Final Report

Program Overview

The algorithm implemented is the best first search branch and bound algorithm to solve the asymmetrical traveling salesman problem. The input is given in the file “input.txt” and is given in the form “[x1,x2,x3]\n[y1,y2,y3]\n[z1,z2,z3]”. Note, the values on the diagonal do not matter since they will be replaced with infinity since visiting the town the tour is currently in is not allowed. Another note, the tour given by the external library is not guaranteed to be the same tour taken by the branch and bound algorithm, as tsp problems can have multiple valid solutions, the important metric to look at is the minimum cost. The algorithm is then run on the adjacency matrix to get a minimum cost and the associated tour. Also included in the program are all the methods used to generate plots and graphics for the presentation and this report, however their input can not be modified using the input file since to run every test would take a very long time. If it is desired to run these tests, simply uncomment them in the main function. The methods are included, yet commented, so that it is clear what was done for testing and graphic generation. To run this program, edit the input file such that the matrix is square and in the form described above, which is also given in the sample input file, and then use the command “python3 TSP.py”. Alternatively the program can be run from any IDE.

Final Implementation

```
class Node:
    def __init__(self, cost, cost_matrix, path, parent):
        self.__children = []
        self.__parent = parent
        self.__path = path
        self.__cost = cost
        self.__cost_matrix = cost_matrix

    def __lt__(self, other):
        return self.get_cost() < other.get_cost()

    def __gt__(self, other):
        return self.get_cost() > other.get_cost()

    def set_children(self, children):...
    def set_parent(self, parent):...
    def set_path(self, path):...
    def set_cost(self, cost):...
    def set_cost_matrix(self, cost_matrix):...
    def get_children(self):...
    def get_parent(self):...
    def get_path(self):...
    def get_cost(self):...
    def get_cost_matrix(self):...

    def append_child(self, child):
        children = self.get_children()
        children.append(child)
        self.set_children(children)

    def append_town(self, town):
        path = self.get_path()
        path.append(town)
        self.set_path(path)
```

The **Node** class is responsible for containing all information necessary to represent a node in the state space tree. The cost, cost matrix, path, parent, and the children of the node are all stored in the node. Additionally there are comparison functions written so that the nodes can easily be compared to be placed in the minimum heap. Finally there are append functions so that children of the node can easily be added, as well as an append town function to add a town onto the list of towns already visited by that node.

```
# This function includes the driver code for the function
def main():
    input_file_name = "input.txt"
    distances = read_input_file(input_file_name)
    tsp_print(distances.copy())
    # validation_test(n_limit=15, repetitions=100)
    # maximum_runtime_test(repetitions=5, max_time=30)
    # random_runtime_test(n_limit=15, repetitions=100)
    # animate_tsp(distances.copy())
```

The **main()** function includes all the driver code for the algorithm. Here the name of the input file can be changed, or the functions used to generate graphics can be uncommented and run.

```
# This function prints runtime, cost, and path for the branch and bound tsp solver and dynamic programming solver
# This function takes an adjacency matrix as an input
def tsp_print(distances):
    print(distances)

    start = timer()
    my_path, my_cost, nodes = tsp(distances.copy())
    end = timer()
    my_elapsed_time = end - start

    start = timer()
    dynamic_path, dynamic_cost = solve_tsp_dynamic_programming(distances.copy())
    end = timer()
    dynamic_elapsed_time = end - start

    print(f"Branch and bound runtime: {my_elapsed_time}")
    print(f"Dynamic programming runtime: {dynamic_elapsed_time}")

    print(f"Branch and bound minimum cost: {my_cost}")
    print(f"Dynamic programming minimum cost: {dynamic_cost}")

    print(f"Branch and bound path: {my_path}")
    print(f"Dynamic programming path: {dynamic_path}")
```

The **tsp_print** function includes all the driver code needed to print the time, cost, and path of the branch and bound algorithm and dynamic programming solver from the external library, python_tsp

```python
# This function reads the input file to generate an adjacency matrix
# This function takes a string file name as the input
# This function returns an adjacency matrix
def read_input_file(input_file_name):
    distances = []
    with open(input_file_name) as f:
        line = f.readline()
        while line:
            distances.append([float(x) for x in line.strip("\n[]").split(",")])
            line = f.readline()
    try:
        distances = np.asarray(distances)
    except:
        print("The input matrix is not square")
        quit(-1)

    i = 0
    while i < len(distances):
        j = 0
        while j < len(distances[i]):
            if i == j:
                distances[i][j] = np.inf
            j = j+1
        i = i + 1
    return distances
```

