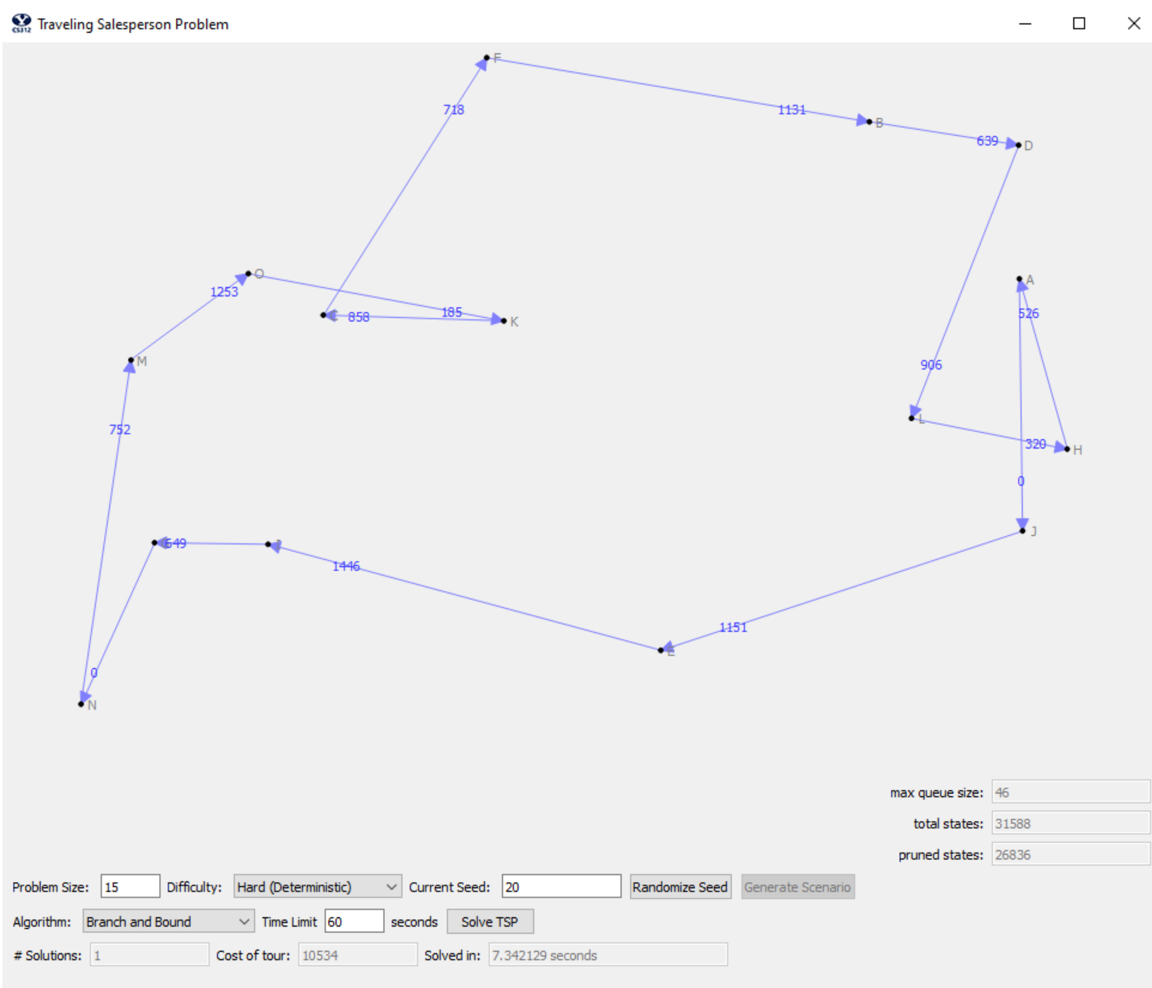
Matthew R. Christensen

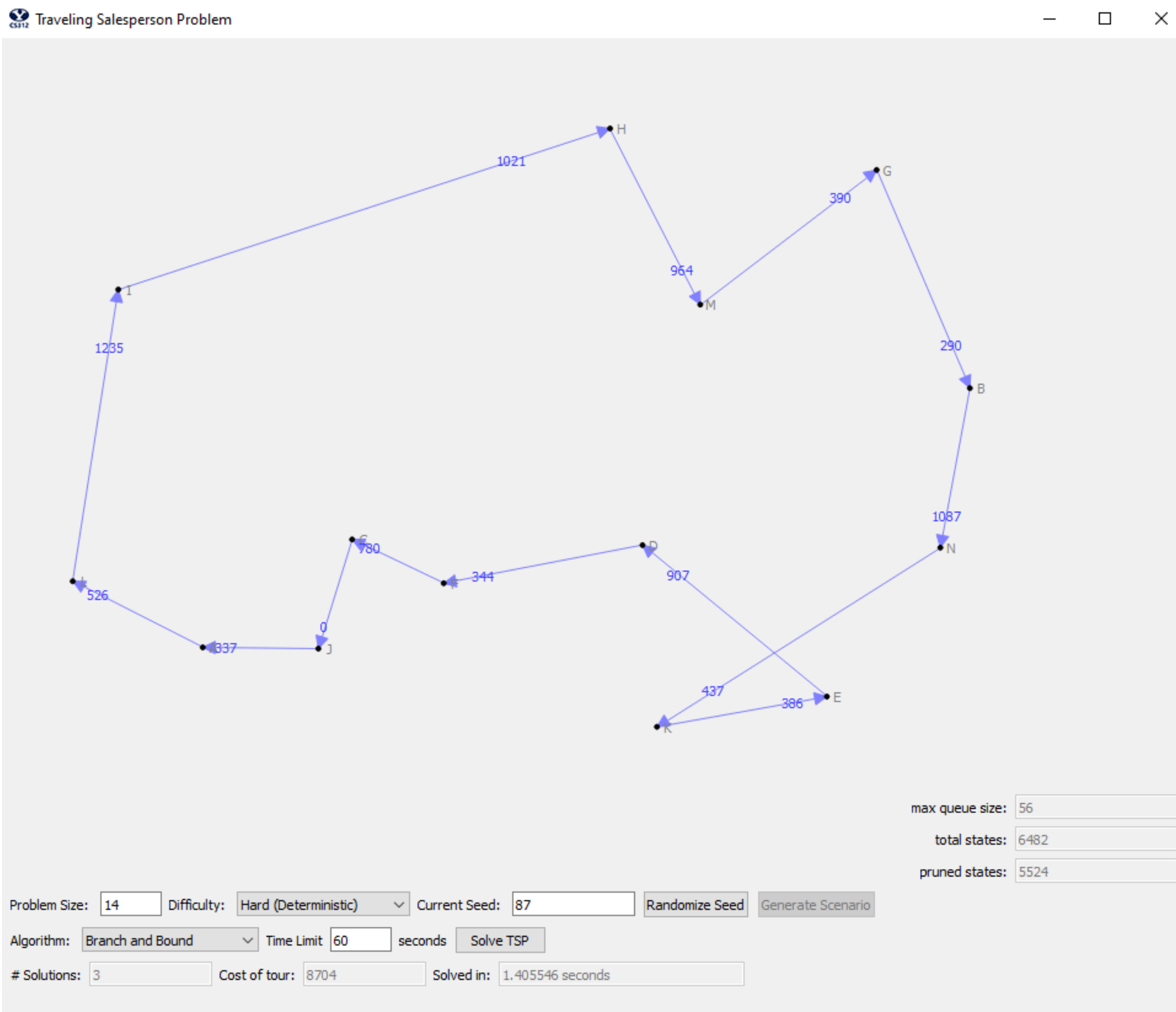
CS 312 (001) – Dr Martinez, Tony R

April 8, 2021

Branch and Bound Traveling Salesperson Algorithm Report

Screenshots





Time and Space Complexity Analysis

First, we will talk about the time and space complexity of our branch and bound traveling salesperson algorithm. This analysis will look at simplified pseudo code that is every similar to the actual algorithm implementation. However, for more details and granularity, please refer to the actual code in the appendix of this paper

This section of the paper will cover the 2nd part of the report (*[10] Explain both the time and space complexity of your algorithm by showing and summing up the complexity of each subsection of your code.*) as well as some additional description elements of parts 3 and 4 respectively (*[5] Describe the data structures you use to represent the states, [5] Describe the priority queue data structure you use and how it works*). Additionally, we talk about the 8th section of the report (*[10] Discuss the mechanisms you tried and how effective they were in getting the state space search to dig deeper and find more solutions early*) in this section of the paper as well.

First is an analysis of the priority queue. For this data structure I chose to use a heap from the heapq python library. Thus, the implementation of our heap is a binary heap, with a push and pop time complexity of O(log n) – the source code can be found here: https://github.com/python/cpython/blob/3.8/Lib/heapq.py. This is accomplished by having a sorted array where the order of said items is maintained whenever adding or removing items (aka, push and pop) – through shifting the tree structure within the array. Because of the ordering of the array, lookups remain a trivially fast operation, which is nicely stated in the documentation, "A nice feature of this sort is that you can efficiently insert new items while the sort is going on, provided that the inserted items are not "better" than the last 0'th element you extracted. This is especially useful in simulation contexts, where the tree holds all incoming events, and the "win" condition means the smallest scheduled time.

