-solutions depend on the set of sub-solutions of other tasks at the given time. Due to the concurrency of the system, the results of determining the compatibility of sub-solutions may differ between executions of the system, and therefore, the solutions produced may differ.

Finally, let us examine the case where the acceptability rating is set to 0%. In this case, the

solutions produced by each subtask are the same, given the same travel constraints, because no sub-solutions are eliminated when the acceptability rating is set to 0%. However, once again, due to the concurrent execution of the system, the order in which subtasks finish and send their solutions to their parent task may differ between executions. The parent task then mixes these solutions together and obtains the same set of mixed solutions as in any other execution. However, the order of the solutions is different. When the transportation task computes the best stay intervals, it chooses the highest ranked interval. In the case where two intervals have the same rank, the first highest ranked interval is chosen. Since the order of the solutions, and hence, the order of the intervals may differ between executions, the interval chosen may be different. Thus, different travel plans may be found. As is the case with many other optimization of constraints problems, there is often more than one equally optimal solution to the problem.

5.6 Approximation of the Optimal Solution

The results returned by TIPS are only approximations to the optimal solution. This is because we may initially eliminate optimal sub-solutions for further use based on their low compatibility with other sub-solutions available at the time. Not all sub-solutions may have been computed yet and so the compatibility results obtained may lead us down the wrong search path. Also, the adequacy of a sub-solution is measured against an arbitrarily assigned acceptability rating. Although a sub-solution may not be compatible with a certain percentage of other sub-solutions, it still may be compatible with some sub-solutions and actually be part of an optimal solution.

We also may not consider some partial solutions that are part of the global optimal solution when we have to stop our search because we have created the maximum number of solutions permitted at any stage of the search. Furthermore, we may possibly eliminate optimal solutions when we choose the best destination order or stay interval collectively. Nevertheless, the approximate solutions obtained by our travel planner are often satisfactory for the user, given that he/she may be more concerned with obtaining useful results that he/she can pick and choose from, rather than optimal

solutions that are unsatisfactory. As can be seen from the travel plan results presented for the scenarios in this chapter, they are somewhat useful to a traveller.

Chapter 6

Evaluation and Conclusions

In this chapter we will discuss the advantages and disadvantages of using the presented architecture for a travel planning application specifically and for other optimization problems in general. We outline some possible future improvements that could be made to the design and present the conclusions of our research.

6.1 Advantages of this Architecture

One of the main advantages of this architecture is that it permits the designer to extend the application rather easily. New optimization strategies can be added effortlessly by extending the abstract classes which already provide most of the methods required to implement the strategy. With a bit more effort, new subtask objects can be added under the existing tasks. Again, many required methods already exist in the Task abstract class and its derived abstract classes. The incremental cost, in terms of the amount of additional code required to implement a new strategy or subtask, is small because of the amount of code reuse that can be achieved.

Another main advantage of this architecture is that it capitalizes on the concurrency inherent in the problem. The overall solution to the TPP comprises specific solutions to many smaller problems. This architecture allows the designer to exploit concurrency present in the problem, especially in the

case where the travel data required by each subtask is disjoint and distributed, perhaps on different machines, which is not uncommon in practise.

This architecture promotes separation of concerns. While creating a new subtask, the designer need not think too much about how its solutions will fit with other subtask solutions. The combination of like subtask solutions is done at a higher level of problem solving. The designer can focus exclusively on producing solutions for a given subtask. In this manner, the system can be built incrementally, adding one subtask at a time to TIPS.

In comparison with the blackboard architecture, the consistency of a subtask's local data is managed by the subtask, so there is less danger of another subtask inadvertently corrupting the data or interleaving access to it with other subtasks. We do not have to synchronize subtasks to access a global data structure, as is required in the blackboard approach. The task of ensuring consistency of the data is simplified, because each subtask protects its own data with synchronized methods.

6.2 Disadvantages of this Architecture

The main disadvantage of this architecture for solving the TPP is that it is not scalable. Due to the combinatorial explosion of possible solutions that results from selecting additional destinations and increasing the travel interval length, TIPS does not perform well. When the limit on the maximum number of intermediate solutions that can be created is reached, alternative solutions, and even, alternative destination orders may not be explored. This reflects the inadequacy of using a breadth-first search for solving large instances of the TPP.

The search strategy for solutions does not guarantee the optimal solution, as was explained in Section 5.6. It is an approximation to the optimal solution. Again, this is because we must limit the number of solutions produced by the breadth-first search, and because we generate as many possible solutions as we can first and then eliminate solutions based on destination order and stay interval times. This may result in the inadvertent elimination of optimal solutions.

Compared to the blackboard architecture, our architecture expends additional overhead for inter-task communication required to build coherent and compatible solutions. This is due to the fact that the data is distributed rather than centralized. Of course, knowledge sources do communicate with each other via the blackboard, but in a more efficient way. They have access to the exact information that they need to work on the problem. With our architecture, on the other hand, a subtask only has some information about the solutions produced by other subtasks and that information must be obtained through several method calls to different subtasks, which must each synchronize access to the subtask's local data. There are efficiency losses incurred by this additional required synchronization.

The results produced by TIPS are somewhat poor in quality and completeness. They do not include intra-city transportation solutions and do not allow for time delays between the arrival in the city, travel to and check-in to an accommodation facility, and travel to an activity. The travel plans suggested by TIPS would be more realistic and useful if they did take these time delays into consideration. Also, the results provide only one activity suggestion for each destination, whether the amount of time spent at a destination is short or long. In general, a user might like to receive suggestions for activities to participate in on other days of the trip.

6.3 Usefulness of Architecture for Solving Optimization Problems

In this section, we discuss three other optimization problems and outline briefly how the architecture presented in this thesis might be used to solve these problems. We refer only to the selection of objects, their organization, and the communication between objects suggested by our architecture when we apply it to these problems. We do not insist that a particular search strategy be employed within this context.

In general, our architecture is best-suited for solving problems with multiple optimization criteria. Although, it could be used for solving problems of optimization of a single criteria, it is often more difficult to take advantage of concurrency in these cases, and a well-known sequential

approach would perhaps be more efficient, because of the additional overhead incurred by inter-task communication. In addition, meeting any or all of the three following main criteria would indicate that a particular problem might be solved by applying our concurrent architecture:

1. Problem solving operations can be divided into disjoint tasks, or
2. The global solution is comprised of disjoint partial solutions, or
3. The input data can be divided into disjoint parts.

