Atkinson, Jenna – Traveling Salesperson

GitHub Link: https://github.com/jennanatkinson/cs312-proj5-traveling-salesperson

1. **Include your well-commented code.**

TSPSolver.py

```python
#!/usr/bin/python3

from queue import PriorityQueue
from TSPBranchAndBound import PriorityEntry, State
from which_pyqt import PYQT_VER
if PYQT_VER == 'PYQT5':
  from PyQt5.QtCore import QLineF, QPointF
elif PYQT_VER == 'PYQT4':
  from PyQt4.QtCore import QLineF, QPointF
else:
  raise Exception('Unsupported Version of PyQt: {}'.format(PYQT_VER))

import time
import numpy as np
from TSPClasses import *

class TSPSolver:
  def __init__( self, gui_view=None ):
    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>
  '''

  # Randomly generate route and check validity while under the time range
  #   Returns the first found solution
  def defaultRandomTour(self, time_allowance=60.0):
```

```
  results = {}
  cities = self._scenario.getCities()
  foundTour = False
  count = 0
  solution = None
  start_time = time.time()
  while not foundTour and time.time() - start_time < time_allowance:
    # create a random permutation
    perm = np.random.permutation(len(cities))
    route = []
    # Now build the route using the random permutation
    for i in range(len(cities)):
      route.append(cities[perm[i]])
    solution = TSPSolution(route)
    count += 1
    if solution.cost < np.inf:
      # Found a valid route
      foundTour = True
  end_time = time.time()
  results['cost'] = solution.cost if foundTour else math.inf
  results['time'] = end_time - start_time
  results['count'] = count
  results['solution'] = solution
  results['max'] = None
  results['total'] = None
  results['pruned'] = None
  return results

''' <summary>
  This is the entry point for the greedy solver, which you must implement for
  the group project (but it is probably a good idea to just do it for the branch-and
  bound project as a way to get your feet wet).  Note this could be used to find your
  initial BSSF.
  </summary>
  <returns>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</returns>
'''

# Returns the first greedy solution found, Time: O(x*n**2)
def greedy(self, time_allowance=60.0, startCity=None):
  # Setup objects
  results = {}
  cities = self._scenario.getCities()
  foundTour = False
  count = 0
  solution = None
  start_time = time.time()
  if startCity == None:
    startCity = cities[0]

  # Time: O(x*n**2)
  while not foundTour and time.time() - start_time < time_allowance:
    unvisitedCitiesSet = set(cities)
    route = []
    currentCity = startCity

    # Build the route greedily, Time: O(n**2)
    for _ in range(len(cities)):
      greedyCost, nextCity = None, None
      # Iterate to find the smallest unvisited edge, Time: O(n)
         for unvisitedCity in unvisitedCitiesSet:
           cost = currentCity.costTo(unvisitedCity)
           # Save the smallest city (or any city, if none have been visited)
           if greedyCost == None or cost < greedyCost:
             greedyCost, nextCity = cost, unvisitedCity

         # Visit the smallest edge, Time: O(1)
         if nextCity != None:
           unvisitedCitiesSet.remove(nextCity)
           route.append(nextCity)
           currentCity = nextCity
         else:
           raise Exception("Unable to visit any city!!")

       solution = TSPSolution(route)
       count += 1
       if solution.cost < np.inf:
         # Found a valid route
         foundTour = True
       else:
         # Choose a new random city as the start city and try again
         startCity = random.choice(cities)

     # Return results
     end_time = time.time()
     results['cost'] = solution.cost if foundTour else math.inf
     results['time'] = end_time - start_time
     results['count'] = count
     results['solution'] = solution
     results['max'], results['total'], results['pruned'] = None, None, None
```

```python
    return results



  ''' <summary>
    This is the entry point for the branch-and-bound algorithm that you will implement
    </summary>
    <returns>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.</returns>
  '''

  # Continues searching for a better solution until the time runs out or the queue is empty, Time: O(qlen*n**3)
  def branchAndBound(self, time_allowance=60.0, givenBssf=None):
    print("**Branch and Bound**")
    # Setup objects
    results:object = {}
    cities:list[City] = self._scenario.getCities()
    bssf:TSPSolution = givenBssf
    start_time:float = time.time()
    rootState:State = State(cities=cities)
    count:int = 0

    # Start with a greedy solution as the bssf, Time: O(x*n**2)
    if bssf == None:
      greedyResult = self.greedy(time_allowance=time_allowance-(time.time() - start_time))
      bssf:TSPSolution = greedyResult['solution']
      print(f"({'{: >5}'.format(round(time.time() - start_time, 2))}s)  BSSF:{bssf}")

    # Priority queue of (priorityNum, State) (Note: anything on the queue is NOT a solution yet)
    pQueue = PriorityQueue()
    pQueue.put(PriorityEntry(0, 0, rootState))
    maxQueueLen:int = 1
    totalStatesCreated:int = 1
    totalStatesPruned:int = 0

    # Continue searching and expanding states on the queue until time is up or nothing is left, Time: O(qlen*n**3)
    while pQueue.qsize() != 0 and time.time() - start_time < time_allowance:
      state:State = pQueue.get().data
      # If the bssf has changed between adding to the queue vs coming off, prune it
      if state.shouldPrune(bssf.cost):
        totalStatesPruned += 1
        del state
        continue
      # print(state.str_routeSoFar())

      # Expand and evaluate "children" aka a next possible unvisitedCity, Time: O(n**3)
      for nextCity in state.unvisitedCitiesSet:
        childState = state.copy() # Time: O(n)
        totalStatesCreated += 1
        childState.visitCity(nextCity) # Time: O(n**2)
        # print(f"    Child:{childState.str_routeSoFar()}", end="")

        # See if there is a solution yet
        route, cost = childState.getSolution() # Time: O(1)
        # If this is not a valid solution yet,
        if route == None or cost == None:
          if not childState.shouldPrune(bssf.cost): # Time: O(1)
            # Prioritize state and put back on the queue
            pQueue.put(PriorityEntry(len(childState.cities)-len(childState.unvisitedCitiesSet), childState.costSoFar, childState))
            # print(f": added to queue")
            if pQueue.qsize() > maxQueueLen:
              maxQueueLen = pQueue.qsize()
          # If it should be pruned, it does not go back on the queue
          else:
            # print(f": pruned")
            totalStatesPruned += 1
            del childState

        # If it is a solution, then see if it is better than bssf, Time: O(1)
        else:
          solution = TSPSolution(route)
          print(f"({'{: >5}'.format(round(time.time() - start_time, 2))}s)  BranchAndBound:{solution}")
          count += 1
          if solution.cost < bssf.cost:
            bssf = solution

    # Return results
    end_time = time.time()
    results['cost'] = bssf.cost
    results['time'] = end_time - start_time
    results['count'] = count
    results['solution'] = bssf
```

```python
      results['max'] = maxQueueLen
      results['total'] = totalStatesCreated
      results['pruned'] = totalStatesPruned
      return results


    ''' <summary>
      This is the entry point for the algorithm you'll write for your group project.
      </summary>
      <returns>results dictionary for GUI that contains three ints: cost of best solution,
      time spent to find best solution, total number of solutions found during search, the
      best solution found.  You may use the other three field however you like.
      algorithm</returns>
    '''

    def fancy(self,time_allowance=60.0):
      pass
```