The space complexity is a little more involved, because each state we generate will have a size of O(n^2), thus our space complexity scale with the number of state necessary to solve the path. If we let *p* represent the depth at which we consider, we will have a total space complexity of n/k ∑ n^(2i) | i = 1, and the summation is from *i* to *p*.

Next is the complexity of the search state data structure. I created a custom search state class that holds all the necessary state space for each level of search, with function for comparison, initializing the cost matrix, storing the cost matrix, checking to see if a city is in the route, getting the length of the route, reducing the cost matrix, and updating the cost matrix. Additionally, it may be worth explaining that the cost matrix an *n* by *n* matrix that holds the cost of travel from one city to another in an array (which is where our space complexity came from in the last section). In all, this class is used to represent the state of a given search route, and this store a reduced cost matrix, the current route, the current cost (or bound) of the route, and the depth of the tree, with means of comparison with the less than operator.

The time and space complexity of each piece of the state class are as follow:

```
# set a given city's cost to infinity
# time complexity: O(1)
# space complexity: O(1)
def set_city_to_infinity(self, row, col):
    self.cost_matrix[col][row] = INF
```

This simple helper function is of constant time for both time and space complexity.

```
# set a column in the cost matrix to infinity
# time complexity: O(n)
# space complexity: O(1)
def set_column_to_infinity(self, column):
    self.cost_matrix[:][column] = INF
```

This simple helper function is also of constant time for space complexity (as we don't need to store anything and we are just modifying the array that has already been stored), while time complexity is O(n), as a whole column is assigned the value of infinity.

```
# set a row in the cost matrix to infinity
# time complexity: O(n)
# space complexity: O(1)
def set_row_to_infinity(self, row):
    self.cost_matrix[row][:] = INF
```

Identical to the last, this simple helper function is also of constant time for space complexity (as we don't need to store anything and we are just modifying the array that has already been stored), while time complexity is O(n), as a whole row is assigned the value of infinity.

```python
# find lowest value in the cost matrix column
# time complexity: O(n)
# space complexity: O(1)
def find_column_minimum(self, column):
    min_index = 0
    minimum = self.cost_matrix[min_index][column]
    for row in range(len(self.cost_matrix)):  # O(n)
        if self.cost_matrix[row][column] < minimum:
            minimum = self.cost_matrix[row][column]
            min_index = row
    return minimum, min_index
```

We must iterate through the whole column to find the minimum value, and thus we have a time complexity of O(n). Additionally, the space complexity is constant because we only store 2 insignificant values (the minimum found and its index).

```python
# find lowest value in the cost matrix row
# time complexity: O(n)
# space complexity: O(1)
def find_row_minimum(self, row):
    minimum = self.cost_matrix[row][0]
    minimum_index = 0
    for column in range(len(self.cost_matrix[row])):  # O(n)
        if self.cost_matrix[row][column] < minimum:
            minimum = self.cost_matrix[row][column]
            minimum_index = column

    return minimum, minimum_index
```

Identical to the last function, in this helper function we must iterate through the whole column to find the minimum value, and thus we have a time complexity of O(n). Additionally, the space complexity is constant because we only store 2 insignificant values (the minimum found and its index).

```python
# reduce column by subtracting the cost and minimum val
# time complexity: O(n)
# space complexity: O(1)
def reduce_column(self, column, minimum):
    for row in range(len(self.cost_matrix)):  # O(n)
        new_cost = self.cost_matrix[row][column] - minimum
        if new_cost == np.nan:
            self.cost_matrix[row][column] = INF
        else:
            self.cost_matrix[row, column] = new_cost
```

For time complexity, we must iterate through the whole column to reduce it, and thus will have a complexity of O(n). Meanwhile, the space complexity is constant because we do not store anything (just modify the matrix/array that has already been created.

```python
# reduce column by subtracting the cost and minimum val
# time complexity: O(n)
# space complexity: O(1)
def reduce_row(self, row, minimum):
    for column in range(len(self.cost_matrix[row])):  # O(n)
        new_cost = self.cost_matrix[row][column] - minimum
        if new_cost == np.nan:
            self.cost_matrix[row][column] = INF
        else:
            self.cost_matrix[row, column] = new_cost
```

Identical to the last function, this helper function, that reduces the row, must iterate through the whole column to reduce it – and thus will have a time complexity of O(n).

