Aron Bauer

CS 3310-2

Final Project Report

**Task #1– Sudoku Puzzle Solver:**

The Sudoku class contains three static methods: Boolean isPromising(), Boolean solveSudoku(), and void print().

The isPromising() method takes a 2D array (the sudoku board), a row and column index, and the value to be added to the board at that position as arguments. The method will, check if this number is unique in the current row and column of the board, then it will check if the given number is already in the current √n x √n sub-grid (3 x 3 sub-grid of the 9 x 9 board) that the given row and column are in within the board, if the number is already there, either in the current row, column or sub-grid, the method will return false, else it will return true.

The solveSudoku() method takes an n x n 2D array representing the board, and n itself as the arguments; n must be a number that has an integer square root, IE: 4, 9, 16, 25… in order for the method to work. The method first finds the next row and column index of the next empty spot on the board, then it will try putting numbers 1 - n by utilizing the isPromising() method until that number is accepted. If there are no empty spaces left, the problem is solved and true is returned. Once the number is accepted, the method will recursively call solveSudoku() from an if statement to find the next number to put in the next empty spot, and the process will repeat. If the recursively called method returns true, the caller method will also return true signifying the puzzle is solved. If that recursively called method returns false, the caller method will try putting the next number between 1 and n in that empty spot. If the end of the loop is reached, the method will return false signifying that there is no solution to the puzzle.

The worst-case time complexity of this algorithm is O(N^(n^2)) where N is the number of rows/columns in the board, for there a N possible options for every empty spot in the board (N will be 9 in this project due to the board being 9 x 9). However, since most Sudoku game boards don’t start empty, the general time complexity will not be that large.

The greatest challenge with this problem was developing the isPromising() method; in particular, developing the correct calculations necessary to determine which sub-grid the given index was in within the board.

**Task #2– 0/1 Knapsack Problem:**

The Knapsack class contains the data members: integers n (the size of the two array data members), W (the Knapsack capacity) and integer arrays p (profits) and w(weights). The class contains two inner classes Node and NodeComparator. The class contains two functions: float bound(), and int solve().

The Node inner class contains data members: integers: level, profit, and weight, float bound, and an Integer ArrayList called solnList. The NodeComparator class implements Java’s Comparator interface. It overrides the interface’s function int compare() where it compares two Node objects’ “bound” data member, in which it returns a greater priority value (-1) if the first Node’s bound is greater, lower (1) if less, or the same (0) if they are equal. This comparator is used for a Java PriorityQueue of Node objects in the solve() method which will be explained later.

The bound() method takes a Node object as an argument. The function will return 0 if the Node’s weight is greater than the Knapsack’s capacity (W). Else the method will calculate the Node’s bound by summing the node’s profit with the profit of every other item in the knapsack that will fit in the knapsack with the added weight of the node. Then that sum is added to the product of the remaining weight in the knapsack and the profit per unit weight ratio of the last item in the knapsack. Then that number is returned as the bound.

The solve() method takes an Integer (maxProfit) and an Integer ArrayList (solnList) as arguments. The method created a Java PriorityQueue of Node objects with the higher the bound, the higher the priority. A Node v (parent Node) and a Node u (child node) are created, and maxProfit is set to zero. Node v is first initialized with a level of -1 (the root of the tree), its profit and weight are set to zero, then its bound is calculated. Node v is next added to the priority queue, and the method enters a loop that will not end until the priority queue is empty. Node v will then be assigned the Node at the top of the priority queue. The loop will continue to the next iteration if v’s bound is less than or equal to maxProfit.

Else Node u is set to the child of v which includes the next item in the list of possible knapsack items, by setting u’s level to one greater than v’s, u’s weight to v’s weight plus the next item’s weight, u’s profit to v’s profit plus the next item’s, and v’s solnList is copied to u’s. If u’s weight is less than or equal to the knapsack’s capacity, and if it’s profit is greater than maxProfit, maxProfit is assigned u’s profit, next u’s solnList is copied to the main solnList, then u’s level+1 (representing the item number in the knapsack) is added to the end of the main solnList. Node u’s bound is now calculated using the bound() function. If u’s bound is greater than maxProfit, u’s level+1 is added to its own solnList, then u is added to the priority queue. Next, Node u is set to the child of Node v but excluding the next item in the knapsack. Node u’s bound is recalculated, and if this u’s bound is also greater than maximum profit, it will also be added to the priority queue. When the loop is finished (the priority queue is empty), the method returns maxProfit.

