Traveling Salesman Problem

November 30, 2023

Gage Moore

Introduction

The Traveling Salesman Problem (TSP) is a classic optimization problem where the goal is to find the shortest possible tour that visits a given set of cities and returns to the starting city, without visiting any city more than once. When measuring time complexity and space complexity for this problem, n will be used to represent the number of cities.

Time Complexity

For the purposes of this lab, I implemented a function to find a solution with a greedy algorithm and a function to find a solution with a branch & bound algorithm. The branch & bound implementation uses a priority queue, which I copied over from my networking lab with slight modifications.

The Greedy Method

The greedy function is implemented within the following function:

```
def greedy(self, time_allowance=60.0):
  results = {}
  cities = self._scenario.getCities()
  found_tour = False
  start_time = time.time()
  route = []

  for start_city in cities: # O(n)
    route = [start_city]

    while not found_tour and time.time() - start_time < time_allowance: # O(n)
      cheapest_neighbor = route[-1]

      for neighbor in cities:
        if neighbor not in route and
           route[-1].costTo(neighbor) < route[-1].costTo(cheapest_neighbor):
          cheapest_neighbor = neighbor

      if route[-1].costTo(cheapest_neighbor) == math.inf: # No more neighbors
        break
      route.append(cheapest_neighbor)

      if len(route) == len(cities) and
         route[-1].costTo(route[0]) < math.inf: # Found a tour
        found_tour = True

    if found_tour:
      break

  solution = TSPSolution(route)
  end_time = time.time()

  results["cost"] = solution.cost if found_tour else math.inf
  results["time"] = end_time - start_time
```

```
  results["count"] = 1 if found_tour else 0
  results["soln"] = solution
  results["max"] = None
  results["total"] = None
  results["pruned"] = None

  return results
```

The function initializes the route list that it will use to store the travel path and enters a loop that iterates through every city, ($O(n)$). For each iteration, the city is set as the starting point, and a seconday loop is run which at its worst case will iterate through every neighbor of that city, ($O(n-1)$). It checks the cost to travel to every unvisited neighbor in order to find the cheapest option, and since this is a greedy algorithm, the cheapest neighbor is chosen as the next destiation. This process is repeated for every neighbor unless a full tour is found.

The algorithm breaks and returns a solution as soon as a tour is found, but if that does not happen until every neighbor of every city has been checked, the function can potentially have a time complexity of $O(n^2)$. A route is constructed for every city that is set as the starting point, but the route is overwritten with each iteration of the initial `for` loop. As a result, the space complexity is bounded by $O(n)$.

The Priority Queue

I used the heap that I created from the networking lab, but instead of setting it up to store nodes in a graph, I created a new `City` class to easily store distance, lower bound, and most importantly the state with each element of the heap:

```
class City:
  def __init__(self, node_id, distance, lower_bound, path, distance_matrix):
    self.node_id = node_id
    self.distance = distance
    self.lower_bound = lower_bound
    self.path = path
    self.distance_matrix = distance_matrix
```

The heap class consists of a method to get the current size, which just returns the length of its internal node list. This gives it a time complexity of $O(1)$ and a space complexity of $O(1)$:

```
def size(self):
        return len(self.nodes)
```

When adding and removing items from the queue, it needed a way to propogate the element to its correct order, (in this case cities are ordered in the heap by their distance). To accomplish this, `bubble_up()` and `bubble_down()` are implemented:

```
def swap(self, i, j):
    temp = self.nodes[i]
    self.nodes[i] = self.nodes[j]
    self.nodes[j] = temp
    self.lookup[self.nodes[i].node_id] = i
    self.lookup[self.nodes[j].node_id] = j

def bubbleUp(self, index):
    parent = (index-1) // 2

    if self.nodes[parent].distance > self.nodes[index].distance:
        self.swap(index, parent) # O(1)
```

```
        self.bubbleUp(parent) # O(log n)

def bubbleDown(self, index):
    left_child = (index * 2) + 1
    right_child = (index * 2) + 2

    # determine which side to sift down
    if left_child > len(self.nodes)-1 or right_child > len(self.nodes)-1:
        min_child =  -1
    elif (self.nodes[left_child].distance < self.nodes[right_child].distance):
        min_child = left_child
    else:
        min_child = right_child

    # exit condition for recursion (node is leaf)
    if min_child < 1 or min_child > len(self.nodes)-1:
        return

    if self.nodes[index].distance > self.nodes[min_child].distance:
        self.swap(index, min_child) # O(1)
        self.bubbleDown(min_child) # O(log n)
```

The `swap()` function runs in constant time and has a space complexity of $O(1)$ as it simply swaps the place of two elements in the heap.

