Optimal Travel

DESIGN DOCUMENT

Kwan, Elena W.C

1. Introduction

The website provides the user with a solution to the travelling salesman problem, based on the inputs that the user provides (i.e. number of cities within the trip and costs of travelling to and from each city). For example, if a user is looking to visit 5 cities, the site would indicate what route they should take to minimize their costs. Since the user specifies the inputs, this cost could either be distance, plane fare, or some other "cost metric". The user simply needs to be consistent about and understand the units he/she wants to apply, but there is no need to provide that information as an input into the calculation. It assumes that the first city provided (city 1) is the starting city, and that each city is visited only once and that the path ends at the starting city.

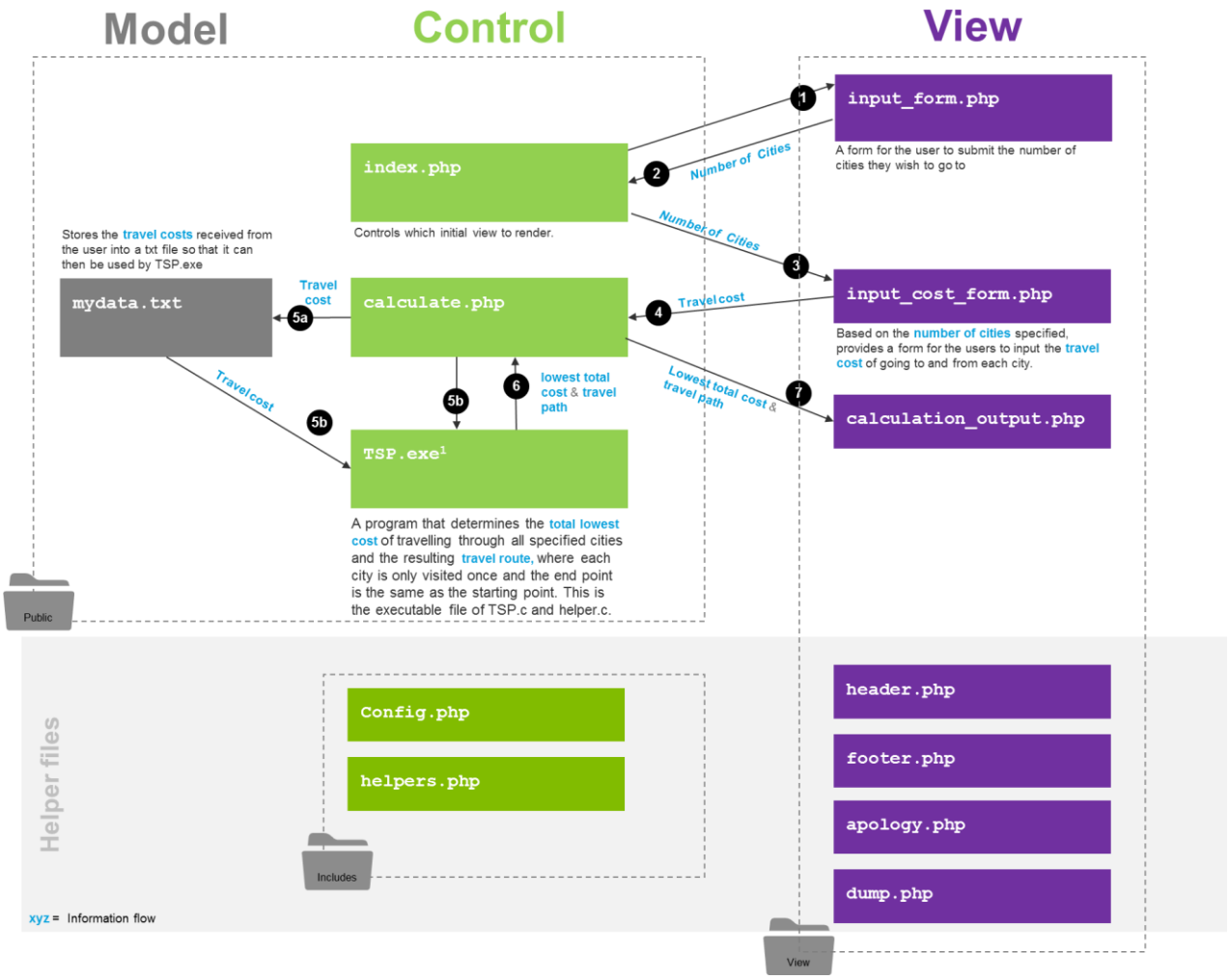
This travelling salesman problem is solved using the branch and bound approach described by the saurabhschool.org. Since this problem is computationally heavy, the program was written in c (TSP.c) for performance reasons. The web interface was written in php, which makes a call to compiled executable code of the c program.

