Nu Le

Professor Tannaz R. Damavandi

Final Project

May 8th, 2020

TASK 1 – SUDOKU PUZZLE SOLVER

1. Code Documentation

In Sudoku Puzzle Solver, the following methods are implemented:

- **boolean SudokuSolved (int[][] grid)** – this method returns true if the parameterized grid has no more cell to fill.
- **boolean isPromising (int[][] grid, int row, int column, int number)** – this method takes in the modified grid, the number in the range 1-9 that is going to be examined, and the row and column of the cell that the given number is in. It returns true if the given number is unique in its corresponding row, column, and its own submatrix.

To arrive at a solution, the algorithm first traverses the matrix to find an empty cell that needs to be filled. If an empty cell is found, the flag isSolved is set to false. It then remembers the row and column value of the mentioned empty cell and proceeds to explore the solution space if the cell is filled with a number from 1 to 9. The solution space is constructed by checking if the number, after being filled in the empty cell, is a promising move. In other words, it will check if the number is unique within its row, column and submatrix. If that's the case, the updated grid with the newly filled cell is passed into a recursive call in order to find the next empty cell. This process repeats until either the whole grid is filled or a number is not promising. If a number is found to be not promising, the algorithm will back track and perform pruning by moving on to a new number. It then carries out the same process all over again.

2. Time Complexity

The recursion tree here will branch into 9 branches and at each level the problem size is reduced by 1. The problem can be designed for a grid size of N*N where N is a perfect square. For such a N, let M = N*N, the recurrence equation can be written as

$$\mathbf{T(M) = 9*T(M-1) + O(1)}$$

where T(N) is the running time of the solution for a problem size of N. Solving this recurrence will yield $O(9^M)$, i.e. $\mathbf{O(9^{N*N})}$. This is an exponential solution.

TASK 2 – 0/1 KNAPSACK PROBLEM

1. Code Documentation

In 0/1 Knapsack Problem, a helper class **KnapsackTools** is implemented to handle all **Best First Search** and **Branch and Bound** operations. The main methods in this class are as follow:

- **int knapsack_BFS_BB(int n, ArrayList profit, ArrayList weight, int W)** – this method returns the maximum profit found in the Knapsack.
- **Item[] sortObject(int n, ArrayList profit, ArrayList weight) –** this method returns the objects that are sorted in descending order in regards to their profit per unit weight value.
- **int bound(int n, ArrayList profit, ArrayList weight, int W, Node node) –** this method return the bound calculated up until that node.
- **void printKnapSack(int W, ArrayList weight, ArrayList profit, int n) –** this method prints the optimal set of objects that made up the maximum profit.

In the first step, all the objects are sorted based on their profit per unit weight value. Then the two arrays of weight and profit are changed to the according order. Next, root node is initialized with level -1 and its appropriate profit, weight, and bound. This is followed by enqueuing the root node into the priority queue. Once we enter the while loop, the removal of the node in front of the queue will ensure the node with largest bound is obtained, which also indicates that there is no unnecessary visits to the other nodes that are unpromising. Here, the bound is achieved by acting greedy much like in the Fractional Knapsack Problem at the node's current level. As the promising node is removed, its children are extracted. The left child profit is set to be the maximum profit if its profit is larger than the maximum profit so far and its weight has not gone over the Knapsack capacity. Furthermore, the children nodes are added into the priority queue only if their bound is larger than the current maximum profit. This process repeats until there is no more node left in the priority queue.

Regarding printing the optimal set, a 2D array is used. Again, much like in the Fractional Knapsack Problem, profits are stored in cells, and maximum profit is traced backwards to extract the appropriate objects that made up the sum.

2. Time Complexity

At each stage - or each level - of the solution tree, a node can have two options, to take the next object (the left child – 1) or to ignore it (the right child – 0). In the worst case, the algorithm can generate all possible solutions; and therefore, go through all the leaves. At that point, the solution tree is complete, which will result in the time complexity of **O(2^N)** where N is the number of objects inside the Knapsack.

TASK 3 – TRAVELING SALESMAN PROBLEM

1. Code Documentation

In Traveling Salesman Problem, similar to task #2, a class TSPTools is created to handle all the **Branch and Bound** operations. The class main methods are as follow:

- **int solve(int[][] costMatrix) –** this method returns the distance of the optimal tour.
- **int calculateCost(int[][] reducedMatrix) -** this method returns the bound/cost of the tour to reach a city.
- **int[][] rowReduction(int[][] reducedMatrix, int[] row) –** this method reduce each row in such a way that there must be at least one zero in each row.
- **int[][] columnReduction(int[][] reducedMatrix, int[] col) –** this method reduce each column in such a way that there must be at least one zero in each column.
- **Node newNode(int[][] parentMatrix, ArrayList<Pair> path, int level, int i, int j) -** this method returns a new node with appropriate cost, vertex, level, path, and reduced matrix in regards to those of its parent.

The bounding function requires that the algorithm obtains a reduced matrix from the original matrix. The lower bound is computed by taking the sum of the cheapest way to leave each city plus any additional cost to enter each city through the calculateCost method. The removal of the node in front of the priority queue guarantees a node with lowest bound so far. After comparing this bound with the current minimum cost, this node child is extracted through the newNode method, along with its bound and corresponding path, level, vertex, reduced matrix value. If the bound of this child node is feasible, it is enqueued into the priority queue. After a city is visited, the path that leads to this node is set to infinity to ensure it will not be visited twice.

2. Time Complexity

Similar to 0/1 Knapsack Problem using Branch and Bound, each node splits the solution into two groups: those that include a particular edge and those that exclude that edge. This will result in 2^N number if nodes for each city, and a tour have N cities that need to be traversed (assuming N is the number of cities in the problem space). Therefore, the time complexity of this algorithm is **$O(2^N * N)$.**