Both the bubble functions have a time complexity of $O(\log n)$ because they recursively iterate through each level of the heap, splitting the number of cities to compare in half with each recursion. The functions do not dynamically allocate any new space, so the space complexity of each is $O(1)$. With these functions in place, inserting and deleting nodes is performed as follows:

```
def insert(self, index, distance, lower_bound, path, distance_matrix):
    self.nodes.append(City(index, distance, lower_bound, path, distance_matrix))
    self.lookup.append(len(self.nodes)-1)
    self.bubbleUp(len(self.nodes)-1) # O(log n)

def deleteMin(self):
    min = self.nodes[0]
    self.nodes[0] = self.nodes[len(self.nodes)-1]
    self.nodes.pop()
    self.bubbleDown(0) # O(log n)
    return min
```

Both of these functions have a time complexity of $O(\log n)$ because they will trigger a `bubble_up` or a `bubble_down()` depending on if an element was added or removed. Both also have a space complexity of $O(1)$ since new space is not dynamically allocated.

The Branch & Bound Method

The branch & bound entry function relies on two helper functions to make the code more readable, one for creating the distance matrix and one for calculating the lower bound.

Representing the states

To hold each iterative distance matrix used by the branch & bound algorithm, a 2-dimensional NumPy array was used. This allowed me to use builtin NumPy array functions such as `ones()`, `copy()`, and `full()`, which made it very easy to make the necessary matrix manipulations required by the branch & bound algorithm.

Distance Matrix Function

The following function creates the distance matrix:

```
def initializeMatrix(self):
  cities = self._scenario.getCities()
  matrix = np.ones((len(cities), len(cities))) * np.inf # O(n^2)

  lower_bound = 0

  for city in range(len(cities)): # O(n^2)
    for next_city in range(len(cities)):
      matrix[city][next_city] = cities[city].costTo(cities[next_city])

  for row in range(len(cities)): # O(n^2)
    minimum = np.inf
    for col in range(len(cities)):
      if matrix[row][col] < minimum: # Found a new minimum
        minimum = matrix[row][col]
    for col in range(len(cities)):
      matrix[row][col] = matrix[row][col] - minimum
    lower_bound += minimum

  for col in range(len(cities)): # O(n^2)
    minimum = np.inf
    for row in range(len(cities)):
      if matrix[row][col] < minimum: # Found a new minimum
        minimum = matrix[row][col]
    for row in range(len(cities)):
      matrix[row][col] = matrix[row][col] - minimum
    lower_bound += minimum

  return matrix, lower_bound
```

The function first creates a matrix filled with infinities where the length and width are n, the number of cities. This operation takes $O(n^2)$ time to complete. A series of doubly-nested `for` loops are then run.

The first loop sets every value in the matrix to the cost of traveling between every city pair, running in $O(n^2)$ time in order to compare n cities across n cities. The next loop subtracts the minimum value of every row out of each row, and the following loop does the same for each column. This is known as a reduced cost matrix, and both loops used for the reduction run in $O(n^2)$ time to iterate through the entire matrix.

The function ultimately creates an entire $n \times n$ matrix for the branch & bound algorithm to work off of, so its space complexity if $O(n^2)$. Since every component that it runs sequentially has a time complexity of $O(n^2)$, the overall time complexity is also $O(n^2)$.

Lower Bound Function

The following function calculates the new lower bound of the given distance matrix:

```
def findLowerBound(self, old_lower_bound, old_distance_matrix, path):
  distance_matrix = np.copy(old_distance_matrix) # O(n^2)
  city_num = distance_matrix.shape[0]
  distance = distance_matrix[path[-2]][path[-1]]

  lower_bound = old_lower_bound + distance

  for i in range(city_num): # O(n)
    distance_matrix[path[-2]][i] = np.inf
    distance_matrix[i][path[-1]] = np.inf

  distance_matrix[path[-1]][path[-2]] = np.inf  # Don't visit the same city twice

  for row in range(city_num): # O(n^2)
    if row == path[-2]: # Skip the row we just added
      continue
    minimum = np.inf
    for col in range(city_num):
      if col == path[-1]: # Skip the column we just added
        continue
      if distance_matrix[row][col] < minimum: # Found a new minimum
        minimum = distance_matrix[row][col]
    if minimum != np.inf: # Found a value to subtract
      for col in range(city_num):
        if distance_matrix[row][col] > 0:
          distance_matrix[row][col] = distance_matrix[row][col] - minimum
        else:
          distance_matrix[row][col] = 0
      lower_bound += minimum

  for col in range(city_num): # O(n^2)
    if col == path[-1]: # Skip the column we just added
      continue
    minimum = np.inf
    for row in range(city_num):
      if row == path[-2]: # Skip the row we just added
        continue
      if distance_matrix[row][col] < minimum: # Found a new minimum
        minimum = distance_matrix[row][col]
    if minimum != np.inf: # Found a value to subtract
      for row in range(city_num):
        if distance_matrix[row][col] > 0:
          distance_matrix[row][col] = distance_matrix[row][col] - minimum
        else:
          distance_matrix[row][col] = 0
      lower_bound += minimum

  return lower_bound, distance_matrix
```