The web interface is organized using the MVC model and is described in the section **website organization and design.** The optimization program, TSP.c, that solves the travelling salesman problem is described in the section **Branch and Bound Approach to Solving the Travelling Salesman Problem**

2. Website Organization and Design

2.1 Conceptual website and file organization and information flow

Since the site only performs one main task (i.e. find the lowest-cost route) and requires successive pieces of information from the user, the site is designed to take the user through a linear journey through the website. This linear flow is described in the table and diagram below.



1	**The user is presented with an initial form:** If the page is reached via a link or redirect, index.php displays a form that prompts them to enter in the **number of cities** they need to go to.
2	**Index.php receives form submission:** Form information is sent to index.php via a post request. The fact that Index.php is now being reached by a post request is used a “signal” that a form was submitted (much like in Pset 7). **Index.php checks for proper inputs:** Index.php checks that the submission wasn’t empty and that an integer between 2 and 25 was provided (there must be at least two cities to travel between and the algorithm only capable of handling 25 cities in less than 30 seconds and performance decreases significantly after that point).
3	**Index.php renders a second form:** If the proper inputs are provided, another form is displayed to the user where he/she must provide the **travel cost** of going to and from each city, with the exception of going to and from the same city (hence the diagonal cells of the form do not require inputs). The form is rendered based on the **number of cities** the user specified.
4	**calculate.php receives the form submission**
5	Once calculate.php receives the **travel costs** from the user, it does two things **5a: calculate.php stores travel cost information in a text file** in a format that the program, TSP.c can read. Since the cost of going to and from the same city is not applicable, this travel option is “negated” by providing as its value the maximum int value (2147483647). For this reason, the user is unable to provide a cost equal or greater than 2147483647. **5b: calculate.php calls an external optimization program** (TSP.exe) which was written in the c-language for performance reasons. As a result, the php program, shell_exec, is used to call this program. This program takes as its arguments (1) the number of cities specified and (2) an appropriately formatted text file with the travel cost information (mydata.txt).
6.	**shell_exec returns a string that contains the outputs of the external program** (i.e. the external program’s statements that would have been printed to the terminal). **calculate.php parses the outputs** of the external program in order **to extract the** (1) **total lowest cost** of the travelling to all the cities and (2) the **travel path** that would result in that total lowest cost.

	TSP.c could have been written to print out only the numeric “answers” to the problem (instead of a full sentence). This would have reduced the complexity of the string to be parsed. However, this would have made outputs of TSP.c less interpretable and harder to use for debugging purposes.
7.	**calculation_output.php displays the calculated outputs** (the **total lowest cost** of the travelling to all the cities and the resulting **travel path**).

2.2 Website design decisions

The website uses php for several reasons:

- **Linear information flow:** The flow of information for this site is very linear in nature. The user must provide the number of cities before can generate the appropriate cost intake form. That cost input is then needed to calculate the lowest-cost path. Since the user goes on a very discrete, sequential journey in this website, creating a website in php (which requires reloading a new page in order to show or obtain new information) was a natural choice. Future iterations of the website may use JavaScript to perform client-side form validation, which would be useful if the user has many cities. In the current site, if the user accidently provides the wrong cost information (e.g. a non-integer value) for one of the travel options he/she would only find out after submitting the form. They would then have to resubmit the entire cost matrix again.

