Report on Traveling Salesperson

Sabal Subedi
Master's in Computer Science
Idaho State University
Pocatello, ID, 83209 USA
sabalsubedi@isu.edu

April 17, 2024

Abstract

This project report summarizes the use of the branch and bound algorithm to solve the traveling salesperson problem.

1 Introduction

Branch and bound is a method for solving optimization problems by breaking them down into smaller sub-problems and using a bounding function to eliminate sub-problems that cannot contain the optimal solution. The algorithm depends on efficient estimation of the lower and upper bounds of regions/branches of the search space. If no bounds are available, the algorithm degenerates to an exhaustive search.
The algorithm explores branches of the tree, which represent subsets of the solution set. Before enumerating the candidate solutions of a branch, the branch is checked against upper and lower estimated bounds on the optimal solution, and is discarded if it cannot produce a better solution than the best one found so far by the algorithm.

2 Pseudocode and Asymptotic Analysis

2.1 Pseudocode for branch and bound algorithm

```
1     def branchAndBound(self, time_allowance=60.0):
2         results = {"pruned": 0, "total": 0, "max": 0,
              "count": 0}
3         cities = self._scenario.getCities()
4         ncities = len(cities)
5         foundTour = False
6         # use priority queue to keep track of next
              subproblem to search
7         priority_queue = []
8         # dictionary mapping priority hashing to sub
              problem
9         priority_dict = {}
10        cost, matrix =
              self.create_initial_matrix(cities)
11        # initial BSSF using greedy approach
12        greedy_bssf = self.greedy()
13        if greedy_bssf["cost"] != np.inf:
14            foundTour = True
15            bssf = BSSF(
16                greedy_bssf["cost"], [city._index for
                      city in greedy_bssf["soln"].route]
17            )
18        else:
19            bssf = BSSF(np.inf, None)
20        start_time = time.time()
21        timed_out = False
22        initial_node = 0
23        cur_path = [initial_node]
24        initial_problem = SubProblem(cost, matrix,
              cur_path)
25        results["total"] += 1
26        # update the priority queue
27        self.add_to_queue(initial_problem,
              priority_queue, priority_dict)
```

```
28          while len(priority_queue) != 0:
29              # sort the priority queue
30              priority_queue.sort(reverse=True)
31              cur_problem_score = priority_queue.pop()
32              cur_problem =
                   priority_dict[cur_problem_score]
33              for j in range(ncities):
34                  # Check timeout
35                  if time.time() - start_time >
                       time_allowance:
36                      timed_out = True
37                      break
38                  # expand the subproblem
39                  problemChild =
                       self.expand_subproblem(cur_problem,
                       j, results)
40                  if problemChild is None:
41                      continue
42                  results["total"] += 1
43                  if len(problemChild.path) == ncities:
44                      if (
45                          problemChild.matrix[problemChild.path[-1],
                               problemChild.path[0]]
46                          != np.inf
47                      ):
48                          foundTour = True
49                          problemChild.cost +=
                               problemChild.matrix[
50                                 problemChild.path[-1],
                                       problemChild.path[0]
51                          ]
52                          # update the BSSF
53                          if problemChild.cost <
                               bssf.cost:
54                              results["count"] += 1
55                              bssf.cost =
                                   problemChild.cost
```

```
56                              bssf.route =
                                   problemChild.path
57                              continue
58                          # prune the child
59                          else:
60                              results["pruned"] += 1
61                      else:
62                          results["pruned"] += 1
63                  elif problemChild.cost >= bssf.cost:
64                      results["pruned"] += 1
65                  else:
66                      self.add_to_queue(problemChild,
                           priority_queue, priority_dict)
67                      if len(priority_queue) >
                           results["max"]:
68                          results["max"] =
                               len(priority_queue)
69              # check timeout
70              if timed_out:
71                  break
72          citiesPath = [cities[index] for index in
               bssf.route]
73          bandbSolution = TSPSolution(citiesPath)
74          end_time = time.time()
75          # update the result
76          results["cost"] = bandbSolution.cost if
               foundTour else math.inf
77          results["time"] = end_time - start_time
78          results["soln"] = bandbSolution
79          return results
```