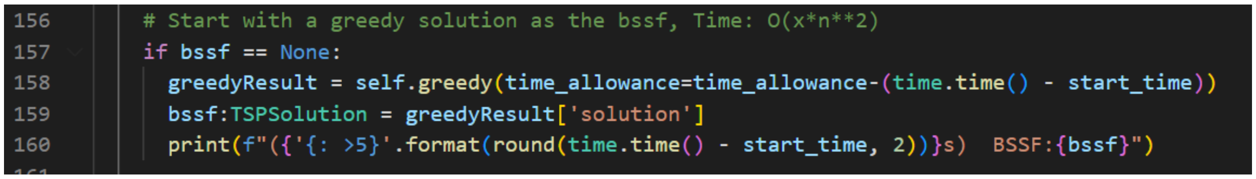
TSPBranchAndBound.py

```python
import math
from TSPClasses import City
from copy import copy, deepcopy

INF_STRING = "-"

# Space: O(1)
class PriorityEntry(object):
  def __init__(self, numVisited:int, costSoFar:int, data):
      self.data = data
      self.numVisited:int = numVisited
      self.costSoFar:int = costSoFar

  def __lt__(self, other): #less than
      if self.numVisited == other.numVisited:
        return self.costSoFar < other.costSoFar
      return self.numVisited > other.numVisited

# Keeps track/adjusts the current route, matrix and cost when adding new cities to the route, Space: O(n)
class State:
  #unvisitedCitiesSet:set, set of cities that haven't been visited yet
  #matrix:dict(tuple(rowIndex, colIndex):reducedCost), dictionary to show which edges can be used next (inf entries do not exist)
  #cities:list[City], reference to the list of all cities
  #routeSoFar:list[City], list of cities in order of what we have visited so far
  #costSoFar:int, the cost we have accumulated on our route
  #isReturnVisitToStart:bool, if the route has finished through all the cities and returned back to the start
  def __init__(self, unvisitedSet:set=set(), matrix:dict=dict(), cities:list[City]=[], route:list=[], costSoFar:int=0):
    self.unvisitedCitiesSet:set = unvisitedSet
    self.matrix:dict = matrix
    self.cities:list[City] = cities # should be readOnly
    self.routeSoFar:list[City] = route
    self.costSoFar:int = costSoFar
    self._isReturnVisitToStart:bool = False # should ONLY be updated if len(unvisitedSet) == 0

    # If given nothing but a list of cities, construct the root state
    if len(cities) != 0 and costSoFar == 0 and len(matrix) == 0 and len(unvisitedSet) == 0 and len(route) == 0:
      self._generateRootStateFromCities() # Time: O(n**2)

  # Given a list of cities, return an init state with the first city as the start, Time: O(n**2)
  def _generateRootStateFromCities(self):
    # Unvisited cities should exclude the start node, so it is not revisited before the end
    self.unvisitedCitiesSet = set(self.cities[1:])

    # Route should include the start node
    self.routeSoFar = [self.cities[0]]

    #Initialize the matrix, Time: O(n**2)
    matrix = dict()
    for city in self.cities:
      for otherCity in self.cities:
        if city != otherCity:
          cost = city.costTo(otherCity)
          if cost != math.inf:
            matrix[tuple((city._index, otherCity._index))] = cost
    self.matrix = matrix
    self._reduceCostOnMatrix() # Time: O(n**2)

  # Returns true if route is impossible or not going to yield a better result, Time: O(1)
  # Should be used if branch is not a solution yet, but trying to determine if we should continue exploring or not
  def shouldPrune(self, bssf:int=math.inf) -> bool:
    return self.costSoFar == math.inf or self.costSoFar >= bssf

  # Sees if this State yields a solution, Time: O(1)
  def isSolution(self) -> bool:
    return len(self.unvisitedCitiesSet) == 0 and self.costSoFar != math.inf and len(self.matrix) == 0 and self._isReturnVisitToStart

  # Return the route if a solution, Time: O(1)
  # (will try to return to start if that is possible and hasn't been done yet)
  def getSolution(self):
    self._tryReturnToStart()
    if self.isSolution():
      return self.routeSoFar, self.costSoFar
    else:
      return None, None

  # If all other nodes have been visited, try to return to the start from the last city (unless already returned), Time: O(1)
  # This will update the matrix and costSoFar if this is impossible
  def _tryReturnToStart(self):
    if self._isReturnVisitToStart: return
    if len(self.unvisitedCitiesSet) == 0 and self.costSoFar != math.inf:
      self.visitCity(self.cities[0]) # Time: O(1) (this will be super fast, bc no reducing)

  # Looks at the matrix and determines if a row or column will be inf (based on visit status), Time: O(1)
  def _isInfinity(self, rowMajor:bool, city:City) -> bool:
    # If the city has been visited (aka at least an edge inbound or outbound, which is the only way to be inf)
    if not (city in self.unvisitedCitiesSet):
      if len(self.routeSoFar) <= 1:
        return False
      # If the city is the startCity
      if city == self.routeSoFar[0] and not self._isReturnVisitToStart:
        # If the route has finished and returned back to the start, then the row and col will be inf
        if self._isReturnVisitToStart:
          return True
        # If startCity hasn't been visited, then the first row will be inf, but not the column
        else:
          return rowMajor
      # If city == most recently visited in route, then only the column will be inf, not the row
      elif city == self.routeSoFar[len(self.routeSoFar)-1]:
        return not rowMajor
        # Otherwise, the city will have an outbound and an inbound edge already logged, so the col and row == inf
        else:
          return True
      else:
        return False

    # Generates the correct cell tuple based on if we are aiming for a row or a column in the loop (see findMinCostAndNormalize), Time: O(1)
    def _generateTuple(self, rowMajor:bool, i:int, j:int):
      if rowMajor:
        return tuple((i, j))
      else: # Rows and columns are reversed here
        return tuple((j, i))

    # Ensures at least one zero on the rows/columns of the matrix (besides inf), Time: O(n**2)
    def _findMinCostAndNormalize(self, rowMajor:bool=True):
      # Look at row each row/column to find the minCost, Time: O(n**2)
      for i in range(len(self.cities)):
```