If at least one of these criteria is met, the distinct operations, partial solutions, or input data can be mapped to tasks and subtasks suitable to the problem to be solved.

We now suggest three problems that can be solved using our architecture. These include problems that involve finding bus routes between bus stations and accompanying schedules, mapping processes to processors and scheduling them to run on a distributed computer, and scheduling meetings for employees of a company.

6.3.1 Bus Scheduling Problem

Suppose a city wishes to develop a local public bus transportation system. The locations of several bus stations where the bus routes will start and end at has been determined. Furthermore, these bus stations have been grouped together into regions based on their proximity to each other. Some of the stations are designated as cross-over stations and connect regions of stations together; bus routes from different regions enter and leave from this station. As such, these stations belong to more than one station group. Within a region, each bus station must be linked to every other bus station in that region via one route. In each case, the fastest route that passes through the most densely populated area is to be chosen. In the case of connecting routes that enter or leave the same station, including cross-over stations, the departure and arrival times should coincide with each other, i.e. the departure time of one bus route from a station should be within a certain time interval after the arrival time of a connecting bus route. Given a set of street intersections, the average times required to travel

between adjoining intersections, and the population statistics for dwellings located along the streets between adjoining intersections, the problem is to determine the bus routes and a schedule for each bus route that satisfy the constraints mentioned above.

The bus scheduling problem is an extension of the shortest paths optimization problem in which the shortest, or minimum cost path, must be determined between a pair of nodes in a graph of connected nodes where each edge connecting two nodes is labelled with a cost. For the bus scheduling problem, the nodes represent street intersections and the cost of each path is a combination of the time required to travel between the pair of intersections and the population along that path.

In order to solve this problem using our architecture, one might choose to represent the part of the problem that solves the route scheduling problem for one group of stations as a task. Thus each bus station region is represented by a separate task. The problem could be further broken down into subtasks which each work on solving the route scheduling problem for a distinct pair of bus stations. Each subtask would be responsible for determining the best route and schedule for the route between two stations in both directions. In doing so, subtasks would query their parent tasks for the compatibility of their schedules with those of connecting routes. Tasks would, in turn, query each other for the compatibility of schedules of their subtasks with those of subtasks belonging to other tasks that are both responsible for the same cross-over stations. Tasks would also query their own subtasks in the case of connecting stations within the same region. The tasks would be responsible for assembling the bus routes and schedules provided by each subtask together for each region and selecting the best combination(s) for the region, which would be passed to the strategy controller. The strategy controller would then choose the best overall bus routes and schedules for all regions based on the route(s) and schedule(s) received from its tasks and pass them on to the application controller.

Different groups of task and subtask objects could be created to implement different optimization strategies, such as those that minimize only route travel time, maximize only population density along routes, or minimize route travel time and maximize population density along routes at the same time. The input constraints passed down to each task and subtask would include the acceptable

time interval between arrival and departure times of connecting bus routes, as well as the required frequency of bus trips at different times of the day and for each day of the week. The other input data would be the set of travel time and population information between street intersections. In this problem, the data used by the subtasks may overlap, but the operations and solutions of the subtasks are disjoint.

6.3.2 Process Mapping Problem

Consider a set of distributed processors, each one linked to the other via some communication channel. Each processor has a set of real-time processes created on that processor that must be scheduled to run at a particular time on any one of the connected processors, and each process has an expected duration and a deadline by which it must complete its execution. We would like to develop some real-time software to map each process to a processor and schedule it for execution at a particular time on that processor. The processor on which a process is created need not be the same as the one on which it will run. The mapping and scheduling should be done at the beginning of a time-slice, and every so often thereafter. There is a communication cost associated with each link connecting two processors, which is incurred when a process runs on a processor other than the one on which it was created. Note that this problem is an extension of the classic job scheduling optimization problem with deadlines.

The process mapper could be implemented using our architecture in the following manner. The possible mappings and schedules of a process are computed by a separate subtask. At the next higher level, the responsibility of computing the mappings and schedules for all of the processes created on a processor belongs to a separate task. A strategy controller is used to select the best mappings and schedules for each processor and send it to the system controller which sends the processes to the selected processors along with their scheduled execution times. Several different strategies could be implemented by different groups of these tasks and subtasks under strategy controllers, including strategies to minimize the total time a process spends in the system, maximize the usage of each processor, and minimize the cost of inter-processor communication.

Each subtask would compute possible processor mapping(s) and schedule(s) for the process based on its knowledge of the expected process duration and deadline. Only at the task-level of problem solving could trade-offs be made between candidate subtask solutions based on the inter-processor communication cost and processor usage, as this information should be known only by the tasks, which are responsible for managing the processor. Subtasks could query their parent tasks for the compatibility of their candidate solution(s) with those computed by other subtasks. Each subtask and task could execute in concurrent threads on the same machine or each task group could run on the processor on which the processes they are scheduling were created. In the latter case, some form of inter-processor communication must be used for inter-task communication in the place of synchronized methods. Perhaps the process mapper could even be used to map and schedule itself to run.

6.3.3 Meeting Scheduling Problem

Finally, take the case of an company with employees that belong to different work groups. They often have to schedule meetings of various lengths with each other to discuss their work. These meetings usually must take place before a certain date in order to meet deadlines. Sometimes the meeting participants are all members of the same work group while, other times, they span different work groups. Not all employees work in the same building, so sometimes they must take into account the time required to travel to the meeting location, if it is to take place in another building. They each have different times when they are available to attend the meeting, depending on other meetings that they have to attend and other work that has to be done. Because the employees often spend more time trying to schedule their meetings than they do actually discussing work-related matters, the company would like to implement an organization-wide meeting scheduler application that automates the process. Given a meeting, its expected duration, the list of participants, and the time limit before which the meeting must be held, the software must determine a date, time, and location for this meeting to be held. This problem is also an extension of the job scheduling optimization problem with deadlines.