Time complexity and space complexity

1. to add cities in priority queue takes $O(1)$ time and $O(n)$ spaces
2. outer loop iterates over all the cities $O(n)$ times
3. to sort and update the cities in priority queue it takes $O(n \log n)$ times and $O(n)$ spaces
4. inner loop iterates over all the expanded state $O(n!)$ times
5. while expanding the subproblem, generating successor states takes $O(n!)$ times and $O(n!)$ spaces and to compute the reduced cost matrix it takes $O(n^2)$ times and $O(1)$ space
6. to update the results it takes $O(n)$ times and $O(n)$ spaces

$$\begin{aligned} Time\_complexity =& O(1) + O(n) * (O(n \log n) + O(n!) * (O(n!) + O(n^2) + O(n))) \\ \approx& O(n!) \\ Space\_complexity =& O(n) + O(n) * (O(n) + O(n!) + O(n!) + O(1) + O(n)) \\ \approx& O(n!) \end{aligned} \tag{1}$$

2.2 Pseudocode for greedy algorithm

```
1     def greedy(self, time_allowance=60.0):
2         cities = self._scenario.getCities()
3         cost, matrix =
             self.create_initial_matrix(cities)  #
             extract initial cost
4         results = {
5             "cost": math.inf,
6             "time": 0.0,
7             "soln": None,
8             "count": 0,
9             "max": None,
10            "total": None,
11            "pruned": None,
12        }
13        start_time = time.time()
14        # Iterate over possible starting nodes
15        for initial_node in range(len(cities)):
16            greedy_matrix = matrix.copy()
17            greedy_cost = cost
18            path = [initial_node]
19
20            while len(path) != len(cities):
21                # Check timeout
22                if time.time() - start_time >=
                     time_allowance:
23                    return results
24
25                min_val, col_index =
                     self.find_next_city(greedy_matrix,
                     path)
26                # No valid path found
27                if min_val == np.inf:
28                    break
29                self.update_greedy_matrix(greedy_matrix,
                     col_index, path)
30                greedy_cost += min_val
```

```
31              # complete tour found
32              else:
33                  greedy_cost +=
                       greedy_matrix[path[-1],
                       initial_node]
34                  cities_path = [cities[i] for i in
                       path]
35                  solution = TSPSolution(cities_path)
36                  if solution.cost < results["cost"]:
37                      results["cost"] = solution.cost
38                      results["time"] = time.time() -
                           start_time
39                      results["soln"] = solution
40                      results["count"] = 1
41          return results
```

Time complexity and space complexity

1. outer loop iterates over each possible starting node which takes $O(n)$ times
2. inner loop iterates over all cities to find the next city with minimum cost edge, taking $O(n)$ times
3. to find next city with minimum cost edge takes $O(n)$ times
4. to update the greedy matrix takes $O(1)$ times
5. to store greedy matrix takes $O(n^2)$ space

$$\begin{aligned} Time\_complexity &= O(n) * O(n) * (O(n) + O(1)) \approx O(n^3) \\ Space\_complexity &= O(n^2) \end{aligned} \tag{2}$$

2.3 Pseudocode for helper functions

```
    # function to expand subproblem
    def expand_subproblem(self, subProblem, col,
       results):
        if subProblem.matrix[subProblem.path[-1],
           col] == np.inf:
            return None

        matrix_copy = subProblem.matrix.copy()
        matrix_copy[subProblem.path[-1]] = np.inf
        matrix_copy[:, col] = np.inf
        matrix_copy[col, subProblem.path[-1]] = np.inf

        reductionCost =
           self.reduce_cost_matrix(matrix_copy)

        new_cost = (
            reductionCost
            + subProblem.cost
            + subProblem.matrix[subProblem.path[-1],
               col]
        )
        new_path = subProblem.path.copy()
        new_path.append(col)
        return SubProblem(new_cost, matrix_copy,
           new_path)
```

```
    # function to compute reduced cost matrix
    def reduce_cost_matrix(self, matrix):
    lower_bound = 0
    for i in range(len(matrix)):
        row_min = min(matrix[i, :])
        if row_min != np.inf:
            matrix[i] -= row_min
            lower_bound += row_min
    for j in range(len(matrix)):
        col_min = min(matrix[:, j])
        if col_min != np.inf:
            matrix[:, j] -= col_min
            lower_bound += col_min
    return lower_bound
```

```
    # function to compute initial matrix
    def create_initial_matrix(self, cities):
        initMatrix = np.full((len(cities),
           len(cities)), np.inf)
        reductionCost = 0
        for i, city_i in enumerate(cities):
            for j, city_j in enumerate(cities):
                if i != j:
                    initMatrix[i, j] =
                       city_i.costTo(city_j)
        reductionCost =
           self.reduce_cost_matrix(initMatrix)
        return reductionCost, initMatrix
```

```
    # function to add items in queue
    def add_to_queue(self, subProblem,
       priority_queue, priority_dict):
        hash_value = subProblem.cost /
           len(subProblem.path)
        priority_queue.append(hash_value)
        priority_dict[hash_value] = subProblem
```

Time complexity and space complexity

1. to expand the subproblem it takes $O(n^2)$ times
   and $O(n^2)$ space to copy matrix
2. to compute the reduced cost matrix it takes $O(n^2)$ times
   and $O(n^2)$ space
3. to compute the initial cost matrix it takes $O(n^2)$ times
   and $O(n^2)$ space
4. to add items in queue it takes $O(1)$ time and $O(1)$ space

3 Observations and Results

3.1 Observation Table

.