```python
        # If the whole row/column is going to be infinities, move on, Time: O(1)
        if self._isInfinity(rowMajor, self.cities[i]):
          continue

        minCost = math.inf

        #Examine each cell in a row/column to find the minCost, Time: O(n)
        for j in range(len(self.cities)):
          cell = self._generateTuple(rowMajor, i, j)
          cost = self.matrix.get(cell)
          if cost != None and cost < minCost:
            minCost = cost
            if minCost == 0:
              break

        # If we are taking on additional costs, we need to readjust the costSoFar and row/column numbers
        if minCost != 0 and minCost < math.inf:
          self.costSoFar += minCost

        # Update all costs in same row/column to normalize, Time: O(n)
          for j in range(len(self.cities)):
            cell = self._generateTuple(rowMajor, i, j)
            cost = self.matrix.get(cell)
            # If not infinity, update cell's cost
            if cost != None:
              self.matrix[cell] = cost - minCost

    # Ensures at least one zero on the rows and columns of the matrix (besides inf), adjusts costSoFar as needed, Time: O(n**2)
    def _reduceCostOnMatrix(self):
      self._findMinCostAndNormalize(rowMajor=True)  # Time: O(n**2)
      self._findMinCostAndNormalize(rowMajor=False) # Time: O(n**2)

    # Marks the city as visited, updates the matrix and then does reduceCost to normalize the matrix again, Time: O(n**2)
    def visitCity(self, cityToVisit:City):
      if (self.costSoFar == math.inf or self._isReturnVisitToStart):
        return
      assert(len(self.routeSoFar) != 0)
      prevCity = self.routeSoFar[len(self.routeSoFar)-1]

      # Ensure that there is actually a path to the other city
      cost = self.matrix.get(tuple((prevCity._index, cityToVisit._index)))
      if cost == None:
        self.costSoFar = math.inf
        return

      # Update cost after traveling to city
      self.costSoFar += cost

      # Remove all the impossible routes, Time: O(n)
      for i in range(len(self.cities)):
        # Update matrix to remove all items from row prevCity
        self.matrix.pop(tuple((prevCity._index, i)), None)

        # Update matrix to remove all items from col cityToVisit
        self.matrix.pop(tuple((i, cityToVisit._index)), None)

      # Update matrix to remove the inverse (cityToVisit -> prevCity) as well
      self.matrix.pop(tuple((cityToVisit._index, prevCity._index)), None)

      # Only allowed to visit the startCity after visiting all other cities
      if cityToVisit == self.cities[0]:
        assert(len(self.unvisitedCitiesSet) == 0)
        assert(len(self.matrix) == 0)
        self._isReturnVisitToStart = True
      else:
        self.unvisitedCitiesSet.remove(cityToVisit)
        self.routeSoFar.append(cityToVisit)
        self._reduceCostOnMatrix() # Time: O(n**2)

    # Makes a shallow copy of all the elements (data structures are new, but if object, then reference is copied), Time: O(n)
    def copy(self):
      result = State()
      result.unvisitedCitiesSet:set = set(self.unvisitedCitiesSet)
      result.matrix:dict = dict(self.matrix)
      result.cities:list[City] = self.cities
      result.routeSoFar:list[City] = copy(self.routeSoFar)
      result.costSoFar:int = self.costSoFar
      result._isReturnVisitToStart:bool = self._isReturnVisitToStart
      return result

    def str_routeSoFar(self):
      string = "State{"
      if len(self.routeSoFar) != 0:
        for city in self.routeSoFar:
          string += f"{city._name}->"
      else:
        string += "*empty*"
      string = string[:-2]
      string += "}"
      return string

    def str_costSoFar(self):
      return f"Cost so far: {self.costSoFar}\n"

    def str_unvisitedCitiesSet(self):
      string = "Unvisited Cities: "
      if len(self.unvisitedCitiesSet) != 0:
        for city in self.unvisitedCitiesSet:
          string += f"{city._name} "
      else:
        string += "*empty*"
      string += '\n'
      return string

    def str_matrix(self):
      string = 'Matrix:\n'
      # Format the matrix printing
      table_data = [[]]
      # Print the names of the cities at the top
      for rowIndex in range(-1, len(self.cities)):
        if rowIndex == -1:
          table_data[0].append(" ")
        else:
          #assert(type(self.cities[i]) == City)
          table_data[0].append(self.cities[rowIndex]._name)

      for rowIndex in range(0, len(self.cities)):
        table_data.append([])
        last = len(table_data)-1
        for colIndex in range(-1, len(self.cities)):
          # Append the name of the city for the row
          if colIndex == -1:
            table_data[last].append(self.cities[rowIndex]._name)
          # Print the same city -> same city as inf
          elif colIndex == rowIndex:
            table_data[last].append(INF_STRING)
          # Lookup in dictionary and see if it exists
          else:
            cost = self.matrix.get(tuple((rowIndex, colIndex)))
            if cost != None:
              table_data[last].append(f"{cost}")
            else:
              table_data[last].append(INF_STRING)

      formatString = "{: >5} "
      for row in table_data:
        for item in row:
          string += formatString.format(item)
        string += '\n'
      string += '\n'
      return string

    def __str__(self) -> str:
      string = self.str_routeSoFar() + '\n'
      string += self.str_costSoFar()
      string += self.str_unvisitedCitiesSet()
      string += self.str_matrix()
      return string
```