The function starts off by creating a copy of the given distance matrix using NumPy which takes $O(n^2)$. This also results in a space complexity of $O(n^2)$ to accommodate the new $n \times n$ matrix. Then, for every city, the distance from the second-to-last city to the current city is set to infinity, and the distance from the current city to the last city is set to infinity. This is to ensure that once a city is visited, it cannot be used to directly connect to other cities in the tour. That is a key part of the partial path approach. This loop ultimately runs in $O(n)$ time.

Then, as in the `initializeMatrix()` function, the matrix is reduced first for every row, then for every column, calculating the lower bound as it goes. These loops both run in $O(n^2)$, generating the reduced cost matrix, and ultimately returning it. In the end, the whole function runs in $O(n^2)$ time and has a space complexity of $O(n^2)$.

Branch & Bound Entry Function

The branch & bound algorithm is implemented as follows:

```
def branchAndBound(self, time_allowance=60.0):
  results = {}
  cities = self._scenario.getCities()
  city_num = len(cities)
  maximum = 1
  total = 1
  pruned = 0

  start_time = time.time()

  # Run greedy to get a starting point
  greedy_results = self.greedy() # O(n^2)
  bssf = greedy_results['soln']
  count = greedy_results['count']

  # Create the initial matrix and lower bound
  initial_matrix, lower_bound = self.initializeMatrix() # O(n^2)
  cities_queue = Heap()
  cities_queue.insert(0, lower_bound, 0, [0], initial_matrix) # O(log n)

  # Run the branch and bound algorithm
  while cities_queue.size() > 0 and time.time()-start_time < time_allowance: # O(n!)
    curr_city = cities_queue.deleteMin() # O(log n)

    if curr_city.lower_bound > bssf.cost:
      pruned += 1
      continue

    if len(curr_city.path) == city_num: # Path is complete
      count += 1
      if curr_city.distance_matrix[curr_city.path[-1]][curr_city.path[0]] < np.inf:
        # Path is a tour
        bssf_attempt = TSPSolution([cities[i] for i in curr_city.path])
        if bssf_attempt.cost < bssf.cost: # Found a better tour
          bssf = bssf_attempt
        break
      else: # Path is not a tour
        pruned += 1
        continue
    else: # Path is not complete
      for i in range(city_num): # O(n)
        if i not in curr_city.path:
          new_path = curr_city.path[:]
          new_path.append(i)
          lower_bound, new_matrix = self.findLowerBound(curr_city.lower_bound,
```

```
curr_city.distance_matrix, new_path) # O(n^2)
          cities_queue.insert(i, lower_bound,
curr_city.distance_matrix[curr_city.path[-1]][i], new_path, new_matrix)
          total += 1

          if cities_queue.size() > maximum:
            maximum = cities_queue.size()

  end_time = time.time()

  results['cost'] = bssf.cost
  results['time'] = end_time - start_time
  results['count'] = count
  results['soln'] = bssf
  results['max'] = maximum
  results['total'] = total
  results['pruned'] = pruned

  return results
```

First, it initializes the necessary values that it will keep track of during the algorithm, and generates a preliminary solution for the problem. This solution is found using the greedy algorithm, described above, and as a result it takes $O(n^2)$ time. This is done so that the branch & bound algorithm has a starting point that it can compare against as it tries to find a new "best solution so far", or bssf.