The **read_input_file()** function is used to read the input from the input file. First the file is opened, and then the function reads the file line by line to extract the distances from the input file. This function also makes sure that the input matrix is square, and then changes the diagonal to infinity, since visiting the same town that the algorithm is currently on is not allowed.

```python
# This is the main branch and bound tsp function
# This function takes an adjacency matrix as an input
# This function returns the path, cost, and a list of nodes to be used to show how the function explores the tree by the animate_tsp function
def tsp(distances):
    nodes = []
    bound = np.inf
    alive_nodes = []
    towns = [i for i in range(0, distances.shape[0])]
    start = 0
    end = 0
    cost = reduce(distances, start, end, 0)

    root = Node(cost, distances.copy(), [], None)
    root.append_town(end)

    # The nodes list is generated for use in animating the tree
    nodes.append(root)
    alive_nodes.append(root)

    answer = root
    while not alive_nodes == []:

        min_node = pop_min(alive_nodes)
        if min_node.get_cost() >= bound:
            return answer.get_path(), bound, nodes

        explore(min_node, towns)

        if min_node is not None:
            children = min_node.get_children()
            nodes += children

            if not children:
                answer = min_node
                bound = min_node.get_cost()

            for child in children:
                insert_node(alive_nodes, child)

    return answer.get_path(), bound, nodes
```

The **tsp()** function implements the branch and bound algorithm to solve the tsp. The function uses a priority queue keyed by the reduced cost of a node implemented by a heap called alive_nodes to get the minimum cost node which will be considered as the e node. If the e node cost is not greater than the bound, the e nodes children will be generated by the explore function. If the e node has no children generated, that means that the e node is also a leaf node and a possible solution has been found. The children of the e node are also added to the heap and the process continues until there are no alive nodes or until the cost of the e node is greater than the bound. If the cost of the e node is greater than or equal to the bound, which starts at infinity, then we return whatever answer we have at that point. Since the algorithm always pulls the minimum cost node, if the minimum cost node is ever greater than the bound, then every other possible node to explore will also have a cost greater than the bound, which means an optimal solution has been found.

```python
# This function explores the children of a give node
# This function takes the node to be explored as an input, as well as the total list of towns
def explore(node, towns):
    if node is None:
        return

    visited_towns = node.get_path()
    start = visited_towns[-1]

    remaining_towns = [i for i in towns if i not in visited_towns]
    for town in remaining_towns:
        cost_matrix = node.get_cost_matrix()
        current_cost_matrix = cost_matrix.copy()
        cost = reduce(current_cost_matrix, start, town, node.get_cost())
        node.append_child(Node(cost, current_cost_matrix, visited_towns + [town], node))
```

The **explore()** function determines what towns have been visited by a node, and gets a list of towns that have not yet been visited. The explore function then generates children of the e node for each town in the list of towns that have not been visited yet.

```python
# This function reduces an adjacency matrix and returns the cost for a given node
# This function takes the cost matrix as an input, as well as the start town, end town, and the previous node cost
def reduce(cost_matrix, start, end, previous_cost):
    edge_cost = cost_matrix[start][end]
    if edge_cost == np.inf:
        edge_cost = 0

    if not end == 0:
        cost_matrix[start] = np.inf
        cost_matrix[:, end] = np.inf
        cost_matrix[end][start] = np.inf

    n = cost_matrix.shape[0]
    reduced_cost = 0
    for i in range(0, n):
        min_row = min(cost_matrix[i])
        if min_row == np.inf:
            continue
        reduced_cost += min_row
        cost_matrix[i] = cost_matrix[i] - min_row

    for i in range(0, n):
        min_column = min(cost_matrix[:, i])
        if min_column == np.inf:
            continue
        reduced_cost += min_column
        cost_matrix[:, i] = cost_matrix[:, i] - min_column

    cost = reduced_cost + edge_cost + previous_cost
    return cost
```