Meanwhile, the space complexity is constant because we do not store anything (just modify the matrix/array that has already been created.

```
# reduce columns of the cost matrix by finding the minimum and reducing if > 0 and < INF
# time complexity: O(n) * O(n + n) = O(n) * O(2n) = O(2n^2) = O(n^2)
# space complexity: O(1)
def reduce_matrix_columns(self):
    for column in range(len(self.cost_matrix[0])):  # O(n)
        minimum, minimum_index = self.find_column_minimum(column)  # O(n)
        if minimum > 0 and minimum != INF:
            self.reduce_column(column, minimum)  # O(n)
            self.best_cost += minimum
```

This simple function simply acts as the driver for the reduction of the columns within the cost matrix/array and contains said logic. This the loop of O(n), in the worst case scenario will have a work load of O(n + n), or O(2n) – which is equivalent to O(n). Thus, the overall time complexity is O(n) * O(n) = O(n^2) – (as seen above). Additionally, the space complexity remains constant, because we are simply modifying the data that is already there.

```
# reduce columns of the cost matrix by finding the minimum and reducing if > 0 and < INF
# time complexity: O(n) * O(n + n) = O(n) * O(2n) = O(2n^2) = O(n^2)
# space complexity: O(1)
def reduce_matrix_rows(self):
    for row in range(len(self.cost_matrix)):  # O(n)
        minimum, minimum_index = self.find_row_minimum(row)  # O(n)
        if minimum > 0 and minimum != INF:
            self.reduce_row(row, minimum)  # O(k)
            self.best_cost += minimum
```

Identical to the last function, this simple helper simply acts as the driver for the reduction of the rows within the cost matrix/array and contains said logic. This the loop of O(n), in the worst case scenario will have a work load of O(n + n), or O(2n) – which is equivalent to O(n). Thus, the overall time complexity is O(n) * O(n) = O(n^2) – (as seen above). Additionally, the space complexity remains constant, because we are simply modifying the data that is already there.

```
# gets the cost from two cities, while setting the row and column to infinity
# time complexity: O(n) + O(n) + O(1) = O(2n + 1) = O(2n) = O(n)
# space complexity: O(1)
def set_cities_to_infinity(self, from_city, to_city):
    self.best_cost += self.cost_matrix[from_city][to_city]
    if self.best_cost != INF:
        self.set_row_to_infinity(from_city)  # O(n)
        self.set_column_to_infinity(to_city)  # O(n)
        self.set_city_to_infinity(from_city, to_city)  # O(1)
```

Once again, this simple function acts as a driver to call the previously discussed function, and thus we can simply add the complexity of the functions together for a time complexity of: O(row_inf) + O(col_inf) + O(city_inf) = O(n) + O(n) + O(1) = O(2n + 1) = O(2n) = O(n)

```
# time complexity: O(1)
# space complexity: O(1)
def set_matrix(self, matrix):
    self.cost_matrix = matrix
```

This function is pretty self-explanatory and operates in constant time in both space and time complexities.

```
# time complexity: O(n)
# space complexity: O(1)
def city_in_route(self, city):
    for val in self.route:  # Worst case: O(n)
        if val._index == city._index:  # O(1)
            return True
    return False
```

This function, in the worst-case scenario (either the city is the last in the list, or not in the list) will have to iterate through the entire route – which at most will be of *n* length. Thus we have a time complexity of O(n), while the space complexity remains constant because we are only traversing existing data (not creating new memory).

```
# initialize cost matrix for cities
# time complexity: O(n) * O(n) * O(1) = O(n^2)
# space complexity: O(n^2)
def init_matrix(self):
    self.cost_matrix = np.zeros(shape=(len(self.cities), len(self.cities)))  # O(n^2)
    row_index = 0
    for fromCity in self.cities:  # O(n)
        col_index = 0
        for toCity in self.cities:  # O(n)
            self.cost_matrix[row_index][col_index] = fromCity.costTo(toCity)  # O(1)
            col_index += 1
        row_index += 1
```

This function initializes the cost matrix array, and thus will have to traves and newly created *n* by *n* array, and thus will have a time and space complexity of O(n^2)

This concludes the discussion on the search state class.

Next is the discussion revolving around generating a first solution to beat, known from here on forward as BSSF (best-solution-so-far). The generate the first initial BSSF we use a greedy algorithm:

```python
# time complexity: O(n) * O(n) = O(n^)
# space complexity: O(n) + O(n) + O(n) = O(3n) = O(n)
def greedy(self, time_allowance=60.0):
    route_found = False
    route = []  # Space: O(n)
    list_of_possible_start_cities = self._scenario.getCities().copy()  # Space: O(n)
    cities = self._scenario.getCities()  # Space: O(n)
    start_city = list_of_possible_start_cities.pop()
    city = start_city
    route.append(city)
    start_time = time.time()
    while route_found is False and (time.time() - start_time) < time_allowance:  # O(n)
        lowest_cost = math.inf
        lowest_city = None
        for neighbor in cities:  # O(n)
            if neighbor is city:
                continue
            if city.costTo(neighbor) < lowest_cost and (neighbor not in route):
                lowest_cost = city.costTo(neighbor)
                lowest_city = neighbor
        if lowest_city is None:  # check to see if can't continue
            if city.costTo(start_city) < lowest_cost:  # check to see if we're done
                route_found = True
                best_sol_so_far = TSPSolution(route)
            else:
                route.clear()
                start_city = list_of_possible_start_cities.pop()
                city = start_city
        else:  # we did find a lowest_city
            route.append(lowest_city)
            city = lowest_city

    end_time = time.time()
    results = {'route': best_sol_so_far.route, 'cost': best_sol_so_far.cost if route_found else math.inf,
               'time': end_time - start_time, 'count': len(route), 'soln': best_sol_so_far, 'max': None,
               'total': None, 'pruned': None}
    return results
```