The function then calls `initializeMatrix()`, ($O(n^2)$), and inserts the first city into the heap, ($O(\log n)$). Then, the algorithm begins by finding the city with the shortest distance via `deleteMin()`, ($O(\log n)$), ignores it if the cost is already greater than that of the bssf, then generates an entirely new path for every unvisited city, updating the distance matrix and lower bound iteratively. If the path is found to be a tour, its cost is compared with that of the bssf, and the bssf is replaced if it is in fact cheaper. Altogether, that final loop has a worst case scenario of $O(n!)$, because every combination of the cities can ultimately be considered.

In the end, the time complexity of the function is $O(n!)$ and the space complexity is $O(n^2 + n)$, (where n^2 accounts for the distance matrix and n accounts for the heap usage). The space complexity can be simplified to $O(n^2)$.

Runtime Analysis

By running the branch & bound algorithm against different city numbers and seeds, the following data was gathered:

# 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	1.077392s	12347*	1453	0	1647	15
16	902	7.819169s	10944*	7270	1	8279	49
18	419	22.611909s	12697*	19637	0	22283	161
20	45	18.033036s	12356*	11469	1	12550	100
22	295	57.801389s	13234*	33260	2	38327	218
22	5	45.013235s	16024*	32840	0	35575	190
24	714	60s (TO)	20018	26442	0	28535	210
26	880	60s (TO)	18781	17514	0	18707	158
28	769	60s (TO)	18227	19150	0	20272	115
40	181	60s (TO)	26204	8441	0	8777	70

By observing the data calculated above, it seems that as the number of cities increases, the other paramters also generally increase. The running time increases with the number of cities, and I noticed that it would occasionally time out around 22 cities, and consistently time out for 24 cities and up. This makes sense, because the traveling salesman problem is NP-hard, and the complexity grows significantly with the problem size.

The cost of the best tour increases with problem size, which makes sense because more cities will cost more to tour. The maximum number of states stored, states created, and states pruned all generally increased with problem size, which makes sense given how more cities will require more iterations of states in the algorithm. The number of bssf updates did not show a clear trend in my data, though it may indicate that my branch & bound algorithm struggled with data sizes larger than 22.

To make my branch & bound algorithm more sophisticated, I tried prioritizing depth over breadth when searching. This would allow me to find the best solutions faster so that I can prune “lost cause” paths earlier while searching. To accomplish this, I selected each new city using the `deleteMin()` function of the priority queue in order to travel to closer cities first. I also explored through entire paths before starting other paths in order to gather more solutions early in the algorithm. The running times of my algorithm are not amazing and there is plenty of work to be done in optimizing it, but those were my attempts to improve its efficiency to some degree.