One way of solving this problem using our architecture is to separate the job of finding acceptable meeting schedules for each employee into different subtasks. Each subtask would be responsible for finding a date, time, and location that would be acceptable based on the employee's agenda, the availability of meeting rooms, and the compatibility of the solutions proposed by other subtasks which it determines by querying the subtasks via their parent tasks. The parent tasks are responsible for finding the best solution for a group of employees represented by its subtasks. The tasks can make trade-offs concerning aspects of the problem that are common to the entire group of employees, such as the location of the meeting, which affects the entire group in the same way, if we assume that all members of a group work in the same building. The strategy controller must choose the best solution that meets the optimization strategy from those candidate solutions it receives from its subtasks. Some optimization strategies that could be implemented by separate groups of tasks and subtasks are: minimize the distance an employee must travel from his/her workspace to the meeting location, attempt to schedule meetings early in the morning, and attempt to schedule meetings early in the afternoon.

What differentiates this problem from the TPP and the other problems presented in this chapter is that the subtasks work on finding one common, global, optimal solution to the problem, rather than finding distinct partial solutions to the problem that are later assembled together to form a complete optimal solution. We have demonstrated that our concurrent architecture can be applied to other applications of practical interest that solve similar optimization of constraints problems, and hence, the usefulness of this architecture.

6.4 Improvements

We now discuss possible improvements to the design of TIPS.

6.4.1 More Efficient Search Strategy

It is apparent that the breadth-first search strategy in combination with the limitation of successors method that we use is inefficient for larger instances of the TPP containing more destinations, longer travel intervals, and larger amounts of travel input data. It is inefficient in terms of consumption of memory and computing power, because it produces too many solutions too soon.

In order to make the application more scalable, we suggest employing a backtracking search strategy (see Section 2.1.1.1.1) in combination with the limitation of sub-solutions method that we use now or some other heuristic evaluation function of the solutions. Each subtask would only construct one complete solution at a time and modify it based on the information it receives about the solutions constructed by other subtasks. The difficulty with this approach is that the subtasks must coordinate their actions with each other. In other words, each subtask must somehow inform the other subtasks of any change it has made to its solution at each step so that compatible solutions are produced.

When the subtasks complete their search, they would then pass their complete solutions up to their parent task, which would build one solution by making trade-offs between the different subtask solutions it receives. The tasks must also inform each other of any changes they make to their solution at each step of problem solving. Finally the tasks would send their solutions to the strategy controller that would put the whole solution together and send it to the application controller.

This search strategy may not be more efficient than the one used in the current design in terms of computing power required, but it would alleviate the strain on memory consumption. Note that this new search strategy can be added to the system by adding new optimization strategies that implement the backtracking search strategy. The other optimization strategies can be left as they are, if desired, because the optimization strategies are completely independent of one another.

6.4.2 LocalTransportation Task and Subtasks

In the present design, intra-city transportation is not taken into account at all. In order to introduce this aspect of the problem into TIPS, a LocalTransportation task and appropriate subtasks such as Taxi, LocalBus, and CarRental could be added. These objects would collaborate to provide local transportation solutions for transportation between inter-city transportation depots, accommodation facilities, and activity venues. Check-in and check-out times for accommodation facilities as well as times for activities would have to be adjusted accordingly to allow for intra-city travel time. Adding this task to the set of problem solving objects would give the user a more comprehensive travel plan and make it more realistic and useful.

6.4.3 Collaboration Between Strategies

Currently, a group of tasks and subtasks applying a particular optimization strategy do not collaborate with other strategies. Strategies do not communicate with each other, but, rather, operate completely independently of one another. Perhaps one strategy could help another one progress faster in its search for the optimal solution by providing it with solutions or partial solutions that it has no use for itself, but which may be of use to the other strategy. This would be advantageous, particularly if strategies performed their search from different travel information databases. The strategies could then provide each other with solutions that they might not otherwise obtain. The application controller could be used to receive the discarded solutions from the strategies and then send them to any other strategy that it thinks might be able to use them.

6.4.4 Additional Strategies and Subtasks

New subtasks could be added to give more variety to the travel plans output by TIPS. Under the NonLocalTransportation task, CarRental and Ship subtasks could be added to find transportation solutions by rental car and boat, respectively. A Campground subtask could be added to the Accommodation task to find campgrounds to stay at. Shows, Sports, and Exhibitions subtasks could

be added to the Activities task to specialize in finding these three types of events, rather than having them all combined into one Events subtask as we have done with the current design.

Other useful optimization strategies could be added to the system. For example, perhaps the traveller(s) are interested in maximizing the length of their stays at the selected destinations. Although TIPS does ensure that the travel plan falls within the travel interval specified by the user, it does not guarantee a travel plan that coincides exactly with the travel interval. For that purpose, a MaxStayTime strategy could be added to the system, by creating appropriate subclasses of the Strategy class and the task and subtask abstract classes. Other useful strategies that could be implemented are ones to minimize the amount of time spent travelling and to provide a travel plan that adjusts the destination orders and travel stay intervals such that it permits the traveller to attend the maximum number of activities at the selected destinations.

6.4.5 Additional Alternative Solutions

The travel planner could offer more alternatives to the user in two ways:

1. suggest more activities for the traveller in each travel plan, and

2. allow the user to request more than one travel plan per optimization strategy.

Currently, only one activity is suggested per travel destination, leaving the traveller without any activity suggestions for the rest of the stay. This was done in order to meet memory constraints. Perhaps, with a backtracking or other search strategy that is not so memory intensive in place, one activity could be provided for each day that the traveller stays at a destination.

In the current implementation, the user can choose to be presented with up to either one travel plan or one travel plan per selected optimization strategy. Perhaps the user could be given more flexibility by being permitted to request either a variable number of travel plans per optimization strategy or a total variable number of travel plans, selected from the first strategies to finish execution.

6.4.6 Reservations

Presently, the travel planner subtasks ensure that there is enough available capacity to accommodate the traveller(s) when creating a particular initial subtask solution. However, no reservations are made for the user when the final travel plan is produced. The travel plans presented to the user are meant to serve as suggestions that the user can later choose to use by making the necessary reservations on his/her own, if he/she so desires. In addition to presenting the user with suggested travel plans, the system could also allow the user to select one travel plan that the user would like to use and make the appropriate reservations on the user's behalf, by updating the database accordingly.