This algorithm simply makes the next best choice by comparing a given city and making the best next choice until finding a complete route. In the worst-case scenario, our while loop will run through all the cities while the worst-case scenario on the inner-loop (i.e., the work on each iteration) is O(n) time complexity as we need to loop through all the neighbors. Thus, our time complexity is O(n) * O(n) = O(n^2). Our space complexity is a little more simple, as we store the route, list of possible start cities, and the list of all cities. Each of these lists are of *n* length, and thus our space complexity is as follows: O(route) + O(len(cites)) + O(len(cities)) = O(n) + O(n) + O(n) = O(3n) = O(n).

Next, and finally, is the entire Branch and bound algorithm all together:

```python
# time complexity: worse case: O(n!) - average: O(p) * (O(log n) + (O(n) * (O(log n) + O(n^2) + O(log n)))) =
#                                               O(p) * (O(log n) + O(n * n^2)) = O(p) * O(n^3) = O(pn^3)
# space complexity: worse case: O(n!) - average: (p) * O(n^2 + n) = O(p * n^2)
def branch_and_bound(self, time_allowance=60.0):
    count = pruned_states = 0
    max_heap_size = total_states = 1
    solution_to_beat = TSPSolution(self.greedy()['route'])  # O(n^2) for time, O(n) for space

    heap = []
    cities = self._scenario.getCities()
    state = SearchState([cities[0]], cities, 0)
    state.init_matrix()
    state.reduce_matrix()
    heappush(heap, state)

    start_time = time.time()
    # time complexity: worse case: O(n!) - average: O(p) * (O(log n) + (O(n) * (# O(log n) + O(n^2) + O(log n))))
    # space complexity: worse case: O(n!) - average: (p) * O(n^2 + n) = O(p * n^2)
    while (time.time() - start_time) < time_allowance and len(heap) > 0:

        # record the biggest heap size we've seen
        max_heap_size = len(heap) if len(heap) > max_heap_size else max_heap_size

        # get next state to analyze
        current_state = heappop(heap)  # O(log n)

        # if state is less costly
        if current_state.best_cost < solution_to_beat.cost:

            # if path contains all cities (make sure it's a valid solution)
            if len(current_state.route) == len(cities):
                last_cost = current_state.route[-1].costTo(current_state.route[0])
                current_state.best_cost += last_cost

                # if state cost is better than our current best solution
                if current_state.best_cost < solution_to_beat.cost:
                    solution_to_beat = TSPSolution(deepcopy(current_state.route))
                    count += 1
```

```python
        # if out path doesn't contain all cities (make it a loop)
        else:
            for city in cities:  # O(n)

                # add cities that are not in our path
                if not current_state.city_in_route(city):
                    total_states += 1
                    new_path = current_state.route.copy().append(city)  # add current city
                    new_state = SearchState(new_path, cities, current_state.best_cost)
                    new_state.cost_matrix = np.copy(current_state.cost_matrix)  # O(log n)
                    city1 = new_state.route[new_state.len() - 2]
                    city2 = new_state.route[new_state.len() - 1]
                    new_state.set_cities_to_infinity(city1._index, city2._index)  # O(1)
                    new_state.reduce_matrix()  # O(n^2)

                    # if the new state could be better than the current solution
                    if new_state.best_cost < solution_to_beat.cost:
                        heappush(heap, new_state)  # O(log n)
                    # if the new state can't beat the current solution then we prune
                    else:
                        pruned_states += 1

    # if there is not improvement (after evaluating the state) then we prune the state
    else:
        pruned_states += 1

end_time = time.time()

results = {'cost': solution_to_beat.cost, 'time': end_time - start_time, 'count': count,
           'soln': solution_to_beat, 'max': max_heap_size, 'total': total_states,
           'pruned': pruned_states + len(heap)}
return results
```

As discussed above, the branch and bound algorithm takes a given path (here generated by the greedy algorithm), and then we expand the search state into deeper levels by looking for possible better decisions and then exploring them in their own state (as previously talked about). Because of the nature of the algorithm, our worst-case scenario will be of order O(n!) for both space and time complexity. However, the average case can be generalized the O(n/k ∑n^i) | p = number of states generated minus the pruned states, and where the sigma goes from i = 1 to p, and where k represents the algorithm’s optimization. This is because, while there are n cities at each depth, we consistently narrow down our search through setting our cost matrix to infinity where possible to speed up computation. We will discuss this p value more in the coming section of the paper.