2. **Explain both the time and space complexity of your algorithm by showing and summing up the complexity of each subsection of your code.**

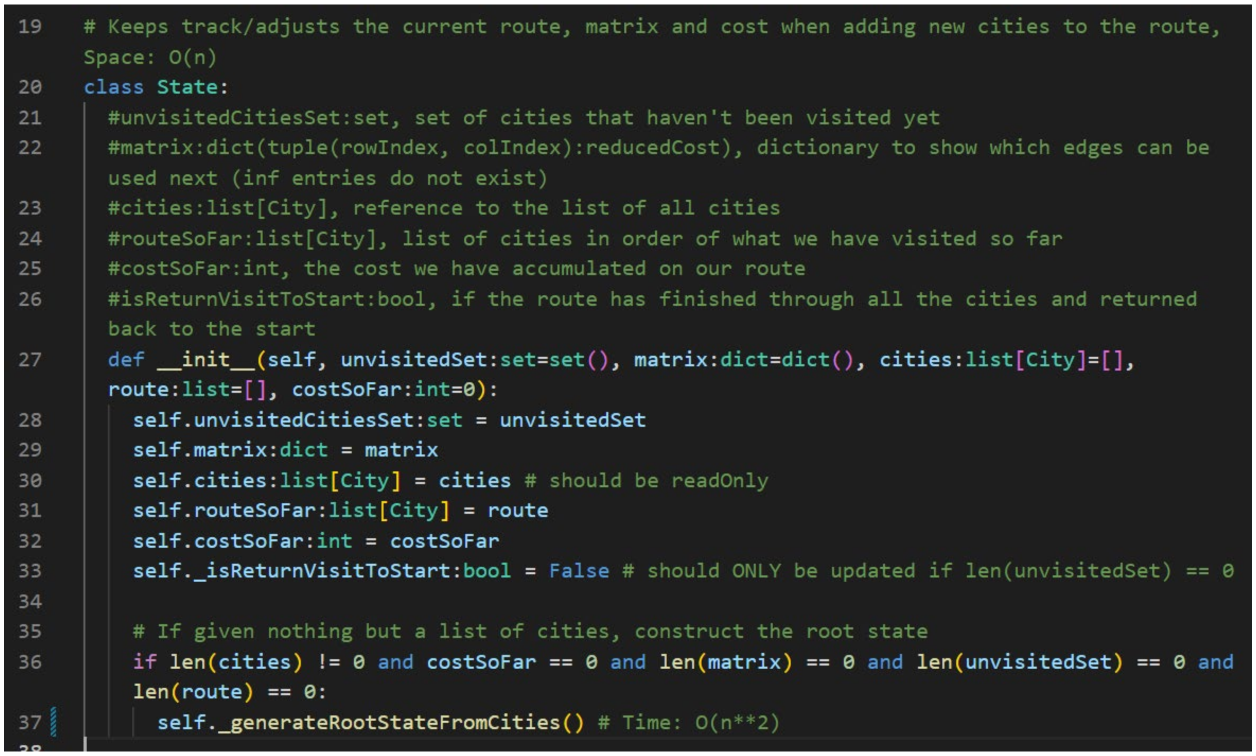
BSSF Initialization



```
# Start with a greedy solution as the bssf, Time: O(x*n**2)
if bssf == None:
  greedyResult = self.greedy(time_allowance=time_allowance-(time.time() - start_time))
  bssf:TSPSolution = greedyResult['solution']
  print(f"({'{: >5}'.format(round(time.time() - start_time, 2))}s)  BSSF:{bssf}")
```