The **reduce()** function reduces the cost matrix to get a cost for a cost matrix, start, end, and previous cost. The function checks the cost matrix to get an edge cost and one of the function

inputs is the previous node cost. To get the total cost of the node the reduce function just needs the reduce cost, which it finds by setting the row, column, and the town end to town start positions in the cost matrix to infinity, because the algorithm should not have the option to travel from the end town or to the start town. The matrix is then reduced by getting the minimum number in each row and subtracting that number from each row, and then doing the same things to the columns. The minimum for the rows and columns are added to an accumulator called the reduced cost, which is then added to the edge cost and previous cost to get the total cost for the node.

```
# This function is used to manage the heap which operates as a priorty queue. This function takes a node and moves it up in the heap
# to maintain the minimum heap property
# This function takes the heap which is a list as an input, as well as the index of the node to be operated on
def sift_up(heap, node_index):
    if not heap:
        return

    parent_index = int(((node_index - 1) / 2))
    if parent_index < 0:
        return

    if heap[node_index] < heap[parent_index]:
        heap[node_index], heap[parent_index] = heap[parent_index], heap[node_index]
        sift_up(heap, parent_index)
```

The **sift_up()** function is used to maintain the minimum heap property. The sift up function checks to see if the parent of a node is smaller, and if it is the node is swapped with its parent until its parent is not smaller.

```
# This function is used to manage the heap. This function takes a node and move it down through the heap to maintain the minimum heap property
# This function takes the heap which is a list as an input, as well as the index of the node to be operated on
def sift_down(heap, node_index):
    size = len(heap)
    min_index = node_index
    left_index = 2*node_index + 1
    right_index = 2*node_index + 2

    if left_index < size and heap[left_index] < heap[min_index]:
        min_index = left_index

    if right_index < size and heap[right_index] < heap[min_index]:
        min_index = right_index

    if not min_index == node_index:
        heap[node_index], heap[min_index] = heap[min_index], heap[node_index]
        sift_down(heap, min_index)
```

The **sift_down()** function is used to maintain the heap property. The sift down function checks to see which of the two children of the node is smaller, and swaps the node with the smaller child until neither child is smaller.

```
# This function inserts a node into the heap and ensures the heap property is satisfied
# This function takes two inputs, the heap list and the node to be inserted
def insert_node(heap, node):
    heap.append(node)
    sift_up(heap, len(heap)-1)
```

The **insert_node()** function is used to insert a node into the heap while maintaining the heap property. The node is appended to the end of the heap, and then sifted up until the heap property is maintained.

```
# This function returns the minimum node in the heap and removes it. It also ensures that the heap property is maintained after
# This function takes one input, the heap list
def pop_min(heap):
    min = heap[0]
    heap[0] = heap[-1]
    del heap[-1]
    sift_down(heap, 0)
    return min
```

The **pop_min()** algorithm is used to get the minimum value from the heap while maintaining the heap property. It maintains the heap property by swapping the root of the heap with the end of the heap, removing the end of the heap, and then sifting the root down.

```
# This function is used to test the different tsp algorithms, it is called by runtime_test This function takes the
# following inputs, the tsp solving algorithm as func, the costs list to record minimum cost at each input size the
# paths list to record the path at each input size, the times list to record the average runtime at each input size,
# and the number of repetitions for the input size
def test(func, costs, paths, times, distances, repetitions):
    path = 0
    cost = 0
    elapsed_time = 0
    for i in range(0, repetitions):
        start = timer()
        try:
            path, cost, nodes = func(distances.copy())
        except:
            path, cost = func(distances.copy())

        end = timer()
        elapsed_time += end - start

    paths.append(path)
    costs.append(cost)
    times.append(elapsed_time / repetitions)
```