6.5 Conclusions

The architecture of the travel planner presented in this thesis has proven to be useful for several reasons. It allows us to solve the TPP in a concurrent fashion using collaborating objects, thus enabling the designer to achieve efficiency gains by using a multi-processor or distributed computer. It promotes the principle of design for change in that new optimization strategies, tasks, and subtasks providing other partial solutions to the TPP can be added rather easily. The architecture makes great gains in terms of code reuse. As well, the design promotes the principle of separation of concerns. The programmer can concentrate on solving one aspect of the overall problem at a time at the subtask level and then compose the overall solution together later on at the task and strategy controller levels rather than concerning him/herself with finding a complete solution right from the start. This principle also applies for managing the consistency of the solution data, which is simplified in comparison to concurrent blackboard approaches. The solutions are privately accessed by problem solving components and are protected by the individual components they belong to in our architecture. In addition, we have shown that the general architecture can be reused to solve other practical optimization of constraints problems. Thus, its range of applicability is widened.

Based on our experience with TIPS, it appears that implementing the travel planner as a production system with distributed control and data and having collaborating knowledge sources is

appropriate, because it does help us solve a rather large problem by breaking it up into parts. The system produces useful results for small instances of the problem, even though they are approximations to the optimal solution. However, it is apparent that the breadth-first search strategy used in this implementation is inadequate for use with larger problem instances. Because of memory constraints that had to be taken into consideration, the results obtained for long travel intervals with many destinations are often unexpected, and sometimes no approximate optimal solution is found. The execution time is rather long in some instances, which is due to the large number of solutions produced by the breadth-first search and the overhead required for querying subtasks about the coherency and compatibility of solutions. Future implementations should employ a more efficient search strategy, such as a backtracking search with an heuristic evaluation function. Based on our experience, it was also made clear that Java is an appropriate development language for this project.

Although this concurrent implementation of a travel planner may not be as efficient in terms of the computing power and memory storage required compared to equivalent sequential implementations, the design increases our understanding of how to solve the problem concurrently and could serve as a basis for future more efficient concurrent implementations.

Bibliography

[1] Per Brinch Hansen. Java's insecure parallelism. *ACM SIGPLAN Notices*, 34(4):38–45, April 1999.

[2] Frank Buschmann et al. *Pattern-Oriented Software Architecture: A System of Patterns*, chapter 2, pages 71–95. John Wiley & Sons, West Sussex, England, 1996.

[3] B. Chandrasekaran, editor. *Blackboard Architectures and Applications*, volume 3 of *Perspectives in Artificial Intelligence*, chapter 6, pages 99–136. Academic Press, San Diego, CA, 1989.

[4] Robert Engelmore and Tony Morgan, editors. *Blackboard Systems*, chapter 25, pages 475–501. The Insight Series in Artificial Intelligence. Addison-Wesley, New York, 1989.

[5] Morris W. Firebaugh. *Artificial Intelligence: A Knowledge-Based Approach*, chapter 10, pages 312–313. PWS-Kent, Boston, MA, second edition, 1989.

[6] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software Design*, chapter 5, pages 315–323. Addison-Wesley professional computing series. Addison-Wesley, Reading, MA, 1995.

[7] Nils J. Nilsson. *Principles of Artificial Intelligence*. Tioga, Palo Alto, CA, 1980.

[8] Rational Software Corporation, Cupertino, CA. *UML Notation Guide*, September 1997.

[9] Yoshiaki Shirai and Jun-ichi Tsujii. *Artificial Intelligence: Concepts, Techniques and Applications*, page 144. Wiley series in computing. John Wiley & Sons, New York, NY, 1984.

[10] Peter van der Linden. *Just Java and Beyond*. Java Series. Sun Microsystems Press, Palo Alto, CA, 1998.

Appendix A

Data Dictionary

Accommodation::Accommodation (*Class*) An abstract class that is responsible for providing strategy-independent operations for creating accommodation solutions and for delivering acceptability ratings for accommodation subtask solutions

Accommodation::Accommodation::buildSolutions (*Operation*) Creates Accommodation solutions by mixing the subtask solutions produced by its subtasks

Accommodation::Accommodation::numSolsToKeep (*Attribute*) The maximum number of accommodation solutions that can be sent to the strategy controller

Accommodation::Accommodation::peerPSol (*Operation*) Returns the percentage of NonLocalTransportation subtask sub-solutions that are compatible with an Accommodation subtask sub-solution

Accommodation::Accommodation::pSol (*Operation*) Returns the percentage of Accommodation subtask sub-solutions that can be used after the given date and time from those in the given city

Accommodation::Accommodation::setIntervals (*Operation*) Assigns appropriate dates and times to accommodation solutions based on the time stay intervals it received and deletes solutions that are not compatible with these intervals

Accommodation::AccommodationSol (*Class*) Stores an accommodation solution that consists of one stay at an accommodation facility for each travel destination and provides operations to modify

and obtain information about it

Accommodation::MinCostAccommodation (*Class*) Creates minimum cost accommodation solutions

Accommodation::MinCostMaxDistAccommodation (*Class*) Creates accommodation solutions by minimizing cost while maximizing distance travelled

Accommodation::MinDistAccommodation (*Class*) Creates accommodation solutions that minimize on distance travelled between cities

Accommodation::Stay (*Class*) Stores information about a particular stay at an accommodation facility and provides information about that stay

Activities::Activities (*Class*) An abstract class that is responsible for providing strategy-independent operations for creating activity solutions and for delivering acceptability ratings for activity subtask solutions

Activities::Activities::buildSolutions (*Operation*) Creates Activities solutions by mixing the subtask solutions produced by its subtasks

Activities::Activities::numSolsToKeep (*Attribute*) The maximum number of activity solutions that can be sent to the strategy controller

Activities::Activities::peerPSol (*Operation*) Returns the percentage of NonLocalTransportation subtask sub-solutions that are compatible with an Activities subtask sub-solution

Activities::Activities::pSol (*Operation*) Returns the percentage of Activities subtask sub-solutions that can be attended after the given date and time from those in the given city

Activities::Activities::setIntervals (*Operation*) Assigns appropriate dates and times to activity solutions based on the time stay intervals it received and deletes solutions that are not compatible with these intervals

Activities::Activity (*Class*) Stores information about an activity that can be attended during the trip and provides information about that activity

Activities::ActivitySol (*Class*) Stores a set of activities that can be attended at each travel destination and provides operations to modify and obtain information about them. It consists of

one activity for each travel destination.