I used the greedy algorithm to initialize my first BSSF, which follows the cheapest route through the cities, evaluating all the edges at each city, for a **Time Complexity of O(x*n**2)**. The x is a variable number of repetitions in case the first greedy attempt doesn't' work, it can randomly choose a new starting city and try again. This number is very low and constant, so it is **functionally O(n**2).**

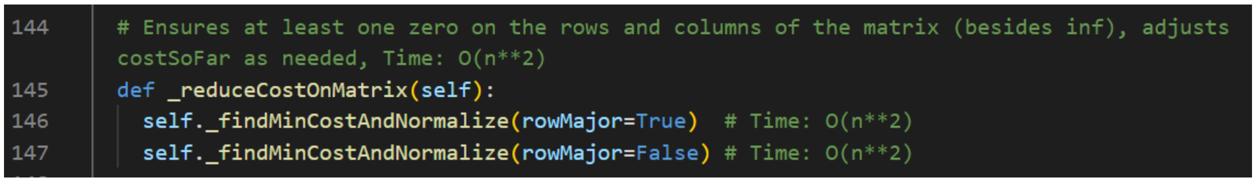
SearchStates

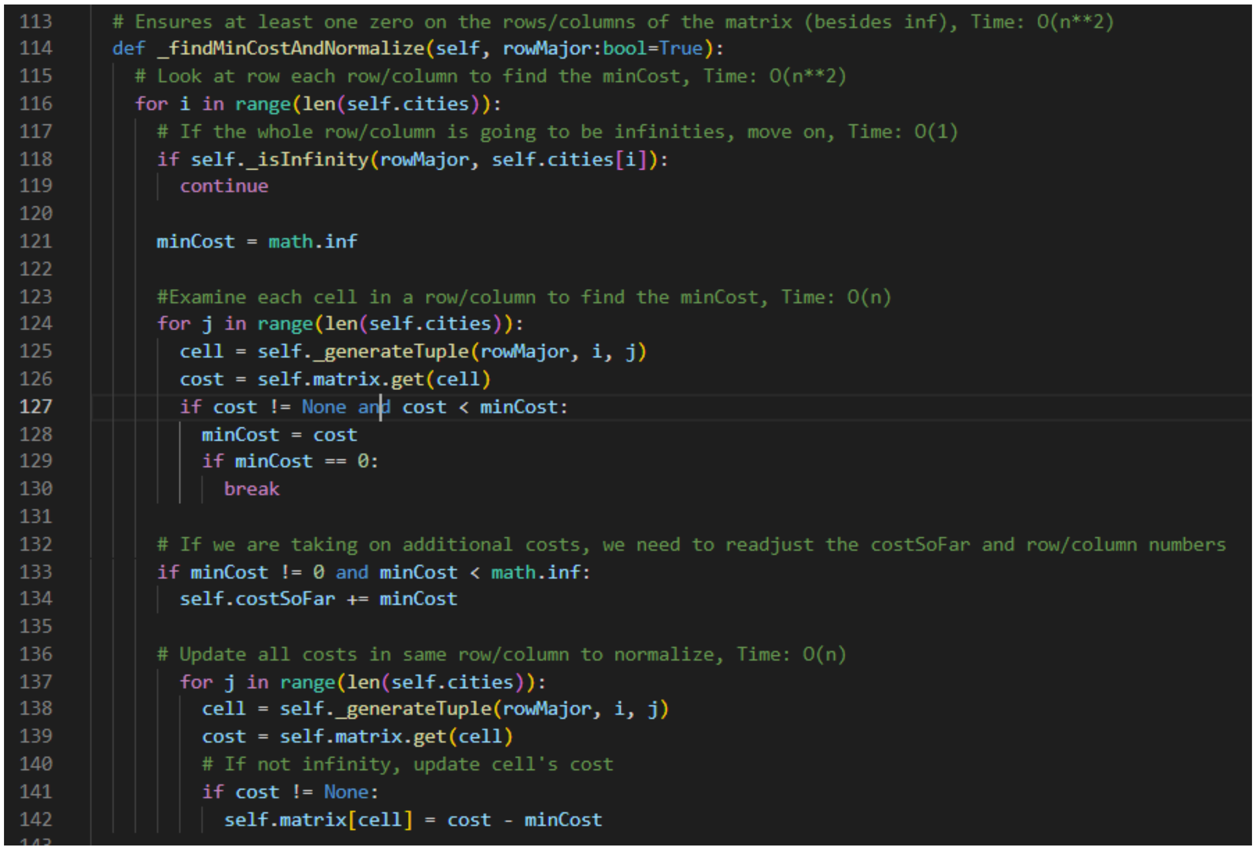


```
# Keeps track/adjusts the current route, matrix and cost when adding new cities to the route,
Space: O(n)
class State:
  #unvisitedCitiesSet:set, set of cities that haven't been visited yet
  #matrix:dict(tuple(rowIndex, colIndex):reducedCost), dictionary to show which edges can be
  used next (inf entries do not exist)
  #cities:list[City], reference to the list of all cities
  #routeSoFar:list[City], list of cities in order of what we have visited so far
  #costSoFar:int, the cost we have accumulated on our route
  #isReturnVisitToStart:bool, if the route has finished through all the cities and returned
  back to the start
  def __init__(self, unvisitedSet:set=set(), matrix:dict=dict(), cities:list[City]=[],
  route:list=[], costSoFar:int=0):
    self.unvisitedCitiesSet:set = unvisitedSet
    self.matrix:dict = matrix
    self.cities:list[City] = cities # should be readOnly
    self.routeSoFar:list[City] = route
    self.costSoFar:int = costSoFar
    self._isReturnVisitToStart:bool = False # should ONLY be updated if len(unvisitedSet) == 0

    # If given nothing but a list of cities, construct the root state
    if len(cities) != 0 and costSoFar == 0 and len(matrix) == 0 and len(unvisitedSet) == 0 and
    len(route) == 0:
      self._generateRootStateFromCities() # Time: O(n**2)
```