The **test()** function takes a function as func, lists as costs, paths, and times, an adjacency matrix as distances, and a number of repetitions. Then the test() function calls whatever function was passed to it with the distances matrix as an argument a number of repetitions times. The function will then populate the costs, paths, and times lists that were passed to it.

```python
# This function compares the minimum costs of the branch and bound function versus the dynamic programming algorithm
# The function takes two inputs, the maximum input test size n_limit as an int, and the number of repetitions at each
# input size
def validation_test(n_limit, repetitions):
    my_times = []
    my_paths = []
    my_costs = []
    dynamic_times = []
    dynamic_paths = []
    dynamic_costs = []
    input_size = []

    for n in range(2, n_limit + 1):
        print(f"Input size = {n}")
        input_size.append(n)
        random_numbers = np.random.randint(0, 100, (n, n))
        distances = []
        for row in random_numbers:
            line = []
            for num in row:
                line.append(float(num))
            distances.append(line)
        distances = np.asarray(distances)
        np.fill_diagonal(distances, np.inf)

        test(tsp, my_costs, my_paths, my_times, distances.copy(), repetitions)
        test(solve_tsp_dynamic_programming, dynamic_costs, dynamic_paths, dynamic_times, distances.copy(), repetitions)

    my_costs = np.asarray(my_costs)
    dynamic_costs = np.asarray(dynamic_costs)

    print(f"For an input size: 2 to {n_limit}")
    print(f"Repetitions = {repetitions}")
    if np.array_equal(my_costs, dynamic_costs):
        print("The minimum costs from each algorithm are all the same")
    else:
        print("The minimum costs from each algorithm are not all the same")
```

The **validate_test()** function generates random adjacency matrices over a range of input sizes and calls the **test()** function on the adjacency matrices with the branch and bound and dynamic programming function as arguments. The validate_test() function then prints whether or not all the minimum costs from the branch and bound and dynamic programming algorithms were the same.

```python
# This function compares the runtime of the branch and bound algorithm versus the dynamic programming algorithm The
# function takes two inputs, the maximum input test size n_limit as an int, and the number of repetitions at each
# input size
def random_runtime_test(n_limit, repetitions):
    my_times = []
    my_paths = []
    my_costs = []
    dynamic_times = []
    dynamic_paths = []
    dynamic_costs = []
    input_size = []

    for n in range(2, n_limit + 1):
        print(f"Input size = {n}")
        input_size.append(n)
        random_numbers = np.random.randint(0, 100, (n, n))
        distances = []
        for row in random_numbers:
            line = []
            for num in row:
                line.append(float(num))
            distances.append(line)
        distances = np.asarray(distances)
        np.fill_diagonal(distances, np.inf)

        test(tsp, my_costs, my_paths, my_times, distances.copy(), repetitions)
        test(solve_tsp_dynamic_programming, dynamic_costs, dynamic_paths, dynamic_times, distances.copy(), repetitions)

    plt.plot(input_size, my_times, label="Branch and Bound")
    plt.plot(input_size, dynamic_times, label="Dynamic Programming")
    plt.title("Runtime vs Input Size")
    plt.xlabel("Input size [n]")
    plt.ylabel("Runtime [s]")
    plt.legend()
    plt.show()
```

The **random_runtime_test()** function generates random adjacency matrices over a range of given input sizes for a number of given repetitions and graphs information about the runtime of the branch and bound algorithm versus the dynamic programming algorithm

```python
# This function compares the runtime of the branch and bound algorithm versus the dynamic programming algorithm The
# function takes two inputs, the maximum input test size n_limit as an int, and the number of repetitions at each
# input size
def maximum_runtime_test(repetitions, max_time):
    my_times = []
    my_paths = []
    my_costs = []
    input_size = []

    current_average_time = 0
    n = 2
    while current_average_time <= max_time:
        print(f"Input size = {n}")
        input_size.append(n)
        random_numbers = np.random.randint(0, 100, (n, n))
        distances = []
        for row in random_numbers:
            line = []
            for num in row:
                line.append(float(num))
            distances.append(line)
        distances = np.asarray(distances)
        np.fill_diagonal(distances, np.inf)

        test(tsp, my_costs, my_paths, my_times, distances.copy(), repetitions)

        current_average_time = my_times[n - 2]
        print(current_average_time)
        n = n + 1

    plt.plot(input_size, my_times, label="Branch and Bound")
    plt.title("Runtime vs Input Size")
    plt.xlabel("Input size [n]")
    plt.ylabel("Runtime [s]")
    plt.legend()
    plt.show()
```

The **maximum_runtime_test()** is almost identical to the random_runtime_test(), but instead of running the test over a given input range, the test is run until the average runtime exceeds a given number of seconds.