Empirical Analysis

In this section of the paper we will cover the 6th and 7th portions of the report *([25] Include a table containing the following columns, [10] 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.*)

First, we show our empirical data:

	# Cities	Seed	Running time (sec.)	Cost of best tour found (*=optimal)	Max # of stored states	# of BSSF updates	Total # of states created	Total # states pruned
1	15	20	7.631081104	10534*	46	1	31588	26836
2	16	902	17.53301644	8362*	74	1	69646	60439
3	10	82	0.012995481	7811*	4	0	91	77
4	15	518	11.80880761	9513*	68	2	49020	41762
5	20	339	55.1789155	10903*	119	4	156980	138469
6	20	951	60.00130248	12071	133	6	160158	135471
7	17	816	28.83160877	10093*	94	7	104351	89859
8	30	403	60.00185466	12373	321	1	89901	69147
9	35	13	60.00237155	16124	415	0	69815	58820
10	40	1	60.00247741	20829	598	10	53778	45947

We can see four cases in which no optimal solution was found because of a 60 second time requirement (see case 6, 8, 9, and 10).

Because depth is prioritizing exploration of cheaper possible solutions, the state space doesn't start to increase rapidly until the number of cities begins to climb (and once done so, the states created start to climb sharply). However, space complexity is held low by focusing on pruning a lot of states early one – meaning the number of max stored states remains very low even when the number of states blows up. Thus, we can see how the total number of states created and pruned are heavily correlated with the problem size. However, the number of BSSF updates depends more on the accuracy of our initial greedy route and how well is does to get to an optimal route. The final observation is that sometimes there are no updates to the BSSF if the greedy approximation is the best is can find within the time frame.

General Discussion

For the general discussion points, sections 3-5 as well as section 8 of the paper requirements (*3. [5] Describe the data structures you use to represent the states. 4. [5] Describe the priority queue data structure you use and how it works. 5. [5] Describe your approach for the initial BSSF. 8. [10] Discuss the mechanisms you tried and how effective they were in getting the state space search to dig deeper and find more solutions early*.) see the previous discussion (these points were interwoven into the other sections, but all the content is there).

Conclusion

In conclusion, we have shown that our algorithm is correctly running a brand and bound implementation in order to solve the traveling salesperson problem. Much care must be taken to expand search state to deeper levels earlier if gains are to be had in time and space complexity. Thus, we can conclude that the analysis of this branch and bound algorithm stating its completeness and compliance.

Appendix

This portion of the paper covers the first part of the report (*[20] Include your well-commented code.*)

