from dataclasses import dataclass, field  
  
import numpy as np  
  
# node compatible with priority queue where path len is priority  
@dataclass  
class Node:  
 matrix: np.ndarray  
 path: list  
 cost: float  
 path\_len: int = field(init=False)  
  
 def \_\_post\_init\_\_(self):  
 self.path\_len = len(self.path)  
  
 def \_\_lt\_\_(self, other):  
 return self.path\_len > other.path\_len

#!/usr/bin/python3  
import itertools  
import time  
from contextlib import suppress  
from typing import List  
  
from models import Node  
from TSPClasses import \*  
from queue import PriorityQueue  
  
  
class TSPSolver:  
 def \_\_init\_\_(self, gui\_view):  
 self.\_scenario = None  
  
 def setupWithScenario(self, scenario: Scenario):  
 self.\_scenario = scenario  
  
 """   
 <summary>  
 This is the entry point for the default solver  
 which just finds a valid random tour. Note this could be used to find your  
 initial BSSF.  
 </summary>  
 <returns>results dictionary for GUI that contains three ints: cost of solution,   
 time spent to find solution, number of permutations tried during search, the   
 solution found, and three null values for fields not used for this   
 algorithm</returns>   
 """  
  
 def defaultRandomTour(self, time\_allowance=60.0):  
 cities = self.\_scenario.get\_cities()  
 ncities = len(cities)  
 foundTour = False  
 count = 0  
 bssf = None  
 start\_time = time.time()  
 while not foundTour and time.time() - start\_time < time\_allowance:  
 # create a random permutation  
 perm = np.random.permutation(ncities)  
 route = [cities[perm[i]] for i in range(ncities)]  
 bssf = TSPSolution(route)  
 count += 1  
 if bssf.cost < np.inf:  
 # Found a valid route  
 foundTour = True  
  
 end\_time = time.time()  
  
 return {  
 "cost": bssf.cost if foundTour else math.inf,  
 "time": end\_time - start\_time,  
 "count": count,  
 "soln": bssf,  
 "max": None,  
 "total": None,  
 "pruned": None,  
 }  
  
 def greedy(self, time\_allowance=60.0):  
 """  
 This is the entry point for the greedy algorithm  
  
 Time complexity: O(n^2)  
 Space complexity: O(n)  
  
 :param time\_allowance:  
 :return: results dictionary for GUI that contains three ints: cost of best solution,  
 time spent to find best solution, total number of solutions found, the best  
 solution found, and three null values for fields not used for this  
 algorithm  
 """  
 start\_time = time.time()  
  
 cities = self.\_scenario.get\_cities().copy()  
 random.shuffle(cities)  
 bssf = None  
  
 with suppress(StopIteration):  
 for start in cities:  
 route = [start]  
 remaining = cities.copy()  
 remaining.remove(start)  
  
 while remaining:  
 if time.time() - start\_time > time\_allowance:  
 raise StopIteration  
 nearest = min(remaining, key=lambda city: start.costTo(city))  
 route.append(nearest)  
 remaining.remove(nearest)  
 start = nearest  
  
 solution = TSPSolution(route)  
 if not bssf or solution.cost < bssf.cost:  
 bssf = solution  
  
 end\_time = time.time()  
  
 return {  
 "cost": bssf.cost,  
 "time": end\_time - start\_time,  
 "count": 1,  
 "soln": bssf,  
 "max": None,  
 "total": None,  
 "pruned": None,  
 }  
  
 def branch\_and\_bound(self, time\_allowance=60.0):  
 """  
 This is the entry point for the branch-and-bound algorithm  
  
 Time Complexity: O(n^3 \* 2^n)  
 Space Complexity: O(n^3 \* 2^n)  
  
 :param time\_allowance: float  
 :return: results dictionary for GUI that contains three ints: cost of best solution,  
 time spent to find best solution, total number solutions found during search (does  
 not include the initial BSSF), the best solution found, and three more ints:  
 max queue size, total number of states created, and number of pruned states.  
 """  
  
 start\_time = time.time()  
  
 queue = PriorityQueue()  
 cities: List[City] = self.\_scenario.get\_cities().copy()  
 # TC: O(n), SC: O(n)  
 city\_map = {city.index: city for city in cities}  
 ncities = len(cities)  
  
 count = 0  
 max\_queue\_size = 0  
 total\_states = 0  
 pruned\_states = 0  
  
 # Create initial matrix  
 # TC: O(n^2), SC: O(n^2)  
 matrix = np.full((ncities, ncities), np.inf)  
 for i, j in itertools.product(range(ncities), range(ncities)):  
 matrix[i, j] = city\_map[i].costTo(city\_map[j])  
  