Activities::MinCostActivities (*Class*) Creates minimum cost activity solutions

Activities::MinCostMaxDistActivities (*Class*) Creates activity solutions that are of minimum cost and, at the same time, maximize on distance travelled

Activities::MinDistActivities (*Class*) Creates activity solutions that minimize on distance travelled between cities

BedBreakfast::BB (*Class*) Stores information about a stay at a particular bed and breakfast on a particular date and time, and provides operations to obtain information about that stay

BedBreakfast::BBSolution (*Class*) Stores a bed and breakfast solution, which is a list of bed and breakfast stays (one for each travel destination), and provides operations to obtain information about those stays and convert itself into an Accommodation solution

BedBreakfast::BedBreakfast (*Class*) An abstract class that is responsible for creating bed and breakfast solutions that meet the traveller's constraints and for providing information about how other sub-solutions match with the list of bed and breakfast stays

BedBreakfast::BedBreakfast::bbs (*Attribute*) The list of bed and breakfast stays that is built

BedBreakfast::BedBreakfast::buildSolutions (*Operation*) Creates bed and breakfast solutions that satisfy the travel constraints using the bed and breakfast travel data

BedBreakfast::BedBreakfast::pSol (*Operation*) Returns the percentage of bed and breakfast stays currently built that can be used after the given date and time from those in the given city

BedBreakfast::MinCostBedBreakfast (*Class*) Creates minimum cost bed and breakfast solutions

BedBreakfast::MinCostMaxDistBedBreakfast (*Class*) Creates bed and breakfast solutions that minimize total cost and maximize total distance travelled between cities

BedBreakfast::MinDistBedBreakfast (*Class*) Creates bed and breakfast solutions that minimize on total distance travelled between cities

Bus::Bus (*Class*) An abstract class that is responsible for creating bus solutions for the TPP and for providing information about how other sub-solutions match with the current list of bus trips

Bus::Bus::buildSolutions (*Operation*) Creates bus solutions that satisfy the travel constraints

using the bus travel data

Bus::Bus::busTrips (*Attribute*) The list of bus trips that is built

Bus::Bus::pSol (*Operation*) Returns the percentage of bus trips currently built from the bus trips destined for the given city that arrive before at least one of the given dates

Bus::BusSolution (*Class*) Stores a bus solution, which is a list of bus trips from the departure city to each destination and then back to the departure city, and provides operations to obtain information about these trips and convert itself into a NonLocalTransportation solution

Bus::BusTrip (*Class*) Stores information about a bus trip between two cities on a particular date and time, and provides operations to obtain information about that bus trip

Bus::MinCostBus (*Class*) Creates minimum total cost bus solutions

Bus::MinCostMaxDistBus (*Class*) Creates bus solutions of minimum total cost and maximum total distance travelled between cities

Bus::MinDistBus (*Class*) Creates bus solutions of minimum total distance travelled between cities

Controller::MinCostMaxDistStrategy (*Class*) Creates one minimum cost and maximum distance travelled travel plan by combining the task solutions it receives, and also provides strategy-specific, task-independent operations for sorting MinCostMaxDist task solutions and ranking MinCost-MaxDist subtask solutions

Controller::MinCostMaxDistStrategy::computeOrderRankings (*Operation*) Returns the rankings of a list of solutions ranked by minimum cost and maximum distance travelled at the same time

Controller::MinCostMaxDistStrategy::orderSolutions (*Operation*) Sorts a list of solutions in order of ascending cost and decreasing distance travelled at the same time

Controller::MinCostStrategy (*Class*) Creates one minimum cost travel plan by combining the task solutions it receives, and also provides strategy-specific, task-independent operations for sorting MinCost task solutions and ranking MinCost subtask solutions

Controller::MinCostStrategy::computeOrderRankings (*Operation*) Returns the rankings of a list of solutions ranked by minimum cost

Controller::MinCostStrategy::orderSolutions (*Operation*) Sorts a list of solutions in order of ascending cost

Controller::MinDistStrategy (*Class*) Creates one minimum distance travelled travel plan by combining the task solutions it receives and also provides strategy-specific, task-independent operations for sorting MinDist task solutions and ranking MinDist subtask solutions

Controller::MinDistStrategy::computeOrderRankings (*Operation*) Returns the rankings of a list of solutions ranked by minimum distance travelled

Controller::MinDistStrategy::orderSolutions (*Operation*) Sorts a list of solutions in order of ascending distance travelled

Controller::Strategy (*Class*) An abstract class that contains methods for storing solutions received from strategy tasks and combining them to create travel plans with mixed task solutions

Controller::Strategy::buildSolutions (*Operation*) Creates travel plans by making combinations of task solutions produced by this strategy's tasks

Controller::Strategy::computeOrderRankings (*Operation*) Returns the rankings of a list of solutions ranked by the optimization constraint of the strategy

Controller::Strategy::controller (*Attribute*) The system controller that strategy solutions are sent to

Controller::Strategy::maxNumSolutions (*Attribute*) The maximum number of solutions that can be built and returned to the controller

Controller::Strategy::numSolutions (*Attribute*) The number of travel plans created by this strategy controller

Controller::Strategy::numTasksFinished (*Attribute*) The number of tasks that have completed their execution and returned their solutions

Controller::Strategy::orderSolutions (*Operation*) Sorts a list of solutions by the values of the constraint associated with the strategy

Controller::Strategy::solutions (*Attribute*) The list of candidate travel plans created by this strategy controller

Controller::Strategy::taskAborted (*Operation*) Performs any actions that must be taken when a task stops execution due to an error that occurred or when all of its subtasks abort

Controller::Strategy::taskFinished (*Operation*) Stores the solutions produced by a task upon its completion

Controller::Strategy::tasks (*Attribute*) The list of tasks that work on sub-divisions of the TPP for this strategy