The worst-case time complexity for this algorithm is O(2^n) where n is the number of items in the knapsack. For a binary tree is created with a depth that is the number of items in the knapsack. In the worst-case scenario all of these nodes will be created in the tree. However, due to optimization of branch and bound pruning and the use of best-first search, the algorithm will rarely reach that upper bound.

The greatest challenge to this algorithm was the conversion of the pseudocode to Java, especially the Node object, for the nodes were just structs in the pseudocode, but the nodes had to be objects in this Java implementation. This lead to problems which requiring the solution of not being able to add Node u directly to the priority queue, but rather requiring a copy of Node u to first be made, then add that copy to the priority queue, because the node that Node u referenced was going to be constantly be overwritten throughout the algorithm’s execution.

**Task #3 - Traveling Salesman Problem:**

The TravelingSalesman class has the data members integer n (number of vertices) and an n x n integer adjacency matrix W. The class contains the inner classes Node, and NodeComparator, and the class contains the functions int length(), int bound(), and int travel().

The Node inner class contains the data members: integers: level, bound, and an Integer ArrayList called path. The NodeComparator class has the same function and use as the NodeComparator inner class from the Knapsack class, except it takes the lower the Node’s bound, the higher the priority, which is the opposite of the Knapsack version.

The length() function takes a Node as its argument. If the Node’s path contains less than 2 elements, the method returns 0. Else, the method will obtain the node’s path’s length by summing the length of the edges from each vertex in the Node’s path to the next. Then the method will return that sum.

The bound() function takes a Node u as its argument. The method creates an ArrayList of Integers that will hold the minimum possible edges of each vertex allowed in the path; it is called minimums. If Node u’s path’s size greater than one, the node’s path’s length is added to minimums. Next the method creates a Boolean array of size n vertices that is called visited, and all the elements in visited that have a corresponding index to a vertex in u’s path, aside from the first vertex, are set to true. The method next enters a nested loop that will complete n^2 operations by searching adjacency matrix W. At each iteration of the outer loop, a Boolean newMinFound is set to false, and an integer min is set to the greatest possible value. If the current row in W corresponds to a vertex in the u’s path, except for the last vertex, the loop will continue to the next iteration. Else, inside the nested loop: if the current row corresponds to the last vertex in the node’s path, and the current column corresponds to vertex 1, continue the nested loop’s iteration. Else, if the current column corresponds to a visited vertex, also continue the loops iteration. Else, if the edge in W[i][j] doesn’t equal zero, and if min is greater than W[i][j], set min to W[i][j], and set newMinFound to true, then continue the loop’s iteration. At the end of the outer loop, if a new minimum was found, add min to the ArrayList minimums. Finally, sum all the elements in minimums, then return that sum, which represents the Node’s bound.

The travel function takes an Integer ArrayList called opttour, and an integer minlength as arguments. This method follows a similar process to the Knapsack class’ solve method. It creates two Node objects v (parent), u (child), and a Priority Queue of Nodes. Node v’s level is set to zero, vertex 1 is added to v’s path, and v’s bound is calculated. Integer minlength is set to the maximum possible value, and v is added to the priority queue. The method then enters a loop that will continue until the priority queue is empty. Node v is assigned the Node at the top of the priority queue, and the loop will continue to the next iteration if v’s bound is greater than or equal to minlength.

Else, Node u is set to a child of v by increasing u’s level to one more than v’s. Then the method enters a nested loop, that will check every vertex from 2 to n. If v’s path contains the current vertex, the nested loop will continue to the next iteration. Else, Node u’s path is set to v’s path, then u adds the current vertex to the end of its path. Node u is now one of two kinds of Nodes, either u’s path completes a tour, or it doesn’t: a final Node or a non-final Node. Node u is considered to have completed a tour if its level is n – 2; that means there is only one non-visited vertex not in the Node’s path. Therefore, that last unvisited node is added to the final u’s path followed by vertex 1 in order to complete the tour (starting from vertex 1). If u’s path’s length greater than or equal to minlength, the nested loop continues. Else minlength is assigned the length of u’s path, and u’s path is copied to opttour (the optimal path).

If Node u is not considered final, it’s bound is calculated. Next, if its bound is less than minlength, a copy of the u is added to the priority queue, then the nested loop continues. Finally, once all the loops have exited, minlength is returned by the method.

The algorithm’s worst-case time complexity is O(n!) where n is the [number of vertices – 1]; this is the case where every possible path is checked. However, just like in Task #2, the optimization gained by branch and bound pruning with best-first search allows this algorithm to often have a time complexity less than factorial.

The greatest challenge to implementing this algorithm was coming up with the bound() method, for its pseudocode was not given. Much trial and error, and rethinking of different approaches was required to develop the said method.