Source Code

The full source code is attached below for reference.

```
#!/usr/bin/python3

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
elif PYQT_VER == 'PYQT6':
  from PyQt6.QtCore import QLineF, QPointF
else:
  raise Exception('Unsupported Version of PyQt: {}'.format(PYQT_VER))


import time
import numpy as np
from TSPClasses import *


class City:
  '''
  Create a city object. Used to represent a node in the queues.

  Time Complexity: O(1)
  Space Complexity: O(1)
  '''
  def __init__(self, node_id, distance, lower_bound, path, distance_matrix):
    self.node_id = node_id
    self.distance = distance
    self.lower_bound = lower_bound
    self.path = path
    self.distance_matrix = distance_matrix


class Heap():
    '''
    Initialize the heap.

    Time Complexity: O(1)
    Space Complexity: O(1)
    '''
    def __init__(self):
        self.nodes = []
        self.lookup = []

    '''
    Return the size of the heap.

    Time Complexity: O(1)
    Space Complexity: O(1)
    '''
    def size(self):
        return len(self.nodes)
```

```python
    '''
    Insert a node into the heap.

    Time complexity: O(log n)
    Space complexity: O(1)
    '''
    def insert(self, index, distance, lower_bound, path, distance_matrix):
        self.nodes.append(City(index, distance, lower_bound, path, distance_matrix))
        self.lookup.append(len(self.nodes)-1)
        self.bubbleUp(len(self.nodes)-1) # O(log n)

    '''
    Return the index of the node with the minimum distance and delete it.

    Time complexity: O(log n)
    Space complexity: O(1)
    '''
    def deleteMin(self):
        min = self.nodes[0]
        self.nodes[0] = self.nodes[len(self.nodes)-1]
        self.nodes.pop()
        self.bubbleDown(0) # O(log n)
        return min

    '''
    Sift the node at given index up the heap until it is above all nodes
    with a greater distance.

    Time complexity: O(log n)
    Space complexity: O(1)
    '''
    def bubbleUp(self, index):
        parent = (index-1) // 2

        if self.nodes[parent].distance > self.nodes[index].distance:
            self.swap(index, parent) # O(1)
            self.bubbleUp(parent) # O(log n)

    '''
    Sift the node at given index down the heap until it is below all nodes
    with a lesser distance.

    Time complexity: O(log n)
    Space complexity: O(1)
    '''
    def bubbleDown(self, index):
        left_child = (index * 2) + 1
        right_child = (index * 2) + 2

        # determine which side to sift down
        if left_child > len(self.nodes)-1 or right_child > len(self.nodes)-1:
            min_child =  -1
        elif (self.nodes[left_child].distance < self.nodes[right_child].distance):
            min_child = left_child
        else:
            min_child = right_child
```

```
        # exit condition for recursion (node is leaf)
        if min_child < 1 or min_child > len(self.nodes)-1:
            return

        if self.nodes[index].distance > self.nodes[min_child].distance:
            self.swap(index, min_child) # O(1)
            self.bubbleDown(min_child) # O(log n)

    '''
    Swap the nodes at the given indices.

    Time complexity: O(1)
    Space complexity: O(1)
    '''
    def swap(self, i, j):
        temp = self.nodes[i]
        self.nodes[i] = self.nodes[j]
        self.nodes[j] = temp
        self.lookup[self.nodes[i].node_id] = i
        self.lookup[self.nodes[j].node_id] = j


class TSPSolver:
  def __init__( self, gui_view ):
    self._scenario = None

  def setupWithScenario( self, scenario ):
    self._scenario = scenario

  '''
    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.
    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.

    Time complexity: O(n!)
    Space complexity: O(n!)
  '''
  def defaultRandomTour( self, time_allowance=60.0 ):
    results = {}
    cities = self._scenario.getCities()
    city_num = 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( city_num )
      route = []
      # Now build the route using the random permutation
      for i in range( city_num ):
```

```
        route.append( cities[ perm[i] ] )
      bssf = TSPSolution(route)
      count += 1
      if bssf.cost < np.inf:
        # Found a valid route
        foundTour = True
    end_time = time.time()
    results['cost'] = bssf.cost if foundTour else math.inf
    results['time'] = end_time - start_time
    results['count'] = count
    results['soln'] = bssf
    results['max'] = None
    results['total'] = None
    results['pruned'] = None
    return results

  '''
    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.
    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.

    Time complexity: O(n^2)
    Space complexity: O(n)
  '''
  def greedy(self, time_allowance=60.0):
    results = {}
    cities = self._scenario.getCities()
    found_tour = False
    start_time = time.time()
    route = []

    for start_city in cities: # O(n)
      route = [start_city]

      while not found_tour and time.time() - start_time < time_allowance: # O(n)
        cheapest_neighbor = route[-1]

        for neighbor in cities:
          if neighbor not in route
             and route[-1].costTo(neighbor) < route[-1].costTo(cheapest_neighbor):
            cheapest_neighbor = neighbor

        if route[-1].costTo(cheapest_neighbor) == math.inf: # No more neighbors
          break
        route.append(cheapest_neighbor)

        if len(route) == len(cities)
           and route[-1].costTo(route[0]) < math.inf: # Found a tour
```

```
                found_tour = True

            if found_tour:
                break

        solution = TSPSolution(route)
        end_time = time.time()

        results["cost"] = solution.cost if found_tour else math.inf