Controller::Tips (*Class*) Responsible for creating the groups of objects that will apply a particular strategy to solve the TPP, for sending any errors or messages reported by the Task subsystem objects to the ConstraintsEntryView object for display, and for sending the travel plan received from each strategy to the ResultsView object for display

Controller::Tips::logToFile (*Operation*) Writes a message to the log file

Controller::Tips::numStrategiesAborted (*Attribute*) The number of strategy controllers that have aborted execution because of an error

Controller::Tips::numStrategiesFinished (*Attribute*) The number of strategy controllers that have terminated execution successfully and have returned their solutions

Controller::Tips::reportError (*Operation*) Requests the user interface to display an error message on the screen

Controller::Tips::strategies (*Attribute*) The array of strategy controllers that were started by this controller

Controller::Tips::strategyAborted (*Operation*) Performs any actions that must be taken when a strategy stops execution due to an error that occurred or when one of its tasks aborts

Controller::Tips::strategyCompleted (*Operation*) Sends the travel plan returned by a strategy to the User Interface subsystem for display to the user

Controller::TravelPlan (*Class*) Stores the NonLocalTransportation, Accommodation, and Activities solution of a travel plan, and provides information about the travel plan

DataStructures::City (*Class*) Stores the name of a city, and the state/province and country in which it is located

DataStructures::ListElement (*Class*) Stores the value of an element of an ObjectList which is a generic object and a reference to the next element of the list

DataStructures::ObjectList (*Class*) Responsible for managing a grow-able size list of generic objects

DateTime::InvalidMsqlDateException (*Class*) An exception that is thrown when an invalid date (a date that does not appear on the Gregorian calendar) has been specified

DateTime::InvalidMsqlTimeException (*Class*) An exception that is thrown when an invalid time (a time that does not fall in the interval 00:00:00 to 23:59:59) has been specified

DateTime::MsqlDate (*Class*) Represents a Gregorian calendar date (DD-MM-YYYY) and provides operations to access and modify this date

DateTime::MsqlDateAndTime (*Class*) Represents a specific instant in time (date and time) and provides operations that access and modify this instant in time

DateTime::MsqlTime (*Class*) Represents a time (HH:MM:SS) on a 24-hour clock and provides operations for accessing and modifying this time

Dining::Dining (*Class*) An abstract class that is responsible for creating dining solutions for the TPP and for providing information about how other sub-solutions match with the current list of restaurants

Dining::Dining::buildSolutions (*Operation*) Creates dining solutions that satisfy the travel constraints using the dining travel data

Dining::Dining::pSol (*Operation*) Returns the percentage of restaurants currently built from the given city that can be used after the given date and time

Dining::Dining::restaurants (*Attribute*) The list of restaurants that is built

Dining::DiningSolution (*Class*) Stores a dining solution, which is a list of restaurants (one for each destination) that a traveller can visit, and provides operations to obtain information about the solution and convert itself into an Activity solution

Dining::MinCostDining (*Class*) Creates minimum cost dining solutions

Dining::MinCostMaxDistDining (*Class*) Creates dining solutions of minimum cost and maximum

distance travelled between cities

Dining::MinDistDining (*Class*) Creates dining solutions of minimum distance travelled between cities

Dining::Restaurant (*Class*) Stores information about a restaurant outing on a particular date and time, and provides operations to obtain information about that restaurant outing

Events::Event (*Class*) Stores information about an event that takes place during a limited time that the traveller can attend, and provides operations to obtain information about that event

Events::Events (*Class*) An abstract class that is responsible for creating solutions that contain time-limited events a traveller can attend for the TPP and for providing information about how other sub-solutions match with the current list of events

Events::Events::buildSolutions (*Operation*) Creates event solutions that satisfy the travel constraints using the event travel data

Events::Events::events (*Attribute*) The list of events to attend that is built

Events::Events::pSol (*Operation*) Returns the percentage of events currently built for the given city that can be attended after the given date and time

Events::EventSolution (*Class*) Stores an event solution, which is a list of time-limited events (one for each destination) that a traveller can attend, and provides operations to obtain information about the solution and convert itself into an Activity solution

Events::MinCostEvents (*Class*) Creates minimum cost event solutions

Events::MinCostMaxDistEvents (*Class*) Creates event solutions of minimum cost and maximum distance travelled between cities

Events::MinDistEvents (*Class*) Creates event solutions of minimum distance travelled between cities

File::FileManager (*Class*) Responsible for returning different kinds of travel data read from files that match certain travel constraints

File::TipsFilenameFilter (*Class*) Responsible for selecting only filenames that have the ".tips" extension for display in a filename chooser of a file dialog box

Hotel::Hotel (*Class*) An abstract class that is responsible for creating solutions that contain hotel stays a traveller can use and for providing information about how other sub-solutions match with the current list of hotel stays

Hotel::Hotel::buildSolutions (*Operation*) Creates hotel solutions that satisfy the travel constraints using the hotel travel data

Hotel::Hotel::hotels (*Attribute*) The list of hotel stays that is built

Hotel::Hotel::pSol (*Operation*) Returns the percentage of hotel stays currently built for the given city that can be used after the given date and time

Hotel::HotelSolution (*Class*) Stores a hotel solution, which is a list of hotel stays (one for each destination), and provides operations to obtain information about the solution and convert itself into an Accommodation solution

Hotel::HotelStay (*Class*) Stores information about a hotel stay, and provides operations to obtain information about the hotel stay

Hotel::MinCostHotel (*Class*) Creates minimum cost hotel solutions

Hotel::MinCostMaxDistHotel (*Class*) Creates hotel solutions of minimum cost and maximum distance travelled between cities

Hotel::MinDistHotel (*Class*) Creates hotel solutions of minimum distance travelled between cities

Motel::MinCostMaxDistMotel (*Class*) Creates motel solutions of minimum cost and maximum distance travelled between cities

Motel::MinCostMotel (*Class*) Creates minimum cost motel solutions

Motel::MinDistMotel (*Class*) Creates motel solutions of minimum distance travelled between cities

Motel::Motel (*Class*) An abstract class that is responsible for creating solutions that contain motels a traveller can stay at and for providing information about how other sub-solutions match with the current set of motel stays