Cities	Seed	Running time (sec)	Cost of best tour found (*=optimal)	Max # of stored states at a given time	# of BSSF updates	Total # of states created	Total # of states pruned
15	20	0.47	9287	48	10	4560	3682
16	902	0.45	7940	53	2	3908	3329
10	150	0.04	8719	25	4	735	506
17	565	40.65	10353	99	9	410791	343510
19	624	60	10229	142	11	564055	476997
30	853	60	14238	629	11	307546	257236
41	256	60	20409	1720	12	209885	157572
50	172	60	20125	5631	1	149878	106740
12	83	0.70	7811	41	4	8896	6753
20	377	18.32	10940	130	7	134338	117916

Figure 1: Observation table with different number of cities and seed

In the figure (in figure 1), the seed represents the randomness, cities represents the number of cities, running time indicates time taken by program to compute branch and bound algorithm, cost of best tour indicates the optimal cost, max of stored states represents the number of item in priority queue, total number of states created and total number of states pruned.
The observation table reflects the exponential nature of the TSP problem i.e. the problem size increases as the number of states increases factorially. The number of pruned states also increases as the problem size increases. The algorithm effectively eliminates suboptimal paths using the lower bound criteria. This helps to reduce the the search space making algorithm more efficient.
However, we can still see the increment in time taken to solve the problem with increased problem size because increased problem size also increases the number of states to explore. This indicates that, the time complexity grows rapidly with problem size due to the factorial growth in the number of states to explore.

Mechanism for state space search
I used the branch and bound algorithm to solve the TSP problem. Unlike the brute force method, this method is effective. First, BSSF is computed using the greedy approach, then the state is branched into multiple smaller subproblems. For each subproblem, the lower bound is computed and then compared with the initial BSSF. All those subproblems with lower bound greater than the BSSF will be pruned. In this way, the algorithm works in two steps: branching the problem and bounding the subproblems.
To get the solution more efficiently, geting the BSSF using greedy approach potentially reduces the search space. Similarly, the lower bound criterion helps to eliminate the suboptimal paths early which reduces the number of states to explore. Also the use of priority queue helps to identify the potential subproblem that might give optimal solution. The cities (subproblems) are sorted in the priority queue which helps the program to choose the best subproblem and eventually get the optimal solution.

3.2 Results and Analysis

1. Priority queue - For insertion and removal of item from priority queue, it takes $O(n \log n)$ time and these operations are performed for every expanded state, which happens at most $O(n!)$ times. Thus, total time complexity is $O(n!)$ and space complexity is $O(n!)$
2. Search States - For generating the successor states, we need to iterate through every cities, which is $O(n)$ times and the operation is performed for every state, which happens at most $O(n!)$ times. Thus, total time complexity is $O(n!)$
3. Reduced Cost Matrix - To update the reduced cost matrix, it takes $O(n^2)$ times and space complexity is $O(n^2)$
4. BSSF Initialization - We use greedy approach i.e. choose the tour based on the minimal edge cost from the starting city. So, the time and space complexity is $O(n^2)$
5. Expanding on SearchState into others - Expanding the subproblem involves generating successor states and updating the priority queue. So, the time complexity is $O(n!)$

The data structure used to represent the states are matrix.

The priority queue data structure I used is the dictionary. Using the subproblem, we compute the hash value and it is added to priority dictionary as key and value pair.

To initialize the BSSF, I used the greedy approach. The starting city is selected randomly then the next city is chosen based on the next closest edge of the neighboring cities.

4 Conclusion

I coded the branch and bound algorithm to solve the traveling salesperson problem. I used the greedy approach to calculate the initial BSSF. The program computes the inital BSSF and checks the criteria i.e compares lower bound against sub optimal paths and prune the state.
The branch and bound algorithm works in two steps: expand the subproblems and then enforcing certain boundaries i.e. bound, the program reflects the exponential nature of the TSP problem. As the problem size increases, the number of possible permutations increases factorially. Then, the program prunes the states that maynot lead to the optimal solution.