Calix Barrus

CS 312

Dr. Ventura

December 1, 2020

Project 5 Writeup

1. def branchAndBound( self, time\_allowance=60.0 ):  
    results = {}  
    cities = self.\_scenario.getCities()  
    ncities = len(cities)  
    count = 0  
    start\_time = time.time()  
     
    # Helper class that will be in the queue  
    class State:  
    *"""  
    State class represents partial solution to the TSP problem  
     
    route: List of cities representing the route constructed so far  
    reduced\_cost\_matrix: matrix of costs between graph vertices. Used to  
    calculate and update the lower\_bound  
    lower\_bound: Pessimistic lower bound on solutions to the TSP along  
    this partial route.  
    """* def \_\_init\_\_(self, route: List, reduced\_cost\_matrix: np.ndarray, lower\_bound: int):  
    *"""  
    This has O(1) temporal complexity and O(n^2) spatial complexity.* ***:param*** *route: List of cities* ***:param*** *reduced\_cost\_matrix: Reduced cost matrix updated with  
    respect to the cities currently in the route* ***:param*** *lower\_bound: Lower bound for the TSP updated with respect  
    to the given reduced\_cost\_matrix  
    """* self.route = route  
    self.reduced\_cost\_matrix = reduced\_cost\_matrix  
    self.lower\_bound = lower\_bound  
    # def \_\_lt\_\_(self, other):  
    # return self.lower\_bound < other.lower\_bound  
     
    def update\_route(self, new\_city: City) -> None:  
    *"""  
    Add the city to the route and update the reduced\_cost\_matrix and  
    lower\_bound accordingly  
     
    O(n^2) temporal complexity and O(1) spaitial complexity (given  
    that that cost has already been paid in the constructor).* ***:param*** *new\_city: City to be added to the route.  
    """* city\_index = cities.index(new\_city)  
    prev\_index = cities.index(self.route[-1])  
     
    # Account for the cost of traveling to the given city  
    self.lower\_bound += self.reduced\_cost\_matrix[prev\_index, city\_index]  
    # Cancel the column from which we travel  
    self.reduced\_cost\_matrix[prev\_index, :] = np.infty  
    # And the column to which we now travel  
    self.reduced\_cost\_matrix[:, city\_index] = np.infty  
     
    self.route.append(new\_city)  
     
    # Make sure there is a 0 in each column and row. Subtract from  
    # the entire column the least element to make it so, if necessary.  
    n = len(self.reduced\_cost\_matrix)  
    for i in range(n):  
    min\_index = np.argmin(self.reduced\_cost\_matrix[i, :])  
    if self.reduced\_cost\_matrix[i, min\_index] == np.infty:  
    # If a row or column not on the route has only infinities, there  
    # is no valid tour given the route so far.  
    if i not in [city.\_index for city in self.route[:len(self.route)-1]]:  
    self.lower\_bound = np.infty  
    elif self.reduced\_cost\_matrix[i, min\_index] > 0:  
    self.lower\_bound += self.reduced\_cost\_matrix[i, min\_index]  
    self.reduced\_cost\_matrix[i, :] -= self.reduced\_cost\_matrix[i, min\_index]  
     
    for i in range(n):  
    min\_index = np.argmin(self.reduced\_cost\_matrix[:, i])  
    if self.reduced\_cost\_matrix[min\_index, i] == np.infty:  
    if i not in [city.\_index for city in self.route[1:]]:  
    self.lower\_bound = np.infty  
    elif self.reduced\_cost\_matrix[min\_index, i] > 0:  
    self.lower\_bound += self.reduced\_cost\_matrix[min\_index, i]  
    self.reduced\_cost\_matrix[:, i] -= self.reduced\_cost\_matrix[min\_index, i]  
     
     
    def child\_state(self, new\_city):  
    *"""  
    Spin of a state similar to the current problem state but with  
    an additional city added to the route.  
    O(n^2) temporal complexity and O(n^2) spatial complexity due to  
    the creation of a whole new reduced\_cost\_matrix.* ***:param*** *new\_city: City to be added to the route* ***:return****: Child problem state of the current State  
    """* result = State(list(self.route), np.array(self.reduced\_cost\_matrix), self.lower\_bound)  
    result.update\_route(new\_city)  
    return result  
     
    cost\_matrix, lower\_bound = self.\_create\_cost\_matrix(cities, 0) # O(n^2) temporally and spatially  
    initial\_problem = State([cities[0]], cost\_matrix, lower\_bound) # O(n^2) temporally and spatially  
    total = 0  
    self.cost\_queue = [ (initial\_problem.lower\_bound, total, initial\_problem) ]  
    deepest\_state = None # Keep track of the best, deepest state seen so far  
    total += 1  
     