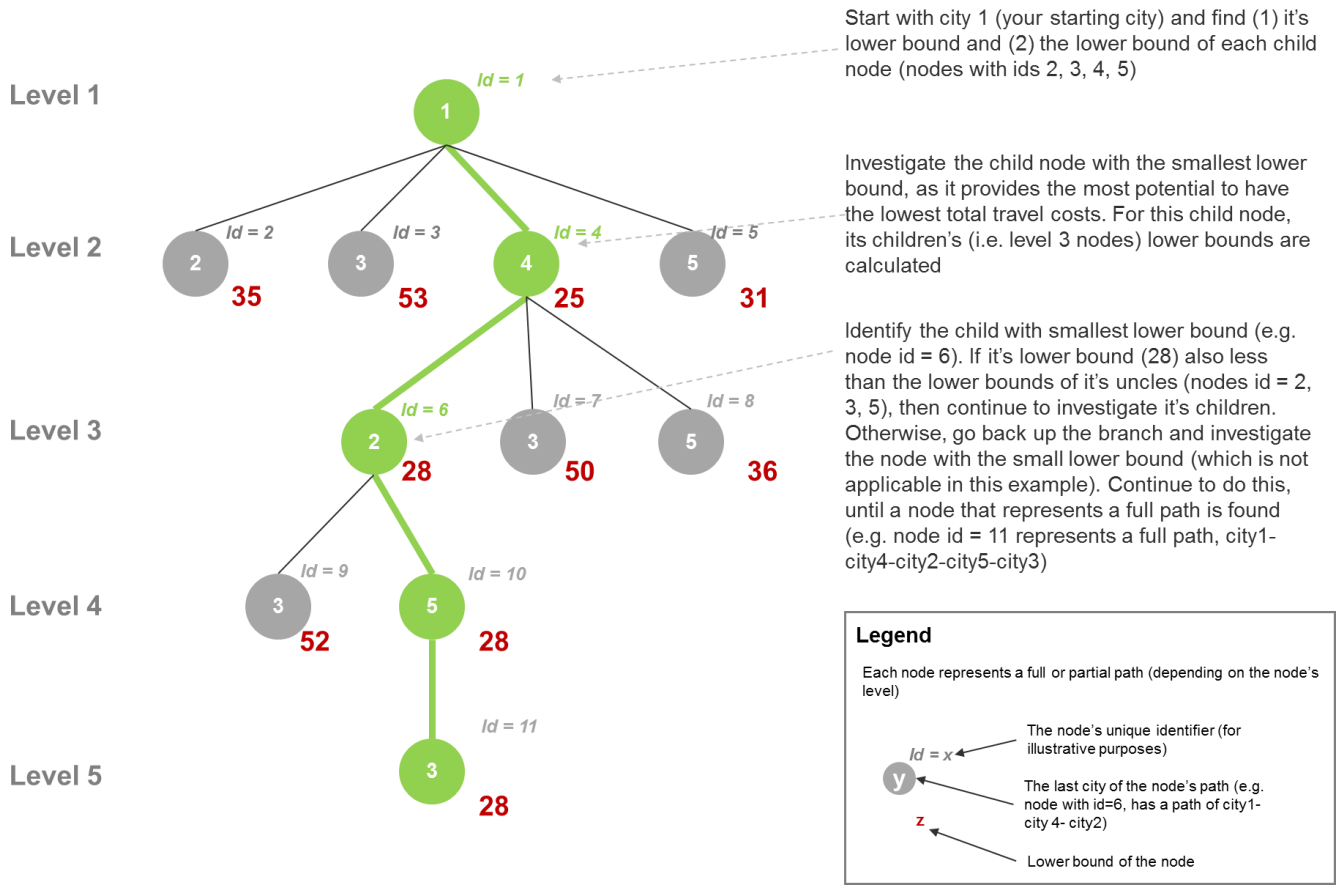
- **Php has functions that allowed external program to be called:** PHP has several programs (e.g. shell_exec) that allows external programs to be called. Since the optimization algorithm was written in the c-language for performance reasons, a method was needed that would allow the web-program (php) to call an external program (TSP.exe). While shell functions might be vulnerable to code-injection attacks, this is mitigated by the fact that the user inputs are checked to ensure that only numbers (i.e. no special characters) are provided.

3. Branch and Bound Approach to Solving the Travelling Salesman Problem

The travelling salesman problem has n! (n factorial) possible solutions, making it a computationally intensive problem. A branch and bound approach can be used to limit the number of solutions investigated. In the travelling salesman problem, solutions can be mapped to a rooted tree. Rather than evaluating all solutions within each branch, a branch's lower estimated bound is evaluated, and the branch is discarded if it cannot produce a better solution than the best one found so far.

In the travelling salesman problem, all nodes represent a full or incomplete path. For example, node with id 6 represents the incomplete path through cities 1-4-3 whereas node with id 11 represents a complete path.