 # Create initial node  
 # TC: O(1), SC: O(1)  
 node = Node(matrix, [cities[0]], 0)  
 queue.put(node)  
  
 # Create initial BSSF  
 # TC: O(n^2), SC: O(n)  
 bssf = self.greedy(time\_allowance)["soln"]  
  
 # TC: O(b^n), SC: O(b^n)  
 while not queue.empty() and time.time() - start\_time < time\_allowance:  
 node = queue.get()  
  
 max\_queue\_size = max(max\_queue\_size, queue.qsize())  
  
 if node.cost >= bssf.cost:  
 pruned\_states += 1  
 continue  
  
 if len(node.path) == ncities:  
 count += 1  
 bssf = TSPSolution(node.path)  
 continue  
  
 current\_city = node.path[-1]  
 i = current\_city.index  
 # TC: O(n), SC: O(1)  
 for j, city in city\_map.items():  
 if city in node.path:  
 continue  
  
 new\_cost = node.cost + node.matrix[i, j]  
  
 # TC: O(n^2), SC: O(n^2)  
 new\_matrix = node.matrix.copy()  
 new\_matrix[i, :] = np.inf  
 new\_matrix[:, j] = np.inf  
 new\_matrix[j, i] = np.inf  
  
 # TC: O(n), SC: O(1)  
 for k in range(ncities):  
 min\_cell = np.min(new\_matrix[k, :])  
 if min\_cell != np.inf:  
 new\_matrix[k, :] -= min\_cell  
 new\_cost += min\_cell  
  
 # TC: O(n), SC: O(1)  
 for k in range(ncities):  
 min\_cell = np.min(new\_matrix[:, k])  
 if min\_cell != np.inf:  
 new\_matrix[:, k] -= min\_cell  
 new\_cost += min\_cell  
  
 new\_path = node.path.copy()  
 new\_path.append(city)  
  
 new\_node = Node(new\_matrix, new\_path, new\_cost)  
 # TC: O(log(b^n)), SC: O(1)  
 queue.put(new\_node)  
  
 total\_states += 1  
  
 end\_time = time.time()  
  
 return {  
 "cost": bssf.cost,  
 "time": end\_time - start\_time,  
 "count": count,  
 "soln": bssf,  
 "max": max\_queue\_size,  
 "total": total\_states,  
 "pruned": pruned\_states,  
 }  
  
 def fancy(self, time\_allowance=60.0):  
 """  
 This is the entry point for the branch-and-bound algorithm that you will implement  
 :param time\_allowance: float  
 :return:  
 """  
 pass

Explain both the **time** and **space** complexity of your algorithm by showing and summing up the complexity of each subsection of your code. Keep in mind the following things:

* Priority Queue: This was implemented using a built-in python heap-based priority queue.
  + Resorting the queue takes log(n) time, so all operations performed take O(logn) time and O(1) space.
* SearchStates: This was implemented using my node class. The most expensive part of this is in
  + creating new nodes, which require O(n^2) matrix creation and updating.
* Reduced Cost Matrix, and updating it: This was implemented using numpy matrices. (All these numbers are according to the
  + Access is O(1) time and space complexity,
  + copy and creation is O(n^2) time and space complexity, and
  + updating takes O(n) time and O(1) space.
* BSSF Initialization: BSSF uses my greedy algorithm to generate a best first guess.
  + My greedy algorithm takes O(n^2) time and O(n) space complexity.
* Expanding one SearchState into others
  + This for loop iterates over each of the cities but only expands children not in the path, so the farther up the tree the closer to the O(n) bound the expansion will take.
  + Creating a new child takes O(n^2) time.
  + Creating O(n) children with each child being O(n^2) creation time puts expansion at O(n^3)
* The full Branch and Bound algorithm
  + The outer for loop will execute O(b^n) times, thanks to branch and bound.
  + The expansion takes O(n^3)
  + Since the algorithm will expand nodes O(n^3) each iteration, the overall time and space complexity is O(n^3\*b^n)

Describe the data structures you use to represent the states.

* The data structures used to represent these states were custom classes with three attributes:
  + The reduced cost matrix, a numpy nxn array
  + The optimistic cost, a python number
  + The path, a python list of cities
* The length of the path was what decided priority on the priority queue. If a path was longer, then it would be the next candidate.

Describe the priority queue data structure you use and how it works.

* The priority queue used a heap queue type data structure. It sorted using the path length of each of the nodes. If one had a longer path than another, then it was a higher priority.

Describe your approach for the initial BSSF.

* My approach for my initial BSSF was to use my greedy algorithm which is efficient at O(n^2) time complexity, to find a solution. The greedy algorithm tries to greedily create a path starting from each of the nodes. Whichever has the best path after trying all start points is the BSSF.

Include a table containing the following columns.