Test Results

Validation Test

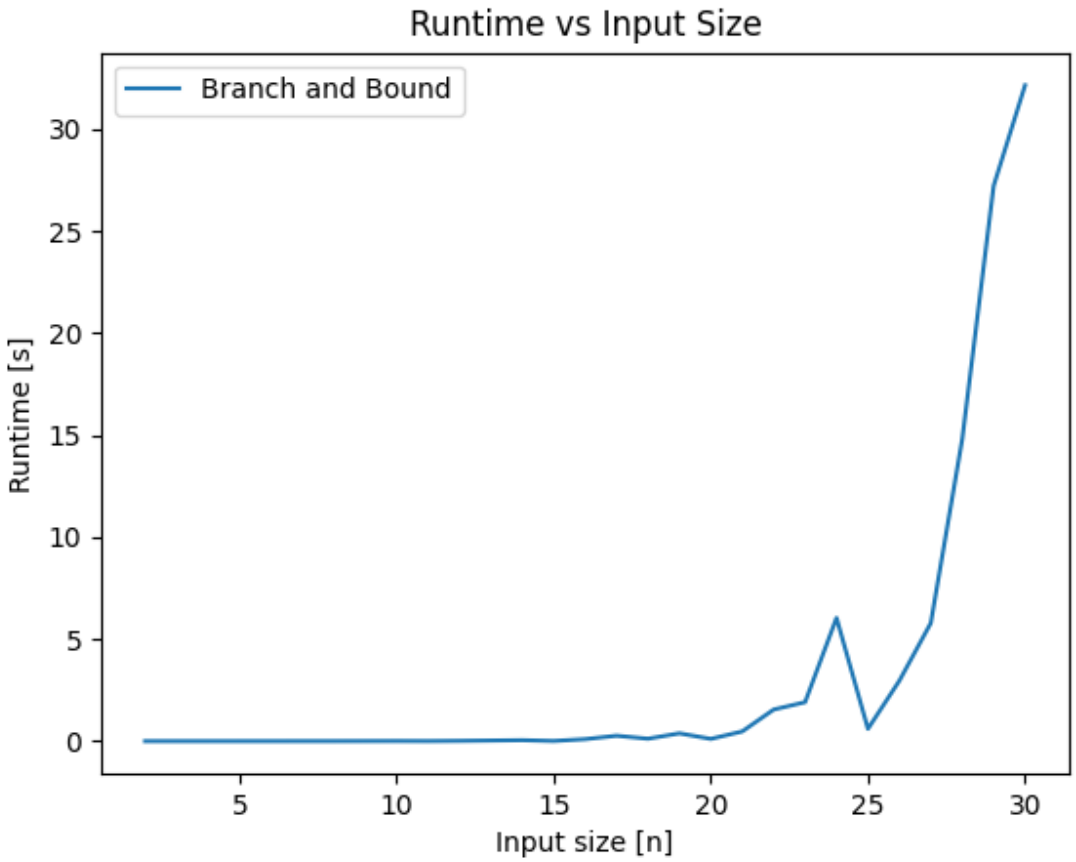
The first test was to ensure that the branch and bound algorithm returned the same minimum cost as the solver given by python_tsp. To perform this test, identical random matrices were given to the branch and bound function and the python_tsp dynamic programming solver. The input size was varied from 2 to 15 for this test as the dynamic programming solver gets quite slow for a test with high repetitions around that input size. Each input size was run for 100 different randomly generated matrices, and the minimum cost for each solution was compared.

```
C:\Users\apoto\PycharmProjects\DesignOfAlgorithms\venv\Scripts\python.exe C:\Users\apoto\PycharmProjects\DesignOfAlgorithms\TSP.py
Input size = 2
Input size = 3
Input size = 4
Input size = 5
Input size = 6
Input size = 7
Input size = 8
Input size = 9
Input size = 10
Input size = 11
Input size = 12
Input size = 13
Input size = 14
Input size = 15
For an input size: 2 to 15
Repetitions = 100
The minimum costs from each algorithm are all the same

Process finished with exit code 0
```