TSPSolver.py

```python
#!/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
else:
    raise Exception('Unsupported Version of PyQt: {}'.format(PYQT_VER))

import time
import numpy as np
from TSPClasses import *
from heapq import heappop, heappush
from copy import deepcopy
import itertools

INF = np.inf


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

    def setupWithScenario(self, 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):
        results = {}
        cities = self._scenario.getCities()
        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 = []
            # Now build the route using the random permutation
            for i in range(ncities):
                route.append(cities[perm[i]])
            bssf = TSPSolution(route)
            count += 1
            if bssf.cost < 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

    ''' <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>
    '''

    # time complexity: O(n) * O(n) = O(n^)
    # space complexity: O(n) + O(n) + O(n) = O(3n) = O(n)
    def greedy(self, time_allowance=60.0):
        route_found = False
        route = []  # Space: O(n)
        list_of_possible_start_cities = self._scenario.getCities().copy()  # Space: O(n)
        cities = self._scenario.getCities()  # Space: O(n)
        start_city = list_of_possible_start_cities.pop()
        city = start_city
        route.append(city)
        start_time = time.time()
        while route_found is False and (time.time() - start_time) < time_allowance:  # O(n)
            lowest_cost = math.inf
            lowest_city = None
            for neighbor in cities:  # O(n)
                if neighbor is city:
                    continue
                if city.costTo(neighbor) < lowest_cost and (neighbor not in route):
                    lowest_cost = city.costTo(neighbor)
                    lowest_city = neighbor
            if lowest_city is None:  # check to see if can't continue
                if city.costTo(start_city) < lowest_cost:  # check to see if we're done
                    route_found = True
                    best_sol_so_far = TSPSolution(route)
                else:
                    route.clear()
                    start_city = list_of_possible_start_cities.pop()
                    city = start_city
            else:  # we did find a lowest_city
                route.append(lowest_city)
                city = lowest_city

        end_time = time.time()
        results = {'route': best_sol_so_far.route, 'cost': best_sol_so_far.cost if route_found else math.inf,
                   'time': end_time - start_time, 'count': len(route), 'soln': best_sol_so_far, 'max': None,
                   'total': None, 'pruned': None}
        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>
    '''

    # time complexity: worse case: O(n!) - average: O(p) * (O(log n) + (O(n) * (O(log n) + O(n^2) + O(log n)))) =
    #                                                O(p) * (O(log n) + O(n * n^2)) = O(p) * O(n^3) = O(pn^3)
    # space complexity: worse case: O(n!) - average: (p) * O(n^2 + n) = O(p * n^2)
    def branch_and_bound(self, time_allowance=60.0):
        count = pruned_states = 0
        max_heap_size = total_states = 1
        solution_to_beat = TSPSolution(self.greedy()['route'])  # O(n^2) for time, O(n) for space

        heap = []
        cities = self._scenario.getCities()
        state = SearchState([cities[0]], cities, 0)
        state.init_matrix()
        state.reduce_matrix()
        heappush(heap, state)

        start_time = time.time()
        # time complexity: worse case: O(n!) - average: O(p) * (O(log n) + (O(n) * (# O(log n) + O(n^2) + O(log n))))
        # space complexity: worse case: O(n!) - average: (p) * O(n^2 + n) = O(p * n^2)
        while (time.time() - start_time) < time_allowance and len(heap) > 0:
```

```python

            # record the biggest heap size we've seen
            max_heap_size = len(heap) if len(heap) > max_heap_size else max_heap_size

            # get next state to analyze
            current_state = heappop(heap)  # O(log n)

            # if state is less costly
            if current_state.best_cost < solution_to_beat.cost:

                # if path contains all cities (make sure it's a valid solution)
                if len(current_state.route) == len(cities):
                    last_cost = current_state.route[-1].costTo(current_state.route[0])
                    current_state.best_cost += last_cost

                    # if state cost is better than our current best solution
                    if current_state.best_cost < solution_to_beat.cost:
                        solution_to_beat = TSPSolution(deepcopy(current_state.route))
                        count += 1

                # if out path doesn't contain all cities (make it a loop)
                else:
                    for city in cities:  # O(n)

                        # add cities that are not in our path
                        if not current_state.city_in_route(city):
                            total_states += 1
                            new_path = current_state.route.copy().append(city)  # add current city
                            new_state = SearchState(new_path, cities, current_state.best_cost)
                            new_state.cost_matrix = np.copy(current_state.cost_matrix)  # O(log n)
                            city1 = new_state.route[new_state.len() - 2]
                            city2 = new_state.route[new_state.len() - 1]
                            new_state.set_cities_to_infinity(city1._index, city2._index)  # O(1)
                            new_state.reduce_matrix()  # O(n^2)

                            # if the new state could be better than the current solution
                            if new_state.best_cost < solution_to_beat.cost:
                                heappush(heap, new_state)  # O(log n)
                            # if the new state can't beat the current solution then we prune
                            else:
                                pruned_states += 1

            # if there is not improvement (after evaluating the state) then we prune the state
            else:
                pruned_states += 1

        end_time = time.time()