* This table includes the first two requirements and then city counts 10-50 with seed 1

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| # 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 | 1.669 | 9525\* | 74 | 9 | 28543 | 24250 |
| 16 | 902 | 3.563 | 8447\* | 75 | 9 | 56709 | 50067 |
| 10 | 1 | 0.069071054 | 8781\* | 28 | 6 | 929 | 734 |
| 11 | 1 | 0.692487955 | 9348\* | 36 | 1180 | 13652 | 7842 |
| 12 | 1 | 0.07187891 | 10085\* | 43 | 7 | 1235 | 1010 |
| 13 | 1 | 0.377291918 | 9937\* | 44 | 5 | 5974 | 5075 |
| 14 | 1 | 1.323231936 | 9635\* | 58 | 9 | 20006 | 17018 |
| 15 | 1 | 0.590727806 | 9211\* | 67 | 3 | 8292 | 7247 |
| 16 | 1 | 5.636322021 | 10431\* | 86 | 29 | 78159 | 67530 |
| 17 | 1 | 12.87562704 | 10938\* | 96 | 26 | 166008 | 146020 |
| 18 | 1 | 60.00021911 | 11023 | 104 | 22 | 734262 | 651632 |
| 19 | 1 | 60.00042009 | 12209 | 120 | 17 | 696299 | 623500 |
| 20 | 1 | 60.00007701 | 11179 | 135 | 26 | 662412 | 595802 |
| 21 | 1 | 60.00073409 | 14092 | 152 | 10 | 634944 | 574017 |
| 22 | 1 | 60.00133991 | 13228 | 157 | 15 | 598777 | 546878 |
| 23 | 1 | 60.00045896 | 15613 | 175 | 4 | 577657 | 529499 |
| 24 | 1 | 60.00039816 | 14993 | 201 | 5 | 546846 | 504556 |
| 25 | 1 | 60.00086403 | 14040 | 195 | 0 | 515750 | 481876 |
| 26 | 1 | 60.00163722 | 14172 | 210 | 0 | 497701 | 465322 |
| 27 | 1 | 60.00108719 | 16338 | 250 | 6 | 502856 | 461684 |
| 28 | 1 | 60.00082517 | 16496 | 261 | 0 | 480712 | 446526 |
| 29 | 1 | 60.00098777 | 17698 | 278 | 0 | 458909 | 430168 |
| 30 | 1 | 60.00093007 | 16337 | 286 | 0 | 448065 | 419507 |
| 31 | 1 | 60.00029516 | 18283 | 331 | 0 | 434920 | 407474 |
| 32 | 1 | 60.00004673 | 17531 | 358 | 0 | 415088 | 392168 |
| 33 | 1 | 60.00159693 | 19299 | 361 | 0 | 410692 | 386337 |
| 34 | 1 | 60.00121117 | 22309 | 394 | 1 | 402320 | 377388 |
| 35 | 1 | 60.00176287 | 21995 | 411 | 0 | 382273 | 362173 |
| 36 | 1 | 60.00070977 | 22099 | 410 | 0 | 376217 | 356066 |
| 37 | 1 | 60.00174594 | 19290 | 432 | 0 | 357981 | 341236 |
| 38 | 1 | 60.00179887 | 22148 | 450 | 0 | 349604 | 333489 |
| 39 | 1 | 60.00211525 | 18986 | 414 | 0 | 326606 | 314165 |
| 40 | 1 | 60.00169706 | 21045 | 444 | 0 | 320638 | 308448 |
| 41 | 1 | 60.00383997 | 19981 | 419 | 0 | 307627 | 296838 |
| 42 | 1 | 60.00412703 | 22225 | 479 | 0 | 305294 | 294284 |
| 43 | 1 | 60.00148988 | 22151 | 522 | 0 | 299647 | 288727 |
| 44 | 1 | 60.00383687 | 21476 | 515 | 0 | 283189 | 273448 |
| 45 | 1 | 60 | 23842 | 629 | 0 | 284815 | 274086 |
| 46 | 1 | 60.00408196 | 24344 | 619 | 0 | 280499 | 270595 |
| 47 | 1 | 60.00294995 | 24739 | 648 | 0 | 278101 | 267664 |
| 48 | 1 | 60.0024581 | 23349 | 700 | 0 | 269090 | 259824 |
| 49 | 1 | 60.00500894 | 24387 | 653 | 0 | 260302 | 252098 |
| 50 | 1 | 60.00620484 | 25505 | 661 | 0 | 259115 | 250554 |

Discuss the results in the table and why you think the numbers are what they are, including how time complexity and pruned states vary with problem size.

* The table depicts a significant amount of variance in time from n=10 to n=17. There does not seem to be a noticeable trend in time increase, this is due to how helpful the starting BSSF derived from the greedy algorithm and how lucky we got in the tree.
  + However, from n=18 to n=50, we see that it is unable to complete the tree in under 60 seconds.
* The max number of states seems to steadily increase as n increases.
* The # of BSSF Updates is consistently greater than 0 until n=24, from which point onwards the # of BSSF Updates is almost entirely 0. This means that the solution submitted is merely the greedy solution found in the beginning.
* The total number of states created and pruned follow a similar curve, lower in the beginning but increasing until n=19, at which time it steadily starts decreasing. This is likely because the greedy algorithm BSSF is taking longer (O(n^2)), giving the branch and bound algorithm less time to run, thus resulting in less nodes created or pruned.

Discuss the mechanisms you tried and how effective they were in getting the state space search to dig deeper and find more solutions early.

* There were two speedups I found helpful
  + First, I used a dictionary in order to be able to be able to index the cities to their index very quickly.
  + Second, my priority queue set path length as priority, or in other words, greater depth was a greater priority. This would cause the algorithm to prioritize getting to leaf nodes, allowing us to update the BSSF more often without filling out the whole tree.