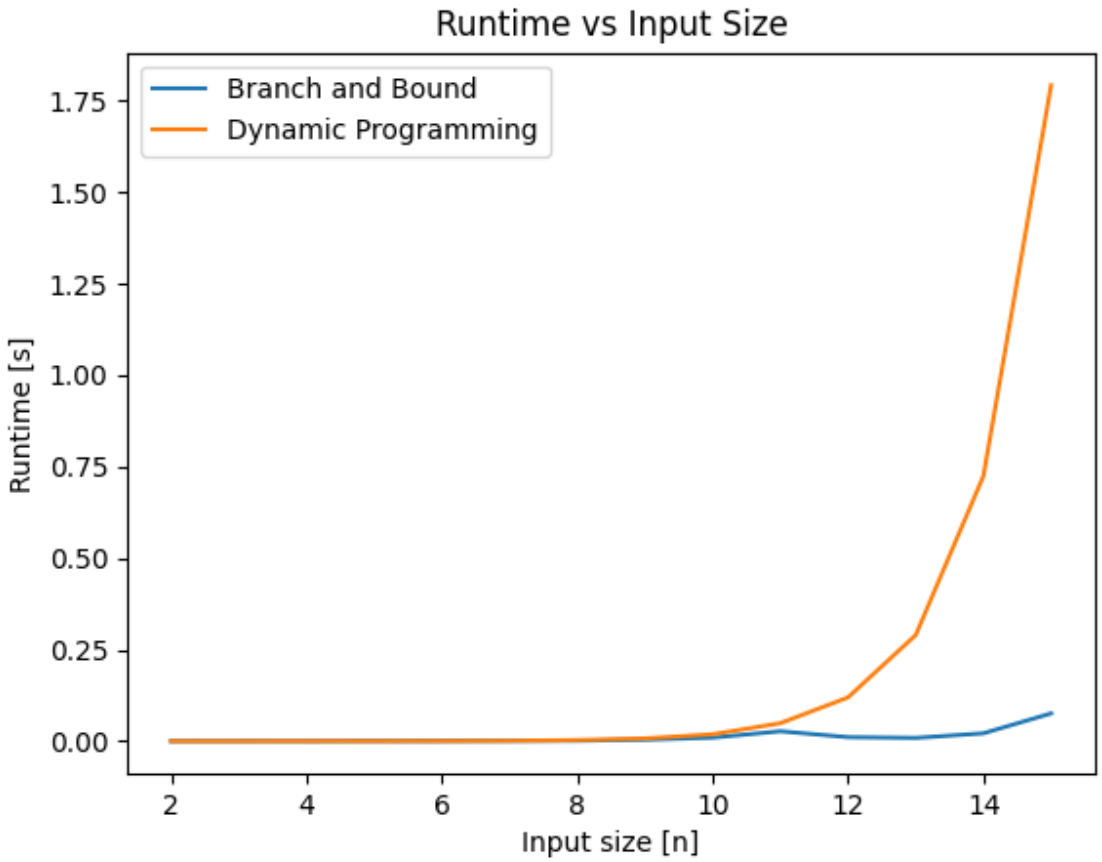
Maximum Town Test

This test involves attempting to get a better idea of what the maximum towns that can be run in a reasonably short time is. The test was run starting at an input size of 2, and repeated each input size 5 times. The amount of repetitions may seem slow, but since the test is for higher runtimes, the length of the test can get very long for high numbers of repetitions. The test was run at increasing input size until the average runtime of the input size went above 30 seconds.



Runtime Comparison Test

The next test involved comparing the average runtimes of the branch and bound algorithm versus the dynamic programming method provided in the external library python_tsp. The test was run for an input size from 2 to 15, as around an input size of 15 the dynamic programming approach starts to get long enough that the test can take a couple of minutes, and for 100 repetitions at each input size.



Result Analysis

Validation Test
The validation test run is obviously not exhaustive, but the minimum costs match for 1400 different adjacency matrices of varying sizes which certainly implies some validation.