Motel::Motel::buildSolutions (*Operation*) Creates motel solutions that satisfy the travel constraints using the motel travel data

Motel::Motel::motels (*Attribute*) The list of motel stays that is built

Motel::Motel::pSol (*Operation*) Returns the percentage of motel stays currently built for the given city that can be used after the given date and time

Motel::MotelSolution (*Class*) Stores a motel solution, which is a list of motel stays (one for each destination), and provides operations to obtain information about the solution and convert itself into an Accommodation solution

Motel::MotelStay (*Class*) Stores information about a stay at a motel, and provides operations to obtain information about the motel stay

NonLocalTransportation::MinCostMaxDistNonLocalTransportation (*Class*) Creates inter-city transportation solutions by minimizing cost while maximizing distance travelled

NonLocalTransportation::MinCostMaxDistNonLocalTransportation::insertIntervals (*Operation*) Adds the cost and distance travelled of a NonLocalTransportation solution to the total for the stay time interval represented by that solution

NonLocalTransportation::MinCostMaxDistNonLocalTransportation::rankIntervals (*Operation*) Ranks the stay time intervals of NonLocalTransportation solutions by minimum cost and maximum distance travelled and returns the interval with the highest rank

NonLocalTransportation::MinCostNonLocalTransportation (*Class*) Creates minimum cost inter-city transportation solutions

NonLocalTransportation::MinCostNonLocalTransportation::insertIntervals (*Operation*) Adds the cost of a NonLocalTransportation solution to the total for the stay time interval represented by that solution

NonLocalTransportation::MinCostNonLocalTransportation::rankIntervals (*Operation*) Ranks the stay time intervals of NonLocalTransportation solutions by minimum cost and returns the interval with the highest rank

NonLocalTransportation::MinDistNonLocalTransportation (*Class*) Creates inter-city transportation solutions that minimize on distance travelled between cities

NonLocalTransportation::MinDistNonLocalTransportation::insertIntervals (*Operation*)

Adds the distance travelled of a NonLocalTransportation solution to the total for the stay time interval represented by that solution

NonLocalTransportation::MinDistNonLocalTransportation::rankIntervals (*Operation*) Ranks the stay time intervals of NonLocalTransportation solutions by minimum distance and returns the interval with the highest rank

NonLocalTransportation::NonLocalTransportation (*Class*) An abstract class that is responsible for providing strategy-independent operations for creating inter-city transportation solutions and for delivering acceptability ratings for NonLocalTransportation subtask solutions

NonLocalTransportation::NonLocalTransportation::buildSolutions (*Operation*) Creates NonLocalTransportation solutions by mixing the subtask solutions produced by its subtasks

NonLocalTransportation::NonLocalTransportation::intervalRanks (*Attribute*) The rankings of each stay interval represented by the NonLocalTransportation solutions based on the optimization strategy

NonLocalTransportation::NonLocalTransportation::intervals (*Attribute*) Lists of pairs of date and times, that form the stay intervals represented by each of the NonLocalTransportation solutions

NonLocalTransportation::NonLocalTransportation::numIntervals (*Attribute*) The number of different stay intervals represented by the NonLocalTransportation solutions

NonLocalTransportation::NonLocalTransportation::peerPSol (*Operation*) Returns the percentage of Accommodation and Activities subtask sub-solutions that are compatible with a NonLocalTransportation subtask sub-solution

NonLocalTransportation::NonLocalTransportation::pSol (*Operation*) Returns the percentage of NonLocalTransportation subtask sub-solutions to the given city that arrive before at least one of the given dates

NonLocalTransportation::TransportSol (*Class*) Stores an inter-city NonLocalTransportation solution that consists of trips starting from the departure city through each destination city and then back to the departure city, and provides operations to modify and obtain information about these trips

NonLocalTransportation::Trip (*Class*) Stores information about a particular trip via some mode of transportation on a given date and time between two cities, and provides information about that trip

Plane::Flight (*Class*) Stores information about a plane flight between two cities on a particular date and time, and provides operations to obtain information about that flight

Plane::MinCostMaxDistPlane (*Class*) Creates plane solutions of minimum cost and maximum distance travelled between cities

Plane::MinCostPlane (*Class*) Creates minimum cost plane solutions

Plane::MinDistPlane (*Class*) Creates plane solutions of minimum distance travelled between cities

Plane::Plane (*Class*) An abstract class that is responsible for creating plane solutions for the TPP and for providing information about how other sub-solutions match with the current set of plane flights created

Plane::Plane::buildSolutions (*Operation*) Creates plane solutions that satisfy the travel constraints using the plane travel data

Plane::Plane::flights (*Attribute*) The list of airplane flights that is built

Plane::Plane::pSol (*Operation*) Returns the percentage of flights currently built to the given city that arrive before at least one of the given dates

Plane::PlaneSolution (*Class*) Stores a plane solution, which is a list of plane flights from the departure city to each destination and then back to the departure city, and provides operations to obtain information about these flights and convert itself into a NonLocalTransportation solution

Sightseeing::MinCostMaxDistSightseeing (*Class*) Creates sightseeing solutions of minimum cost and maximum distance travelled between cities

Sightseeing::MinCostSightseeing (*Class*) Creates minimum cost sightseeing solutions

Sightseeing::MinDistSightseeing (*Class*) Creates sightseeing solutions of minimum distance travelled between cities

Sightseeing::Sight (*Class*) Stores information about a tourist sight that can be visited by the

traveller, and provides operations to obtain information about that sight

Sightseeing::Sightseeing (*Class*) An abstract class that is responsible for creating solutions that contain tourist sights a traveller can visit, and for providing information about how other sub-solutions match with the current list of tourist sights

Sightseeing::Sightseeing::buildSolutions (*Operation*) Creates sightseeing solutions that satisfy the travel constraints using the sightseeing travel data

Sightseeing::Sightseeing::pSol (*Operation*) Returns the percentage of sights currently built for the given city that can be visited after the given date and time

Sightseeing::Sightseeing::sights (*Attribute*) The list of tourist sights to visit that is built

Sightseeing::SightseeingSolution (*Class*) Stores a sightseeing solution, which is a list of tourist sights (one for each destination) that a traveller can visit, and provides operations to obtain information about the solution and convert itself into an Activity solution