My State class holds many members, but the most space expensive is the set, matrix and two lists of the cities. The matrix only holds ints and each of the sets and lists hold only references to the Cities and not the objects themselves. However, these data structures grow in size with n, for a **Space Complexity of O(n)**. Initializing the first state involves setting each of these members and looping through the combinations of cities to generate the matrix, for a **Time Complexity of O(n**2).**

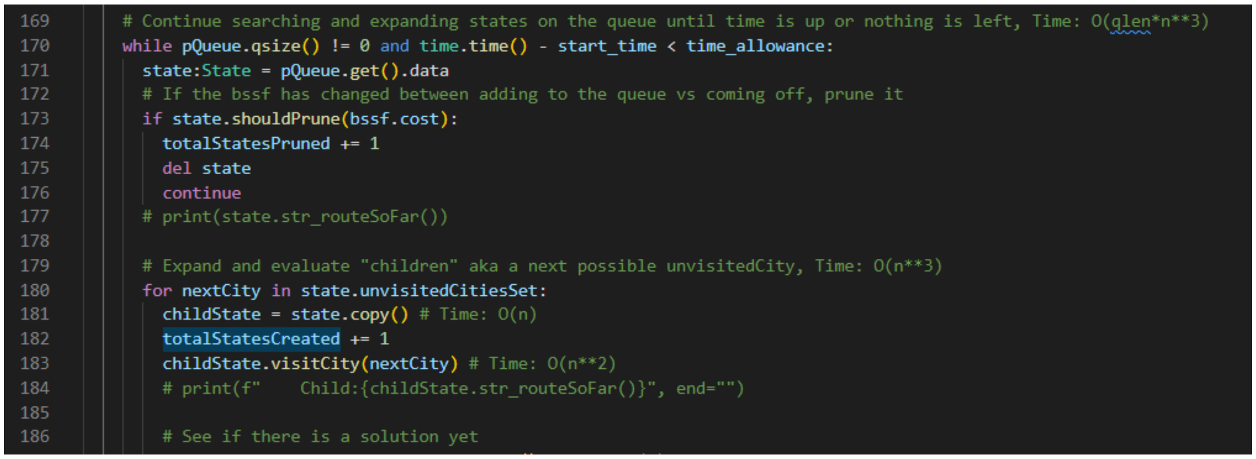
Reduced Cost Matrix, and updating it


```python
# Ensures at least one zero on the rows and columns of the matrix (besides inf), adjusts costSoFar as needed, Time: O(n**2)
def _reduceCostOnMatrix(self):
  self._findMinCostAndNormalize(rowMajor=True)  # Time: O(n**2)
  self._findMinCostAndNormalize(rowMajor=False) # Time: O(n**2)
```


```python
# Ensures at least one zero on the rows/columns of the matrix (besides inf), Time: O(n**2)
def _findMinCostAndNormalize(self, rowMajor:bool=True):
  # Look at row each row/column to find the minCost, Time: O(n**2)
  for i in range(len(self.cities)):
    # If the whole row/column is going to be infinities, move on, Time: O(1)
    if self._isInfinity(rowMajor, self.cities[i]):
      continue

    minCost = math.inf

    #Examine each cell in a row/column to find the minCost, Time: O(n)
    for j in range(len(self.cities)):
      cell = self._generateTuple(rowMajor, i, j)
      cost = self.matrix.get(cell)
      if cost != None and cost < minCost:
        minCost = cost
        if minCost == 0:
          break

    # If we are taking on additional costs, we need to readjust the costSoFar and row/column numbers
    if minCost != 0 and minCost < math.inf:
      self.costSoFar += minCost

    # Update all costs in same row/column to normalize, Time: O(n)
      for j in range(len(self.cities)):
        cell = self._generateTuple(rowMajor, i, j)
        cost = self.matrix.get(cell)
        # If not infinity, update cell's cost
        if cost != None:
          self.matrix[cell] = cost - minCost
```