The nodes are evaluated using the branch and bound approach as follows:



For more information on how the lower bounds are calculated, please consult the following resource: saurabhschool.org.

3.1 TSP.c design overview

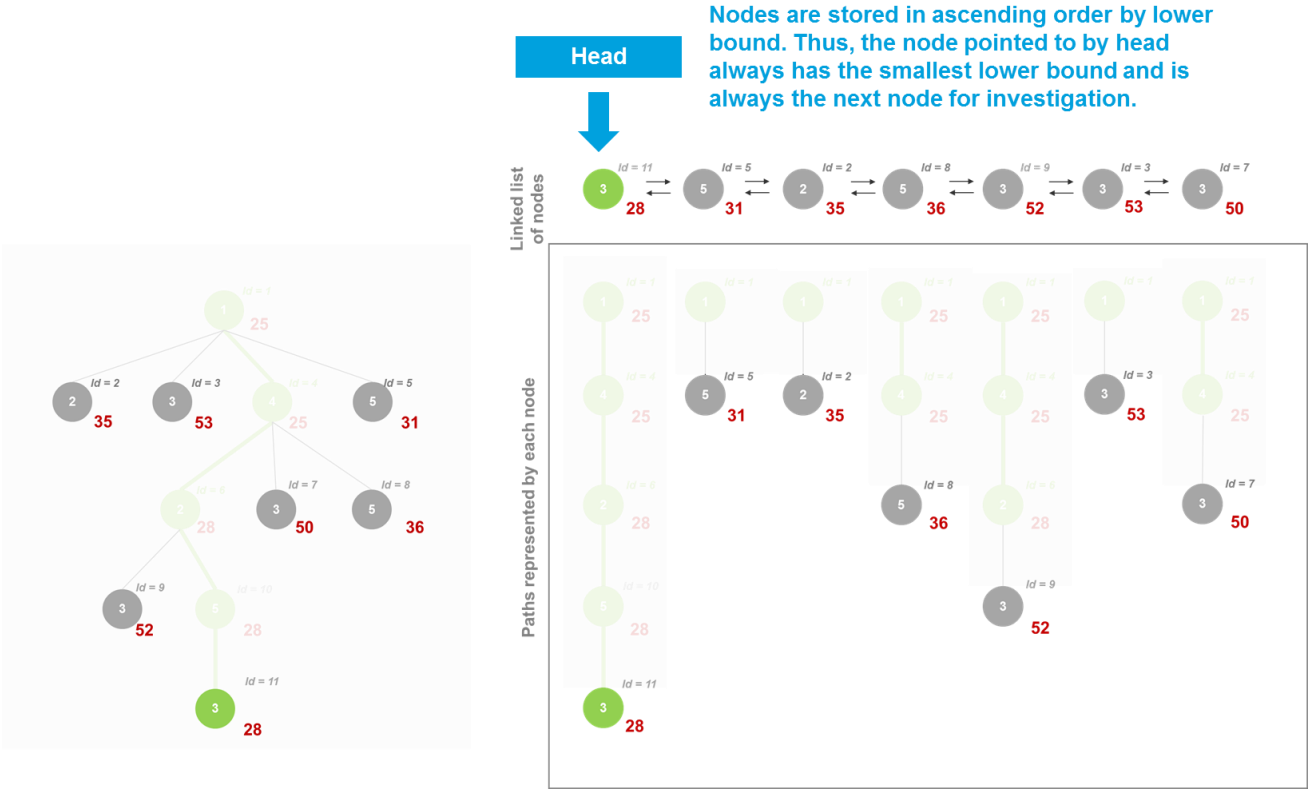
While nodes can be conceptually thought of as belonging to a tree (as above), for the purposes of this program, the nodes are stored in a linked list in ascending order (based on their lower bound value). This way, the program simply needs to evaluate the first node in linked list, as this node will always have the smallest lower bound. This avoids the need perform recursion on a tree of nodes and avoids the need to compare a child's lower bounds with (1) its siblings nodes and (2) its uncle nodes. This is done implicitly by storing the node in its proper place in the linked list.