    # best solution so far  
    self.bssf = self.greedy()['soln'] # Worst case O(n^2) temporally, avg. case O(n) temporally. O(n) spatially in any case.  
     
    cost\_or\_depth = True  
    max\_queue\_length = 0  
    states\_trimmed = 0  
    while not len(self.cost\_queue) == 0 and time.time()-start\_time < time\_allowance:  
    # Alternate prioritizing lower bound or tree depth  
    if 0 != np.random.randint(0, 10): # About 1 in 10 times look at the deepest state  
    current = heapq.heappop(self.cost\_queue)[2] # O(log(n)) temporally  
    if deepest\_state is not None and current == deepest\_state[2]:  
    deepest\_state = None  
    else:  
    if deepest\_state == None:  
    current = heapq.heappop(self.cost\_queue)[2] # O(log(n)) temporally  
    else:  
    current = deepest\_state[2]  
    self.cost\_queue.remove(deepest\_state) # O(n) temporally  
    deepest\_state = None  
    heapq.heapify(self.cost\_queue) # O(n) temporally  
     
    if current.lower\_bound > self.bssf.cost:  
    states\_trimmed += 1  
    continue  
     
    # Complete tours will skip this for loop  
    # At worst, this loop (combined with the outer loop) will go through all n! different partial tour possibilites  
    for city in np.setdiff1d(cities, current.route, assume\_unique=True): # Look at cities not in the route  
    if current.route[-1].costTo(city) == np.infty: # No edge to city  
    continue  
     
    child = current.child\_state(city) # O(n^2) temporally and spatially  
    if child.lower\_bound < self.bssf.cost:  
    new\_state = (child.lower\_bound, total, child)  
    heapq.heappush(self.cost\_queue, new\_state) # O(log n)  
     
    # Remember the best, deepest state  
    if deepest\_state is None:  
    deepest\_state = new\_state  
    elif len(new\_state[2].route) >= len(deepest\_state[2].route) and new\_state[0] > deepest\_state[0]:  
    deepest\_state = new\_state  
    total += 1  
    else:  
    states\_trimmed += 1  
     
    # Complete tours  
    if len(current.route) == ncities:  
    deepest\_state = None  
    solution = TSPSolution(current.route) # O(n)  
    count += 1  
    if solution.cost < self.bssf.cost:  
    self.bssf = solution  
    if len(self.cost\_queue) > max\_queue\_length:  
    max\_queue\_length = len(self.cost\_queue)  
     
     
    if len(self.cost\_queue) > max\_queue\_length:  
    max\_queue\_length = len(self.cost\_queue)  
     
    states\_trimmed += len(self.cost\_queue)  
     
    end\_time = time.time()  
    results['cost'] = self.bssf.cost  
    results['time'] = end\_time - start\_time  
    results['count'] = count  
    results['soln'] = self.bssf  
    results['max'] = max\_queue\_length  
    results['total'] = total  
    results['pruned'] = states\_trimmed  
    return results  
     
   def \_create\_cost\_matrix(self, cities, starting\_city\_index) -> (np.ndarray, int):  
    *"""  
    Temporal and spatial complexity O(n^2)  
    """* n = len(cities)  
    lower\_bound = 0  
    result = np.zeros((n, n))  
    for i in range(n):  
    for j in range(n):  
    result[i, j] = cities[i].costTo(cities[j])  
     
    for row in result:  
    min\_index = np.argmin(row)  
    if row[min\_index] != np.infty and row[min\_index] > 0:  
    lower\_bound += row[min\_index]  
    row -= row[min\_index]  
     
    for col in np.rollaxis(result, 1):  
    min\_index = np.argmin(col)  
    if col[min\_index] != np.infty and col[min\_index] > 0:  
    lower\_bound += col[min\_index]  
    col -= col[min\_index]  
     
    return result, lower\_bound
2. The total temporal and spatial complexity of my Branch and Bound algorithm is O(n!n^2). I tried to include comments throughout the code to illustrate the temporal and spatial complexity of different parts.   
   The part of the algorithm that dominates the worst is the O(n^2) step in the center of the pair of loops. The loops iterate over all O(n!) possible partial path combinations, and the worst operation performed at the center of said loops is the O(n^2) creation of problem expansions to be added to the queue (this is the . child = current.child\_state(city) line of code). This function call dominates all other operations inside of the inner for loop, and the loops will repeat at worst O(n!) times, leaving us with O(n! n^2) complexity (temporal and spatial) for the algorithm.   
     