        results["time"] = end_time - start_time
        results["count"] = 1 if found_tour else 0
        results["soln"] = solution
        results["max"] = None
        results["total"] = None
        results["pruned"] = None

        return results

    '''
        This is the entry point for the branch-and-bound algorithm that you will
implement
        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.

        Time complexity: O(n!)
        Space complexity: O(n^2)
    '''
    def branchAndBound(self, time_allowance=60.0):
        results = {}
        cities = self._scenario.getCities()
        city_num = len(cities)
        maximum = 1
        total = 1
        pruned = 0

        start_time = time.time()

        # Run greedy to get a starting point
        greedy_results = self.greedy() # O(n^2)
        bssf = greedy_results['soln']
        count = greedy_results['count']

        # Create the initial matrix and lower bound
        initial_matrix, lower_bound = self.initializeMatrix() # O(n^2)
        cities_queue = Heap()
        cities_queue.insert(0, lower_bound, 0, [0], initial_matrix) # O(log n)

        # Run the branch and bound algorithm
        while cities_queue.size() > 0 and time.time()-start_time < time_allowance: #
O(n!)
            curr_city = cities_queue.deleteMin() # O(log n)
```

```
      if curr_city.lower_bound > bssf.cost:
        pruned += 1
        continue

      if len(curr_city.path) == city_num: # Path is complete
        count += 1
        if curr_city.distance_matrix[curr_city.path[-1]][curr_city.path[0]] < np.inf:
          # Path is a tour
          bssf_attempt = TSPSolution([cities[i] for i in curr_city.path])
          if bssf_attempt.cost < bssf.cost: # Found a better tour
            bssf = bssf_attempt
          break
        else: # Path is not a tour
          pruned += 1
          continue
      else: # Path is not complete
        for i in range(city_num): # O(n)
          if i not in curr_city.path:
            new_path = curr_city.path[:]
            new_path.append(i)
            lower_bound, new_matrix = self.findLowerBound(curr_city.lower_bound,
curr_city.distance_matrix, new_path) # O(n^2)
            cities_queue.insert(i, lower_bound,
curr_city.distance_matrix[curr_city.path[-1]][i], new_path, new_matrix)
            total += 1

            if cities_queue.size() > maximum:
              maximum = cities_queue.size()

    end_time = time.time()

    results['cost'] = bssf.cost
    results['time'] = end_time - start_time
    results['count'] = count
    results['soln'] = bssf
    results['max'] = maximum
    results['total'] = total
    results['pruned'] = pruned

    return results

  '''
    Create the initial distance matrix and lower bound.
    Returns distance matrix, lower bound.

    Time complexity: O(n^2)
    Space complexity: O(n^2)
  '''
  def initializeMatrix(self):
    cities = self._scenario.getCities()
    city_num = len(cities)
    matrix = np.full((city_num, city_num), np.inf) # O(n^2)

    lower_bound = 0

    for city in range(city_num): # O(n^2)
```

```
    for next_city in range(city_num):
      matrix[city][next_city] = cities[city].costTo(cities[next_city])

  for row in range(city_num): # O(n^2)
    minimum = np.inf
    for col in range(city_num):
      if matrix[row][col] < minimum: # Found a new minimum
        minimum = matrix[row][col]
    for col in range(city_num):
      matrix[row][col] = matrix[row][col] - minimum
    lower_bound += minimum

  for col in range(city_num): # O(n^2)
    minimum = np.inf
    for row in range(city_num):
      if matrix[row][col] < minimum: # Found a new minimum
        minimum = matrix[row][col]
    for row in range(city_num):
      matrix[row][col] = matrix[row][col] - minimum
    lower_bound += minimum

  return matrix, lower_bound

'''
  Get the lower bound for the given path.
  Returns lower bound, distance matrix.

  Time complexity: O(n^2)
  Space complexity: O(n^2)
'''
def findLowerBound(self, old_lower_bound, old_distance_matrix, path):
  distance_matrix = np.copy(old_distance_matrix) # O(n^2)
  city_num = distance_matrix.shape[0]
  distance = distance_matrix[path[-2]][path[-1]]

  lower_bound = old_lower_bound + distance

  for i in range(city_num): # O(n)
    distance_matrix[path[-2]][i] = np.inf
    distance_matrix[i][path[-1]] = np.inf

  distance_matrix[path[-1]][path[-2]] = np.inf # Don't visit the same city twice

  for row in range(city_num): # O(n^2)
    if row == path[-2]: # Skip the row we just added
      continue
    minimum = np.inf
    for col in range(city_num):
      if col == path[-1]: # Skip the column we just added
        continue
      if distance_matrix[row][col] < minimum: # Found a new minimum
        minimum = distance_matrix[row][col]
    if minimum != np.inf: # Found a value to subtract
      for col in range(city_num):
        if distance_matrix[row][col] > 0:
          distance_matrix[row][col] = distance_matrix[row][col] - minimum
```

```
      else:
        distance_matrix[row][col] = 0
    lower_bound += minimum

for col in range(city_num): # O(n^2)
  if col == path[-1]: # Skip the column we just added
    continue
  minimum = np.inf
  for row in range(city_num):
    if row == path[-2]: # Skip the row we just added
      continue
    if distance_matrix[row][col] < minimum: # Found a new minimum
      minimum = distance_matrix[row][col]
  if minimum != np.inf: # Found a value to subtract
    for row in range(city_num):
      if distance_matrix[row][col] > 0:
        distance_matrix[row][col] = distance_matrix[row][col] - minimum
      else:
        distance_matrix[row][col] = 0
    lower_bound += minimum

return lower_bound, distance_matrix
```