        results = {'cost': solution_to_beat.cost, 'time': end_time - start_time, 'count': count,
                   'soln': solution_to_beat, 'max': max_heap_size, 'total': total_states,
                   'pruned': pruned_states + len(heap)}
        return results

    def fancy(self, time_allowance=60.0):
        pass


class SearchState:
    def __init__(self, path, cities, best_cost):
        super().__init__()
        self.cost_matrix = np.zeros(shape=(len(self.cities), len(self.cities)))
        self.best_cost = best_cost
        self.route = path
        self.cities = cities

    # comparison function
    # time complexity: O(1)
    # space complexity: O(1)
    def __lt__(self, value):
        if len(self.route) is not len(value.route):
            return len(self.route) > len(value.route)
        else:
            return self.best_cost < value.best_cost

    # initialize cost matrix for cities
```

```python
    # time complexity: O(n) * O(n) * O(1) = O(n^2)
    # space complexity: O(n^2)
    def init_matrix(self):
        self.cost_matrix = np.zeros(shape=(len(self.cities), len(self.cities)))  # O(n^2)
        row_index = 0
        for fromCity in self.cities:  # O(n)
            col_index = 0
            for toCity in self.cities:  # O(n)
                self.cost_matrix[row_index][col_index] = fromCity.costTo(toCity)  # O(1)
                col_index += 1
            row_index += 1

    # time complexity: O(1)
    # space complexity: O(1)
    def __str__(self):
        return str(self.cost_matrix)

    # time complexity: O(1)
    # space complexity: O(1)
    def len(self):
        return len(self.route)

    # time complexity: O(n)
    # space complexity: O(1)
    def city_in_route(self, city):
        for val in self.route:  # Worst case: O(n)
            if val._index == city._index:  # O(1)
                return True
        return False

    # time complexity: O(1)
    # space complexity: O(1)
    def set_matrix(self, matrix):
        self.cost_matrix = matrix

    # gets the cost from two cities, while setting the row and column to infinity
    # time complexity: O(n) + O(n) + O(1) = O(2n + 1) = O(2n) = O(n)
    # space complexity: O(1)
    def set_cities_to_infinity(self, from_city, to_city):
        self.best_cost += self.cost_matrix[from_city][to_city]
        if self.best_cost != INF:
            self.set_row_to_infinity(from_city)  # O(n)
            self.set_column_to_infinity(to_city)  # O(n)
            self.set_city_to_infinity(from_city, to_city)  # O(1)

    # reduce matrix by reducing columns and rows
    # time complexity: O(n^2) + O(n^2) = O(2n^2) = O(n^2)
    # space complexity: O(1)
    def reduce_matrix(self):
        # skip reducing if best cost is infinity (it's not going to get better)
        if self.best_cost != INF:
            self.reduce_matrix_rows()  # O(n^2)
            self.reduce_matrix_columns()  # O(n^2)

    # reduce columns of the cost matrix by finding the minimum and reducing if > 0 and < INF
    # time complexity: O(n) * O(n + n) = O(n) * O(2n) = O(2n^2) = O(n^2)
    # space complexity: O(1)
    def reduce_matrix_rows(self):
        for row in range(len(self.cost_matrix)):  # O(n)
            minimum, minimum_index = self.find_row_minimum(row)  # O(n)
            if minimum > 0 and minimum != INF:
                self.reduce_row(row, minimum)  # O(k)
                self.best_cost += minimum

    # reduce columns of the cost matrix by finding the minimum and reducing if > 0 and < INF
    # time complexity: O(n) * O(n + n) = O(n) * O(2n) = O(2n^2) = O(n^2)
    # space complexity: O(1)
    def reduce_matrix_columns(self):
        for column in range(len(self.cost_matrix[0])):  # O(n)
            minimum, minimum_index = self.find_column_minimum(column)  # O(n)
            if minimum > 0 and minimum != INF:
                self.reduce_column(column, minimum)  # O(n)
                self.best_cost += minimum

    # reduce column by subtracting the cost and minimum val
```

```
    # time complexity: O(n)
    # space complexity: O(1)
    def reduce_row(self, row, minimum):
        for column in range(len(self.cost_matrix[row])):  # O(n)
            new_cost = self.cost_matrix[row][column] - minimum
            if new_cost == np.nan:
                self.cost_matrix[row][column] = INF
            else:
                self.cost_matrix[row, column] = new_cost

    # reduce column by subtracting the cost and minimum val
    # time complexity: O(n)
    # space complexity: O(1)
    def reduce_column(self, column, minimum):
        for row in range(len(self.cost_matrix)):  # O(n)
            new_cost = self.cost_matrix[row][column] - minimum
            if new_cost == np.nan:
                self.cost_matrix[row][column] = INF
            else:
                self.cost_matrix[row, column] = new_cost

    # find lowest value in the cost matrix row
    # time complexity: O(n)
    # space complexity: O(1)
    def find_row_minimum(self, row):
        minimum = self.cost_matrix[row][0]
        minimum_index = 0
        for column in range(len(self.cost_matrix[row])):  # O(n)
            if self.cost_matrix[row][column] < minimum:
                minimum = self.cost_matrix[row][column]
                minimum_index = column

        return minimum, minimum_index

    # find lowest value in the cost matrix column
    # time complexity: O(n)
    # space complexity: O(1)
    def find_column_minimum(self, column):
        min_index = 0
        minimum = self.cost_matrix[min_index][column]
        for row in range(len(self.cost_matrix)):  # O(n)
            if self.cost_matrix[row][column] < minimum:
                minimum = self.cost_matrix[row][column]
                min_index = row
        return minimum, min_index

    # set a row in the cost matrix to infinity
    # time complexity: O(n)
    # space complexity: O(1)
    def set_row_to_infinity(self, row):
        self.cost_matrix[row][:] = INF

    # set a column in the cost matrix to infinity
    # time complexity: O(n)
    # space complexity: O(1)
    def set_column_to_infinity(self, column):
        self.cost_matrix[:][column] = INF

    # set a given city's cost to infinity
    # time complexity: O(1)
    # space complexity: O(1)
    def set_city_to_infinity(self, row, col):
        self.cost_matrix[col][row] = INF
```