In order for this methodology to work:

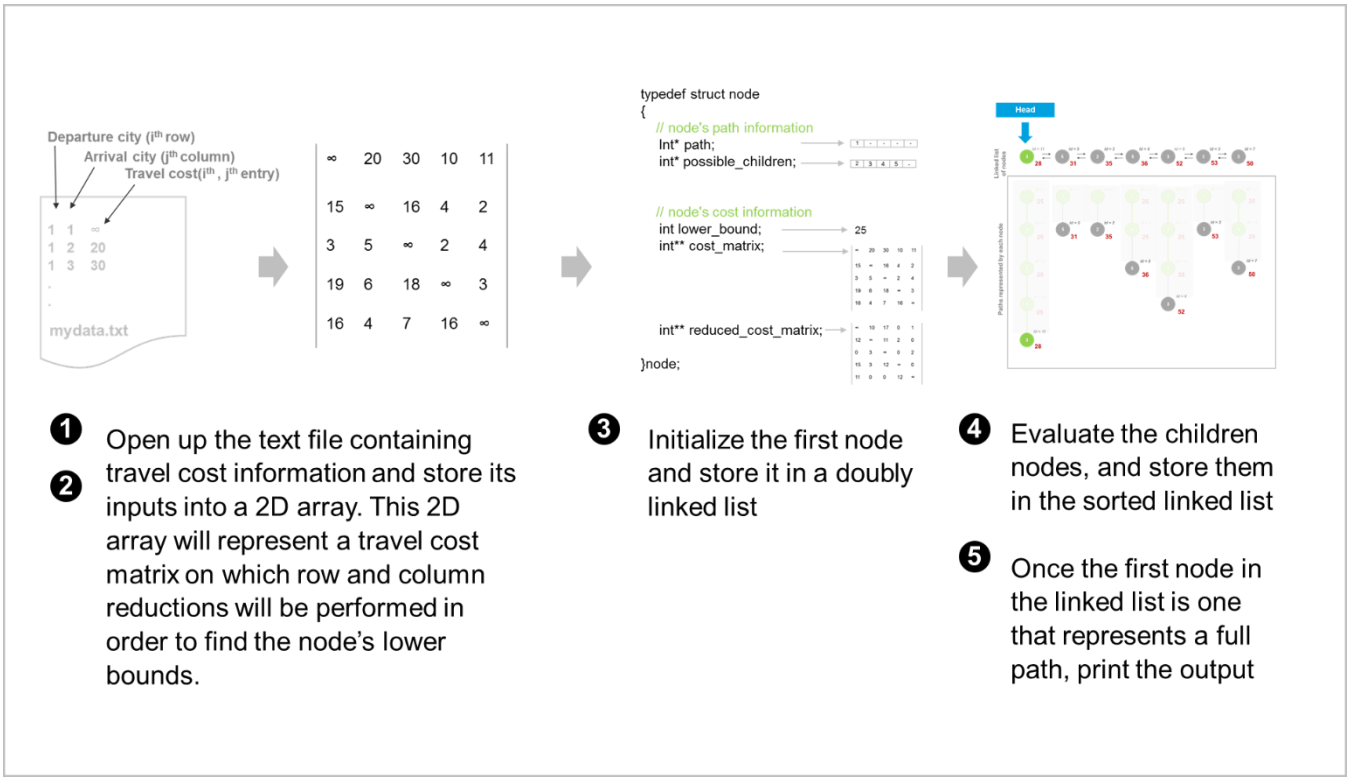
(1) **Only the *leaf nodes* can be stored in the linked list**. For example, node with id = 11 is stored in the linked list, but not its parent (node with id = 10). Only paths that still need to be evaluated are stored in the linked list. Since the *parent* of node with id = 11 was already evaluated (in order to get to node id = 11), it should not be stored in the linked list.

(2) **Each node must be a struct** which contains several pieces of information. Since each evaluation of a node is performed 'net new' (i.e. without the parental context provided by recursion), information about the node must be stored in a struct. Specifically, the following information about a node must be persistent: (1) the path represented by the node, (2) its possible children and (3) its reduced cost matrix.

(3) The program stops evaluating when a node representing a full path is at the beginning of the linked list.



To implement this methodology, the main function of TSP.c performs the 5 main steps:



1. Open up the text file containing
2. travel cost information and store its inputs into a 2D array. This 2D array will represent a travel cost matrix on which row and column reductions will be performed in order to find the node's lower bounds.

3. Initialize the first node and store it in a doubly linked list

4. Evaluate the children nodes, and store them in the sorted linked list

5. Once the first node in the linked list is one that represents a full path, print the output