Reducing the matrix involves looking at each row to reduce and each column to reduce (see in findMinCostAndNormalize). To do that, you must look at each element within that row and if there is not a 0, adjust that whole row for a Time Complexity of O(n). Because this must be done for each row, the Time Complexity for findMinCostAndNormalize is O(n**2). And because we must normalize the rows and columns, this process is done twice, for a **total Time Complexity of O(n**2)**.

Expanding one SearchState into others

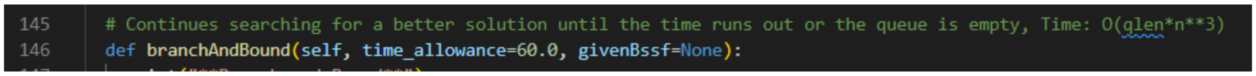

```
# Continue searching and expanding states on the queue until time is up or nothing is left, Time: O(qlen*n**3)
while pQueue.qsize() != 0 and time.time() - start_time < time_allowance:
  state:State = pQueue.get().data
  # If the bssf has changed between adding to the queue vs coming off, prune it
  if state.shouldPrune(bssf.cost):
    totalStatesPruned += 1
    del state
    continue
  # print(state.str_routeSoFar())

  # Expand and evaluate "children" aka a next possible unvisitedCity, Time: O(n**3)
  for nextCity in state.unvisitedCitiesSet:
    childState = state.copy() # Time: O(n)
    totalStatesCreated += 1
    childState.visitCity(nextCity) # Time: O(n**2)
    # print(f"    Child:{childState.str_routeSoFar()}", end="")

    # See if there is a solution yet
```

The first item is popped of the queue and then eventually expanded into a new childState. The original state is copied (Time complexity of O(n) because it is a shallow copy), and then a new unvisited city is visited (Time complexity of O(n**2) because the matrix needs to be reduced afterwards). This gives a **total Time Complexity of O(n**3)** for expanding and evaluating the child state.

Priority Queue

I used the default Python Priority Queue which has a **Time Complexity of O(logn)** because Python uses a Heap. The Queue has a **Space Complexity of O(qlen*n)** because it holds a qlen number of States.

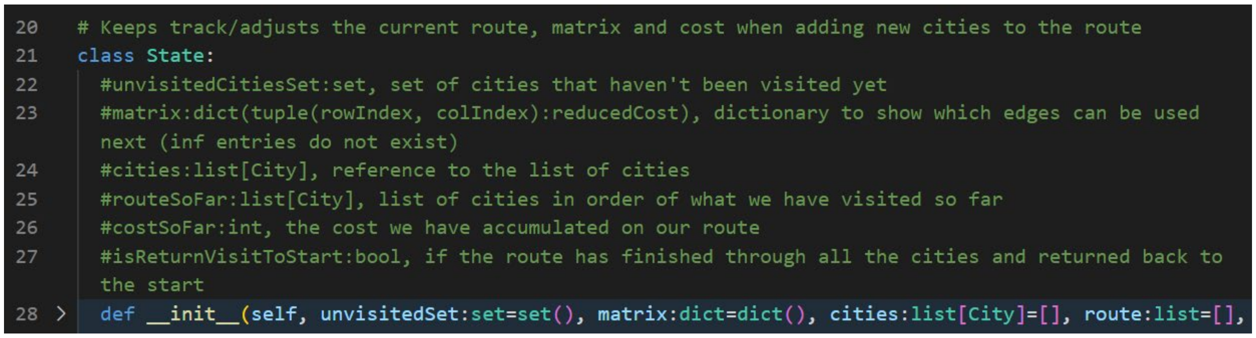
The full Branch and Bound algorithm


```
# Continues searching for a better solution until the time runs out or the queue is empty, Time: O(qlen*n**3)
def branchAndBound(self, time_allowance=60.0, givenBssf=None):
```

The full Branch and Bound algorithm is dominated by the complexity of expanding new states, because we are using a queue instead of looping. If that time complexity is O(n**3), then the total **Time Complexity for the algorithm is O(qlen*n**3)** where qlen is the maximum number of States on the queue at given time.

