Lab 5: Gene Sequencing

Garrett Price

December 3, 2022

Branch and Bound Analysis

Priority Queue

The implementation of the priority queue in my project was through the use of a heap. Insertion and removal from a heap queue both have a time complexity of $O(log(n))$. The space complexity of the heap is $O(n!)$, with n being the amount of cities, as the worst case would include every possible state in the queue.

SearchStates

```
    class State:
    def __init__(self, matrix, path, score, index):
    self.matrix = matrix
    self.path = path
    self.cost = score
    self.city_index = index

    def __lt__(self, other):

    return self.cost / len(self.path) < other.cost / len(other.path)
```

Above is the structure I used to represent each search state. Each state consists of a reduced cost matrix, an array representing the path up to this state, an integer value for the current cost, and another integer that represents the index of the city represented by the path. The matrix creation has the time complexity $O(n^2)$, with n being the amount of cities in the problem. The space complexity is the same, as n by n edges must be represented. The path array be at most n units since it will need to hold each city for it to be a complete tour. The score and index are both integers, and are negligible for both time and storage compared to the arrays above. When expanding one state into all potential next states, the time complexity is $O(n)$ as every next city must be checked.

Reduced Cost Matrix

The reduced cost matrix is an n by n matrix. The initial creation has the complexity $O(n^2)$ and it has the same space complexity. Updating the matrix at worst would also be $O(n^2)$ since it is possible that every element would need to be updated. In my implementation, a reduced cost matrix is created and then reduced with every state creation.

Initial BSSF

To obtain the initial BSSF, I implemented a greedy search. This greedy search takes the shortest edge to an unvisited node until a full tour has been formed. This has the time and space complexity of $O(n)$ as it needs to visit and store each city in the order it is visited. This is fast but not optimal, so it provides a good upper bound. It may need to run more than one if the first one is not a complete tour. Worst case would be that it

never finds a tour and it times out while searching for a greedy solution, but that is very unlikely. Overall, it is still O(n) with some constant factor depending on the amount of times it must search.

Complete Algorithm

The complete branch and bound algorithm puts all of these pieces together. First, the first BSSF is obtained by running a greedy search, O(n). Next, the values are all initialized. There are a few integers that take O(c) space and then there is the reduced-cost matrix that takes O(n^2) space. The matrix also takes O(n^2) to be created and O(n^2) to be updated. This initial creation of the matrix is done as part of creating the first state. The creation of the state objects also takes O(n) time and space due to the creation of the path array, but that is dominated by the O(n^2) time and space of the matrix created above. With every state creation, there is an insertion into the priority queue, which is complexity O($log(n)$). Up until this point, the time complexity is O($2n^2 + 2n + log(n)$), simplified to be simply O(n^2).

After the initial state is created, the algorithm begins iterating through the priority queue. Going through the queue can be at worst O($n!$) if every possible state is stored and searched. The queue can also take up O($n! * n^2$) space if every state is scored. Inside each step of the queue, there is a removal of the next object, O($log(n)$), followed by a loop that iterates through the next possible states. This loop has the time complexity of O(n) since it checks every city. A new state is created, O(n^2), in each iteration of the loop. The amount of states created does decrease as the current path gets longer and there are less next potential states, but worst case $n - 1$ states are created. Lastly, the states are added to the heap with the time complexity O($log(n)$), so we will simplify the expansion of states to be O($log(n) * n - 1 * n^2$) or O(n^3).

Overall, the time complexity of the branch and bound algorithm is O($n^2 + (n! * n^3)$). O(n^2) from the creation and setup before the expansions, O($n!$) from iterating through the queue, and O(n^3) for the work done in each iteration. This can be simplified to O($n!$). The branch and bound method in practice can be faster than that due to the elimination of states and their substates before expanding them.

Results

Cities	Seed	Running Time	Cost of best tour	Max queue	BSSF updates	Total states created	Total states pruned
15	1	2.54	9536*	67	3	8250	6972
16	2	13.67	9934*	98	12	40881	33876
17	3	15.085	10381*	97	5	39181	33889
18	4	60	10688	115	3	152201	128860
19	5	37.172	10343*	122	13	78578	69529
20	6	60	10514	145	10	127735	109263
21	7	60	10709	157	11	111383	97684
22	8	60	12321	265	15	114740	93900
14	9	8.458	9373*	62	4	30070	24556
15	10	13.054	11469*	70	6	42311	35423

The pattern is that as the amount of cities increase, the time needed to find a solution increases. The increase in time is not at a linear rate, and it can appear random. While this problem is of O(n!) complexity, rarely are all n! states actually checked with the branch and bound method. The closer the inital BSSF is to the optimal solution, the more aggressively states can be pruned away. This leads to much smaller amounts of states being created and checked. An interesting comparison is the first and last tours I checked, both with length 15. The first one completed in just under 3 seconds, and created about 5 times less states than the last one. The time was also about 5 times less. This difference is likely due to the initial estimate done by the greedy search. As it starts from a random node, there is little control over the tour. Sometimes it can return a really close answer; as you can see, the first 15 tour only had to update the BSSF 3 times after the initial greedy search, showing it was very close and was an effective pruning method.