Maximum Town Test
The plot implies that at around 30 towns, the average runtime of the algorithm for a random adjacency matrix is around 30 seconds. From running multiple runtime tests, it is clear that the contents of the input adjacency matrix heavily influences the runtime of the branch and bound algorithm. Increasing the amount of towns also in general increases the runtime of the algorithm, but there were many instances that the effect that the input has on the runtime are much stronger than the effect of adding one more town. For instance, at an input size of 24 the runtime might be 90 seconds, and at an input size of 25 the runtime could be as low as 5 seconds. This difference in runtime is accounted for by how few nodes the algorithm has to explore. As the algorithm explores more nodes, the runtime starts to tend towards the worst case time complexity of the branch and bound algorithm, which is O(n!) just like the brute force algorithm. This worst case time complexity comes from the fact that the entire state space tree might have to be explored. To prove the point that the contents of the input can have much more

of an impact than the input size, the below output was run with a 100x100 town matrix of all the same distances which took less than 10 seconds.

```
[[inf  1.  1. ...  1.  1.  1.]
 [ 1. inf  1. ...  1.  1.  1.]
 [ 1.  1. inf ...  1.  1.  1.]
 ...
 [ 1.  1.  1. ... inf  1.  1.]
 [ 1.  1.  1. ...  1. inf  1.]
 [ 1.  1.  1. ...  1.  1. inf]]
Branch and bound runtime: 9.8807023
Branch and bound minimum cost: 100.0
Branch and bound path: [0, 1, 99, 98,

Process finished with exit code 0
```

Note: The tour was not included as it is much too long to fit on the screen.

And here is an output run with a 13x13 matrix of alternative rows of [2,1,2,1…],[1,2,1,2…] which took more than 2 minutes despite having 87 fewer towns to look at.

```
[[inf  1.  2.  1.  2.  1.  2.  1.  2.  1.  2.  1.  2.]
 [ 1. inf  1.  2.  1.  2.  1.  2.  1.  2.  1.  2.  1.]
 [ 2.  1. inf  1.  2.  1.  2.  1.  2.  1.  2.  1.  2.]
 [ 1.  2.  1. inf  1.  2.  1.  2.  1.  2.  1.  2.  1.]
 [ 2.  1.  2.  1. inf  1.  2.  1.  2.  1.  2.  1.  2.]
 [ 1.  2.  1.  2.  1. inf  1.  2.  1.  2.  1.  2.  1.]
 [ 2.  1.  2.  1.  2.  1. inf  1.  2.  1.  2.  1.  2.]
 [ 1.  2.  1.  2.  1.  2.  1. inf  1.  2.  1.  2.  1.]
 [ 2.  1.  2.  1.  2.  1.  2.  1. inf  1.  2.  1.  2.]
 [ 1.  2.  1.  2.  1.  2.  1.  2.  1. inf  1.  2.  1.]
 [ 2.  1.  2.  1.  2.  1.  2.  1.  2.  1. inf  1.  2.]
 [ 1.  2.  1.  2.  1.  2.  1.  2.  1.  2.  1. inf  1.]
 [ 2.  1.  2.  1.  2.  1.  2.  1.  2.  1.  2.  1. inf]]
Branch and bound runtime: 141.4655171
Branch and bound minimum cost: 14.0
Branch and bound path: [0, 5, 6, 11, 12, 1, 2, 7, 10, 3, 4, 9, 8]

Process finished with exit code 0
```

Runtime Comparison Test

The runtime of the branch and bound algorithm for a random adjacency matrix seems to beat that of the dynamic programming algorithm. The dynamic programming approach runtime plots were very consistent, usually running in roughly the same amount of time at each input size, and always tended upwards in an exponential fashion, which is to be expected. The runtime plots of the branch and bound method generally tended upwards, but many times this rule was broken and the graphs had a sawtooth shape somewhere in the plot. Additionally, at a given input size the runtime could possibly vary by minutes. This is because of the aforementioned influence of the contents of the input on the runtime, being as important as, if not more important, than the input size.

Conclusion

The best first search branch and bound algorithm seems to outperform the dynamic programming algorithm, and certainly the brute force algorithm for a random adjacency matrix in terms of shortest runtime. In none of the random runtime tests run did the dynamic programming approach ever beat the branch and bound approach at an input size around 10 and above. The branch and bound algorithms worst case scenario input rarely shows up in the random tests, but some adjacency matrices are certainly worse than others. During certain tests with a low repetition, it was clear that the runtime could vary by minutes depending on the adjacency matrix. The branch and bound algorithm seems to work very quickly for practically any random matrix up to an input size of about 25 on my computer, but it can certainly be applied to matrices with a larger input size if the input contents are not too close to the worst case scenario for the algorithm.