Sorting::Indicators (*Class*) Provides the service of assigning an integer ranking by value of a list of optimization constraints, such as counts, costs, and distances, that can be represented by a list of integer or floating point numbers

Sorting::InsertionSorter (*Class*) Provides sorting routines for sorting arrays of integer and floating point numbers and objects associated with arrays of integer and floating point numbers using the InsertionSort algorithm

Sorting::QuickSorter (*Class*) Provides sorting routines for sorting arrays of integer and floating point numbers and objects associated with arrays of integer and floating point numbers using the QuickSort algorithm

Sorting::Sorter (*Interface*) An interface that provides operations for sorting arrays of integer and floating point numbers and objects associated with arrays of integer and floating point numbers

Task::Solution (*Interface*) An interface that provides operations supported by all subtask and task solutions

Task::SubTaskSolution (*Interface*) An interface that provides operations supported by all subtask solutions

Task::SubTaskSolution::toTaskSolution (*Operation*) Converts the subtask solution into an instance of its parent task solution

Task::Task (*Class*) An abstract class that represents a sub-division of the work of solving the TPP and produces solutions that solve that aspect of the TPP. It stores the traveller's constraints and the solutions to the part of the problem being solved, and provides operations common to tasks and subtasks such as storing solutions received from subtasks, accumulating rankings of destination city orders, and deleting solutions that are not useful.

Task::Task::acceptabilityRating (*Attribute*) The minimum percentage of sub-solutions from other subtasks that a given subtask sub-solution must be compatible with in order for it to be retained in the search for solutions

Task::Task::bestIntervals (*Attribute*) The list of stay intervals for each destination that was chosen as being the best for this optimization strategy

Task::Task::bestOrder (*Attribute*) Contains the list of destination cities in the order that it was determined collectively that they should be visited

Task::Task::computeBestOrder (*Operation*) Determines the order of destination cities with the highest overall ranking

Task::Task::controller (*Attribute*) The system controller to which this task or subtask reports any errors

Task::Task::getOrder (*Operation*) Returns the best order of destination cities based on the cumulative rankings of the subtasks

Task::Task::intervalsSet (*Attribute*) Indicates whether or not the time stay intervals have been determined yet

Task::Task::maxNumSolutions (*Attribute*) The maximum number of solutions that may be built by this subtask or task

Task::Task::numSolutions (*Attribute*) The number of solutions that have been created by this task or subtask

Task::Task::numSubTasksAborted (*Attribute*) The number of subtasks of this task that have

halted their execution due to some error or the inability to find any solutions

Task::Task::numSubTasksFinished (*Attribute*) The number of subtasks that have completed their execution and sent their solutions to this parent task

Task::Task::orderRank (*Attribute*) The rankings of the destination city orders present in the subtask solutions for this task

Task::Task::orderRanking (*Operation*) Stores the rankings of destination city orders computed by a subtask on its solutions

Task::Task::orders (*Attribute*) The lists of destination city orders represented by all of the subtask solutions for this task

Task::Task::overallOrderComputed (*Attribute*) Indicates whether or not all of the subtasks have sent their destination city order rankings so that the overall order has been computed

Task::Task::parent (*Attribute*) The task which this subtask must send its solutions

Task::Task::peers (*Attribute*) The list of peer tasks working on another part of the TPP

Task::Task::setBestIntervals (*Operation*) Stores the time stay intervals that were determined by the NonLocalTransportation task and must be assigned to the solutions of this task

Task::Task::solutions (*Attribute*) The list of solutions created to solve a part of the TPP

Task::Task::strategy (*Attribute*) The strategy controller for this task or subtask

Task::Task::subTaskAborted (*Operation*) Performs any actions that must be taken when a subtask stops execution due to an error that occurred

Task::Task::subTaskFinished (*Operation*) Stores the solutions sent by a subtask upon its completion

Task::Task::subTasks (*Attribute*) The list of subtasks working on a sub-division of this task's problem and providing solutions for that part of the problem

Task::Task::taskOrderRank (*Attribute*) The rankings of the destination city orders present in the solutions of the subtasks of this task's peer tasks

Task::Task::taskOrderRanking (*Operation*) Stores the combined destination city order rankings of a peer task computed by all of its subtasks on their solutions

Task::Task::taskOrders (*Attribute*) The lists of destination city orders present in the subtask solutions of this task's peer tasks

Task::TaskSolution (*Interface*) An interface that provides operations supported by all task solutions, namely, NonLocalTransportation, Accommodation, and Activity solutions

Task::TaskSolution::setIntervals (*Operation*) Assigns dates and times to the task solution such that it is compatible with the list of best stay intervals for each destination

Train::MinCostMaxDistTrain (*Class*) Creates train solutions of minimum cost and maximum distance travelled between cities

Train::MinCostTrain (*Class*) Creates minimum cost train solutions

Train::MinDistTrain (*Class*) Creates train solutions of minimum distance travelled between cities

Train::Train (*Class*) An abstract class that is responsible for creating train solutions for the TPP and for providing information about how other sub-solutions match with the current set of train trips created

Train::Train::buildSolutions (*Operation*) Creates train solutions that satisfy the travel constraints using the train travel data

Train::Train::pSol (*Operation*) Returns the percentage of train trips currently built arriving at the given city that arrive before at least one of the given dates

Train::Train::trips (*Attribute*) The list of train trips that is built

Train::TrainSolution (*Class*) Stores a train solution, which is a list of train trips from the departure city to each destination and then back to the departure city, and provides operations to obtain information about these train trips and convert itself into a NonLocalTransportation solution

Train::TrainTrip (*Class*) Stores information about a train trip between two cities on a particular date and time, and provides operations to obtain information about that train trip

UserInterface::ConstraintsEntryView (*Class*) A user interface window responsible for displaying the user interface controls manipulated by the user to enter travel constraints and sends the constraints to the system controller (Tips). It also displays error messages for errors in the user input or errors that occur during the search for results, and information messages about the progress

of the search.

UserInterface::ErrorReporter (*Class*) A dialog box that displays error messages to the user

UserInterface::ResultsView (*Class*) A user interface window that displays one travel plan result for an optimization strategy search for travel solutions. It also permits the user to save this result to a file.