I tried using a randomly generated tour first for my initial BSSF, but the results were too inconsistent and I could never guarantee that my initial BSSF was actually going to prune anything. I decided to implement the greedy algorithm for the first search to get a good answer and it sped things up drastically and made the overall algorithm more consistent. I also changed the order in which the priority queue returned the next state many times. I tried doing a first-in first-out queue, but that was slow since it did a breadth-first style tour. I then switched it to a last-in first-out queue and improved it, but since it was just based on the order states were inserted I had no idea if the state I was choosing was good. I then implemented a priority queue and changed the key a few times. The first I had it sort by the current cost of the state, but that was pretty much just an implementation of A* and was slow. I then changed it to prioritize depth, but I ran into a similar problem with my LIFO queue. I then decided to mix the two and use a ratio of cost to depth to determine the next state to be expanded. This ratio worked great for quick and effective pruning since it could pick the state doing best for its relative depth and quickly check if it continued at a good rate for the rest of the tour.

Source Code

```
    def branchAndBound(self, time_allowance=60.0):
        # Get initial bssf, currently using random but if
        # I have the time I'll change it to greedy
        results = {}
        cities = self._scenario.getCities()
        random_result = self.greedy()
        bssf_cost = random_result['cost']
        bssf_route = random_result['soln']

        # Initialize needed values
        queue = []
        solution_count = 0
        max_queue_size = 0
        current_queue_size = 0
        minimum_cost = 0
        pruned = 0
        nodes_generated = 0
        ncities = len(cities)
        reduced_cost_matrix = np.empty((ncities, ncities))
        for i in range(0, ncities):
            for j in range(0, ncities):
                reduced_cost_matrix[i][j] = cities[i].costTo(cities[j])

        # creates initial reduced cost matrix
        reduced_cost_matrix, minimum_cost = self.minimizeMatrix(
                                            reduced_cost_matrix)
        start_time = time.time()

        #Generate initial state with first city
        starting_city = cities[0]
        init_state = State(reduced_cost_matrix, np.array([starting_city])
                                            , minimum_cost,
                                            starting_city._index)

        #Initialize queue with first state
        heapq.heappush(queue, init_state)
        current_queue_size += 1
        max_queue_size = 1

        #Begin searching
        while (len(queue) != 0 and time.time()-start_time <
                                            time_allowance):

            #Pop next possible solution off queue
            current_state = heapq.heappop(queue)
            current_queue_size -= 1

            #Check if current state is still good
            if(current_state.cost < bssf_cost):

                #Start creating next potential states
```

```
            for city in cities:

                #If the city has already been visited,
                #don't create a state
                if city not in current_state.path:

                    #If there is no connecting edge, go to next city
                    distance = current_state.matrix[current_state.
                                                    city_index,
                                                    city._index]
                    if(distance == math.inf):
                        continue

                    #Create a substate with every valid city
                    nodes_generated += 1

                    #Copy the previous state's matrix
                    new_matrix = np.copy(current_state.matrix)

                    #Reduce the copied matrix
                    new_matrix[:,city._index] = math.inf
                    new_matrix[current_state.city_index,:] = math.inf
                    new_matrix[city._index, current_state.city_index]
                                                     = math.inf
                    new_matrix, cost_to_adjust = self.minimizeMatrix(
                                                    new_matrix)

                    #Copy the path from the previous state
                    #Add the current state's city to its path
                    new_path = np.copy(current_state.path)
                    new_path = np.append(new_path, city)

                    #Calculate the lowerbound of the current state
                    new_score = current_state.cost + distance +
                                                    cost_to_adjust


                    #Create state object
                    new_state = State(new_matrix, new_path, new_score
                                                    , city.
                                                    _index)

                    #Check if it is a complete path
                    if(len(new_state.path) == ncities and city.costTo
                                                    (
                                                    starting_city
                                                    ) != math.
                                                    inf):

                        #Update bssf if needed
                        if(new_state.cost < bssf_cost):
                            bssf_route = TSPSolution(new_state.path)
                            bssf_cost = bssf_route.cost
```

```
                    solution_count +=1

                #If the path is not complete,
                #add or prune this state
                elif new_state.cost < bssf_cost:
                    heapq.heappush(queue, new_state)
                    current_queue_size += 1
                    if(current_queue_size > max_queue_size):
                        max_queue_size = current_queue_size
                else:
                    pruned += 1
```