3. Describe the data structures you use to represent the states.

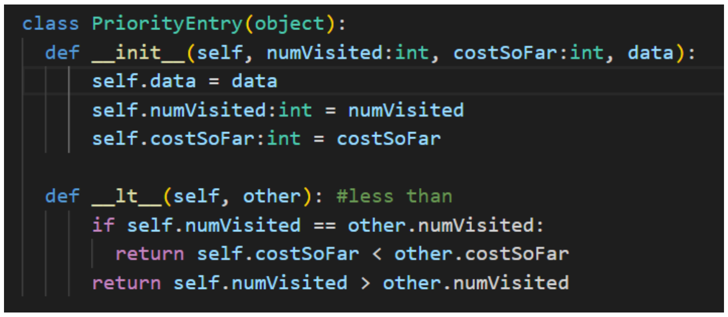
I created a State class which handles keeping track of the States route, cost and matrix so far. It also handles visiting a new city (which updates the route, and updates/reduces the matrix etc.) and checking if the current State is a solution. Below are some comments explaining the purpose of each of my data members:



```python
# Keeps track/adjusts the current route, matrix and cost when adding new cities to the route
class State:
    #unvisitedCitiesSet:set, set of cities that haven't been visited yet
    #matrix:dict(tuple(rowIndex, colIndex):reducedCost), dictionary to show which edges can be used
    next (inf entries do not exist)
    #cities:list[City], reference to the list of cities
    #routeSoFar:list[City], list of cities in order of what we have visited so far
    #costSoFar:int, the cost we have accumulated on our route
    #isReturnVisitToStart:bool, if the route has finished through all the cities and returned back to
    the start
>   def __init__(self, unvisitedSet:set=set(), matrix:dict=dict(), cities:list[City]=[], route:list=[],
```

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

I used the built-in python Priority Queue for my queue and created my own PriorityEntry object for each entry in the queue to prioritize how I wanted it. The queue prioritizes based on the larger numVisited, and then if that number is the same, then it prioritizes the next smallest costSoFar.



```python
class PriorityEntry(object):
    def __init__(self, numVisited:int, costSoFar:int, data):
        self.data = data
        self.numVisited:int = numVisited
        self.costSoFar:int = costSoFar

    def __lt__(self, other): #less than
        if self.numVisited == other.numVisited:
            return self.costSoFar < other.costSoFar
        return self.numVisited > other.numVisited
```

5. Describe your approach for the initial BSSF.

I implemented a greedy algorithm which starts with the first city and then takes the next smallest route avaliable. If a path is not found with that method, then a new random city is chosen for the start city and the greedy algorithm tries again. The function returns the first solution found and that is used for the initial BSSF. I chose this approach because it would be more optimal starting bssf than a random path because there are some decisions about choosing a smaller path.

6. Include a table containing the following columns.

Running Branch and Bound on Difficulty Hard, with time limit = 60

**=optimal (if the algorithm doesn't time out)*

# Cities	Seed	Running Time	Cost of best tour found	# of BSSF updates	Max Queued States at a Given Time	Total States Created	Total States Pruned
15	20	0.631937 s	10104*	12	69	7732	6543
16	902	0.273056 s	7937*	8	67	3364	2904
18	371	14.822297 s	10513*	5	100	157636	139917
20	912	20.238040 s	10665*	10	135	187715	166051
25	173	60.000998 s	11847	9	210	308236	282947
30	953	60.000475 s	13212	7	324	366576	330054
35	261	60.000179 s	16455	4	440	453008	355187
40	811	60.001034 s	18402	11	568	342150	302377
45	628	60.000018 s	21017	7	749	302111	258454
50	813	60.000902 s	21857	15	920	314788	225295

7. 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.

Looking at this table, we can see that as the number of cities grows, the number of total States created grows pretty dramatically and then starts to taper off at the end. By increasing the number of cities, we exponentially increase the amount of possible States we could create. However, in the last few rows of data, because we have a limited running time, most of the computational time is probably spent evaluating the states already on the queue and pruning them, rather than creating new ones. Expanding states and reducing the matrix becomes more expensive in time complexity as we grow.

The max queued at a time also increases as we increase the number of cities, because each child expansion (if not pruned) is then added back to the queue. As we saw above, the queue size impacts time complexity if we allowed the algorithm to continue looking at all of the states.

The pruned states stay a pretty consistent ratio to the created states, which means that the bssf is pruning well and properly, and the priority method for the queue is successful.

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

I tried to first create a ratio between the numVisited / costSoFar to determine the priority of the queue, so that hopefully the bigger cost would result in less priority. However, that method resulted in too similar priorities between states and not as many solutions were found. I changed the prioritization to first prioritize the larger numVisited, and then if that number was the same, then prioritize the next smallest costSoFar. That resulted in much better prioritizations. Because of this, I get the bulk of my solutions in the first 10 seconds (even when the number of cities increases), and then it takes the remaining time to spread out and look for more