   To prioritize different problem states, we used a priority queue, which allowed O(log(n)) insertion and min\_deletion. Unfortuntely, as part of our heuristic, we grabbed and deleted a node not at the front of the queue, and so had to repeatedly heapify the array ( an O(n) operation).   
     
   Each Search State costs the algorithm O(n^2) temporally and spatially to create. The search states track the cities traversed so far, the lower\_bound on a solution to the TSP, and a reduced\_cost\_matrix that could be used to update said lower\_bound. The creation of the reduced\_cost\_matrix is what gives the state creation such a high temporal and spatial cost.   
     
   For the creation of the BSSF, I used a greedy algorithm that I coded up. Since the greedy algorithm only remembers a single route at a time, it as spatial complexity of O(n). Since it can at worst try the greedy algorithm starting from every state, it is has worst case temporal complexity of O(n^2), but on average merely O(n). This is dominated in the algorithm by the fat O(n!n^2) operation.
3. For the states, I a simple python list to keep track of the route so far, and a numpy array to keep track of reduced cost matrix. Though the creation of the state is O(n^2), the elements of the matrix have to be iterated over quite a few times. The use of a numpy array here ensures speedy iterations over the matrix and results in clean, slick python code.
4. I used the python heapq library to keep track of my priority queue. Each problem state was accompanied in a tuple by the lower\_bound and the sequence with which it was entered into the queue. Tuple comparison compares items sequentially, so the use of a tuple allowed the heapq library to sort the states by lower\_bound, and secondarily by insertion order. The use of insertion order was necessary to avoid ties, so that the states are never compared directly (as such a comparison would fail).
5. For the initial BSSF I coded and used the greedy algorithm that we talked about in class.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| # Cities | Seed | Running Time (s.) | Cost (\*= optimal) | Max # States Stored | # BSSF Updates | # States Created | # States Pruned |
| 15 | 20 | .511 | 10135\* | 1790 | 2 | 2240 | 2644 |
| 16 | 902 | 5.415 | 8760\* | 14429 | 5 | 17500 | 17185 |
| 20 | 542 | 60.041 | 12118 | 112818 | 5 | 126676 | 123972 |
| 25 | ﻿542 | 60.021 | 16666 | 126850 | 0 | 137498 | 134482 |
| 30 | 473 | 60 | 17004 | 92046 | 0 | 97840 | 97417 |
| 25 | 840 | 120 | 14255 | 149237 | 2 | 162372 | 157253 |
| 20 | 148 | 35.466 | 12383\* | 38060 | 14 | 47092 | 71575 |
| 20 | 390 | 120 | 12498 | 115872 | 1 | 134235 | 144869 |
| 22 | 787 | 120 | 18396 | 150065 | 4 | 165122 | 155725 |
| 22 | 536 | 120 | 14583 | 157299 | 1 | 173993 | 166263 |

1. One of the most surprising things to me is the variability of the time that it takes to find an optimal solution. For trials with ~15 states, the algorithm was very good, and could generally find an optimal solution given 60 seconds. Increasing to even the small number of 25 or 30 states drastically reduced, practically eliminated, the possibility that the algorithm would finish in time. One surprising result was one trial where the TSP problem was solved with 20 cities in about 35 seconds. Other similar trials failed to yield results in 120 seconds. It appears to be very much hit or miss with regard to whether the algorithm gets to take home the average case or something more along the lines of the worst case, with respect to temporal complexity. Note that the number of sites processed/pruned/stored get reduced as problem size increases as well. This tracks because each site takes n^2 time to process, so increasing the size of sites means we can hit substantially few sites. What surprised me is that the slowdown when we increased the number of sites doesn’t quite seem proportional to the O(n^2) of the storage and processing requirements. I wonder if this has to do with optimization under the hood of the numpy library.
2. Another mechanism I tried was to add to the priority of each result, a bigger number for those that were deeper, a smaller number to those that were more shallow. This attempt never really resulted in actually getting the queue to drill down to solutions however, as there would be a few hundred on the tip of the queue, all of whom were only ok, inflated because of the depth factor, and then most of whom wouldn’t end up even being paths anyways. After several attempts with no solutions, we switched to a strategy where we will remember and expand the best deepest node about every tenth run through the loop. This turned out to be quite successful, giving fairly good results.